PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

Complex goals for planning in nondeterministic domains: preferences and strategies

Dzmitry Shaparau

Advisor:

Prof. Paolo Traverso

Fondazione Bruno Kessler

Co-Advisor:

Prof. Marco Pistore

Fondazione Bruno Kessler

February 2008

Acknowledgements

First of all, I would like to thank my advisors, Marco Pistore and Paolo Traverso, for their support, patience, and encouragement throughout these years. I am grateful to them for allowing me to do research in their group. Without their brilliant guidance, this dissertation could not be done.

I sincerely thank all my colleagues and collaborators from SOA team for providing a friendly environment, where it has been a great pleasure for me to work.

I wish to thank my thesis reviewers, Daniel Borrajo, Enrico Giunchiglia, and Roberto Sebastiani, who have spent much time in reviewing and provided useful feedback.

I am grateful to all my friends who have contributed to this thesis with their comments and advises.

Finally, I would like to thank my parents for their life-long support and warm encouragement. I want to especially acknowledge my wife, Olga, for her love and patience. This dissertation is dedicated to her.

Abstract

Modern applications require automated planning techniques to deal with uncertain environment, actions with nondeterministic outcomes, and objectives with a very complex structure. Widely spread uncertainty requires planning for extended goals that are not simply a set of states to be reached, but also a set of conditions to be satisfied during the plan execution. This thesis addresses the problem of planning for two classes of extended goals: goals with preferences and goals obtained by fusing of procedural and declarative constructs.

The importance of the problems of planning with actions that have nondeterministic effects and of planning with goal preferences has been widely recognized, and several works address these two problems separately. However, combining planning under uncertainty with goal preferences adds some new difficulties to the problem. Indeed, even the notion of optimal plan is far from trivial, since plans in nondeterministic domains can result in several different behaviors satisfying conditions with different preferences. Planning for optimal conditional plans must therefore take into account the different behaviors, and conditionally search for the highest preference that can be achieved. In this work, we address this problem. We formalize the notion of optimal conditional plan, and we describe a correct and complete planning algorithm that is guaranteed to find optimal solutions.

Another important class of planning objectives for nondeterministic do-

mains is motivated by the fact that in most planning approaches, goals and plans are different objects: goals are declarative requirements on what has to be achieved, and plans are procedural specification on how to achieve goals. This is the case as for classical reachability goals as for more complex setting, such as temporally extended goals. However, we might need to specify some planning objectives as procedures to be executed and vice versa. The combination of procedural constructs with declarative temporally extended goals can significantly increase the expressiveness of the goal language and simplify the process of goal development for human domain expert. Moreover, the procedural approach allows for embedding the domain specific knowledge in the goal to lighten the plan search process. We propose a novel language for expressing temporally extended goals for planning in nondeterministic domains. The key feature of this language is that it allows for an arbitrary combination of declarative goals expressed in temporal logic and procedural goals expressed as plan fragments.

We implement the proposed techniques as a framework that includes: the novel goal language for each class of extended goals we addressed in the thesis, the formal model for these goal languages, and planning algorithms for such kind of goals. We perform a set of experimental evaluations that show the potentialities of our approaches.

Keywords

Planning under uncertainty, planning with extended goals, planning with preferences

Contents

List of Figures

Chapter 1

Introduction

Automated Planning is a process of automated reasoning that chooses and schedules actions based on their preconditions and expected outcomes in order to archive as best as possible preliminarily defined objectives. The modern artificial intelligence systems are characterized by a large amount of independent or partially dependent agents with specific responsibilities and life-cycles. Moreover, in many cases the supervisor software has a very restricted access to the system under control. These requirements inspire the evolution of approaches and techniques for automated planning of the behavior of large space domains in uncertain environment.

With respect to uncertainty, most of application domains are nondeterministic. In this case actions can have more than one possible outcome that cannot be predicted during the planning phase. In many cases nominal outcomes do not exist at all, because it is impossible to guarantee that something exceptional is not happen. Therefore, the possibility to deal with nondeterminism became an obligatory property for a planning framework to be applied to the real-life problems.

In recent works, there are two major approaches to model nondeterminism: (i) all action outcomes can happen with equal probability, and (ii) the probability is assigned to each action outcome. Although the probabilistic

approach gives an opportunity to define a planning domain more precisely, the planning techniques based on the qualitative model are traditionally much more faster. Moreover, in many domains there is no possibility to specify the exact probabilities for action outcomes, therefore the best model for such applications is a qualitative one. In this thesis we do not consider the probabilistic model and assume that all action outcomes can happen with equal probability.

Planning in nondeterministic domains leads to the fact that the execution trace of the generated plan can result in more than one sequence of domain states. In this case the plan requires a conditional representation where the next action to execute depends on the current domain state. One of the most promising approaches for planning under uncertainty is a planning based on the *model checking*.

Model checking is a set of formal techniques to verify whether a system satisfies a given logical formula [35]. It was successfully applied to the problem of requirements and design verification in complex hardware and software systems [6]. The major benefit of the model checking techniques is that it is extremely fast and can deal with large space application domains. Planning as model checking has shown outstanding results in nondeterministic domains [33, 73, 9, 10, 30]. The proposed approach is based on the idea of symbolic model checking [36, 48, 84] which models a planning domain as a state-transition system in terms of propositional logic formulas encoded as Ordered Binary Decision Diagrams (BDD) [21, 22]. In this work we exploit advantages of the symbolic model checking techniques and apply them to new classes of planning goals.

In past decades, a lot of planning techniques and approaches has been proposed. One of the most important criteria to evaluate usefulness and powerfulness of the planning framework is a range of planning objectives that can be solved using this framework. Planning for classical reachability

goals has been deeply investigated, but in many cases such trivial goals can not express planning objectives of the real-life applications. The famous example of complicated planning goals is *temporally extended goals* that are not simply a set of states to be reached, but also a set of conditions or sub-goals to be satisfied during the plan execution. For instance, in a robotic application, we may need to specify that a robot should "move from the location A to the location B avoiding the wet surfaces along the path, and if the robot meets a human it has to salute him". This goal can be represented as a combination of the *reachability goal* "move from the location A to the location B", the *maintainability goal* "avoid the wet surfaces", and the *conditional goal* "if the robot meets a human it has to salute him". Moreover, goals should take into account nondeterminism and express behaviors that should hold for all plan execution paths and behaviors that should hold only for some of them. Typically, such goals is expressed in terms of temporal logics [39], but there are special goal languages that adapts major benefits of temporal logic concepts for the planning needs and improves them by adding new types of extended goals (see, for instance, [55]).

With respect to the fact that the effectiveness of extended goals has been widely recognized, in this dissertation we deal with two novel classes of extended goals in uncertainty environment that was not addressed in the literature before.

1.1 Contributions

In this dissertation we develop an approach to apply planning to two novel classes of extended goals in nondeterministic domains: *goals with preferences*, and *procedural goals* intermixed with *declarative goals*. In both cases we provide a theoretical framework that includes (i) a novel goal language

for each class of extended goals we discussed, (*ii*) a formal model for this goal language, and (*iii*) a planning algorithm that can be used to resolve such kind of planning goals. The proposed approach is evaluated on a large space domains to show its effectiveness and powerfulness.

Planning with preferences

The importance of the problem of *planning with goal preferences* has been widely recognized (see, for instance, [18, 85, 80, 17]). In planning with preferences, the user can express preferences over goals and situations, and the planner must generate plans that meet these preferences. More and more research is addressing these two important problems separately, but the problem of *planning with goal preferences under uncertainty* has not been addressed yet. This is clearly an interesting problem, and most applications need to deal both with nondeterminism and with preferences. However, providing planning techniques that can address the combination of the two aspects requires to solve some difficulties, both conceptually and practically.

Conceptually, even the notion of optimal plan is far from trivial. Plans in nondeterministic domains can result in several different behaviors, some of them satisfying conditions with different preferences. Planning for optimal conditional plans must therefore take into account the different behaviors, and conditionally search for the highest preference that can be achieved. In deterministic domains, the planner can plan for (one of) the most preferred goal, and in the case there is no solution, it can iteratively look for less preferred plans. This approach is not possible with nondeterministic domains. Since actions are nondeterministic, the planner will know only at run-time whether a preference is met or not, and consequently a plan must interleave different levels of preferences, depending on the different action outcomes. In this setting, devising planning algorithms that can work *in*

practice, with large domains and with complex preference specifications, is even more an open challenge.

In the first part of the thesis we address this problem, both from a conceptual and a practical point of view. Differently from the decision theoretic approach taken, e.g., in [15] we allow for an explicit and qualitative model of goal preferences. The model is *explicit*, since it is possible to specify explicitly that one goal is better than another one. It is *qualitative*, since preferences are interpreted as an order over the goals. Moreover, the model takes into account the nondeterminism of the domain. If we specify that a goal g_1 is better than g_2, we mean that the planner should achieve g_1 in all the cases in which g_1 can be achieved, and it should achieve g_2 in all the other cases.

We define formally the notion of *strong* and *strong cyclic* optimal conditional plan, and we devise planning algorithms for the corresponding planning problems. The algorithms are complete and correct, i.e., they are guaranteed to find only optimal solutions, and to find an optimal plan if it exists. We have designed the algorithms to deal in practice with complex problems, i.e., with large domains and complex goal preference specifications.

We perform a preliminary set of experimental evaluations with some examples taken from two different domains: robot navigation and web service composition. We evaluate the performances w.r.t. the dimension of the domain, as well as w.r.t. the complexity of the goal preference specification (increasing the number of preferences). The experimental evaluation shows the potentialities of our approach.

Fusing procedural and declarative goals

In most planning approaches, goals and plans are different objects: goals are declarative requirements on what has to be achieved, and plans are

procedural specification on how to achieve goals. This is the case of classical planning, where goals are conditions on states to be reached and plans specify sequences of actions. This is also the case of more expressive and complex settings, such as planning with temporally extended goals in nondeterministic domains, where goals are, e.g., formulas in a temporal logic, and plans are, e.g., policies or conditional and iterative combinations of actions, see, e.g. , [49, 74, 37, 51].

However, it is often important to have the possibility to combine declarative goals and procedural plans, and this is especially useful for planning in nondeterministic domains for temporally extended goals. Indeed, in nondeterministic domains, it is often useful to specify partial plans, i.e. plans of actions to be executed only for a subset of the possible outcomes of the plan execution, while we might need to specify declarative goals to be achieved when uncovered states are reached. For instance, we can specify directly nominal plans that are interleaved with declarative conditions to be satisfied in case of failure. Vice versa, we can interleave a declarative goal specification with procedures to be executed as exception handling routines that recover from dangerous failures.

In the case of temporally extended goals, parts of the goals can be better specified directly as procedures to be executed than as temporal formulas to be satisfied. For instance, a goal for a robot that has to visit periodically some locations in a building, can be easily specified as a procedural iteration interleaved with a declarative specification of the states to be reached, rather than as a temporal formula with nested maintenance and reachability conditions.

In this dissertation, we propose a novel language for expressing temporally extended goals for planning in nondeterministic domains. The key feature of this language is that it allows for an arbitrary combination of declarative goals expressed in temporal logic and procedural goals

expressed as plan fragments. It combines declarative goals expressed in temporal logics (in particular, we adopt the EaGLe language [37] for temporal goals) with procedural specifications such as conditional and iterative plans, policies and failure recovery control constructs. We provide a formal definition of the language and its semantics.

We also propose an approach for planning with this language in nondeterministic domains. The idea is to construct control automata that represent properly the interleaving of procedural and declarative goals. We can then use such automata to control the search for a plan in a similar way as in [37]. We evaluate the algorithm performances w.r.t. the dimension of the domain. The experimental evaluation shows the potentialities of our approach: a rather simple and natural combination of procedural and declarative goals allows us to scale up of order of magnitudes w.r.t. fully declarative goals.

1.2 Thesis Organization

The thesis is organized in the following way. In Chapter 2 we give a brief overview of automated planning areas connected with the topic of the thesis. This overview starts from the formalization of the general conceptual model for a planning framework and describes the main challenges for planning under uncertainty. Chapter 3 reviews basic definitions of planning in nondeterministic domains, i.e. the definition of nondeterministic domain, typical planning problem with reachability goals, and different types of solutions: strong, strong-cyclic and weak. In Chapter 4 we propose a novel framework for *planning with goal preferences* in nondeterministic domains. This proposal includes a formal model for a novel goal language and two planning algorithms to generate the optimal plan in respect to the goal preferences. Chapter 5 gives a background in planning for temporally ex-

tended goals. In particular, the chapter gives an intuition of the planning approach for temporally extended goals which is based on the plan search process guided by the control automata. In Chapter 6 we propose an approach to deal with a novel class of extended goals that allows for fusing procedural and declarative goals. Chapter 7 presents the implementation and evaluation of the planning techniques proposed in this thesis. Concluding remarks and directions for future work are discussed in Chapter 8.

Chapter 2

State of the Art

In this chapter we give a brief overview of automated planning areas connected with the topic of the thesis. We introduce a general conceptual model of the planning framework and describe the main challenges for planning under uncertainty. As we will se, one of the key points in the planning under uncertainty is a language for extended goals. One of the main impacts of this work is a new goal language that allows mixing procedural and declarative approaches for goal definitions. Therefore, we revise the related planning techniques in this area, in particular Hierarchical Task Network planning, action language Golog, and search control knowledge. We describe their main concepts and compare them in respect to this work. Finally, we give an introduction to the planning with preferences, which is also a central research area of this work.

2.1 Automated planning

Automated planning is an area of AI that investigates the process of automated reasoning over a set of available actions by organizing their executions to satisfy a planning goal. The action is characterized by preconditions that describe the requirements to the system state where it can be executed, and effects that describe changes in the system caused by

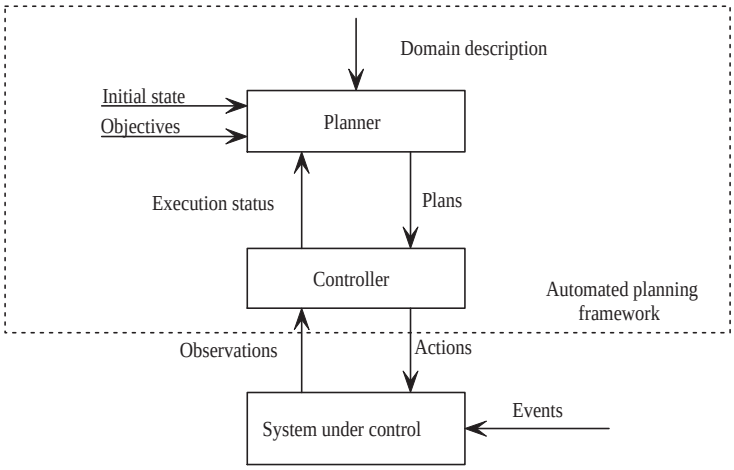


Figure 2.1: A general conceptual model of the planning framework.

the action execution. There are a lot of different types of applications which need automated reasoning and, hence, there are various types of planning, i.e. motion planning [56, 28], perception planning [5], planning in space application [1, 19], scheduling [24], automated service composition [68, 75, 72, 79, 82, 57, 58], automated synthesis of controllers [50] and etc. Different planning forms are based on different principles of the problem representation and solving techniques, therefore planning techniques that adapt and exploit domain specific knowledge usually show better results in such domains in comparison with universal planning approaches, i.e. domain-independent planning. But it is quite costly and unfeasible to develop a new planning area for each narrow direction. Therefore recent research is concentrated on the domain-independent techniques for automated planning. A general conceptual model of the planning framework is explicitly described in [66]. It can be represented as shown on Figure 2.1. It consists of three main components: *planner*, *controller*, and *system under control*.

The planner accepts a planning domain, an initial domain state and objectives as input parameters. The planning domain is a formal model

emulating the system under control. Typically it is represented in terms of system states and actions that describe the behavior of system transitions between states. The planning domain depending on the real system under control can be:

- finite or infinite. Almost of planning techniques require the finiteness of the domain.

- deterministic or nondeterministic. Actions in nondeterministic domain can have more than one outcome. The deterministic domains are simpler for planning, but unfortunately, most of real-life systems are nondeterministic.

- static or dynamic. Dynamic domains are characterized by a non-empty set of external events that can be happen in the system unpredictably due to some external reasons. Such events are uncontrollable for the planning framework. On the contrary, static domains do not model external events, therefore any system change is modeled as a result of the action execution, which is controlled by the planning framework.

The initial domain state defines an initial state of the system under control. The objectives express the planning goal. Typically, the planning goal is a constraint for the final domain state, i.e. it describes the state to which the system has to be transited. The planner generates a plan that is an instruction set to transit the domain from the initial state to the state satisfying specified objectives.

The controller is a component which controls the plan execution process. It observes the current system state and provides an action to be performed according to the synthesized plan. In the *full observable* system the controller exactly knows current system state, but there is a very important

class of real-life applications which are characterized by the *partial observability*. This means that the controller in some cases can only detect only some parameters of the domain state. In such cases the controller can send the current plan execution status to the planner and request the planner to tune the plan according to the obtained partial observable situation.

If we assume that planning objectives are system states that has to be reached and the system under control is finite, deterministic, static, and fully observable then we get a problem of *classical planning*. It is well known and deeply investigated [66]. But the assumptions of the classical planning are very restrictive and unacceptable in the real-life. For this reason, one of the central areas in the world of automated planning is *planning under uncertainty*.

2.2 Planning under Uncertainty

Planning under uncertainty relaxes assumptions of the classical planning about domain determinism, full observability and reachability goals.

In most of real-world applications we can not guarantee the unique outcome for each action. In this case we have a *nondeterministic* domain where actions can have more than one possible result that cannot be predicted at planning time. Sometimes nominal outcomes do not exist at all, sometimes planning model should take into account possible action failures. Indeed, ever a very simple action in our real life such as "login to PC" can end up in at least three states: "login is successful", "your password has been expired", and "your account has been deactivated by the administrator". In this work we are based on the most general assumption for nondeterministic outcomes: all possible outcomes can happen with equal probability. The alternative approach is *probabilistic planning* [62, 63, 13, 14, 60]. It requires to model probabilities in the action outcomes, while our point is

to have a qualitative model of the domain.

Another typical challenge for real applications is *partial observability*. It means that the controller can not unambiguously define the current state of the system, because it gathers information from sensors and some domain variables can be unobservable. For example, a robot typically can investigate the outward things only using its personal sensors having restricted work range, e.g. video cameras or microphones. Therefore, the robot does not know the state of the world objects hidden from its sensors. For example, in order to detect whether the door is opened or not the robot should take place near of the door and use sensors. In this case current system state is represented by the set of all possible states according to the last observations, i.e. some system states become indistinguishable due to insufficient information about current domain state. It makes the planning difficult, because it should deal with a search space as a power of set of states. In this work we do not touch the problem of partial observability and deal only with full observable domains.

Widely spread uncertainty in the real-life applications forces us to perform planning for *extended goals* that are not simply a set of states to be reached, but also a set of conditions to be satisfied during the plan execution. Because of domain nondeterminism, we sometimes need to define a goal that can be satisfied only in some of possible plan execution paths, i.e. "try reach" strategy that gives only a chance to reach a goal, but does not guarantee this. However, we often need to satisfy a goal in all possible execution paths in spite of nondeterminism, i.e. "do reach" strategy that guarantees the goal satisfiability. Additional temporal conditions that should be maintained during the plan execution is also very useful. For example, we can define a goal to reach some final states but be sure that safety conditions are not broken along the plan execution path. Temporal conditions allow for specifying complex goal types such as cyclic goals,

conditional goals, sequential goals, and etc.

Several approaches from the classical planning like satisfiability planning [26, 63, 61, 44, 45] or graph-based planning [86, 87] can be applied to these challenges, but they are able to solve only some restricted forms of uncertainty. There are a set of works dedicated to the heuristic forward search [47] and theorem proving techniques [78]. One of the most promising approaches for the planning under uncertainty is a planning based on the *model checking*.

2.2.1 Planning as model checking

Model checking is a set of formal techniques to verify whether a system satisfies a given logical formula [35]. It was successfully applied to the problem of requirements and design verification in complex hardware and software systems [6].

A typical model checking algorithm accepts a system model and property that the system is expected to satisfy as input parameters. It returns as output either "TRUE", if the system model satisfies the specified system property, or a counterexample which details how the system can falsify the specified property. The system property is typically expressed by a temporal logic formula [39]. The process of counterexample synthesis can be considered as a planning process to build a plan that transits the system to the state where the given property falsifies. The major benefits of model checking that forces to re-use its ideas in planning are:

- Model checking is extremely fast and can deal with extremely large domains [48].

- The model definition in model checking is very similar to the planning domain definition.

- The logical formula describing the system property for model checking can be easily adapted to the goal definition for automated planning.

Planning as model checking was first introduced in [33]. Proposed algorithms are implemented in the planner MBP [8] based on the model checker NuSMV [29, 31, 32]. The new approach has shown outstanding results dealing with all classes of uncertainty: nondeterminism, partial observability and extended goals [33, 73, 9, 10, 30]. The proposed planning technique is based on the idea of symbolic model checking [36, 48, 84] which models a state-transition system in terms of propositional logic formulas encoded as Ordered Binary Decision Diagrams (BDD) [21, 22]. BDD allows encoding large space domains in a very compact way [48, 65]. In this case the system is modeled in terms of boolean propositions and each system state is represented by the propositional formula. Moreover, a set of system states and transitions can be encoded as a single propositional formula, i.e. compactly and efficiently. The planner based on the symbolic model checking searches through the domain state space using logical operations over propositional formulas that are efficiently realized in BDD, i.e. the planner searches sets of states and sets of transitions at once, rather the single state and transition.

2.2.2 Extended goals

One of the key points in the planning under uncertainty is a language for extended goals. Its expressiveness defines a scope of problems that can be solved. The most widely used approach to express extended goals is encoding them as a temporal logic formula, i.e. temporally extended goals. Indeed, Liner Temporal Logic (LTL) [39] provides a set of temporal operators such as "next" (X), "finally" (F), "globally" (G), "until" (U) that can be used to define constrains not only on the terminal state to reach, but

also on some intermediate states passed during the plan execution. As example of combination of reachability, sequential and maintainability goals, let us consider following planning goal: *"reach state g_1, then reach state g_2; and be sure that property p holds in all states passed by the plan execution trace"*. It can be expressed by the LTL formula $(\mathtt{F}(g_1) \rightarrow \mathtt{X}(\mathtt{F}(g_2))) \wedge \mathtt{G}(p)$. Moreover, LTL also covers a various useful types of temporally extended goals such as cyclic goals and reactive goals [25, 3, 4].

Computation Tree Logic (CTL) [39] gives the ability to express temporal behaviors that take into account nondeterminism of the application domain [73, 70]. In contrast to LTL, CTL temporal operators can be used only together with path quantifiers:

- Universal quantifier (A) states that the temporal operator should hold in all possible execution paths.
- Existential quantifier (E) states that the temporal operator should hold at least in one of possible execution paths.

For example, the goal AFg requires a plan that guarantees reachability of the state g in spite of nondeterministic domain behavior. The alternative variant EFg is satisfied by the plan that gives at least one chance to reach the state g. We note that LTL and CTL have different expressiveness, i.e. there is a set of goal types that can be expressed in LTL and can not be expressed in CTL, and vice versa.

However, in many practical cases the expressiveness of the temporal logic is not adequate. For example, the classical interpretation of existential quantifier is too "weak" for planning problems. Most of nondeterministic planning problems requires not only to find a plan that gives at least one chance to reach the goal, but it requires "to do everything possible" to satisfy a goal. It means that the goal needs to define some intentional aspects for the resulting plan, and such aspects can not be expressed by

the temporal logic formula.

Moreover, ever if we have a plan that actually tries to reach a given goal, its execution path can nevertheless have a chance to fail if some nondeterministic action transits a system to a state where a given goal becomes truly unreachable. In this case we need a possibility to define alternative recovery or compensation goals to finish the plan execution correctly. This point is especially important because nondeterministic outcome can not be predicted on the planning phase and the planner has to take into account all "undesired" system behaviors. For example, the goal for a coffee machine "*(try to make coffee)* ***FAIL*** *(send a help request to a service center)*" can be interpreted as "The machine has to do its best to make coffee, but if it becomes impossible, a help request has to be sent to a service center".

According to these reasons a new goal language EaGLe was developed [55]. It keeps the benefits of temporal logics, adapts them for the planning needs and improves them by adding new types of extended goals. As example, EaGLe has operators like *TryReach* and *Fail* which allow to express described above complex goals.

2.3 Procedural approaches in planning

Traditionally, the goal and the resulting plan are considered as completely independent and unrelated objects. There is no possibility to reuse previously obtained plan in definition of the new goal and vice versa. Ever if the human domain expert has idea about some steps that should be in plan, he can not express this knowledge in the goal. This is a very important open problem that was not directly addressed before in the literature. One of the main impacts of the thesis is dedicated to this problem. In next subsections we give a short overview of related works that are based on some procedural approaches in different planning aspects, i.e. definition of

planning domains, goals and search control heuristics.

2.3.1 Hierarchical Task Network planning

The main difference of Hierarchical Task Network (HTN) planning [83, 40, 67] from others kinds of planning is that the objective is not to archive a set of goals but to perform a set of *tasks*. The planner accepts a set of *methods* as an additional domain description, where each method describes how to decompose some task into a set of smaller tasks. The planning process recursively decomposes each task into smaller and smaller subtasks until *primitive tasks* are reached, i.e. tasks that do not have a method for the decomposition. Hence, all tasks are organized into hierarchical levels. HTN methods generally describe the standard operating procedures that one would normally use to perform tasks in some domain. Such representations are usually more appropriate for many real-world domains than are classical planning operators, as they better characterize the way that users think about problems [67].

HTN planning is based on the assumption that the planning domain is deterministic and fully observable. It can be easily extended to accommodate restricted sets of extended goals. For example, maintainability goal "except dangerous room A" can be satisfied if we add additional precondition "room $\neg A$" to the *method* "moveTo(room)". Goals like "visit room A at least three times" can be expressed by introducing predicate symbols, function symbols and integer arithmetic. Moreover, there is an extension of the forward-chaining planning algorithm based on the HTNs for nondeterministic domains [52]. Recently, there was developed a novel approach which combines the power of the HTN-based search-control strategies with BDD-based symbolic model checking techniques [53].

In comparison with classical planning, the main advantage of HTN planning is that it provides a convenient way to write the problem solving

heuristics for the human domain experts. A variety of classical and non-classical problems can be efficiently represented and solved with a good set of HTNs. Therefore, this technique was mostly used in the planning applications [88]. Main disadvantage of the HTN planning is the domain specification takes more time for the definition, because domain author should describe not only domain operators but also methods.

HTNs provide an evidence that goals and plans can be interconnected with a great benefit. In this work we adapt and extend the idea of HTN tasks to the planning goal definition. We define a planning goal as a hierarchial combination of procedural operators and declarative reachability goals. We will also see that the idea of decomposing of the complex goal into a set of simple ones can be effectively applied to the planning for extended goals.

2.3.2 Golog

Golog [59] is a procedural logical programming language for systems whose design is based on the logic of actions. It is based on Reiter's version of the situation calculus [64, 77]. Golog was developed for robot programming and to support high-level robot task planning (e.g. [23]). Golog is equipped with control structures like sequence, loops, conditionals, but also less standard constructs like the nondeterministic choice of actions. Such constructions help to assemble primitive action calls into complex actions and procedures. There are a lot of extensions for dealing with continuous change [46], decision-theoretic planning [16], concurrency [43], allowing for exogenous and many others.

The execution of Golog programs generates a sequence of primitive action calls that transits the domain from an initial state to a goal state. In spirit it is very close to planning. In this work we adapt the idea of procedural control structures to the goal declaration in planning under

uncertainty.

2.3.3 Search control knowledge

The most promising way to optimize the plan search process is to provide to the planner an additional knowledge about the planning domain. HTN and Golog in spirit are based on this idea, i.e. smart selection of the HTN domain specific tasks and Golog procedures built over primitive action calls can be used to optimize the planning process. A set of planners allows providing additional search control knowledge in parallel to the application domain definition, e.g. TLPlan [3], TALPlanner [54], or SHOP2 [67]. One of the most promising approaches is to define additional search control knowledge as a temporal logic formula [4, 51]. In this case the additional modality GOAL is introduced. GOAL(f) means that the temporal logic formula f is true in all goal states. For instance, we can define a search control rule for an abstract robot navigation domain such as "whenever getting object x at location y is part of the current goal, the robot, if it happens to be holding x, should hold on to it until he is at y" [51]. Note the usage of first-order quantifications "whenever" and "until", and GOAL modality "x at location y is part of the current goal". This rule declares that if box x has to be in location y in all goal states, then if the robot holds x then it has to hold it until it reaches location y, i.e. the robot can not throw such box in a location different from y. The set of such search control rules defined by the human domain experts allows significantly reducing the planning search space. In this work we propose an alternative idea for the search control knowledge. It is based on the fusing procedural and declarative approaches for planning goal definition.

2.4 Planning with goal preferences

The importance of the problem of planning with preferences has been widely recognized [47, 78, 80]. Planning with preferences is extremely important for planning domains were all goals defined by human operator can not be satisfied due to some reasons. Many application domains assume that the planning goal is a conjunction of some properties that need to be satisfied. It is a widely spread situation when the resulting real-life problem can not be resolved, i.e. there is no plan to reach a state where all properties encoded in the planning goal are satisfied. In this case one of the possible solutions is to find plans that satisfy only a subset of goal properties. This problem is known as partial satisfiability planning.

The intuitive examples of application that requires planning with preferences are NASA planning problems [80]. These problems are characterized by the large number of possible goals of different values and the planning system must choose only a subset that can be accomplished within limited time and resources.

The most common approach in this case is to find a plan that satisfies as many goal properties as possible [47]. The alternative approach is to define the preferences among different goal properties [18]. There are two ways to establish preferences between goals: quantitative (i.e. using numbers) and qualitative (i.e. using ordering relations between goal properties). The main benefit of the qualitative approach is that it is quite easy for human operator, but it becomes a nightmare in the complex problems with dozens of different goal properties. Quantitative goals preferences are more general. In this case the planner can imply relations between different goal properties based on their preference numbers. The intuitive way to set up goals preferences is to assign an utility value to each goal property. In this case we need to find a plan that transits the system to the state

with maximal sum of utility values.

Most of existed approaches to planning with goal preferences do not address the problem of conditional planning, but are restricted either to deterministic domains, and/or to the generation of sequential plans. This is the case of planning as satisfiability with preferences [44], of planning for multiple criteria [76], of the work on over-subscription planning [85, 80], of CSP-based planning for qualitative specifications of conditional preferences [18], and of preference-based planning in the situation calculus [11]. In the field of answer set programming, [38] addresses the problem of generating sequences of actions that are conformant optimal plans in domains where nondeterministic actions have associated costs. [81] proposes a language for expressing plan preferences over plan trajectories, whose foundations are similar to those of general rank-based languages for the representation of qualitative preferences [20].

Planning in deterministic domains becomes quite trivial. Intuitively, we iterate through all possible goals one by one in decreasing order in respect to the preferences and perform planning for each goal until we find a resolvable one. So, the planning with preferences can be translated to the set of usual planning problems. In this case the planning challenge is to find the optimal plan as fast as possible.

Unfortunately, this idea can not be applied to nondeterministic domains, because the failure of the plan execution can be detected only during its execution. Therefore, the same plan can satisfy different user preferences depending on outcomes of the nondeterministic actions. In this case we need to measure the plans quality taking into account all possible plan execution paths.

In this work we propose a solution to the problem of conditional planning with goal preferences in nondeterministic domains. We provide a novel goal language to setup user preferences, a notion of the optimal plan in

nondeterministic domains, and planning algorithms to solve such kind of planning problems.

Decision theoretic planning, e.g., based on MDP [15] is a different approach that can deal with preferences in nondeterministic domains. However, there are several differences with our approach, both conceptual and practical. First, in most of the work on decision theoretic planning goals are not defined explicitly, but as conditions on the planning domain, e.g., as rewards/costs on domain states/actions, while our goal preference model is explicit. Second, we propose a planner that works on a qualitative model of preferences. In decision theoretic planning, optimal plans are generated by maximizing an expected utility function. Finally, from the practical point of view, the expressiveness of the MDP approach is more difficult to be managed in the case of large state spaces.

Chapter 3

Planning in Nondeterministic Domains

The aim of this section is to review basic definitions of planning in nondeterministic domains which we use in the thesis. All of them are taken, with minor modifications, from [34]. We start from the definition of nondeterministic domain. Then we introduce a typical planning problem with reachability goal and different types of solutions: strong, strong-cyclic and weak. We compare these types in respect to the quality and describe a basic planning algorithm for each of them. In next chapters we will actively use the notion of strong, strong-cyclic and weak plans. These concepts are critically important for the planning with preferences in nondeterministic domains.

3.1 Planning Domains and Planning Problems

We model a *(nondeterministic) planning domain* in terms of *propositions*, which characterize system *states*, of *actions*, and of a *transition relation* describing system evolution from one state to possible many different states.

Definition 1 (Planning Domain). *A planning domain $\mathcal{D}$ is a 4-tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ where*

- $\mathcal{P}$ *is the finite set of basic propositions,*
- $\mathcal{S} \subseteq 2^{\mathcal{P}}$ *is the set of states,*
- $\mathcal{A}$ *is the finite set of actions,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *is the transition relation.*

We denote with $\mathrm{Act}(s) = \{a : \exists s'.\mathcal{R}(s, a, s')\}$ the set of actions that can be performed in state s, and with $\mathrm{Exec}(s, a) = \{s' : \mathcal{R}(s, a, s')\}$ the set of states that can be reached from s performing action $a \in \mathrm{Act}(s)$.

An example of the planning domain is defined as follows.

Example 1. *Consider the simple robot navigation domain represented in Figure 3.1. It consists of a building of 6 rooms and of a robot that can move between these rooms performing the actions described in Figure 3.1. Notice that action "goRight" performed in "Hall" moves the robot either to "Room 1" or to "Room 2" nondeterministically. Notice also that there is a door between "Room 3" and "Store" which is difficult for the robot to pass through, therefore the action "goRight" performed in "Room 3" is also nondeterministic.*

The state of the domain in Example 1 is defined by the room the robot is currently in. Therefore, the domain is defined by the set of propositions for each possible robot position {`Hall, Lab, Store, Room1, Room2, Room3`}. To remove meaningless domain states we require that only one proposition from the set is true in each domain state. Hence, a domain state is defined by the propositional formula over basic propositions.

The set of actions $\mathcal{A}$ in the domain described in Example 1 is {`goRight`, `goDown`}. Each action is characterized by the transition relation $\mathcal{R}$ that describes action preconditions (i.e. the set of domain states where given action can be executed) and action effects (i.e. the set of possible states in

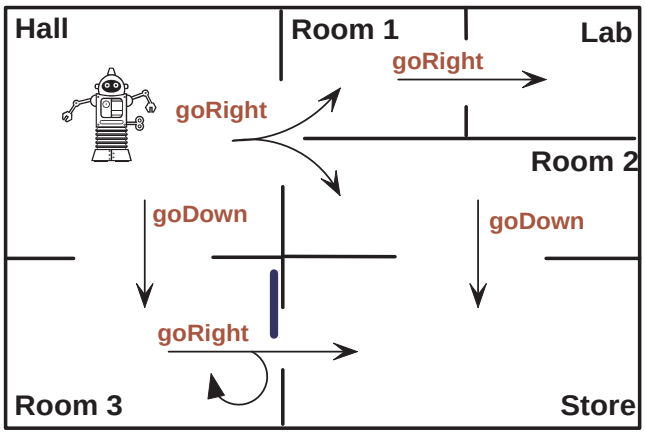


Figure 3.1: A simple domain

which the domain transits after action execution). Therefore, the transition relation is defined as follows:

$$\mathcal{R}(Hall,\ goRight,\ Room1),$$

$$\mathcal{R}(Hall,\ goRight,\ Room2),$$

$$\mathcal{R}(Room1,\ goRight,\ Lab),$$

$$\mathcal{R}(Room3,\ goRight,\ Store),$$

$$\mathcal{R}(Room3,\ goRight,\ Room3),$$

$$\mathcal{R}(Hall,\ goDown,\ Room3),$$

$$\mathcal{R}(Room2,\ goDown,\ Store).$$

Typically, a planning problem requires to find a plan for action executions which transits a domain from the initial state to the goal state. Therefore, a planning problem is defined by a planning domain $\mathcal{D}$, a set of initial states $\mathcal{I}$ and a set of goal states $\mathcal{G}$.

Definition 2 (Planning Problem). *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. A planning problem for $\mathcal{D}$ is a triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.*

Example 2. *A planing problem for the domain of Example 1 is the following:*

- $\mathcal{I}: \ room = Hall$
- $\mathcal{G}: \ room = Store$

The intuition is that the robot should move from "Hall" to "Store".

Intuitively, a solution to a planning problem is a plan which can be executed from any state in the set of initial states $\mathcal{I}$ to reach states in the set of goal states $\mathcal{G}$.

3.2 Weak, Strong and Strong Cyclic Solutions

In nondeterministic domains a plan can not be represented as a sequence of actions. The next action to execute depends on the outcome of the previous one. Therefore, we represent plans by state-action tables, or policies, which associate to each state an action that has to be performed. The plan execution is an iterative process where on each iteration we detect the current domain state and execute an action associated to this state in the state-action table. The plan execution terminates whenever the domain appears in the state which is not defined in the state-action table.

Definition 3 (Plan). *A plan π for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ is a state-action table which consists of a set of pairs $\{\langle s, a \rangle : s \in \mathcal{S}, a \in Act(s)\}$*

We denote with StatesOf(π) $= \{s : \exists a. \langle s, a \rangle \in \pi\}$ the set of states in which plan π can be executed. We describe the possible executions of a plan with an execution structure, i.e, a Kripke Structure [39].

Definition 4 (Execution Structure). *Let π be a plan for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. The execution structure induced by π from the*

set of initial states $\mathcal{I} \subseteq \mathcal{S}$ *is a tuple* $K = \langle Q, T \rangle$*, where* $Q \subseteq \mathcal{S}$ *and* $T \subseteq \mathcal{S} \times \mathcal{S}$ *are inductively defined as follows:*

1. *if* $s \in \mathcal{I}$*, then* $s \in Q$*, and*
2. *if* $s \in Q$ *and* $\exists \langle s, a \rangle \in \pi$ *and* $s' \in \mathcal{S}$ *such that* $\mathcal{R}(s, a, s')$*, then* $s' \in Q$ *and* $T(s, s')$*.*

A state $s \in Q$ *is a terminal state of* K *if there is no* $s' \in Q$ *such that T(s,s').*

Due to the nondeterminism in the domain, we need to specify the "quality" of the solution by applying additional restrictions on "how" the set of goal states should be reached. In particular we distinguish *weak*, *strong*, and *strong cyclic* solutions. A weak solution does not guarantee that the goal will be achieved, it just says that there exists at least one execution path which results in a terminal state that is a goal state. A strong solution guarantees that the goal will be achieved in a finite number of action executions in spite of nondeterminism, i.e., all execution paths of the strong solution always terminate and all terminal states are in a set of goal states. A strong cyclic solution is similar to the strong one, but it can contain loops in its execution structure and, therefore, can cause the infinite execution paths. This is a formalization of trial-and-error strategy where we assume that any loop in the plan execution structure will eventually terminate and the plan execution always reaches the goal state.

Definition 5 (Weak, Strong and Strong Cyclic Solutions).
Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ *and* $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ *be a planning domain and problem respectively. Let* π *be a plan for* $\mathcal{D}$ *and* $K = \langle Q, T \rangle$ *be the corresponding execution structure.*

1. π *is a weak solution to* P *if some of the paths in* K *terminate with states in* $\mathcal{G}$*.*

2. *π is a strong solution to P if all the paths in K are finite and their terminal states are in $\mathcal{G}$.*

3. *π is a strong cyclic solution to P if from any state in Q some terminal state is always reachable and all terminal states are in $\mathcal{G}$*

We call a state-action pair $\langle s, a\rangle \in \pi$ *strong* if all execution paths started from $\langle s, a\rangle$ terminate in the set of goal states. We call it *weak* if it is not strong, and at least one execution path started from $\langle s, a\rangle$ terminates in the set of goal states. Intuitively, a weak solution contains at least one weak state-action pair, while a strong (or strong cyclic) solution consists of strong state-action pairs only.

Example 3. *Consider the following plan π_1 for the domain of Example 1:*

state	*action*
$room = Hall$	$goRight$
$room = Room2$	$goDown$

Plan π_1 causes the robot to move from "Hall" to "Room 2" and after that move down to "Store". This is a weak solution for the planning problem from Example 2, indeed the action "goRight" performed in "Hall" can nondeterministically lead to the "Room 1" where plan execution terminates without reaching the "Store".

Consider another plan π_2 for the domain of Example 1:

state	*action*
$room = Hall$	$goDown$
$room = Room3$	$goRight$

Plan π_2 causes the robot to move from "Hall" to "Room 3" and after that move right to "Store". This is a strong cyclic solution for the planning problem from Example 2, because the action "goRight" performed in "Room

3" can nondeterministically lead back to "Room 3". Hence, the plan execution can infinitely repeat the action "goRight" in "Room 3". But if we assume that this loop eventually terminates, then π_2 guarantees that "Store" will be eventually reached.

From point of quality the strong solutions are the most preferable ones. Unfortunately, most of real-life planning problems do not have strong solutions. Ever a very simple planning domain from Example 2 has indeed no strong solutions. The weak solutions gives only a chance to reach the goal, therefore they have worst quality. Strong cyclic solutions is a compromised approach between unreal strong and definitely bad weak solutions. It is motivated by the fact that all states in the real-life planning domains are usually strongly connected, i.e. ever if undesired action outcome is happened we have a chance to transit the domain to the initial state and try again with hope that eventually we get a desired action outcome.

3.3 Algorithms for Strong and Weak Planning

Both algorithms are based on a breadth-first backward search from the goal states towards to the initial states. We first discuss the algorithm for strong planning. The intuition of the algorithm is shown in the following example.

Example 4. *Suppose we have the planning domain $\mathcal{D}$ depicted on Figure 3.2. It is identical to the one from Example 1 with only one difference: the action "goRight" in "Room 2" deterministically moves robot to "Store". Now the planning problem from Example 2 has a strong solution and we show how it can be found.*

We start from the empty plan π and iteratively add some state-action pairs to π until we get a strong solution for the planning problem. On the first step we build a set of state-action pairs $\langle s, a\rangle$ such that all possible

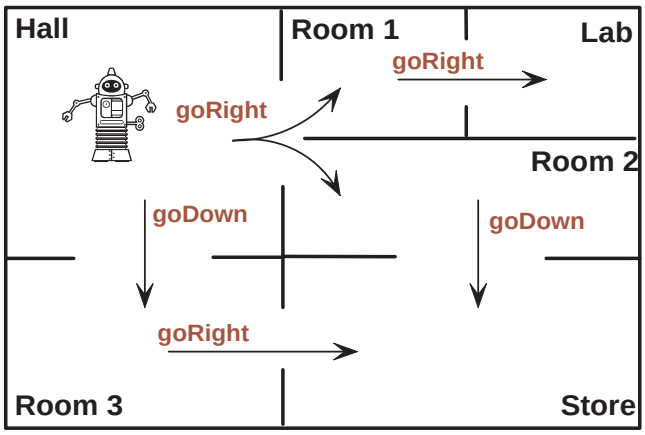


Figure 3.2: A simple domain with strong plan

outcomes of the execution of a in s lead to the goal state "room=Store". We denote this operation as a strong pre-image *computation. It is easy to see that such pre-image consist of two state-action pairs:* $\langle room = Room2, goDown\rangle$ *and* $\langle room = Room3, goRight\rangle$*. We add all state-action pairs from the pre-image to the plan* π*. On the second step we build a strong pre-image for states* $\mathcal{G} \vee StatesOf(\pi)$ *and obtain three state-action pairs:* $\langle room = Room2, goDown\rangle$*,* $\langle room = Room3, goRight\rangle$*,* $\langle room = Hall, goDown\rangle$*. The first two state-action pairs is not interested for us, because we already have a state-action pairs for states "room=Room2" and "room=Room3", but we do not have state-action pairs defined for the state "room=Hall", therefore we add* $\langle room = Hall, goDown\rangle$ *to the plan* π*. On the next step we again build a strong pre-image for states* $\mathcal{G} \vee StatesOf(\pi)$ *and obtain the same set of state-action pair as on the previous step:* $\langle room = Room2, goDown\rangle$*,* $\langle room = Room3, goRight\rangle$*, and* $\langle room = Hall, goDown\rangle$*. All of them are not interested for us, because we already have state-action pairs for states "room=Room2", "room=Room3", and "room=Hall". It means that we reached a fixed point. Now we only need to check whether the initial state "room=Hall" in* $\mathcal{G} \vee StatesOf(\pi)$*.*

It is true, therefore the generated plan π is a strong solution. The resulting plan is defined as follows:

state	*action*
$room = Hall$	$goDown$
$room = Room3$	$goRight$
$room = Room2$	$goDown$

We note that this plan has a state-action pair $\langle room = Room2, goDown\rangle$ which is never will be executed if execution starts from the initial state "room=Hall". This is because presented algorithm allows for finding a solution for all possible initial states.

The formal algorithm definition for strong planning is following:

```
01 function StrongPlanning(D,I,G);
02   π := ∅;
03  do
04   π' := π;
05   preImage := {⟨s,a⟩ : 0 ≠ Exec(s,a) ⊆ (StatesOf(π) ∨ G)};
07   prunedPreImage := {⟨s,a⟩ ∈ preImage ∧ s ∉ StatesOf(π) };
08   π := π ∪ prunedPreImage;
09  while(π != π');
10  if I ∈ (StatesOf(π) ∨ G)
11   return π;
12  else
13   return ⊥;
14  fi;
15 end;
```

Now we give a description of all steps in the algorithm:

- Line 02: We start planning with empty plan.
- Line 03-09: We iteratively add new state-action pairs to the plan until fixed point is reached.

- Line 04: We use π' to check whether the fixed point in plan building is reached.
- Line 05: We build a strong pre-image, i.e. a set of state-action pairs $\langle s, a\rangle$ such that all outcomes of the execution of a in s lead to the goal states or to the states StatesOf(π) where solution is already known.
- Line 06: We remove from the pre-image state-action pairs defined for states for which we already have a solution. This step guarantees that only a shortest solution from any state appears in the resulting plan.
- Line 10-14: We check whether the initial states are included in the StatesOf(π) $\vee$ $\mathcal{G}$. If it is true then we return the resulting plan π as a strong solution [line 11]. Otherwise, no strong solution exists [line 13].

The weak planning algorithm is identical to the strong one, except that we use a *weak pre-image* computation:

$$\text{preImage} = \{\langle s, a\rangle : \text{Exec}(s, a) \ \cap (\text{StatesOf}(\pi) \vee G) \neq 0\}$$

The weak pre-image is a set of state-action pairs that have at least one outcome leading either directly to the goal states or to the states for which a solution was already built on the previous iterations.

3.4 Algorithm for Strong Cyclic Planning

In case of strong cyclic planning we have a more complex situation in respect to strong or weak planning. If we adopt the idea of pre-image computation proposed in previous section then the main issue is to choose the right kind of pre-images. It is easy to see that the strong cyclic pre-image has to consist of a sub-set of state-action pairs from a weak pre-image. In this way, any action from strong cyclic pre-image has at least

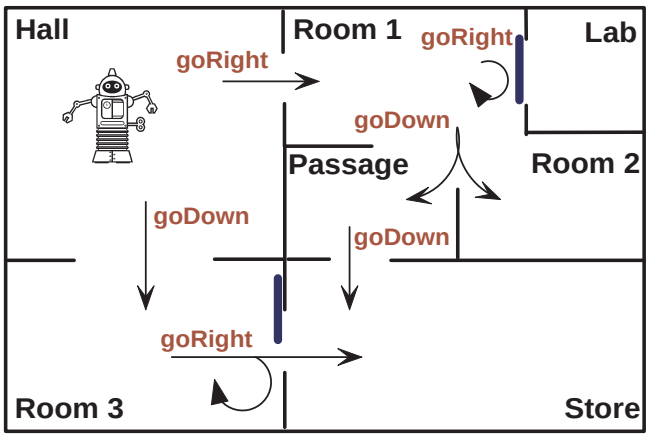


Figure 3.3: An example domain for strong cyclic planning

one outcome that lead to progress, i.e either to the goal states or to the state for which a solution was already built. But we need to add additional restriction to all others outcomes: they have to lead to the states that are non-terminal for the resulting strong cyclic solution, i.e all these states has to be added to the plan in the future.

The following example gives an intuition of the strong cyclic planning algorithm.

Example 5. *Suppose we have the robot navigation domain $\mathcal{D}$ depicted on Figure 3.3. We note that the door in between "Room 1" and "Lab" is locked, therefore the action "goRight" performed in "Room 1" does not change the robot position. We need to find a strong cyclic solution for the planning problem from Example 2.*

According to the strong cyclic plan definition, any strong cyclic plan is a weak plan, therefore the intuition of the strong cyclic algorithm is to build a state action table which describes all possible *weak plans. If the strong cyclic solution exists then it is a part of the constructed state action table which describes all possible weak plans. To do so, we use the weak planning algorithm, but do not perform pruning of the pre-image. After*

this procedure we get following plan π_1*:*

state	*action*
$room = Hall$	$goDown$
$room = Hall$	$goRight$
$room = Room3$	$goRight$
$room = Room1$	$goDown$
$room = Room1$	$goRight$
$room = Passage$	$goDown$

After that we iteratively remove from π_1 *state-action pairs which are definitely wrong, i.e. the state-action pairs which lead to the terminal state different from the goal states. In our case we have only one such state-action pair:* $\langle room = Room1, goDown \rangle$*. Hence, we get a plan* π_2*:*

state	*action*
$room = Hall$	$goDown$
$room = Hall$	$goRight$
$room = Room3$	$goRight$
$room = Room1$	$goRight$
$room = Passage$	$goDown$

Now we need to remove non-progress state-action pairs that can appear in π_2*, i.e. loops that do not give a chance to reach goal states. In our example, the infinite cycle caused by cyclical performing the action "goRigth" in "Room1". To remove non-progress states we build a resulting plan* π *using a weak planning algorithm, but* π *can contains only state-action pairs from* π_2*. After this operation we get a plan* π*:*

state	*action*
$room = Hall$	$goDown$
$room = Room3$	$goRight$
$room = Passage$	$goDown$

In fact, last two steps (i.e. elimination of wrong and non-progress state-action pairs) have to be performed cyclically until fixed point is reached. Because each time when we remove a wrong state-action pair a new non-progress one can appear and vice versa. Once we get a fixed point we only need to check whether the initial state "room=Hall" is in StatesOf(π) ∨ $\mathcal{G}$. It is true, therefore, π is a strong cyclic solution.

The formal definition of the strong cyclic planning algorithm is follows.

```
01 function StrongCyclicPlanning(D,I,G);
02   π := AllWeakPlans(D,G);
03   do
04     π' := π;
05     π := RemoveWrongStrongCyclicStateActions(π,G);
06     π := RemoveNonProgress(D,G,π);
07   while(π ≠ π')
08   if I ∈ (StatesOf(π) ∨ G)
09     return π;
10   else
11     return ⊥;
12   fi;
13 end;
```

The procedure *AllWeakPlans* builds all possible weak plans. It is identical to the weak planning algorithm, except that we do not perform pruning of the pre-image:

```
01 function AllWeakPlans(D,G);
02     π := ∅;
03   do
04     π' := π;
05     preImage := {⟨s,a⟩ : Exec(s,a) ∩ (StatesOf(π) ∨ G)≠ 0};
06     π := π ∪ preImage;
07   while(π != π');
08   return π;
09 end;
```

The procedure *RemoveWrongStrongCyclicStateActions* removes state-action pairs that have at least one outcome leading to a terminal state different from the goal state:

```
01 function RemoveWrongStrongCyclicStateActions(π,G);
02   while (∃⟨s,a⟩ ∈ π : ∅ ≠ Exec(s,a) \ (StatesOf(π) ∪ G) )
03     π = π \ ⟨s,a⟩;
04   end;
05   return π;
06 end;
```

The last procedure *RemoveNonProgress* removes non-progress state-action pairs which generate loops in the execution structure, but do not give a chance to reach goal states. This procedure is similar to the weak planning algorithm, but without image pruning and all state-action pairs of the resulting plan have to be in the plan obtained on the previous step:

```
01 function RemoveNonProgress(D,G,π_super);
02     π := ∅;
03   do
04     π' := π;
05     preImage := {⟨s,a⟩ ∈ π_super : Exec(s,a) ∩ (StatesOf(π) ∨ G)≠ 0};
06     π := π ∪ prunedPreImage;
07   while(π != π');
08   return π;
09 end;
```

Finally, [lines 08-12] we check whether the generated plan is a solution.

In this work, in particular in the part dedicated to the planning with preferences in nondeterministic domains, we will refer to some techniques that were successfully applied to strong, weak and strong cyclic planning for reachability goals.

Chapter 4

Planning with Goal Preferences

In this chapter we propose the innovative framework for *planning with goal preferences* in nondeterministic domains. It consists of two fundamental elements: a new goal language which is able to describe planning goals with respect to the user preferences, and planning algorithms which are guaranteed to find the *optimal* plan. The chapter is structured as follows. We first give an intuitive definition of main challenges and open issues of planning with preferences. Then we propose a new goal language, its motivation and syntax formalization. We analyze the problem of planning for the goals defined in the new language and propose two algorithms to find the optimal strong and strong cyclic solutions.

4.1 Motivation and Problem Definition

Widely spread nondeterminism in real-life planning domains causes the fact, that ever very simple reachability goals in most cases can not be satisfied by *strong* plans. For example, if we plan a business trip it is impossible to find a solution that guarantees the reachability of the destination on time, because any flight or train can be delayed or canceled due to weather or technical circumstances. Therefore, in most cases we forced to use *weak* plans with the hope that nondeterministic actions will outcome to desired

states. The possibility of the plan termination in unknown fail state makes the *weak* plan unattractive for the real-life applications. In this work we propose an intermediate approach that generates plans balanced between *strong* and *weak* criteria. The idea is to apply planning with preferences concepts.

The intuitive approach for planning with preferences is to define a set of alternative reachability goals and to order them according to user preferences. For example, planning a business trip we can define three goals: "get to the destination point on time", "get to the destination point with delay less than one day", "reimburse all travel expanses in case of delay more than one day". To satisfy such set of goals we need a plan that does its best to meet user preferences. It means that the plan has to guaranty termination in one of the goal state, i.e. it is a *strong* plan for total set of goal states. Moreover, the plan has to try to satisfy the most preferable goal and only if it becomes unreachable due to domain nondeterminism the plan can satisfy the less preferable goal. In general, any planning problem in nondeterministic domain can have a set of different solutions therefore we need an approach to measure all possible plans to choose the *optimal* one.

Figure 4.1 illustrates differences in planning with preferences in deterministic and nondeterministic domains. In case of deterministic domain depicted on Figure 4.1a, we can split the given problem in a set of independent planning problems without preferences, i.e. We first try to find a plan to reach the goal state g_1. It is easy to see that such plan does not exist. Then we try to find a plan to reach the goal state g_2. Such plan exists and we can stop the planning process, because there are no reasons to plan for less preferable goal g_3.

In case of nondeterministic domain depicted on Figure 4.1b the situation is much more complex. In this simple domain there are three possible plans

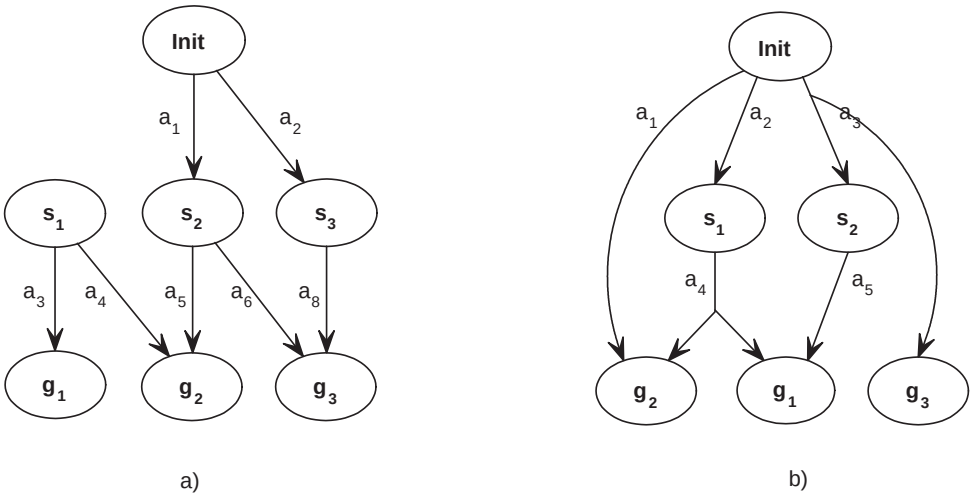


Figure 4.1: Example of deterministic (a) and nondeterministic (b) planning problem. The state "Init" is an initial state. Goal states g_1, g_2, and g_3 are ordered according to user preferences.

to reach goal states:

- The plan p_1=$\{\langle Init, a_1\rangle\}$ definitely transits the system to g_2
- The plan p_2=$\{\langle Init, a_2\rangle; \langle s_1, a_4\rangle\}$ transits the system either to g_1 or g_2.
- The plan p_3=$\{\langle Init, a_3\rangle; \langle s_2, a_5\rangle\}$ transits the system either to g_1 or g_3.

The key challenge is to choose which plan satisfies the user preferences in the best way. The intuitive approach is to compare goal sets which can be satisfied by each plan and to choose the most preferable one. For instance, p_2 is better than p_1 since it can satisfy all possible goals of p_1 and additionally it gives an opportunity to satisfy the most preferable goal g_1. It is easy to see that p_2 is also better than p_3, because the only difference in their sets of goals is that p_2 can lead to g_2 whereas p_3 can lead to the less preferable goal g_3. Therefore, p_2 is the optimal plan for the given planning

problem. But it is quite difficult to compare plans p_1 and p_3, because we need to choose whether it is better to have an unambiguous satisfiability of g_2 or it is better to try fortune in the hope of g_1, but with possibility to end up in g_3.

Conceptually, the notion of optimal plan is far from trivial. Plans in nondeterministic domains can result in several different behaviors, some of them satisfying conditions with different preferences. Planning for optimal conditional plans must therefore take into account the different behaviors, and conditionally search for the highest preference that can be achieved. Since actions are nondeterministic, the planner will know only at run-time whether a preference is met of not, and consequently a plan must interleave different plans for different levels of preferences depending on the different action outcome.

Moreover, taking into account preferences makes the already hard problem of taking into account multiple action outcomes even harder. Devising planning algorithms that can work in practice with these two dimensions of complexity in large domains and with complex preference specifications is even more an open challenge.

One of the simplest, but extremely important example of reachability goals with preferences is a combination of "main" and "recovery" goals. Defining a planning problem we need to be prepared to get a situation when during the plan execution we found out that the main goal became unreachable. Hence, we need a possibility to start a compensation plan which moves the system to some of acceptable stable states where the plan execution can terminate safely. Such set of states is called a recovery goal. Moreover, each goal can be expressed by a set of more detailed sub-goals that cover different aspects of a planning problem. For instance, ordering a business trip at a travel agency it is not enough to specify the source and destination points. To obtain a comfortable plan for the trip we need

to concrete the goal by providing additional constraints to the transport type, payment type, hotel type, and etc. Moreover, we need to express our preferences in scope of each constraint such as "traveling by plane is more preferable than traveling by train" or "payment by credit card is more preferable than payment by cash". In general, the planning problem can define arbitrary many groups of constraints that can be grouped hierarchically in a tree-like structure. To illustrate and motivate this example consider following goal:

1. Main goal

    1a. Travel from Rome to Paris.

    1b. Choose between traveling by plane, train, or bus.

    1c. Choose between payment by credit card and cash.

2. Recovery goal

    2a. All transactions have to be canceled.

On the top level we have two goals: "main" and "recovery". Hence, we need a plan which satisfies the main goal for traveling, but if it becomes unreachable the plan has to cancel all requested transactions in spite of the possible nondeterministic domain behavior. The main goal describes a goal to find out a plan to reach Paris from Rome considering the traveling transport and payment type, where only one of listed payment and transport types can be used. In this example goals are ordered according to our preferences, i.e. main goal is more preferable than recovery one, plane is more preferable than train, and etc.

This example illustrates two different grouping modes: disjunctive and conjunctive. The disjunctive grouping assumes that the planner has to satisfy at least one goal from the group. In the example above it is enough to satisfy main goal, buy a ticket only for only type of transport, and choose

only one option for the payment. The conjunctive grouping assumes that all goals from the group have to be satisfied. Therefore, to satisfy the main goal for traveling the planner has to satisfy sub-goals 1a, 1b and 1c. There is a possibility to define some hybrid grouping, such as "satisfy as many goals from the group as possible", but in this work we address only basic grouping modes.

4.2 Goal Language with Preferences

We encode the goal as a hierarchical structure where any goal can have an unbounded set of nested goals. Formally, the hierarchy is organized by the combination of two operators: **OneOf** and **All**. The operator **All** is used to conjunct subgoals when all of them have to be satisfied, i.e. it represents conjunctive grouping mode. Therefore, in the example above **All** has to be used to define the main goal as conjunction of 1a, 1b and 1c. **OneOf** operator is used for the goal composition when only one of them has to be satisfied, i.e. for disjunctive grouping mode. Therefore it can be applied to formalize goals 1b, 1c and the top level goals: main and recovery. As we mentioned in the previous section, the **OneOf** construction provides a possibility to specify a secondary (less preferable, compensation or recovery) goals to manage a situation when the focused goal became unreachable as result of the nondeterministic action outcome. To express user preferences between disjunctive goals **OneOf** assigns to each its subgoal a natural number. The lager number indicates more preferable goals. The usage of natural numbers for expression user preferences enables to set up a total ordering relation not only between goals from the same **OneOf** operator, but also between combination of goals from different hierarchy levels. For instance, in order to compare goals "travel by plan and pay by cash" and "travel by train and pay by credit card" we need a quantitative expression

of user preferences for each atomic goal. The goals such as 1a, 2a or any subgoal from 1b and 1c are basic. They describe just a set of domain states and can be represented by propositional formulas over basic propositions. We propose the formal definition of the goal language that extends classical reachability goals by adding preferences and compositional operators as follows.

Definition 6 (**Goal Language with Preferences).**
Let $\mathcal{P}$ be the set of basic propositions. The propositional formulas p and the goal with prefernece $g \in \mathcal{G}$ over $\mathcal{P}$ are defined as follows:

1. $p := \top \mid \bot \mid b \mid \neg p \mid p \wedge p \mid p \vee p \mid$
2. $g := p \mid \mathbf{OneOf}(g_1, c_1; ...; g_n, c_n) \mid \mathbf{All}(g_1, ..., g_n)$

where $c_i \in \mathbb{N}$ describes a goal preference, and $b \in \mathcal{P}$ is a basic proposition.

The goal from the travel agency example can be expressed in the new goal language in the following way:

OneOf(
  All(
    'from Rome to Paris',
    OneOf(plane,9; train,6; bus,1),
    OneOf(CC,2; cash,1)
  ),1;
  'All transactions are canceled',0
)

In order to make propositional formulas more readable we replaced them by intuitively understandable expressions. We assigned preference values to each type of transport and payment, and finally we assigned preference values to the main and recovery goals. There are seven sets of states which are satisfies the goal above:

1. 'from Rome to Paris' $\wedge$ plane $\wedge$ CC

...

6. 'from Rome to Paris' $\wedge$ bus $\wedge$ cash

7. 'All transactions are canceled'

But some of these sets of states are more preferable than others in consequence to user preferences and we need a technique to compare them based on the preferences specified inside of **OneOf** operators.

4.2.1 Goal Satisfaction

We now give semantics to the goal language defined above. This is done in two steps. First, we define when a state satisfies the goal. Second, we define an ordering relation among the states satisfying the goal. This relation defines which state satisfying the goal is "better" than another one, and hence, captures the meaning of preferences expressed in the goal.

The definition of the goal satisfiability in a domain state is very simple: the goal is evaluated on a state (or, equivalently, an assignment to basic propositions) interpreting operator **All** as a propositional $\wedge$ and operator **OneOf** as a propositional $\vee$.

We generalized the conception of the state to the conception of the *assignment* for domain propositions $\mathcal{P}$ that can describe the system state as well as a set of system states. In particular, we focused on the *goal assignments* which encode the subset of goal states. In this case, we declare that an assignment s satisfies a goal g. The goal is hierarchical, therefore the process of assignment satisfiability checking is based on recursive satisfiability checking for each subgoal. Thus, to satisfy **OneOf** goal the assignment has to satisfy at least on of its subgoals. To satisfy **All** goal the assignment has to satisfy all its subgoals. Finally, to satisfy the basic

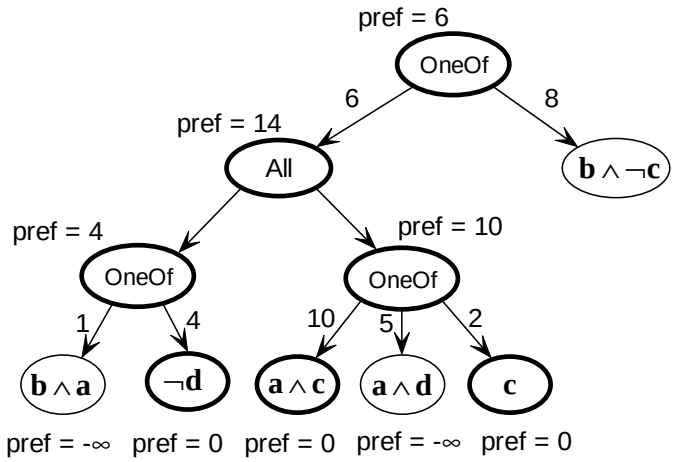


Figure 4.2: Example of the goal-tree representation. Nodes with bold border are satisfied by assignment $\{a, c\}$.

goal as a propositional formula the assignment has to make this formula true.

Definition 7 (**Goal Satisfaction**)**.**
Let $s \subseteq \mathcal{P}$ be an assignment and g be a goal on $\mathcal{P}$. Let $b \in \mathcal{P}$ be a basic proposition. We say that s satisfies g, and we write $s \models g$, according to the following rules:

1. $s \models \boldsymbol{OneOf}(g_1, c_1; ...; g_n, c_n)$ *iff* $\exists i : s \models g_i$;
2. $s \models \boldsymbol{All}(g_1, ..., g_n)$ *iff* $\forall i : s \models g_i$;
3. $s \models \bot, \top, \neg p, p \wedge p, p \vee p$ *is defined using the standard rules of logic operators;*
4. $s \models b$ *iff* $b \in s$.

The goal can be represented by a graph-tree, as shown on Figure 4.2, where leaf nodes are propositional formulas and others denote **All** and **OneOf** operators. The previous definition does not consider the goal pref-

erences. To do this we define a preference function that measures the preference of the given assignment in respect to the given goal.

Definition 8 (Preference Function).
Let $s \subseteq \mathcal{P}$ be an assignment and g be a goal on $\mathcal{P}$. The preference function $pref(s, g) : \mathcal{P} \times \mathcal{G} \to \mathbb{N} \cup \{-\infty\}$ is defined as follows:

$$pref(s,g) = \begin{cases} -\infty & \text{if } s \not\models g, \\ 0 & \text{if } s \models g \text{ and } g \text{ is a propositional formula}, \\ \max\limits_{s \models g_i} \{c_i\} & \text{if } s \models g \text{ and } g = \textbf{\textit{OneOf}}(g_1, c_1; ...; g_n, c_n), \\ \sum pref(g_i, s) & \text{if } s \models g \text{ and } g = \textbf{\textit{All}}(g_1, ..., g_n). \end{cases}$$

According to this definition, the preference function for an assignment that does not satisfy the goal is $-\infty$, while the preference function for an assignment satisfying the goal is a natural number. The goal language does not specify preferences for propositional formulas, therefore the preference function for an assignment that satisfies a propositional formula is 0. In the case of **OneOf** goals, the preference function is equal to the maximal preference function value calculated for sub-goals. Finally, in case of **All** operator we sum up all values of preference function for sub-goals. Some examples of priorities are shown on Figure 4.2.

On top of the definition of assignment preference function, we define a total ordering relation on the assignments. We use a notation $s_1 \succ_g s_2$ if an assignment s_1 is more preferable than an assignment s_2 with respect to the goal g, and notation $s_1 \simeq_g s_2$ if assignments s_1 and s_2 are equally preferable with respect to the goal g. We define that two assignments are equally preferable if we can not distinguish that one of them is more preferable than another one (i.e., $s_1 \simeq_g s_2 \Leftrightarrow (s_1 \not\succ_g s_2) \wedge (s_2 \not\succ_g s_1)$). We remark that this total ordering relation cannot be defined taking into account only the preference function. Indeed, Figure 4.3.a depicts a goal g where assignments p and q have equal values for the preference function,

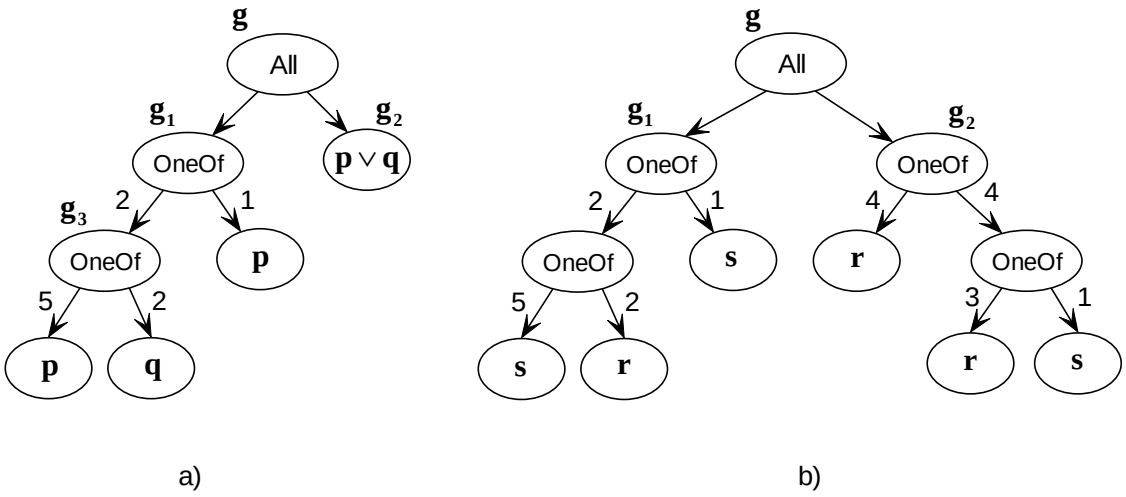


Figure 4.3: The example of **All** goal where a) *pref*(p, g) = *pref*(q, g) and $p \succ_g q$; b) *pref*(s, g) = *pref*(r, g) and $s \simeq_g r$.

i.e. *pref*(p, g) = *pref*$(q, g) = 2$. But, the goal g_3 is the most preferable one in respect to the goal g_1 and $p \succ_{g_3} q$. Therefore, it is natural to declare that $p \succ_{g_1} q$, and, hence, $p \succ_g q$.

To compare two assignments we first need to check its preference function values. If they are not equal we can definitely order these assignments, otherwise we have to compare assignments in respect to all subgoals of the top goal operator. In some cases, there is no possibility to decide which assignment is more preferable than another one. Such example is shown on Figure 4.3.b where assignments r and s are equally preferable in respect to the goal g, since *pref*(r, g) = *pref*$(s, g) = 6$ and $s \succ_{g_1} r$ and $r \succ_{g_2} s$.

Definition 9 (**Assignment Ordering Relation).**
An assignment s_1 is (strictly) preferable than an assignment s_2 with respect to a goal g, written $s_1 \succ_g s_2$, if

$$s_1 \succ_g s_2 \Leftrightarrow$$

$$\Leftrightarrow \begin{cases} pref(s_1, g) > pref(s_2, g) & , or \\ pref(s_1, g) = pref(s_2, g) & and\ g = \boldsymbol{All}(g_1, ..., g_n) \\ & and\ \forall i : \left(s_1 \not\succ_{g_i} s_2\right) \\ & and\ \exists i : \left(s_1 \succ_{g_i} s_2\right), or \\ pref(s_1, g) = pref(s_2, g) & and\ g = \boldsymbol{OneOf}(g_1, c_1; ...; g_n, c_n) \\ & and\ \forall i : (c_i = pref(s_1, g) \wedge (s_1 \models g_i)) \Rightarrow \\ & \quad \Rightarrow s_1 \not\succ_{g_i} s_2 \\ & and\ \exists i : c_i = pref(s_1, g) \wedge s_1 \models g_i \wedge s_1 \succ_{g_i} s_2 \end{cases}$$

where $s_1 \not\succ_g s_2$ *means that* $s_2 \succ_g s_1$ *is not true. Assignments* s_1 *and* s_2 *are equally preferable, written* $s_1 \simeq_g s_2$*, if* $s_1 \not\succ_g s_2$ *and* $s_2 \not\succ_g s_1$*.*

According to this definition, if two assignments s_1 and s_2 have the same preference function values with respect to the goal $g = \mathbf{All}(g_1, ..., g_n)$, then we prefer assignment s_1 only if s_1 is preferable on some of the sub-goals (condition $s_1 \succ_{g_i} s_2$) and s_2 is not preferable on any sub-goal (condition $s_1 \not\succ_{g_i} s_2$). In the case $g = \mathbf{OneOf}(g_1, c_1; ...; g_n, c_n)$ the definition is similar, i.e., we require that s_1 is sometimes strictly preferable and that s_2 is never strictly preferable, however we do this only on those sub-goals with maximum preference function value that are satisfied by assignment s_1.

Theoretically, any goal can have several representations in the proposed goal language. For example, it is obvious that goals $g_1 = \mathbf{All}(p_1)$ and $g_2 = \mathbf{All}(\mathbf{All}(p_1))$ can be considered as *equivalent* goals. It is similar to the case that any propositional formula can have several syntactical representation, i.e. $p_1 \equiv p_1 \wedge p_1$, or $p_1 \wedge (p_2 \vee p_3) \equiv (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$. The notion of equivalent goals has to capture the fact that equivalent goals express equivalent preferences. For instance, goals $g_1 = \mathbf{All}(p_1, \mathbf{OneOf}(p_2, 1; p_3, 2))$ and $g_2 = \mathbf{OneOf}(p_1 \wedge p_2, c_1; p_1 \wedge p_3, c_2)$ can be equivalent or not, depending on the values of c_1 and c_2. Therefore, we consider two goals as equivalent if and only if they describe equivalent sets of states and equivalent preferences between them.

Definition 10 (**Goal Equivalency**).
The goals $g_1, g_2 \in \mathcal{G}$ are equivalent, written $g_1 \equiv g_2$, iff they have equal set of assignments and $\forall s_1, s_2 : s_1 \succ_{g_1} s_2 \Leftrightarrow s_1 \succ_{g_2} s_2$.

According to the definition the goal equivalency is a symmetrical relation. In most cases it is enough to use asymmetrical modification of this relation called *goal implication*.

Definition 11 (**Goal Implication**).
The goal g_2 is implied by the goal g_1, written $g_1 \Rightarrow g_2$, iff both goals have equal set of assignments and $\forall s_1, s_2 : s_1 \succ_{g_1} s_2 \Rightarrow s_1 \succ_{g_2} s_2$.

The implied goals keep only strict ordering relation between assignments. The replacement of the planning goal by the implied one is a fair operation from the planning point of view. It is motivated by the fact, that if two assignments are equal in respect to the goal ($s_1 \simeq_g s_2$), then we are free to make a decision on the planner side and replace this relation by a strict one ($s_1 \succ s_2$, or $s_2 \succ s_1$). Therefore, without loosing generality we can replace the original planning goal by an implied one.

We state that the replacement of any sub-goal in the goal with preference by an equivalent sub-goal transforms this goal with preference to an equivalent one.

Lemma 1. *Let g be a **OneOf**$(g_1, c_1; \ldots; g_n, c_n)$ or **All**$(g_1, \ldots, g_n)$ goal. If we replace an arbitrary sub-goal g_j by an equivalent sub-goal g'_j, then the obtained goal g' will be an equivalent goal with respect to g.*

Proof. According to Definition 7 the set of assignments for g' is a same one as for g. Therefore we only need to prove that $\forall s_1, s_2 : s_1 \succ_g s_2 \Leftrightarrow s_1 \succ_{g'} s_2$.

First, consider the case where $g =$**OneOf**$(g_1, c_1; \ldots; g_n, c_n)$. Let s_1 and s_2 be two arbitrary assignments such as $s_1 \succ_g s_2$. According to definition of the preference function Definition 8, $pref(s_1, g) = pref(s_1, g')$ and

$pref(s_2, g) = pref(s_2, g')$. Therefore, $pref(s_1, g) > pref(s_2, g) \Rightarrow pref(s_1, g') > pref(s_2, g')$, and, hence, $pref(s_1, g) > pref(s_2, g) \Rightarrow (s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2)$. Otherwise, i.e. if $pref(s_1, g) = pref(s_2, g)$, the statement $s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2$ holds iff

$$\forall i : (c_i = pref(s_1, g) \wedge (s_1 \models g_i)) \Rightarrow s_1 \not\succ_{g_i} s_2, \text{ and}$$
$$\exists i : c_i = pref(s_1, g) \wedge s_1 \models g_i \wedge s_1 \succ_{g_i} s_2$$

But, this is true because of the fact that the replaced goal g'_j is equivalent to the g_j, i.e. $s_1 \succ_{g_j} s_2 \Leftrightarrow s_1 \succ_{g'_j} s_2$. Using this line of reasoning it is easy to show that the backward statement $s_1 \succ_{g'} s_2 \Rightarrow s_1 \succ_g s_2$ is also true.

Second, in the case where $g =$**All**$(g_1, \ldots, g_n)$ the statement of the lemma is proved analogously to the case where $g =$**OneOf**$(g_1, c_1; \ldots; g_n, c_n)$. □

Theorem 1. *Let g be an arbitrary goal with preferences. Let g_1 be an arbitrary sub-goal on arbitrary level of the hierarchical goal structure of g. If we replace g_1 by an equivalent sub-goal g'_1 then the obtained goal g' will be an equivalent to the original goal g.*

Proof. The proof is based on Lemma 1. Let g_2 be a parent goal for g_1 and after replacement of g_1 by g'_1 it is transformed to g'_2. According to Lemma 1, $g_2 \Leftrightarrow g'_2$. Repeating this rule to all parent sub-goals on the way from g_2 to the root goal g implies that $g \Leftrightarrow g'$. □

Unfortunately, the transformation technique based on the replacement of nested sub-goals can not be applied using implications of these sub-goals.

In order to standardize the process of planing with goals described in the proposed goal language we transform a goal to another goal with a special structure, optimal for planning. We call such kind of goals "flat goals". The only requirement to the goal transformation is that the resulting flat goal has to be an implication of the original goal.

4.2.2 Flat Goals

A simple goal structure simplifies the planning process. The most convenient goals from the planning point of view that still capture preferences are the so called *flat* goals: they consist of a list of propositional formulas ordered by preference.

Definition 12 (**Flat Goal**).
We call a goal $g \in \mathcal{G}$ *"flat" if it has a form* **OneOf**$(p_1, c_1; ...; p_n, c_n)$*, where* p_i *is a propositional formula.*

The main advantage of the flat goal is that all its subgoals are propositional formulas. Therefore, the planner can work with a sequence of usual reachability goals in decreasing order on preference values. In order to take the advantages of flat goals we propose the algorithm that converts any goal expressed in the proposed goal language to the flat one which is an implication of the original goal. Such goal transformation is called *flattening*.

The flattening algorithm consists of two steps. In the first step we eliminate **All** operators from the goal and obtain a goal that contains only **OneOf** operators and propositional formulas. In the second step we eliminate all **OneOf** operators except of the top one, thus obtaining the flat goal. To guarantee that resulting flat goal is an implication of the original one, we developed goal transformation rules guaranteeing that an output goal is implied by an input goal.

Definition 13 (**All Elimination**).
The ***All*** *elimination is based on the following goal transformation rules:*

- *Rule 1: Flattening of the* ***All*** *operator with nested* ***All*** *operators.*
  $\boldsymbol{All}(\boldsymbol{All}(g_1, ..., g_n), g'_1, \ldots, g'_k) \rightarrow \boldsymbol{All}(g_1, ..., g_n, g'_1, \ldots, g'_k)$

- *Rule 2: Pushing* ***All*** *operators to the leafs of the goal-tree structure.*

  All(
  $\quad$ ***OneOf***$(g_1^1, c_1^1; \ldots; g_{n_1}^1, c_{n_1}^1)$,
  $\quad \ldots,$
  $\quad$ ***OneOf***$(g_1^k, c_1^k; ...; g_{n_k}^k, c_{n_k}^k)$,
  $\quad p_1, \ldots, p_m$
) $\rightarrow$
  OneOf(
  $\quad$ ***All***$(g_1^1, g_1^2, \ldots, g_1^k, p_1 \wedge \cdots \wedge p_m)$,$c_1^1 + c_1^2 + \cdots + c_1^k$;
  $\quad \ldots$
  $\quad$ ***All***$(g_{n_1}^1, g_{n_2}^2, ..., g_{n_k}^k, p_1 \wedge \cdots \wedge p_m)$,$c_{n_1}^1 + c_{n_2}^2 + \cdots + c_{n_k}^k$
)

Notice that, in the second rule, the resulting **OneOf** goal contains $n_1 * ... * n_k$ subgoals that represent all possible combinations of subgoals of the nested **OneOf** operators in the original goal. The preference value of any of such combinations is the sum of the preference values of the subgoals. Intuitively, both rules are based on the propositional logic axioms, i.e. $((b_1 \wedge b_2) \wedge b_3) \equiv (b_1 \wedge b_2 \wedge b_3)$ and $((b_1 \vee b_2) \wedge (b_3 \vee b_4)) \equiv ((b_1 \wedge b_3) \vee (b_1 \wedge b_4) \vee (b_2 \wedge b_3) \vee (b_2 \wedge b_4))$.

A special case of the second rule is when all subgoals of the **All** operator are propositional formulas, i.e., $k = 0$. In this case, the rule can be specialized as follows:

$$\mathbf{All}(p_1, ..., p_m) = \mathbf{OneOf}(p_1 \wedge \cdots \wedge p_m, 0)$$

This rule used to remove **All** operators when they reach the leaves of the goal-tree.

The following result shows that the application of the flattening rules for **All** operator preserves the goal equivalency requirement.

Proposition 1. *Let g be an arbitrary goal and g' be a goal obtained from g applying one of the rules in Definition 13. Then $g \Leftrightarrow g'$.*

Proof. We prove by contradiction that rules in Definition 13 do not change the set of assignments. Consider an arbitrary assignment s, such that $s \models g$ and $s \not\models g'$. It means following:

- Rule 1: According to the Definition 7 $s \not\models g'$ means that $\exists g'_i : s \not\models g'_i$ or $\exists g_i : s \not\models g_i$, but it means that $s \not\models g$ (again according to the Definition 7). It is a contradiction to our assumption that $s \models g$.

- Rule 2: According to the Definition 7 $s \models g$ means that $\forall p_i : s \models p_i$ and in each nested **OneOf** goal of the input goal g there exists at least one goal g_i^j such that $s \models g_i^j$. The output goal g' contains all possible combinations of subgoals of the nested **OneOf** operators from the input goal g. Therefore, there is a combination that contains all g_i^j such that $s \models g_i^j$. Therefore $s \models g'$, and this is a contradiction to our assumption that $s \not\models g'$.

Analogously, it is easy to show that there is no assignment s, such that $s \not\models g$ and $s \models g'$.

The proof of $g \Rightarrow g'$. Let s_1, s_2 be arbitrary assignments. We prove that $\forall s_1, s_2 : s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2$.
It is obvious for Rule 1.
Consider Rule 2. We note that original goal g has a structure:

$$g = \textbf{All}(\ \textbf{OneOf}(g_1^1, c_1^1; \ldots; g_{n_1}^1, c_{n_1}^1),\ \ldots,\ \textbf{OneOf}(g_1^k, c_1^k; ...; g_{n_k}^k, c_{n_k}^k)),$$

where we denote with g_i a sub-goal of g:

$$g_i = \textbf{OneOf}(g_1^i, c_1^i; \ldots; g_{n_1}^i, c_{n_1}^i).$$

The goal obtained as a result of transformation g':

$$g' = \mathbf{OneOf}(\ \mathbf{All}(g_1^1, \ldots, g_1^k), c_1^1 + \cdots + c_1^k;\ \ldots;$$
$$\mathbf{All}(g_{n_1}^1, \ldots, g_{n_k}^k)\ , c_{n_1}^1 + \cdots + c_{n_k}^k\ ),$$

where we denote with g'_i is a sub-goal of g':

$$g'_i = \mathbf{All}(g_{j_1}^{i_1}, \ldots, g_{j_k}^{i_k}).$$

To simplify the goal structure we removed propositional formulas p_i from the transformation rule. We can do this without loosing the generality, because propositional formulas does not influence on the transformation equivalency property.

It is easy to see that $\forall s, s \models g : \mathit{pref}(s, g) = \mathit{pref}(s, g')$. Therefore, if $\mathit{pref}(s_1, g) > \mathit{pref}(s_2, g)$ then $\mathit{pref}(s_1, g') > \mathit{pref}(s_2, g')$ and so $s_1 \succ_{g'} s_2$. In case if preference functions are equal for both assignments, $s_1 \succ_g s_2$ means that there exists at least one **OneOf** sub-goal g_i such that $s_1 \succ_{g_i} s_2$, and for all others sub-goals g_j: $s_2 \not\succ_{g_j} s_1$. The output goal g' contains all possible combinations of subgoals of the nested **OneOf** operators from the input goal g. Therefore, there is a sub-goal g'_i of g' such that $s_1 \succ_{g'_i} s_2$, and for all others sub-goals g'_j: $s_2 \not\succ_{g'_j} s_1$, and so $s_1 \succ_{g'} s_2$.

The proof of $g' \Rightarrow g$. Let s_1, s_2 be arbitrary assignments. We prove that $s_1 \succ_{g'} s_2 \Rightarrow s_1 \succ_g s_2$. As in the previous case, if $\mathit{pref}(s_1, g') > \mathit{pref}(s_2, g')$ then it is obvious that $\mathit{pref}(s_1, g) > \mathit{pref}(s_2, g)$ and, hence, $s_1 \succ_g s_2$. If $\mathit{pref}(s_1, g') = \mathit{pref}(s_2, g')$ then, according to the definition of preference function, $\forall g'_i$: $s_2 \not\succ_{g'_i} s_1$, and $\exists g'_j$: $s_1 \succ_{g'_j} s_2$. Without loosing generality we can re-enumerate all goals in such way that the goal g'_1 with a structure:

$$g'_1 = \mathbf{All}(g_1^1, g_1^2, \ldots, g_1^k),$$

satisfies statement $s_1 \succ_{g'_1} s_2$. Therefore, $\mathit{pref}(s_1, g') = \mathit{pref}(s_2, g') = c_1^1 + c_1^2 + \cdots + c_1^k$, and following statements are true:

- In respect with g_1, $\nexists g_j^1$: $s_1 \models g_j^1 \wedge c_j^1 > c_1^1$. Because, otherwise, there would be a goal g'_i:

$$g'_i = \mathbf{All}(g^1_j, g^2_1, \ldots, g^k_1),$$

where $s_1 \models g'_i \wedge c^1_j + c^2_1 + \cdots + c^k_1 > c^1_1 + c^2_1 + \cdots + c^k_1 = \mathit{pref}(s_1, g')$.

- In respect with g_1, $\nexists g^1_j$: $c^1_1 = c^1_j$, and $s_1 \not\models g^1_j \wedge s_2 \models g^1_j$. Because, otherwise, there would be a goal g'_i:

$$g'_i = \mathbf{All}(g^1_j, g^2_1, \ldots, g^k_1),$$

where $s_1 \not\models g'_i \wedge s_2 \models g'_i \wedge c^1_j + c^2_1 + \cdots + c^k_1 = \mathit{pref}(s_1, g')$. This is a contradiction to our base assumption that $s_1 \succ_{g'} s_2$.

Therefore, according to the definition of the assignment ordering relation, $s_1 \succ_{g'} s_2 \Rightarrow s_2 \not\succ_{g_1} s_1$. In a similar way, we prove that $s_1 \succ_{g'} s_2 \Rightarrow \forall g_i : \; s_2 \not\succ_{g_i} s_1$. Moreover, $s_1 \succ_{g'_1} s_2$ implies that there is at least one g_i such that $s_1 \succ_{g_i} s_2$, and therefore $s_1 \succ_{g'} s_2 \Rightarrow s_1 \succ_g s_2$. □

In the flattening process, we apply the two rules of the previous definition to each **All** node of the goal, until a fixed point is reached and the resulting goal contains only **OneOf** operators and propositional formulas. According to Theorem 1, the order in which the rules are applied to the different nodes of the goal preserves the equivalency requirement for the goal transformation. In our implementation we apply them from the top of the tree to the leaves.

In the second step of our flattening algorithm we transform a goal that contains only **OneOf** operators to a flat one. Such transformation is based on the iterative application of the rule described below until a fixed point is reached. As in the previous step, the order of subgoals to which rule is applied is not important. In our implementation, we apply this rule to the top goal until it becomes the flat one.

Definition 14 (OneOf Flattening).

*The **OneOf** flattening is based on the following goal transformation rule:*

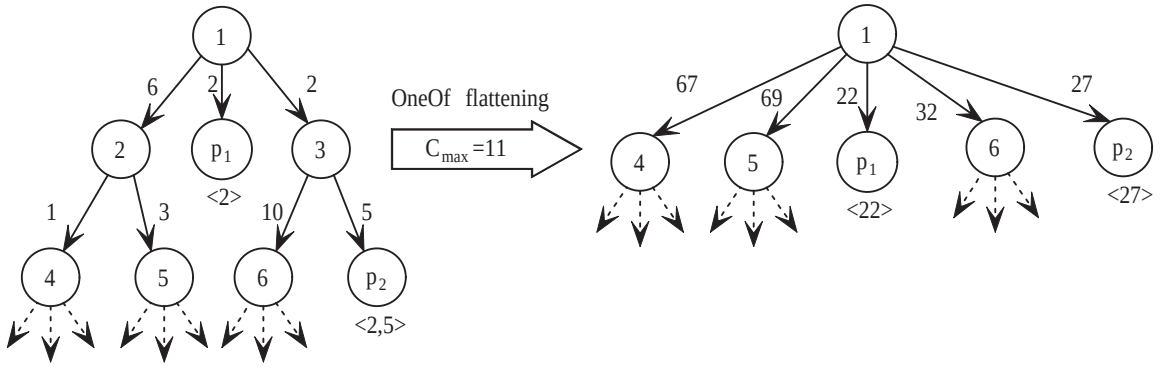


Figure 4.4: Example of usage of the **OneOf** flattening rule, where p_i are propositional formulas and **OneOf** operators are enumerated by integers.

- *Flattening of nested* ***OneOf*** *operators.*
  $\boldsymbol{OneOf}(\boldsymbol{OneOf}(g_1^1, c_1^1; \ldots; g_{n_1}^1, c_{n_1}^1), c_1; \ldots; \boldsymbol{OneOf}(g_1^k, c_1^k; ...; g_{n_k}^k, c_{n_k}^k), c_k; p_1, c'_1; \ldots; p_m, c'_m) \rightarrow$
  $\boldsymbol{OneOf}(g_1^1, \overline{c}_1^1; ...; g_{n_1}^1, \overline{c}_{n_1}^1; \ldots; g_1^k, \overline{c}_1^k; ...; g_{n_k}^k, \overline{c}_{n_k}^k;\ \ p_1, c'_1 * c_{max}; ...; p_m, c'_m * c_{max})$,
  where $\overline{c}_i^j = c_i^j + c_j * c_{max}$,
  and $c_{max} = max\{c_i^j\} + 1$

An example of usage of the **OneOf** flattening rule is shown on Figure 4.4.

Notice that, to calculate new preference values for sub-goals of **OneOf** operators we use c_{max} number, which guarantees that if **OneOf** operator j has bigger priority than operator k ($c_j > c_k$) then, after flattening, any priority $\overline{c}_i^j$ of the goal obtained from operator j is bigger than any priority $\overline{c}_l^k$ of the goal obtained from operator k. The example of the goal transformation depicted on Figure 4.4. In this example we can see that assignments $\{p_1\}$ and $\{p_2\}$ are equally preferable in respect to the source goal, but strictly comparable in respect to the goal obtained after transformation.

Proposition 2. *Let g be an arbitrary goal and g' be a goal obtained from*

g applying the rules in Definition 14. Then $g \Rightarrow g'$.

Proof. We prove by contradiction that rule in Definition 14 does not change the set of assignments. Consider an arbitrary assignment s, such that $s \models g$ and $s \not\models g'$. According to the Definition 7 $s \not\models g'$ means that s does not satisfy all sub-goals of g' (i.e $s \not\models g_i^j$ and $s \not\models p_i$ for all possible i,j). But $s \models g$ means that there exists at least one $g_{subgoal} = g_i^j$ or $g_{subgoal} = p_i$, such that $s \models g_{subgoal}$. And so, we get a contradiction. Analogously we prove that assumption about existence of s such that $s \not\models g$ and $s \models g'$ is also wrong.

In order to prove that $\forall s_1, s_2 : s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2$ we first prove that $\forall s_1, s_2 : \mathit{pref}(s_1, g) > \mathit{pref}(s_2, g) \Rightarrow \mathit{pref}(s_1, g') > \mathit{pref}(s_2, g')$. Consider two arbitrary assignments $s_1, s_2 : s_1 \succ_g s_2$. Without loosing generality we can assume that $s_1 \models g_1^1$ and $\mathit{pref}(s_1, g) = c_1$. Otherwise, we can reorder operators and goals inside of the g. Moreover, if $\exists p_i : s_1 \models p_i$ and $\mathit{pref}(s_1, g) = c_i'$ then we can replace the p_i by the equivalent goal $\mathbf{OneOf}(p_i)$ and reorder **OneOf** operators to satisfy assumption that $s_1 \models g_1^1$ and $\mathit{pref}(s_1, g) = c_1$. Analogously, we assume that $s_2 \models g_k^1$ and $\mathit{pref}(s_1, g) = c_k$. $\mathit{pref}(s_1, g) > \mathit{pref}(s_2, g)$ means that $c_1 > c_k$, therefore

$$\overline{c}_1^1 = c_1^1 + c_1 * c_{max} \geqslant$$

$$c_1^1 + (1 + c_k) * c_{max} = c_1^1 + c_{max} + c_k * c_{max} \geqslant$$

$$c_1^1 + 1 + c_k^1 + c_k * c_{max} = c_1^1 + 1 + \overline{c}_k^1 > \overline{c}_k^1$$

$\overline{c}_1^1 > \overline{c}_k^1$ implies $\mathit{pref}(s_1, g') > \mathit{pref}(s_2, g')$. And so, $s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2$ if $\mathit{pref}(s_1, g) > \mathit{pref}(s_2, g)$. Analogously, we can show that $s_1 \succ_g s_2 \Rightarrow s_1 \succ_{g'} s_2$ if $\mathit{pref}(s_1, g) = \mathit{pref}(s_2, g)$. □

To conclude the flattening procedure we prove that algorithm described above always terminates and it is correct.

Theorem 2. *The flattening algorithm terminates and preserves satisfaction and priority.*

Proof. An arbitrary goal g contains a finite number of **OneOf** and **All** operators and, hence, the depth of the goal-tree is finite. **All** rules either decrease the depth of one of the subgoal-trees (first rule) or move **All** to one level below keeping invariant the depth of the goal-tree (second rule). Therefore after finite number of **All** rules applications the resulting goal will contains only **OneOf** operators and propositional formulas.

OneOf rule decreases the depth of the goal-tree by one until the goal become flat. Hence the flattening algorithm always terminates. Since the goal contains neither **All** operators nor nested **OneOf** operators, the resulting goal is flat.

An iterative application of propositions 1 and 2 guarantees that the flattening procedure does not change the set of satisfying assignments and the preference among assignments. □

The results of this section allow us to transform any goal with preferences g into a flat goal, which is implied from g and consists of a list of pairs (states,preference value), i.e. $\mathbf{OneOf}(p_1, c_1; ...; p_n, c_n) = \langle p_1, c_1; ...; p_n, c_n \rangle$.

In the following section, we address the problem of planning for goals with preferences. In doing so, we will (safely) focus on goals of the form $\langle p_1, c_1; ...; p_n, c_n \rangle$. Moreover, we can order all goals by its preference values and consider the goal of the form $\langle p_1, ..., p_n \rangle$.

4.3 Strong Planning For Goals with Preferences

This section addresses the problem of finding solutions to the problem of planning with preference goals. We start by giving a definition of a planning problem with preferences and of plans satisfying such problem. Then

we will move to the plan optimality challenge. In the previous section we showed that any goal described in the new goal language with preferences can be replaced by the implied *flat* goal. Therefore, without loosing generality we can consider only goals that consists of a list of reachability goals ordered by preferences.

Definition 15 (**Reachability Goal with Preferences**).
A reachability goal with preferences $\mathcal{G}_{list}$ *is an ordered list* $\langle g_1, ..., g_n \rangle$*, where* $g_i \subseteq S$*. The goals in the list are ordered by preferences, where* g_1 *is the most preferable goal and* g_n *is the worst one.*

For simplicity, we consider a totally ordered list of preferences. If we obtained the reachability goal with preferences from the flat goal containing some sub-goals with equal preference values then we can set up the preference order between such sub-goals in an arbitrary way.

Definition 16 (**Planning Problem with Preferences**).
Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ *be a planning domain. A planning problem with preferences for* $\mathcal{D}$ *is a triple* $\langle \mathcal{D}, \mathcal{I}, \mathcal{G}_{list} \rangle$*, where* $\mathcal{I} \subseteq \mathcal{S}$ *and* $\mathcal{G}_{list} = \langle g_1, ..., g_n \rangle$ *is a reachability goal with preferences.*

Example 6. *Let us consider a goal with preferences for the domain of Example 1 which consists of two preference goals* $\mathcal{G}_{list} = \langle g_1, g_2 \rangle$*, where:*

- $g_1 = \{room = Store\}$
- $g_2 = \{room = Lab\}$

The intuition of this goal is that the robot has to move to "Store" or "Lab", but "Store" is a more preferable goal and the robot has to reach "Lab" only if "Store" becomes unsatisfiable. The planning problem problem with preferences is defined by the triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G}_{list} \rangle$ *.*

Plans as state-action tables are expressive enough to be solutions for planning problems for reachability goals with preferences.

Definition 17 (Solution).
A plan π is a solution for the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \{g_1, ..., g_n\} \rangle$ if it is a strong solution to the planning problem $\langle \mathcal{D}, \mathcal{I}, \bigvee_{1 \leq i \leq n} g_i \rangle$.

Such plan definition does not take into account goal preferences. All plans are equally preferable regardless from which goals from the list are satisfied. To overcome this limit, we set up an ordering relation between plans and define a notion of plan optimality.

In the definition of the ordering relation among plans, we have to take into account that, due to nondeterminism, different goal preferences can be reached by considering different executions of a plan from a given state. In our approach, we take into account the "extremal" goal preferences that are reached by executing a plan in a state, namely the goal with the best preference and that with the worst preference. It is motivated by the fact that one of the most typical cafeterias for quality measurement of the solution under uncertainty are:

- The best possible benefits that can be reached in case of favourable nondeterminism resolution.

- The worst possible losses that can be reached in case of unfavourable nondeterminism resolution.

Formally, we denote with $\mathit{pref}(\pi, s)^{best}$ the goal with best preference (i.e., of minimum index in the goal list) achievable from s:

$$\mathit{pref}(\pi, s)^{best} = \min\{i : \exists s' \subseteq g_i \text{ and } s', \text{ is a terminal state of the execution structure for } \pi \text{ that can be reached from } s\}.$$

The definition of goal $\mathit{pref}(\pi, s)^{worst}$ with worst preference reachable from s is similar. If $s \notin \text{StatesOf}(\pi)$ then $\mathit{pref}(\pi, s)^{best} = \mathit{pref}(\pi, s)^{worst} = -\infty$.

In the following definition, we compare the quality of two plans π_1 and π_2 in a specific state s of the domain. We use an *optimistic behavior assumption*, i.e., we compare the plans according to the goals of best preferences reached by the plans (i.e., $\mathit{pref}(\pi_1, s)^{best}$ and $\mathit{pref}(\pi_2, s)^{best}$). In case the maximum possible goals are equal, we apply a *pessimistic behavior assumption*, i.e., we compare the plans according to the goals of worst precedence (i.e., $\mathit{pref}(\pi_1, s)^{worst}$ and $\mathit{pref}(\pi_2, s)^{worst}$).

Definition 18 (**Plans Total Ordering Relation in a State**).
Let π_1 and π_2 be plans for a problem P. Plan π_1 is better than π_2 in state s, written $\pi_1 \prec_s \pi_2$, if:

- $\mathit{pref}(\pi_1, s)^{best} < \mathit{pref}(\pi_2, s)^{best}$, *or*
- $\mathit{pref}(\pi_1, s)^{best} = \mathit{pref}(\pi_2, s)^{best}$ *and* $\mathit{pref}(\pi_1, s)^{worst} < \mathit{pref}(\pi_2, s)^{worst}$

If $\mathit{pref}(\pi_1, s)^{best} = \mathit{pref}(\pi_2, s)^{best}$ and $\mathit{pref}(\pi_1, s)^{worst} = \mathit{pref}(\pi_2, s)^{worst}$ then π_1 and π_2 are equivalent in state s, written $\pi_1 \simeq_s \pi_2$.
We write $\pi_1 \preceq_s \pi_2$ if $\pi_1 \prec_s \pi_2$ or $\pi_1 \simeq_s \pi_2$.

We can now define relations between plans π_1 and π_2 by taking into account their behaviors in the common states $S_{common}(\pi_1, \pi_2) = \text{StatesOf}(\pi_1) \cap \text{StatesOf}(\pi_2)$.

Definition 19 (**Plans Ordering Relation**).
Let π_1 and π_2 be plans for a problem P. Plan π_1 is better than plan π_2, written $\pi_1 \prec \pi_2$, if:

- $\pi_1 \preceq_s \pi_2$ *for all states* $s \in S_{common}(\pi_1, \pi_2)$, *and*
- $\pi_1 \prec_{s'} \pi_2$ *for some state* $s' \in S_{common}(\pi_1, \pi_2)$.

If $\pi_1 \simeq_s \pi_2$ for all $s \in S_{common}(\pi_1, \pi_2)$, then the two plans are equally good, written $\pi_1 \simeq \pi_2$.

Example 7. *Let us assume that the door in the domain of Example 1 can never be "Locked" and, hence, the action "goRight" performed in "Room 3" deterministically moves the robot to "Store". Therefore the plan π_1 defined in Example 1 is a solution for the planning problem with preferences defined in Example 6. We also consider another solution π_2 that is defined as follows:*

state	*action*
$room = Hall$	$goRight$
$room = Room2$	$goDown$
$room = Room1$	$goRight$

These plans have only one common state $s = \{room = Hall\}$. We have $pref(\pi_1, s)^{best} = pref(\pi_1, s)^{worst} = 1$ because all execution paths satisfy g_1. However $pref(\pi_2, s)^{best} = 1$, but $pref(\pi_2, s)^{worst} = 2$ because there is an execution path that satisfies g_2. Therefore $\pi_1 \prec \pi_2$.

Notice that the plans ordering relation is not total: two plans are incomparable if there exist states $s_1, s_2 \in S_{common}(\pi_1, \pi_2)$ such that $\pi_1 \prec_{s_1} \pi_2$ and $\pi_2 \prec_{s_2} \pi_1$. However, in this case we can construct a plan π such that $\pi \prec \pi_1$ and $\pi \prec \pi_2$, as follows:

- if $\langle s, a\rangle \in \pi_1$ and either $s \notin \text{StatesOf}(\pi_2)$ or $\pi_1 \preceq_s \pi_2$, then $\langle s, a\rangle \in \pi$;
- if $\langle s, a\rangle \in \pi_2$ and either $s \notin \text{StatesOf}(\pi_1)$ or $\pi_2 \prec_s \pi_1$, then $\langle s, a\rangle \in \pi$.

As a consequence, there exists a plan which is better or equal to any other plan and we call such plan *optimal*.

Definition 20 (Optimal Plan)**.**
Plan π for problem P is an optimal plan if $\pi \preceq \pi'$ for any another plan π' for the same problem P.

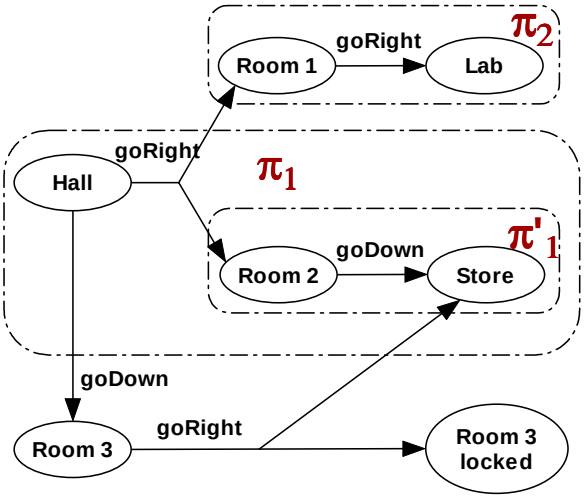


Figure 4.5: Intuition of the strong planning algorithm.

Example 8. *If the door in the domain of Example 1 can be "Locked", then plan π_2 of Example 7 is optimal for the problem defined in Example 6.*

We remark that one could adopt more complex definitions for ordering plans. For instance, one could consider not only the goals with maximum and minimum preferences reachable from a state, but also the intermediate ones. Another possibility would be to compare the quality of plans taking into account the probability to reach goals with a given preference, and/or to assign revenues to the goals in the preference list. This however requires to model probabilities in the action outcomes, while our starting point is to have a qualitative model of the domain and of the goal. With respect to these and other possible models of preferences, our approach has the advantage of being simpler, and the experiments show that it is sufficient in practice to get the expected plans.

4.4 Strong Planning Algorithm

We now describe a planning algorithm to solve a planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}_{list} \rangle$. Our approach consists of building a state-action table π_i for each goal g_i, and then to merge them in a single state-action table π. To build tables π_i, we will follow the same approach exploited in Section 3.3 for the case of strong goals without preferences, i.e., we perform a backward search for the plan that guarantees to add a state to a plan only if we are sure that a plan exists from that state. In the context of planning with goal preferences, however, performing a backward search also requires to start from the goal of lowest preferences and to incrementally consider goals of higher preference, as shown in the following example.

Example 9. *Suppose we have the domain presented in Example 1 and the planning problem with preferences of Example 6, i.e., the goal consists of two preferences* $\mathcal{G}_{list} = \langle g_1 = \{room = Hall\}, g_2 = \{room = Lab\} \rangle$.

Our requirement on the plan is that the robot should at least *reach the Lab. For this reason, we start by searching for states from which a strong solution* π_2 *for a goal* g_2 *exists. We incrementally identify states from which* g_2 *is reachable with a strong plan of increasing length. This procedure stops when all the states have been reached from which a strong plan for* g_2 *exists. An example of such a plan* π_2 *on our robot navigation domain is depicted on the top right part of Figure 4.5. In this case, the only state from which the Lab can be reached in a strong way is* $Room1$.

We then take goal g_1 *into account, and we construct a plan* π_1*, which is weak for* g_1*, but which is a strong solution for a goal* $g_1 \vee g_2$*. As in the previous step, we initially build* π_1 *as a strong solution for the goal* g_1*, using a backward-search approach. This way, we select those actions that guarantee to reach our preferred goal (see plan* π'_1 *in Figure 4.5). Once such strong plan for* g_1 *cannot be further extended, we perform a* weakening

of the plan, i.e., we try to find a weak *state-action pair for π_1 which is, at the same time, a* strong *action-pair for $\pi_1 \cup \pi_2$. Intuitively, if strong planning becomes impossible, we look for a weak extension of the π_1 which leads either to the strong part of the π_1 or to the plan for the less preferable goal π_2. In our example, we add state-action pair $\langle room = Hall, goRight \rangle$ during the* weakening *process. Notice that having already computed π_1 is a necessary condition to apply the weakening.*

After the weakening, we continue by looking again for strong extensions of plan π_1, until a new weakening is required. Strong planning and weakening are performed cyclically until a fixed point is reached. In our simple example, we reach the fixed point with a single weakening, and the final plan π_1 is shown in Figure 4.5.

In case of an arbitrary goal $\mathcal{G}_{list} = \langle g_1, ..., g_n \rangle$, the weakening process for the plan π_i consists of several iterations. We first try apply weakening using $\pi_i \cup \pi_{i+1}$, then $\pi_i \cup \pi_{i+1} \cup \pi_{i+2}$, and so on, until at least one appropriate state-action pair is found.

In the final step of our algorithm we check whether the initial states $\mathcal{I}$ are covered by plans π_1 and π_2 and, if this is the case, we extract the final plan π combining π_1 and π_2 in a suitable way.

We now describe the algorithm implementing the idea just described in Example 9. The core of the algorithm is function *BuildStrongPlan*, which is shown on Listing 4.1.

This function accepts a domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, a list of goals, and returns a list of plans, i.e. one plan for each goal. The algorithm builds a plan $pList[i]$ for each goal $gList[i]$ [lines 02-23], starting from the worst one ($i = |gList|$) to the best one ($i = 1$). For each goal $gList[i]$ we do following steps:

- We first do strong planning for goal $gList[i]$ and store the result in variable SA [line 03]. Function *StrongPlan* is defined in Section 3.3.

```
01 function BuildPlan(D, gList);
02   for (i:=|gList|; i>0; i--) do
03     SA := StrongPlan(D, gList[i]);
04     oldSA := SA;
05     wSt := StatesOf(SA) ∪ gList[i];
06     j:=i+1;
07     while (j≤|gList|)
08       wSt := wSt ∪ StatesOf(pList[j]) ∪ gList[j];
09       WeakPreImg := {⟨s,a⟩ : Exec(s,a) ∩ StatesOf(SA)≠0};
10       StrongPreImg := {⟨s,a⟩ : 0 ≠ Exec(s,a) ⊆ wSt};
11       image := StrongPreImg ∩ WeakPreImg;
12       SA := SA∪{⟨s,a⟩ ∈ image: s ∉ StatesOf(SA)};
13       if (oldSA ≠ SA) then
14         SA := SA ∪ StrongPlan(D, StatesOf(SA) ∪ gList[i]);
15         oldSA := SA;
16         wSt := StatesOf(SA) ∪ gList[i];
17         j := i+1;
18       else
19         j++;
20       fi;
21     done;
22     pList[i] := SA;
23   done;
24   return pList;
25 end;
```

Listing 4.1: BuildStrongPlan function

- When a fixed point is reached by function *StrongPlan*, a weakening step is performed. Variable $oldSA$ is initialized to SA [line 04] — this is necessary to check whether the weakening process is successful.

- In the weakening [lines 05-21], we incrementally consider goals of lower and lower preference, starting from goal with index $j = i+1$, until the weakening is successful. Along the iteration, we accumulate in variable wSt the states against which we perform the weakening. Initially, we define wSt as those states for which we have a plan for $gList[i]$, namely, the states in SA and those in $gList[i]$ [line 05]. We then incrementally add to wSt the states for $gList[j]$ [line 08].

- During the weakening process, we look for states-action pairs [line 11] which lead to SA in a weak way [line 09], and that, at the same time, lead to wSt in a strong way [line 10]. We add to SA those state-action pairs that correspond to states not already considered in SA [line 12] — as explained in Section 3.3, removing pairs already contained in SA is necessary to avoid loops in the plans.

- If we find at least one new state-action pair [lines 13-18], then we extend SA performing strong planning again [line 14], and re-start weakening, re-initializing $oldSA$, j, and wSt [line 15-17].

- If we reach a fixed point, and we cannot increment SA neither by strong planning nor by the weakening process, then we save SA as plan $pList[i]$ [line 22] and start planning for the goal $gList[i-1]$.

The top level planning function is the following:

```
function StrongPlanning(D,I,gList);
  pList := BuildStrongPlan(D,gList);
  if I ∈ ⋃_{1≤i≤|gList|} (StatesOf(pList[i])∪gList[j])
    π = extractPlan(gList,pList);
    return π;
```

```
  else
    return ⊥;
  fi;
end;
```

The function *StrongPlanning* accepts a planning domain $\mathcal{D}$, a set of domain initial states I, and a goal list $gList$, and returns a plan π, if one exists, or $\bot$. Essentially, the function *Planning* checks if plans computed by *BuildStrongPlan* for all goals in the list $gList$ are enough to cover the initial states I. If yes, then a plan π is extracted and returned. Otherwise, $\bot$ is returned. The function *extractPlan* is shown on Listing 4.2 builds a plan by merging the state-action tables in $pList$ in a suitable way. More precisely, it guarantees that a state-action pair from $pList[i]$ is added to π only if this state is not managed by a plan for a "better" goal: $\langle s, a\rangle \in \pi$ only if $\langle s, a\rangle \in pList[i]$ for some i, and $s \notin \text{StatesOf}(pList[j]) \cup gList[j]$ for all $j < i$.

```
function extractPlan(gList, pList);
  π = ∅;
  for (int i=0; i<|gList|; i++) do
    foreach (⟨s,a⟩: ⟨s,a⟩ ∈ pList[i]) do
      if s ∉ StatesOf(π) then π := π ∪ ⟨s,a⟩;
    end;
  end;
  return π;
end;
```

Listing 4.2: extractPlan function

The following theorems state the termination, soundness and completeness of the proposed algorithm.

Theorem 3 (Termination). *Function StrongPlanning*$(D, I, gList)$ *always terminates.*

Proof. There are two possible causes of non-termination in the proposed algorithm:

- "while" loop in function StrongPlan at lines 03-08. During the iteration the variable SA can not be reduced (line 07). If it is not changed at line 07 then the guardian condition of the loop ($oldSA \neq SA$) becomes true and it terminates. Therefore, the permanent increasing of SA is the only one possibility for non-termination. But this is not possible, because the set of domain states and domain actions is finite, and so, the loop will eventually terminate.

- "while" loop in function BuildStrongPlan at lines 07-21. This loop can be infinite only if condition at line 13 becomes true infinitely often. But, as in previous case, the variable SA can not be reduced (line 12) and it can not be increased infinitely. Therefore, eventually the condition at line 13 will become permanently false and the loop will terminate.

□

Theorem 4 (Completeness). *If $StrongPlanning(D, I, gList) = \perp$, then planning problem $\langle D, I, gList\rangle$ admits no strong solutions according to Definition 16.*

Proof. Let $P = \langle D, I, \langle g_1, \ldots, g_n\rangle\rangle$ be an arbitrary planning problem with preferences. We prove this theorem by contradiction.
Assume that $StrongPlanning(D, I, gList) = \perp$ and there is a plan π which is a strong solution for the goal $g' = \bigvee_{1\leq i\leq n} g_i$. It means that there is a state $s_1 \in I$ such that $s_1 \in StatesOf(\pi)$ and $s_1 \notin \bigcup_{1\leq j\leq n} (StatesOf(\pi_j) \cup g_j)$. We denote as $\langle s_1, a_1\rangle$ a state-action pair from π defined for state s_1. Since s_1 was added to no one of sub-plans π_i, there is a state s_2 such

that $s_2 \in Exec(s_1, a_1)$ and $s_2 \notin \bigcup_{1 \leq j \leq n} (StatesOf(\pi_j) \cup g_j)$. Otherwise, if $\forall s : s \in Exec(s_1, a_1) \Rightarrow \exists \pi_i : s \in StatesOf(\pi_i)$, then the state s_1 would be definitely added to some of sub-plans $\langle \pi_1, \ldots, \pi_n \rangle$ according to the definition of *StrongPlanning*. We denote as $\langle s_2, a_2 \rangle$ a state-action pair from π defined for state s_2.

We apply the same reasoning to the state s_2 and prove that there is a state s_3 such that $s_3 \in Exec(s_2, a_2)$ and $s_3 \notin \bigcup_{1 \leq j \leq n} (StatesOf(\pi_j) \cup g_j)$. Continuing this process we build an infinite execution path for the plan π where each passed state s_i holds property $s_i \notin \bigcup_{1 \leq j \leq n} (StatesOf(\pi_j) \cup g_j)$. But, this is a contradiction to the definition of the strong solution that states that any possible execution path has a bounded length. Therefore, our base assumption that there is a state $s_1 \in I$ such that $s_1 \in StatesOf(\pi)$ and $s_1 \notin \bigcup_{1 \leq j \leq n} (StatesOf(\pi_j) \cup g_j)$ is false. Hence, if there is at least one strong solution π then $\forall s : s \in I \Rightarrow s \in \bigcup_{1 \leq j \leq n} (StatesOf(\pi_j) \cup g_j)$, and therefore $StrongPlanning(D, I, gList) \neq \bot$. □

Theorem 5 (Soundness). *If* $StrongPlanning(D, I, gList) = \pi \neq \bot$, *then* π *is an optimal strong plan for the problem* $\langle D, I, gList \rangle$, *according to Definition 20.*

Proof. π is a strong plan for the goal $g_{total} = \bigvee_{1 \leq i \leq n} g_i$ according to its construction. Indeed, π does not have cycles and it always terminates in a state from g_{total}.

We prove that π is an optimal plan by contradiction. Assume that π is not an optimal plan. It means that there is another plan π' which is an optimal one and it is better than π in some state s. Let $pref(\pi', s)^{best} = i$ and $pref(\pi', s)^{worst} = j$. Thus π' is a strong plan from the state s to goal states $g' = \bigvee_{i \leq k \leq j} g_k$ and, at the same time, it is a weak plan for g_i. Building a state-action table π_i for the sub-goal g_i we perform a weakening using

strong pre-image computation for g' and merge obtained pre-image with a weak pre-image for g_i. Hence, we find *all possible* states from which there exists a strong plan for the goal g' which is, at the same time, a weak plan for g_i. Therefore, the algorithm guarantees that the state-action table for g_i is a strong solution for g' from s and a weak for g_i from s. It means that the resulting plan π is also a strong solution for the g' and weak for g_i. Hence, $\mathit{pref}(\pi, s)^{best} = i$ and $\mathit{pref}(\pi, s)^{worst} = j$. But it means that $\pi \simeq_s \pi'$. We got a contradiction with our assumption that $\pi' \prec_s \pi$. □

4.5 Strong Cyclic Planning with Preferences

In this section we extend the problem of planning with preferences to support strong cyclic plans. The main difference is that resulting plans can cause infinite execution paths, i.e. the plan execution structure can have loops. But we require that any infinite plan execution path always has a chance to terminate, i.e. we do not allow for loops that once entered have no chance to reach the goal. The definition of the planning problem with preferences is the same as it was proposed in Definition 16. The definition of the strong cyclic solution is follows.

Definition 21 (**Strong Cyclic Solution).**
A plan π is a strong cyclic solution for the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \{g_1, ..., g_n\} \rangle$ if it is a strong cyclic solution to the planning problem $\langle \mathcal{D}, \mathcal{I}, \bigvee_{1 \leq i \leq n} g_i \rangle$.

Comparing the quality of strong cyclic solutions we can eliminate from considering the infinite execution paths, and therefore we can apply the same approach to distinguish an optimal strong cyclic plan as were proposed in Definition 19 and Definition 20.

We remark that in the definition of the plans ordering we do not distinguish plans from different classes. For instance as an alternative, we

could add an additional rule that two plans π and π' with equal preference characteristics, $pref(\pi)^{best} = pref(\pi')^{best}$ and $pref(\pi)^{worst} = pref(\pi')^{worst}$, can be strongly ordered if one of them contains loops, but another one does not. But the process of planning for these plan classes is very different, and it is usually implemented as two different planning algorithms. Therefore, the planning process usually consists of two steps: (i) on the first step we build the optimal strong plan; (ii) on the second step we build the optimal strong cyclic plan and check whether the $pref()^{best}$ and $pref()^{worst}$ has been improved in respect to the strong plan. After that we can make a decision what kind of plan is more preferable according to the given problem.

4.6 Strong Cyclic Planning Algorithm

The key idea of the strong cyclic planning with preferences is similar to the strong planning technique proposed in the previous section. We again build a state-action table π_i for each goal $g_i \in \mathcal{G}_{list}$, and then merge them to the resulting state-action table π. We remark that proposed strong planning algorithm for building of π_i consists of two major steps:

- Incremental extension of the π_i by performing the strong planning search process until the fixed point is reached.
- Extension of π_i by performing weakening process.

The planner performs these two steps cyclically until the point where π_i can not be extended neither by strong planning nor by weakening.

The strong cyclic algorithm is based on the same schema, but the process of strong cyclic weakening is different. The next example gives an intuition of the strong cyclic algorithm.

Example 10. *Suppose we have the domain depicted on Figure 4.6 and the planning problem with preferences where* $\mathcal{G}_{list} = \langle g_1, g_2 \rangle$ *and* $\mathcal{I} = \{s_{init}\}$.

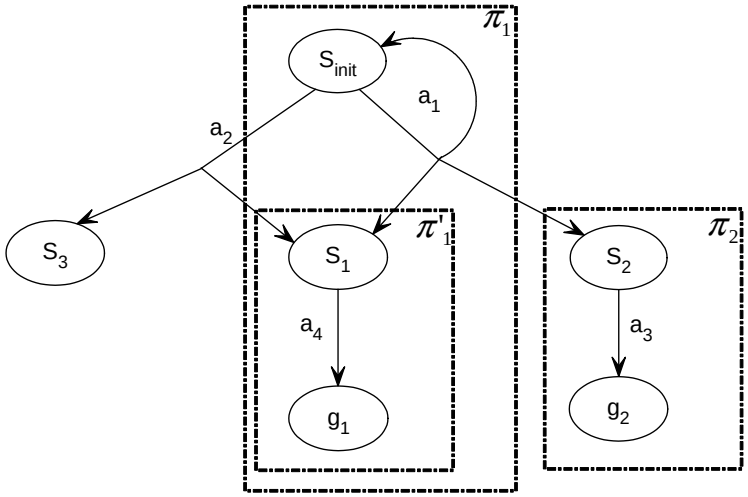


Figure 4.6: Example domain for strong cyclic planning.

As in the case of strong planning, we start from building of the plan for the less preferable goal g_2. In fact, π_2 is a strong cyclic plan for the goal g_2. This is motivated by the same reason as for strong planning, i.e. the resulting plan has to guarantee that at least one goal from the list has to be reached in spite of nondeterminism. Let's discuss in details the process of construction of π_1.

We start from assumption that $\pi_1 = \emptyset$ and cyclically perform next steps until the fixed point is reached.

1. *We compute the next extension of π_1 as a strong plan for $g_1 \vee StatesOf(\pi_1)$. The result of this step π'_1 is depicted on Figure 4.6.*

2. *We perform a strong cyclic weakening process with respect to the goal g_1. The weakening state-action pair has to satisfy following criteria:*

   - *This state-action pair has to be at least a weak one with respect to the goal $g_1 \vee StatesOf(\pi_1)$.*
   - *This state-action pair has to lead to states from which the goal g_1 can be reached strong cyclically.*

Let $\langle s, a \rangle$ be a candidate state-action pair for a weakening. To check satisfiability of the second criterion we need to check whether there is a strong cyclic plan to g_1 from each possible outcome state of the execution of $\langle s, a \rangle$. The only possibility to do this is to build in advance a state-action table that contains all possible strong cyclic plans for the goal g_1. This can be done using the strong cyclic algorithm described in Section 3.4. Concerning our example domain, there are two actions satisfying the first criterion: $\langle s_{init}, a_1 \rangle$ and $\langle s_{init}, a_2 \rangle$, but both of them do not satisfy the second criterion because there is no strong cyclic plan to g_1 from s_2 or s_3.

3. *We perform a strong cyclic weakening process with respect to the goal $g_1 \vee g_2$. The weakening state-action pair has to satisfy following criteria:*

   - *This state-action pair has to be at least a weak one with respect to the goal $g_1 \vee StatesOf(\pi_1)$.*
   - *This state-action pair has to be at least a weak one with respect to the goal $g_2 \vee StatesOf(\pi_2)$.*
   - *This state-action pair has to lead to states from which the goal $g_1 \vee g_2$ can be reached strong cyclically.*

   As on the previous step, to satisfy the third criterion we need to build in advance a state-action table that contains all possible strong cyclic plans for the goal $g_1 \vee g_2$. It is easy to see that only state-action pair $\langle s_{init}, a_1 \rangle$ satisfies all three criteria, therefore we can add it to π_1. Since, the weakening is successful we have to start the plan search process from the first step.

In this very simple example we reach the fixed point after the first execution of described above steps. The obtained plan π_1 is following:

state	*action*
s_{init}	a_1
s_1	a_4

The key element of the proposed approach is a weakening process. Whenever we build a plan π_i for the goal $g_i \in \mathcal{G}_{list} = \langle g_1, g_2, \ldots, g_n \rangle$ we first start weakening with respect to the goal g_i, if it is not successful we perform weakening with respect to the goal $g_i \vee g_{i+1}$, and etc., the last possible goal for weakening is $g_i \vee g_{i+1} \vee \cdots \vee g_n$. Moreover, the state-action pair can be used for weakening with respect to the goal $g_i \vee \cdots \vee g_j$ only if it guarantees that the goal $g_i \vee \cdots \vee g_j$ is reachable strong cyclically from all its outcomes in spite of nondeterminism. Therefore, we need to build in advance a set of temporal helper state-action tables $\pi^{all}_{i,i}, \pi^{all}_{i,i+1}, \ldots, \pi^{all}_{i,n}$, where $\pi^{all}_{i,j}$ contains all possible strong cyclic plans for the goal $g_i \vee g_{i+1} \vee \cdots \vee g_j$ and can be constructed using the algorithm described in Section 3.4. Hence, the state-action pair $\langle s, a \rangle$ can be used for weakening with respect to the goal $g_i \vee \cdots \vee g_j$ only if $\langle s, a \rangle \in \pi^{all}_{i,j}$.

The top level planning function for strong cyclic planning with preferences is shown on Listing 4.3.

The function *StrongCyclicPlanning* accepts a planning domain $\mathcal{D}$, a set of domain initial states I, and a goal list $gList$, and returns a strong cyclic plan π, if one exists, or $\perp$. This function is identical to the one for strong planning with only one difference: it uses the function *BuildStrongCyclicPlan* to compute the plan π_i for each goal from the list. This function is depicted on Listing 4.4.

The construction of π_i for each goal from the list is done in two steps:

1. We build a list of temporal state-action tables *helperList* using function HelperFullStrongCyclicPlans depicted on Listing 4.5 as follows:
   - $helperList[j] = \emptyset$, if $j < i$,

```
function StrongCyclicPlanning(D, I, gList);
  pList := BuildStrongCyclicPlan(D, gList);
  if I ∈ ⋃_{1≤i≤|gList|} (StatesOf(pList[i]) ∪ gList[j])
    π = extractPlan(gList, pList);
    return π;
  else
    return ⊥;
  fi;
end;
```

Listing 4.3: Strong cyclic planning with preferences

```
function BuildStrongCyclicPlan(D, gList);
  for (int i=0; i<|gList|; i++) do
    pList[i] = ∅;
    helperList := HelperFullStrongCyclicPlans(D, gList, pList, index);
    OptimalCtrongCyclicPlan(univSA, gList, helperList, index);
  end;
  return pList;
end;
```

Listing 4.4: The construction of the state-action table π_i for each goal g_i

- $helperList[j] = \pi^{all}_{i,j}$, if $j \geq i$.

where $\pi^{all}_{i,j}$ is a state-action table, which contains all possible strong cyclic plans for a goal $\{g_i \cup ... \cup g_n\}$. It is build using the strong cyclic algorithm described in Section 3.4.

2. Finally, we build π_i using function OptimalStrongCyclicPlan shown on Listing 4.5. In this function we cyclically perform following steps:

   - We extend the π_i using strong planning algorithm described in Section 3.3
   - Weakening using the ideas proposed in Example 10.

The following theorems state the termination, soundness and completeness of the proposed algorithm for strong cyclic planning with preferences.

Theorem 6 (**Termination).** *Function* $StrongCyclicPlanning(D, I, gList)$ *always terminates.*

Proof. The proof is identical to the proof of Theorem 3. □

Theorem 7 (**Completeness).** *If* $StrongCyclicPlanning(D, I, gList) = \perp$*, then the planning problem* $\langle D, I, gList\rangle$ *admits no strong cyclic solutions according to Definition 21.*

Proof. Let $P = \langle D, I, \langle g_1, \ldots, g_n\rangle\rangle$ be an arbitrary planning problem with preferences. We prove this theorem by contradiction.
Assume that $StrongCyclicPlanning(D, I, gList) = \perp$ and there is a plan π which is a strong cyclic solution for the goal $g' = \bigvee\limits_{1\leq i\leq n} g_i$. It means that there is a state $s_1 \in I$ such that $s_1 \in StatesOf(\pi)$ and
$s_1 \notin \bigcup\limits_{1\leq j\leq n} (StatesOf(\pi_j) \cup g_j)$. Let g_m be the most preferable goal which can be reached from s_1 according to the plan π. Let $\langle s_1, a_1\rangle$ be a state-action pair from π defined for the state s_1.

Since $s_1 \notin StatesOf(\pi_m)$, all possible outcomes of execution $\langle s_1, a_1\rangle$ are neither in π_m nor in g_m, i.e $\forall s : s \in Exec(s_1, a_1) \Rightarrow s \notin (StatesOf(\pi_m) \cup g_m)$. According to the definition of the strong cyclic plan, each execution path has a chance to terminate in a goal state, i.e. there is at least one state $s_2 : s_2 \in Exec(s_1, a_1)$ such that s_2 is different from s_1 and there is an execution path from s_2 to g_m which does not pass through s_1. Let $\langle s_2, a_2\rangle$ be a state-action pair from π defined for the state s_2.

We apply the same reasoning to the state s_2 and prove that there is a state s_3 such that $s_3 \in Exec(s_2, a_2)$, s_3 is different from s_2 and s_1, and $s_3 \notin (StatesOf(\pi_m) \cup g_m)$. Continuing this process we build an infinite set of different states such that each state s holds property:

$s \notin (\pi_m \cup g_m)$. But this is impossible because the planning domain can contain finite number of states. Therefore, s_1 is definitely in π_m. It means that if there is at least one strong cyclic solution for the goal g' then $StrongCyclicPlanning(D, I, gList) \neq \perp$. □

Theorem 8 (Soundness). *If* $StrongPlanning(D, I, gList) = \pi \neq \perp$, *then* π *is an optimal plan for problem* $\langle D, I, gList \rangle$, *according to Definition 20.*

Proof. The constructed plan π is a not a strong cyclic solution for the problem $\langle D, I, gList \rangle$ if and only if

- π has a cycle that leads to the infinite execution without possibility to reach any of goal states, or
- π can terminate in a state different from the goal states.

The first option is not possible due to the plan construction algorithm. Consider construction of arbitrary sub-plan π_i. We prove that any state-action pair from π_i always gives a chance to reach g_i. We prove this statement by induction. Let $S_1, \ldots, S_m$ be a list of weak pre-images that were built for π_i construction. Let $S'_1, \ldots, S'_m$ be a list of sets of state-action pairs that were added to π_i and $\forall j : S'_j \subseteq S_j$.

Case $j = 1$: According to the algorithm $\forall \langle s, a \rangle \in S'_1 : \langle s, a \rangle$ is a weak state-action pair for the set of states described by the goal g_i. Therefore $\langle s, a \rangle$ always gives a chance to reach the goal states.

Case $j > 0$: By the inductive hypothesis, for all $k < j$ we have $\forall \langle s, a \rangle \in S'_k : \langle s, a \rangle$ always gives a chance to terminate in goal states. But, according to the algorithm, $\forall \langle s, a \rangle \in S'_j : \langle s, a \rangle$ is a weak state-action pair for the set of states $\bigcup_{1 \leq k \leq j} StatesOf(S'_k)$. It means that $\langle s, a \rangle$ always gives a chance to reach the goal states.

Now we prove that π can not terminate in a state different from the goal states. Assume that an arbitrary sub-plan π_i contains a state-action pair $\langle s', a' \rangle$ which can transit the domain to the state $s_0 \notin \bigvee\limits_{1 \leq j \leq n} g_i$ and $\nexists a_0 \in \mathcal{A} : \langle s_0, a_0 \rangle \in \pi$. $\langle s', a' \rangle$ was added to π_i during the weakening process, because all state-action pairs added during the strong planning process transit the domain either to the goal state g_i or to $StatesOf(\pi)$. According to the algorithm, there is a strong cyclic plan containing $\langle s', a' \rangle$ for the goal $\bigvee\limits_{i \leq j \leq n} g_i$, therefore there is a strong cyclic plan from state s_0 to $\bigvee\limits_{i \leq j \leq n} g_i$. But, according to the Theorem 8, either $s_0 \in \bigvee\limits_{1 \leq j \leq n} g_i$, or $s_0 \in StatesOf(\pi)$. We have a contradiction with the assumption $s_0 \notin \bigvee\limits_{1 \leq j \leq n} g_i$ and $\nexists a_0 \in \mathcal{A} : \langle s_0, a_0 \rangle \in \pi$. $\square$

```
function HelperFullStrongCyclicPlans(D, gList, index);
  for (int i:=1; i<index; i++) do
    helperList[i] := ∅;
  done;
  G := ∅;
  for (int i:=index; i<|gList|; i++) do
    G := G ∪ gList[i];
    helperList[i] := StrongCyclicPlanning(D, I=∅, G);
  done;
  return helperList;
end;
```

```
function OptimalStrongCyclicPlan(D, gList, helperList, index);
  do
    pList[index] := StrongPlaning(pList[index], D, I=∅, gList[index]);
    plan_old := pList[index];
    states := gList[index] ∪ StatesOf(pList[index]);
    weakPreImg := {⟨s,a⟩: Exec(s,a) ∩ StatesOf(states) ≠ ∅ }
    i := index;
    while (pList[index]=plan_old && i<|gList|) do
      pList[index] := pList[index] ∪ {⟨s,a⟩ ∈weakPreImg :
            s ∉pList[index]∧⟨s,a⟩ ∈ helperList[i]};

      i++;
    done;
  while (pList[index] ≠ plan_old);
  return pList;
end;
```

Listing 4.5: The construction of temporal state-action tables *helperList* and final building of π_i

Chapter 5

Background for Planning with Extended Goals

The aim of this section is to review basic definitions and concepts of planning with extended goals in nondeterministic domains. All of them are taken, with minor modifications, from [71] and [73]. We briefly discuss problem of planning with extended goals and give a definition of the plan for such kind of goals. In this dissertation we propose a novel language for extended goals that supports constructions from EaGLe goal language [55], therefore we give a short introduction to EaGLe. Finally, we describe a promising framework for planning with extended goals based on the control automata. In this work we re-use and extend it.

5.1 Plans for Extended Goals

Intuitively, temporally extended goals are goals that express conditions on the whole execution of plans, not just on the final states. Alternatively, they can be thought as goals that dynamically changes during execution.

The plan for such kind of goals can not be represented as a state-action table according to Definition 3, because the action depends not only on the domain state, but also on a goal that is needed to be satisfied at that

moment. Traditionally, such evolving plans are modeled though different "states" of the plan executor, where each "state" corresponds to a specific "current" goal. In the following, these execution "states" are called contexts, and plans are represented by two relations, one which defined the action to be executed depending on the current domain state and execution context, and one which defined the evolution of the execution context depending on the outcome of an action.

Definition 22 (Plan). *A plan π for a planning domain $\mathcal{D}$ is a tuple $\langle \mathcal{C}, c_0, act, ctxt \rangle$ where*

- *$\mathcal{C}$ is a set of contexts,*
- *c_0 is the initial context,*
- *$act : \mathcal{S} \times \mathcal{C} \rightharpoonup \mathcal{A}$ is the action function,*
- *$ctxt : \mathcal{S} \times \mathcal{C} \times \mathcal{S} \rightharpoonup \mathcal{C}$ is the context function.*

If s is the current domain state and c is the current plan execution context, then $act(s, c)$ returns the action to be executed by the plan, while $ctxt(s, c, s')$ associates to each reached state s' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan due to domain nondeterminism.

Example 11. *Consider a simple sequential goal for the domain of Example 1 "Move the robot from Hall to Store, and after that move to Lab". The plan for such goal is following:*

act	$ctxt$
$(Hall, c_0) \rightarrow goRight$	$(Hall, c_0, Lab) \rightarrow c_0$
	$(Hall, c_0, Room1) \rightarrow c_0$
$(Lab, c_0) \rightarrow goDown$	$(Lab, c_0, Room1) \rightarrow c_0$
$(Room1, c_0) \rightarrow goDown$	$(Room1, c_0, Store) \rightarrow c_1$
$(Store, c_1) \rightarrow goUp$	$(Store, c_1, Room1) \rightarrow c_1$
$(Room1, c_1) \rightarrow goUp$	$(Room1, c_1, Lab) \rightarrow c_1$

This plan defines two contexts, i.e one context per each reachability goal. Firstly, the robot moves from Hall to Store in context c_0. When it reaches the "Store" the context is changed to c_1 and robot moves to "Lab".

We describe the possible executions of a plan with an execution structure, i.e, a Kripke Structure [39].

Definition 23 (**Execution Structure).** *Let π be a plan for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. The execution structure induced by π from the set of initial states $\mathcal{I} \subseteq \mathcal{S}$ is a tuple $\mathcal{K} = \langle Q, T \rangle$, where $Q \subseteq \mathcal{S} \times \mathcal{C}$ and $T \subseteq Q \times \mathcal{A} \times Q$ are inductively defined as follows:*

1. *if $s \in \mathcal{I}$, then $\langle s, c_0 \rangle \in Q$, and*
2. *if $\langle s, c \rangle \in Q$, and $\exists a = act(s, c) \in \pi$, and $\exists s' \in \mathcal{S}$ such that $\mathcal{R}(s, a, s')$, and $\exists c' = ctxt(s, c, s')$ then $\langle s', c' \rangle \in Q$ and $\langle s, c \rangle \xrightarrow{a} \langle s', c' \rangle \in T$.*

A state $s \in S$ is a terminal state of K in the context c if there are no $\langle s', c' \rangle \in Q$, and $a \in \mathcal{A}$, such that $\langle s, c \rangle \xrightarrow{a} \langle s', c' \rangle \in T$.

Example 12. *The Execution Structure for the plan from Example 11 is depicted on the Figure 5.1.*

In Section 2.2.2 we noted that temporal logic is a typical approach to express the extended goal. We also pointed out that temporal logic can

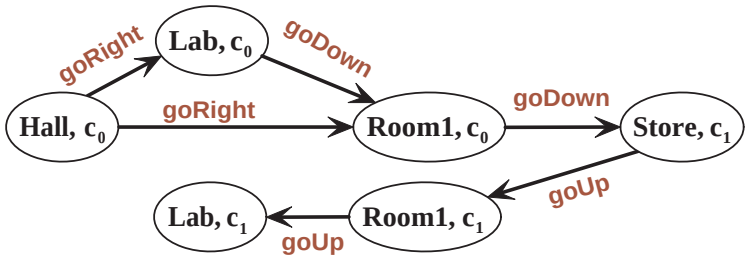


Figure 5.1: The execution structure for the plan from Example 11

not express some aspects that critically important for planning. Therefore, Extended Goal Language (EaGLe) was developed [55]. It keeps benefits of temporal logics, adapts them for the planning needs, and improves them by adding new types of extended goals.

5.2 EaGLe Goal Language

Let $\mathcal{B}$ be the set of basic propositions. The *propositional formula* p and *extended goal* g over $\mathcal{B}$ are defined as follows:

$p := \top \mid \bot \mid b \mid \neg p \mid p \wedge p \mid p \vee p$

$g := \mathbf{DoReach}\ p \mid \mathbf{TryReach}\ p \mid \mathbf{DoMaint}\ p \mid \mathbf{TryMaint}\ p \mid$
$g\ \mathbf{Then}\ g \mid g\ \mathbf{Fail}\ g \mid \mathbf{Repeat}\ g \mid g\ \mathbf{And}\ g$

The formal definition for each EaGLe construction can be found in [55]. We now give an informal description of this extended goal language. EaGLe language can express following classes of planning goals:

- *Reachability goals.* The goal "**DoReach** p" requires a plan that transits the domain to the state that holds p in spite of nondeterminism, i.e. it requires a *strong solution*. In comparison with CTL logic, **DoReach** is equivalent to **AF** operator. The operator **TryReach** requires a *weak solution* but with additional intentional aspect that the generated plan has to do "everything possible" to archive a goal. This is very different from CTL operator **EF**.

- *Maintainability goals.* The goal "**DoMaint** p" requires that all states that can be passed during the plan execution hold p. The construct **TryMaint** extends CTL operator **EG** by adding intentional aspects in the same way as **TryReach**.

- *Sequential goals.* The construct "g_1 **Then** g_2" is used to define a sequence of goals.

- *Failure goals.* The operator **Fail** is used to model recovery from failure. It is extremely important in nondeterministic domains where goal failure can be detected only at execution time. **Fail** is especially useful in combination with **TryReach** and **TryMaint**. As example, we can define a goal for a coffee machine as: (**TryReach** '*make coffee*') **Fail** (**DoReach** '*send a help request to a service center*'). This goal is interpreted as "*Do everything possible to make coffee, but if it becomes unreachable, guarantee that a help request is sent to a service center*". In this case, the goal **DoReach** '*send a help request to a service center*' is the recovery goal that will be satisfied only if the primary goal fails.

- *Cyclic goals.* The goal "**Repeat** g" specifies that the goal g has to be satisfied cyclically until it fails.

- *Parallel goals.* The construct "g_1 **And** g_2" requires a plan that is a solution for g_1 and g_2 at the same time.

Planning goals expressed in EaGLe have been successfully applied to planning problems from different application areas, such as robot navigation [55], automated Web Service composition [69] and many others. In this work we developed a new powerful language for extended goals that supports all EaGLe constructions.

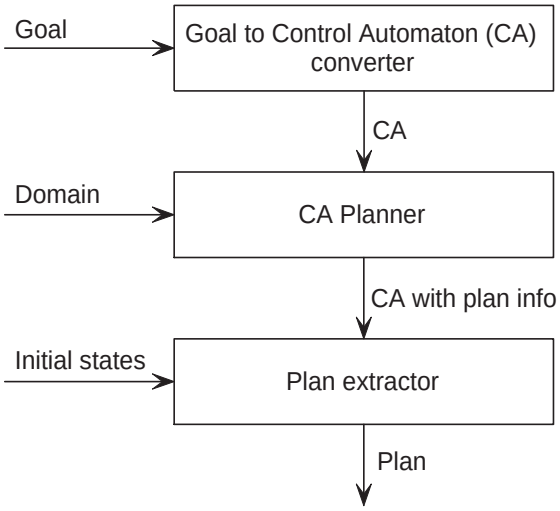


Figure 5.2: The conceptual model of the planning framework

5.3 Planning Framework Based on Control Automata

In previous section we noted that there is a set of approaches to express extended goals, i.e CTL or LTL formulas or specialized goal languages such as EaGLe. [71] proposed an idea of a framework for planning with extended goals based on the *control automata*.

The conceptual model of this planning framework is depicted on the Figure 5.2. It consists of tree main components:

- *Goal to Control Automaton Converter*. It is used to transform a goal into the control automaton that encodes goal requirements. Each control state describes some intermediate sub-goal that the plan intends to achieve. The transitions between control states defines constraints on the domain states to satisfy the *source* sub-goal and to start satisfiability of the *target* sub-goal. The resulting control automaton is delegated to the CA Planner.

- *CA Planner*. It associates to each control state a set of domain states for which a plan exists for the sub-goal encoded in this control state.

We start from assumption that all domain states are compatible with all control states and iteratively remove incomparable domain states until a fixed point is reached.

- *Plan Extractor*. It considers each control state as a plan execution context and extracts the resulting plan based on the set of domain states associated to the control states.

In fact, *CA Planner* and *Plan Extractor* are independent from the goal language. The main benefit of this approach is scalability: in order to develop a new goal language we only need to extend *Goal to Control Automaton Converter*.

We now describe each step of the planning process in details.

5.3.1 Control Automata Construction

The automaton is used to control the plan search process performed by *CA Planner*. Nodes of the control automaton represent control states of the planner that can be passed during the search. In particular, each control state corresponds to some sub-goal that has to be resolved. Transitions between states correspond to possible evolutions of the planner state in the search process, i.e. the change of the control state indicates the change of a sub-goal that has to be resolved. Moreover, each transition has a guardian condition that indicates when the planner can transit from one state to another one.

Definition 24 (Control Automaton)**.** *A control automaton is a tuple* $\langle C, c_0, T, RB \rangle$*, where*

- C *is a set of control states;*
- c_0 *is the initial control state;*

- $T(c) = \langle t_1, t_2, \ldots, t_n \rangle$ *is the ordered list of transitions from control state* c. *Each transition can be either normal, i.e.* $t_i \in \mathcal{B} \times (C \times \{\circ, \bullet\})*$, *or immediate, i.e.* $t_i \in \mathcal{B} \times (C \cup \{succ, fail\}$
- $RB = \langle rb_1, ..., rb_m \rangle$ *is the list of sets of control states marked as red blocks.*

Each control state represents a plan execution context according to the Definition 22. The *normal* transitions correspond to the action execution in the plan. Each target control state of the *normal* transition is marked either as $\circ$ or as $\bullet$. A domain state s satisfies the *normal* transition guarded by proposition formula p if it satisfies p and there is an action a from s such that (i) all possible action outcomes are compatible with some of the target control states; and (ii) some of action outcomes is compatible with each target state marked by $\bullet$. The *immediate* transitions describe internal changes in the plan execution and cause only the change of the plan execution context. The *red block* states indicates that any plan execution path should eventually leave these control states, i.e. the plan execution can not stay forever in the *red block* state.

Consider a simple sequential goal from Example 11 "Move the robot from Hall to Store, and after that move to Lab". In EaGLe it can be defined as "(**TryReach** 'the robot is in Store') **Then** (**DoReach** 'the robot is in Hall')". The initial state is "the robot is in Hall". We proposed **TryReach** operator for the first reachability goal because we showed in Example 3 that the strong version of this goal (i.e. **DoReach**) can not be resolved. We now give an intuition of the construction process of control automata based on this simple goal.

All EaGLe operators can be divided in two groups:

- *Atomic operators* do not contain another operators inside. This is the case for **DoReach**, **TryReach**, **DoMaint**, and **TryMaint**.

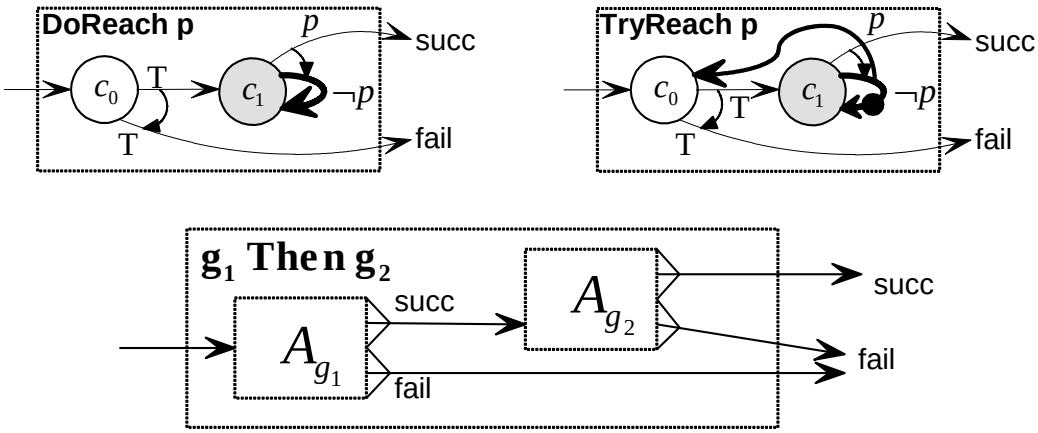


Figure 5.3: Control automata for **DoReach**, **TryReach**, and **Then** operators

- *Compositional operators* compose another operators. This is the case for **Then**, **Fail**, **Repeat**, and **And**.

Any EaGLe goal can be represented as a hierarchial tree-like structure where leaf-nodes are atomic operators and other nodes are compositional operators. To build the control automaton for the goal we first build automata for each atomic operator and then iteratively build automata for compositional operators until the automaton for the root operator is constructed.

Therefore, to build the control automaton for the example goal we need to build the automata for goals **TryReach** '*the robot is in Store*' and **DoReach** '*the robot is in Hall*', and after that compose these two automata to express **Then** operator. The graphical interpretation of automata for **DoReach**, **TryReach**, and **Then** is shown on Figure 5.3.

DoReach and **TryReach** automata consist of two states: c_0 (initial control state) and c_1 (reachability *red block* state). There are two transitions outgoing from the red block state c_1 in **DoReach** automaton. The order between this transitions is depicted by the small array from on transition to another one. The first transition is guarded by condition p. It is

a success transition that corresponds to the domain states where p holds. This transition is *immediate* because our goal task immediately becomes satisfied whenever the domain appears in the state where p holds. The second transition is guarded by condition $\neg p$. It represents the case where p does not hold in the current state, and therefore, in order to achieve goal **DoReach** p, we have to assure that the goal can be achieved in all the next states. This transition is a *normal* since it requires the execution of an action in the plan. **TryReach** differs from **DoReach** only in definition of the transition from c_1 guarded by condition $\neg p$. In this case we do not require that goal **TryReach** p holds for all next states, but only for some of them. Therefore, the transition has two possible targets, namely control states c_1 (corresponding to the next states were we expect to achieve **TryReach** p) and c_0 (for the other next states). The semantics of goal **TryReach** p requires that there should be always at least one next state that satisfies **TryReach** p; that is, target c_1 of the transition is marked by $\bullet$ in the control automaton. This non-emptiness requirement is represented in the diagram with the $\bullet$ on the arrow leading back to c_1. The preferred transition from control state c_0 is the one that leads to c_1. This ensures that the algorithm will try to achieve goal **TryReach** p whenever possible.

In case of **Then** operator, all successful transitions of the automaton for the first sub-goal g_1 leads to the initial state of the automata for the second sub-goal g_2. All successful transitions of the automaton for g_2 leads to the successful state for the top goal g_1 **Then** g_2. Finally, all failure transitions either for g_1 or for g_2 falsify the top goal g_1 **Then** g_2.

Now we are ready to build a control automaton for the example goal "(**TryReach** 'the robot is in Store') **Then** (**DoReach** 'the robot is in Hall')". It is shown on Figure 5.4.

The formal definition of the control automata construction process for all kinds of goals expressed in EaGLe can be found in [55].

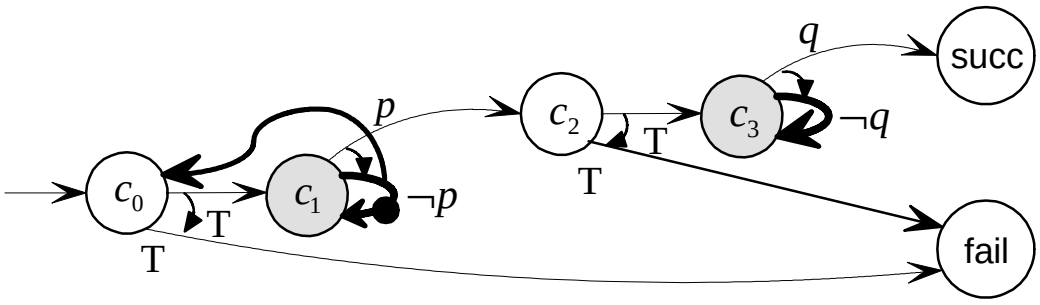


Figure 5.4: Control automaton for the example goal "(**TryReach** 'the robot is in Store') **Then** (**DoReach** 'the robot is in Hall')". The proposition formula p encodes 'the robot is in Store' and propositional formula q encodes 'the robot is in Hall'

5.3.2 Plan Search Process

Once the control automaton for a planning goal is built, *CA planner* searches for the plan by associating to each its state a set of domain states. The listing of the planning algorithm is depicted on the Figure 5.5.

- Line 02: The algorithm starts with an optimistic association that assigns all domain states to the each automaton context
- Line 03: The state "fail" represents a goal failure, therefore we assign an empty assignment to this state.
- Lines 04-05: All control states are grouped by blocks, i.e. the set of red blocks is extended by one more block called "green block" that groups non-red block states.
- Lines 06-11: The algorithm iteratively refines the associations for each block of control states until a fix-point is reached, i.e. the function *need-refinement(B)* evaluates to false for the each automaton context.
- Line 07: If a block has to be refined then all red block states from this block change its associations to the empty set. It is motivated by the

```
01  function planning(D,CA) : assoc
02    foreach c ∈ CA.C do assoc[c]:= ⊤;
03    assoc[CA.fail] := ⊥;
04    green−block := {c ∈ C : ∀B ∈ CA.RB . c ∉ B};
05    bloks := C.RB ∪ {green−block};
06    while (∃B ∈ blocks . need−refinement(B)) do
07      if B ∈ CA.RB then foreach c ∈ B do assoc[c]:= ∅;
08      while (∃c ∈ B : need−update(c)) do
09        assoc[c] := update−ctxt(CA, assoc, c);
10      end;
11    end;
12    return assoc;
13  end;
```

Figure 5.5: The planning algorithm for a goal represented by a control automaton

fact that we require that any red block context has to be eventually left.

- Lines 08-10: The algorithm iteratively refines the associations for each state from the given block until a fix-point is reached, i.e. the function *need-update(c)* evaluates to false for each context form the given block.
- Line 12: The resulting association contains all information about the plan.

The core element of the algorithm is a refinement function *update-ctxt(CA, assoc, c)*. It takes in input the automaton CA, the current association of domain states *assoc*, a context that has to be refined $c \in CA.C$, and returns the new set of domain states to be associated to c. This function is defined as follows:

$$\textit{update-ctxt(CA, assoc, c)}=$$
$$\{q : \exists t \in CA.T(c).q \in \textit{trans-assoc(t, assoc)}\}$$

It means that a new association q for the domain state c satisfies the condition of some transition t outgoing from this automaton state c. If $t = (p, c')$ is an immediate transition, then:

$$\textit{trans-assoc(t, assoc)}= \{q : q \models p \wedge q \in \textit{assoc}[c']\}$$

According to this definition, we require that a new association q satisfies property p and it is compatible with association of the target transition context c'. If the transition $t = (p, \langle (c'_1, k'_1), \ldots, (c'_n, k'_n)\rangle)$ is a normal transition, then:

$$\textit{trans-assoc(t, assoc)}=$$
$$\{q : q \models p \wedge \exists a \in \textit{Act}(q).(q, a) \in \textit{gen-pre-image}(\langle (c'_1, k'_1), \ldots, (c'_n, k'_n)\rangle)\}$$

where:

$$\textit{gen-pre-image}(\langle (c'_1, k'_1), \ldots, (c'_n, k'_n)\rangle)= \{(q, a) :$$
$$\textit{Exec}(q, a) \subseteq (\textit{assoc}[c'_1] \cup \cdots \cup \textit{assoc}[c'_n]) \wedge$$
$$\forall k'_i.k'_i = \bullet : \emptyset \neq (\textit{Exec}(q, a) \cap \textit{assoc}[c'_i]\}$$

It means that we require that normal transition also satisfy the guardian property p. Moreover, the association consists of the domain state where some action a can be executed such that all action outcomes satisfy following conditions:

- Any action outcome is compatible with an association of some of the transition target control states
- Some of action outcomes is compatible with an association of each transition target control state marked by $\bullet$

These two conditions are motivated by the following requirement: a state-action pair (q,a) appears in *gen-pre-image* only if all possible action outcomes do not break the plan execution process, i.e. the plan stays in one of

the target contexts for given transition, and for each obligatory target transition context (marked by $\bullet$) there is an action outcome that is compatible with this target transition context.

Concerning the algorithm functions *need-refinement* and *need-update*, we note that at the beginning of the algorithm all states have to be marked as "need-update". Moreover, each time when the association for some state c is changed then all states that have at least one transition leading to c have to be updated. Finally, a block of states has to be refined if it contains at least one state that has to be updated.

5.3.3 Plan Extraction

Once the plan search process is terminated by reaching a fix-point, a plan can be extracted from the domain states associated to the each state of the control automaton. We now give an intuition for the *Plan Extractor* algorithm.

To check that the plan has been found, we need to check whether all initial states of the planning problem are associated to the initial state of the control automaton. If this is true the plan exists, otherwise the plan does not exist. The information necessary to define plan functions *act* and *ctxt* are implicitly computed during the plan search process. Indeed, function *update-ctxt* determines the action $act(q, c)$ to be performed from a given state q in a given context c, and the next plan context $ctxt(q, c, q')$ for any possible next state q'. The preference order among the transitions associated to each control state is exploited in this phase, in order to guarantee that the resulting plan satisfies the plan in the best possible way. Therefore, the plan can be obtained from a given assignment by executing one more step of the refinement function and to collect these information.

Chapter 6

Fusing Procedural and Declarative Goals

In this chapter, we propose a novel language for expressing temporally extended goals for planning in nondeterministic domains. The key feature of this language is that it allows for an arbitrary combination of declarative goals expressed in temporal logic and procedural goals expressed as plan fragments. We provide a formal definition of the language and its semantics, and we propose an approach to planning with this language in nondeterministic domains.

6.1 Motivation and Problem Definition

In most planning approaches, goals and plans are different objects: goals are declarative requirements on what has to be achieved, and plans are procedural specification on how to achieve goals. This is the case of classical planning, where goals are conditions on states to be reached and plans specify sequences of actions. This is also the case of more expressive and complex settings, such as planning with temporally extended goals in nondeterministic domains, where goals are, e.g., formulas in a temporal logic, and plans are, e.g., policies or conditional and iterative combinations of

actions, see, e.g. , [49, 74, 55, 51].

However, it is often important to have the possibility to combine declarative goals and procedural plans, and this is especially useful for planning in nondeterministic domains for temporally extended goals. Indeed, in nondeterministic domains, it is often useful to specify partial plans, i.e. plans of actions to be executed only for a subset of the possible outcomes of the plan execution, while we might need to specify declarative goals to be achieved when uncovered states are reached. For instance, we can specify directly nominal plans that are interleaved with declarative conditions to be satisfied in case of failure. Vice versa, we can interleave a declarative goal specification with procedures to be executed as exception handling routines that recover from dangerous failures. In the case of temporally extended goals, parts of the goals can be better specified directly as procedures to be executed than as temporal formulas to be satisfied. For instance, a goal for a robot that has to visit periodically some locations in a building, can be easily specified as a procedural iteration interleaved with a declarative specification of the states to be reached, rather than as a temporal formula with nested maintenance and reachability conditions.

The mostly wide spread classes of temporally extended goals are *sequential* goal, where a goal is defined as a sequence of subgoals that have to be satisfied one by one, and *conditional* goals, where the next goal to satisfy is selected in respect to the guardian condition. As example, in order to deliver the box from the room A to the room B in a robot navigation domain we can specify the following goal: "Reach the room A; if there is a box then deliver it to the room B, otherwise, return to the initial position". It is a natural way to express such goals using procedural operators instead of a complex declarative formula. Moreover, there are some classes of temporally extended goals which either can not be expressed declaratively or can be expressed with a serious limitations, i.e. *recovery* goals

which define an alternative goal that need to be satisfied if the main goal becomes unreachable due to nondeterminism, or *cyclic* goals which define a goal that has to be satisfied cyclically while the guardian condition is true.

The declarative goals deal with domain states only and can not operate with domain actions. Therefore, there is no possibility to use previously generated or manually written plans in definition of the new goals. Even if the human domain experts have a clear idea about some steps that have to be in the plan, they can not express this knowledge in the goal. But it can be easily done with aid of procedural constructions typical for a plan definition. Hence, fusing declarative and procedural approach not only increases expressiveness of the goal language, but it also increases its usability and scalability. Moreover, using a procedural approach in the goal definition gives a possibility to help the planner in reasoning by dividing a complex goal into smaller ones and resolving them separately. In many cases a planning problem have more than one solution, and there is no universal utility function to estimate plans quality to choose the best one. The last decision is made by the human domain expert. Therefore, the possibility to write manually some critical parts of the plan and use them in the goal definition helps to get a more preferable plan.

To solve all challenges mentioned above we need to develop a novel goal language for expressing temporally extended goals for planning in nondeterministic domains. This language has to allow for an arbitrary combination of declarative goals expressed in temporal logic and procedural goals expressed as plan fragments. We also need to develop a planning framework which generates plans for the goals defined in the new goal language.

6.2 Extended Goal Language

The basic assumption for the new goal language is that it has to support a declarative definition of the temporally extended goals. In this work, we support CTL goals proposed in [74] and EaGLe goals as defined in [55]. To realize the possibility to embed the fragments of the plans in the goal definition we include in the new language constructions typical for the plan definition, i.e. primitive action call, sequence, conditional, cyclic, and check-point operators. Other goal constructs cover such important aspects of the temporally extended goals as failure recovery and search control policies. We define the new language as follows.

Definition 25 (**Extended Goal Language**). *Let $\mathcal{B}$ be the set of* basic propositions *and $(p \in)\mathcal{P}$ be a* propositional formula *over $\mathcal{B}$. Let $\mathcal{A}$ be the set of actions, and $\mathcal{G}$ be the set of temporally extended goals. The* extended goal tasks *$(t \in)\mathcal{T}$ over $\mathcal{A}$ and $\mathcal{G}$ are defined as follows:*

goal g	*temporally extended goal*
doAction a	*primitive action call*
$t_1; t_2$	*sequences*
if p ***do*** t_1 ***else*** t_2 ***end***	*conditionals*
while p ***do*** t_1 ***end***	*cycles*
check p	*check-point*
try t_0 ***catch*** p_1 ***do*** t_1 ***end***	*failure recovery*
policy f ***do*** t_1 ***end***	*search control policy*

We now provide intuition and detailed description for each construction of the language.

6.2.1 Temporally extended goal

The language includes temporally extended CTL goals proposed in [74] and EaGLe goals described in Section 5.2. To define such kind of goals we provide operator **goal** g, where g is a temporally extended goal.

6.2.2 Primitive action call

The primitive action call operator is a basic operator in the plan definition, therefore it is critical to support such construction in the goal language. The wise usage of primitive actions can eliminate critical branching points in the plan search and dramatically improve the planning time. The key idea for the goal optimization is to eliminate planning if it is not needed. If we require or know in advance that goal *"Put down the box"* has to be satisfied by only one primitive action then we can put it in the goal definition: **doAction** *put_down_box*.

6.2.3 Goal sequence

The sequence of goals is used to support plan fragments in the goal. The simplest example is a sequence of primitive action calls. But the sequence operator is a powerful pattern to manage the planning process and to increase the resulting plan quality. One of the most intuitive and effective idea for plan search improvements is to split a complex goal into a sequence of more simple goals. For example, a goal *"Deliver box from room A to room B"* can be redefined as sequence *"Reach the room A; Find the box; Take the box; Reach the room B; Put down the box"*. Goals *"Reach the room A"* and *"Find the box"* can be resolved independently, as they require different search strategies (pathfinding and scanning), therefore the right goal split in a sequences can significantly reduce planning search space.

6.2.4 Conditional operator

The conditional goal **if-else** provides the possibility to switch between different goals based on the reached domain state.

6.2.5 Cyclic operator

We often need to define a cyclic goal which has to be performed until some condition is true. This is the case for instance for the goal *"While there exists a box in the room A pick up any box and deliver it to the room B"*. In such case we can add some search knowledge information about the fact that boxes have to be delivered one by one. Hence, the goal can be defined as *"**while** ('the room A is not empty') **do** **DoReach** 'the robot loaded a box in A'; **DoReach** 'the robot is in B'; **doAction** 'unload the robot' **end** "*.

6.2.6 Check-point

The goal **check** checks whether the current domain state is allowed according the the check-point condition. For example, if we have a goal "t_1; **check** (p)" it means that we require a plan which satisfies t_1 in such way that guaranties that finally the domain has to be in the state p. The most intuitive reason to use check-point is to check the correctness of the plan fragment in the goal definition.

6.2.7 Failure recovery

Construction **try-catch** is used to model recovery from failure due to domain nondeterminism. This operator is very different from classical disjunction "satisfy goal t_1 or t_2". It captures intentional aspects of the goal satisfiability, i.e. does everything that is possible to achieve goal task t_1 and only if it becomes truly unreachable then satisfies t_2. Moreover, for

one **try** we can define many **catch** blocks to specify different recovery goal tasks for different failure conditions.

6.2.8 Search control policy

The last construction of proposed language is **policy** which can be defined for any goal task t. The policy is a formula on current states, actions, and next states (operator **next**). Intuitively, policy is an additional restriction to the domain transition function, which defines relation between the current and next domain states. Therefore we allow for usage of actions in the policy definitions to make them simpler and readable. For example, for a goal task "Reach room A" we can define a policy "(**next** (position) != position) $\vee$ **action**=*look-around*". It means that we restrict planner to consider only actions which change robot position or particular action named *look-around*, and therefore reduce the search space. Effectiveness of the policies increases if we associate them to simple sub-goals of a complex goals. The reason is that simpler goals can be restricted by stronger policies.

6.3 Formal Model

We now provide a formal semantics for the proposed language. In order to formalize whether the plan π satisfy the goal task t we assign to each domain state s of the plan execution structure two sets: $\mathcal{S}_t^\pi(s)$ and $\mathcal{F}_t^\pi(s)$. They represent finite plan execution paths which satisfy or falsify the goal task t from the state s following the plan π. Hence, the plan satisfies the goal task t if there is no execution path which leads the domain to the failure state.

Definition 26. *A plan π satisfies a goal task t in a state s if and only if $\mathcal{F}_t^\pi(s) = \emptyset$.*

We now consider all kinds of goal tasks and define $\mathcal{S}_t^\pi(s)$ and $\mathcal{F}_t^\pi(s)$ by induction on the structure of the goal task.

$t = \textbf{goal}\ g$ is satisfied by π if π satisfies the goal g written either in CTL or EaGLe goal languages.

$t = \textbf{doAction}\ a$ is satisfied by π in state s and execution context c if there exists a transition from the state $\langle s, c\rangle$ guarded by action a and there are no transitions guarded by others actions, otherwise it fails. Formally, $\mathcal{S}_t^\pi(s) = \{\sigma : \exists\langle s', c'\rangle.\sigma = \langle s, c\rangle \xrightarrow{a} \langle s', c'\rangle\}$. $\mathcal{F}_t^\pi(s) = \{\langle s, c\rangle\}$ if $\langle s, c\rangle$ is a terminal state of the execution structure, otherwise $\mathcal{F}_t^\pi(s) = \{\sigma : \exists b, \langle s', c'\rangle.\sigma = \langle s, c\rangle \xrightarrow{b} \langle s', c'\rangle\}$.

$t = t_1; t_2$ requires to satisfy first sub-goal task t_1 and, once t_1 succeeds, to satisfy next sub-goal task t_2. Formally, $\mathcal{S}_t^\pi(s) = \{\sigma = \sigma_1; \sigma_2 : \sigma_1 \in \mathcal{S}_{t_1}^\pi(s) \wedge \sigma_2 \in \mathcal{S}_{t_2}^\pi(last(\sigma_1))\}$. $\mathcal{F}_t^\pi(s) = \{\sigma_1 : \sigma_1 \in \mathcal{F}_{t_1}^\pi(s)\} \cup \{\sigma = \sigma_1; \sigma_2 : \sigma_1 \in \mathcal{S}_{t_1}^\pi(s) \wedge \sigma_2 \in \mathcal{F}_{t_2}^\pi(last(\sigma_1))\}$

$t = \textbf{if}\ p\ \textbf{do}\ t_1\ \textbf{else}\ t_2\ \textbf{end}$ requires to satisfy the sub-goal task t_1 if formula p holds in the current state, or the alternative sub-goal task t_2 if p does not hold in the current state. Formally, if $s \models p$ then $\mathcal{S}_t^\pi(s) = \mathcal{S}_{t_1}^\pi(s)$ and $\mathcal{F}_t^\pi(s) = \mathcal{F}_{t_1}^\pi(s)$. Otherwise, $\mathcal{S}_t^\pi(s) = \mathcal{S}_{t_2}^\pi(s)$ and $\mathcal{F}_t^\pi(s) = \mathcal{F}_{t_2}^\pi(s)$.

$t = \textbf{while}\ p\ \textbf{do}\ t_1\ \textbf{end}$ requires cyclic satisfiability of the sub-goal task t_1 while condition p holds. Moreover, it requires that the loop has to be finite. Formally, if $s \not\models p$ then $\mathcal{S}_t^\pi(s) = \{\langle s, c\rangle\}$ and $\mathcal{F}_t^\pi(s) = \emptyset$. Otherwise, $\mathcal{S}_t^\pi(s) = \{\sigma : last(\sigma) \not\models p \wedge \exists\sigma_1; ...; \sigma_n = \sigma.\forall i = 1..n : first(\sigma_i) \models p \wedge \sigma_i \in \mathcal{S}_{t_1}^\pi(first(\sigma_i))\}$ and $\mathcal{F}_t^\pi(s) = \{\sigma : \exists\sigma_1; ...; \sigma_n = \sigma.\forall i = 1..n-1 : first(\sigma_i) \models p \wedge \sigma_i \in \mathcal{S}_{t_1}^\pi(first(\sigma_i)) \wedge \sigma_n \in \mathcal{F}_{t_1}^\pi(last(\sigma_{n-1}))\}$.

$t = \textbf{check}\ p$ requires that formula p holds in the current state of the plan execution $\langle s, c\rangle$. Formally, $\mathcal{S}_t^\pi(s) = \{\sigma : \sigma = \langle s, c\rangle \wedge s \models p\}$. $\mathcal{F}_t^\pi(s) = \{\sigma : \sigma = \langle s, c\rangle \wedge s \not\models p\}$.

$t = \textbf{try}\ t_0\ \textbf{catch}\ p_1\ \textbf{do}\ t_1\ ...\ \textbf{catch}\ p_n\ \textbf{do}\ t_n\ \textbf{end}$ requires to satisfy the "main" sub-goal task t_0, but in case if t_0 fails, it requires to satisfy one of

the "recovery" sub-goal tasks $t_1, ..., t_n$ according to the domain state obtained after t_0 fail. Formally,
$\mathcal{S}_t^\pi(s) = \{\sigma : \sigma \in \mathcal{S}_{t_0}^\pi(s)\} \cup \{\sigma = \sigma_0; \sigma' : \sigma_0 \in \mathcal{F}_{t_0}^\pi(s) \wedge \exists i. last(\sigma_0) \models p_i \wedge \sigma' \in \mathcal{S}_{t_i}^\pi(last(\sigma_0))\}$ and
$\mathcal{F}_t^\pi(s) = \{\sigma : \sigma \in \mathcal{F}_{t_0}^\pi(s) \wedge \forall i = 1..n : last(\sigma) \not\models p_i\} \cup \{\sigma = \sigma_0; \sigma' : \sigma_0 \in \mathcal{F}_{t_0}^\pi(s) \wedge \exists i. last(\sigma_0) \models p_i \wedge \sigma' \in \mathcal{F}_{t_i}^\pi(last(\sigma_0))\}$.
$t =$ **policy** f **do** t_1 **end** requires to satisfy sub-goal task t_1 following the transition rule f, that does not change a set of failure execution paths, but narrows the set of successful execution paths in order to reduce the search space. Formally, $\mathcal{S}_t^\pi(s) = \{\sigma : \sigma \in \mathcal{S}_{t_1}^\pi(s) \wedge \forall s_i, a, s_{i+1}. f \mid_{s_i,a,s_{i+1}} = \top\}$ and $\mathcal{F}_t^\pi(s) = \mathcal{F}_{t_1}^\pi(s)$, where s_i, s_{i+1} are connected by a states from the execution path σ.

6.4 Planning Framework

In this work we re-used the planning approach that was proposed in Section 5.3. We built our framework on top of a formal model, where a planning goal is represented by a control automaton. This formalism was developed with aim to uniform the planning goal structure and to separate the plan search techniques from the concrete planning goal definition. In this formalism the planning framework is modeled as a 3-layer structure, where the first layer represents a planning goal defined in some goal language, the second layer represents a control automaton built according to the given planning goal, and the third layer represents a plan search algorithm that is guided by the control automaton. Since *CA Planner* and *Plan Extractor* components from the planning framework depicted on the Figure 5.2 are independent from the goal language, we only need to define the control automaton construction process for proposed goal language and re-use algorithms for CA Planner and Plan Extractor proposed

in Section 5.3. The main benefit of this approach is scalability: in order to add new features to the goal language we only need to improve Control Automaton Converter.

6.4.1 Control Automaton Construction

We model a *control automaton* in terms of *control states* which characterize atomic sub-goals, *transition function* which define possible evolutions between control states, and guardian *propositional* and *policy formulas* which control a planning search process.

Definition 27 (**Control Automaton with Policies**). *Let F be a policy formula defined over domain states $\mathcal{S}$, next states $\boldsymbol{next}(\mathcal{S})$, and domain actions $\mathcal{A}$. A control automaton is a tuple $\langle C, c_0, T, RB\rangle$, where*

- *C is a set of control states;*
- *c_0 is the initial control state;*
- *$T(c) = \langle t_1, t_2, \ldots, t_n\rangle$ is the ordered list of transitions from control state c. Each transition can be either normal, i.e. $t_i \in \mathcal{B} \times F \times (C \times \{\circ, \bullet\})$, or immediate, i.e. $t_i \in \mathcal{B} \times (C \cup \{succ, fail\}$*
- *$RB = \langle rb_1, ..., rb_m\rangle$ is the list of sets of control states marked as red blocks.*

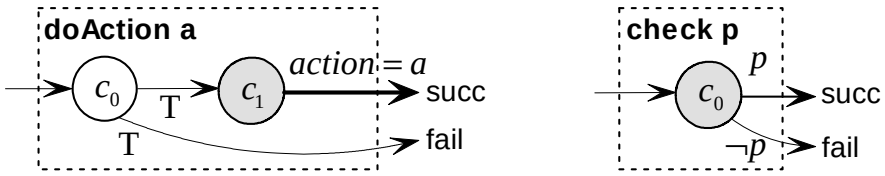
In comparison with Definition 24 the normal transition can be guarded by a policy formula, which restricts the set of actions that can be performed according to this transition.

Each control state represents a plan execution context according to the Definition 22. The *normal* transitions correspond to the action execution in the plan. Domain state s satisfies the *normal* transition guarded by proposition formula p if it satisfies p and there is an action a from s such

that all possible action outcomes are compatible with some of the target control states. Moreover, the *normal* transition is additionally guarded by the policy formula, which restricts the set of actions that can be performed according to this transition. The *immediate* transitions describe the internal changes in the plan execution and cause only the change of the plan execution context, therefore they can not be guarded by policy formulas. The *red block* states indicates that any plan execution path should eventually leave these control states, i.e. the plan execution can not stay forever in the *red block* state.

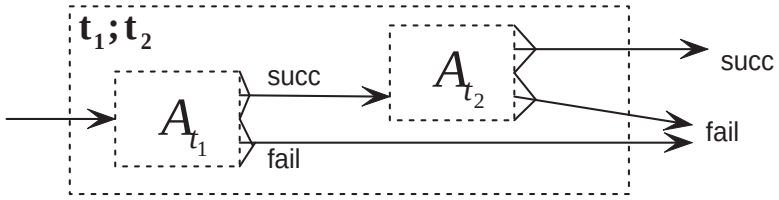
In the following, we use the graphical notations of [55] for control automata definitions. We start from the **goal** operator. In fact, it represent the declarative goal expressed in the EaGLe goal language. The process of automata construction for the EaGLe goal is described in Section 5.3.1.

The control automaton for **doAction** a and **check** p are following:



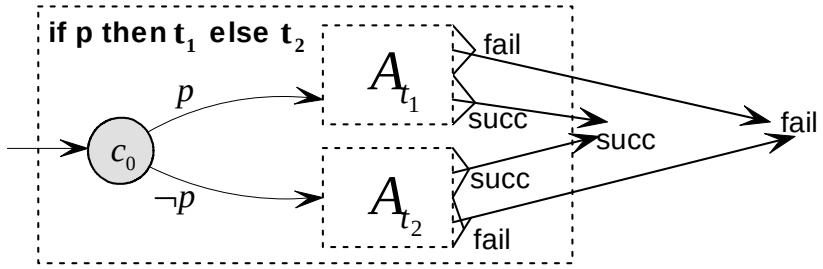
We notice that, in **doAction** a, the transition from control state c_1 guarantees that a domain state is acceptable only if next state is achieved by execution of the action a. In **check** p all transitions are immediate, because it does not require action performing. It only checks that the current domain state satisfies formula p.

The control automata for compound goals is modeled as a composition of automata designed for sub-tasks. The sequence $t_1; t_2$ is represented by following control automaton:
All successful transitions of the automaton for t_1 lead to the initial state of the automaton for t_2. All successful transitions of the automaton for t_2



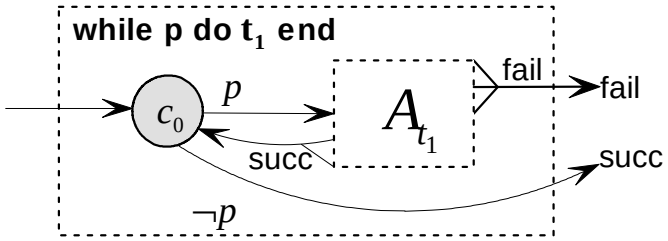
lead to the successful state for the compound goal $t_1; t_2$. Finally, all failure transitions either for t_1 or for t_2 falsify $t_1; t_2$.

The conditional goal task **if-else** is modeled as follows:



The context c_0 immediately moves plan execution to the initial context of one of the control automata for goal tasks t_1 or t_2 according to the current domain state, i.e. whether the property p holds in the current domain state or not. We notice that if **else** part of the **if-else** construction is absent then transition $\neg p$ leads directly to success.

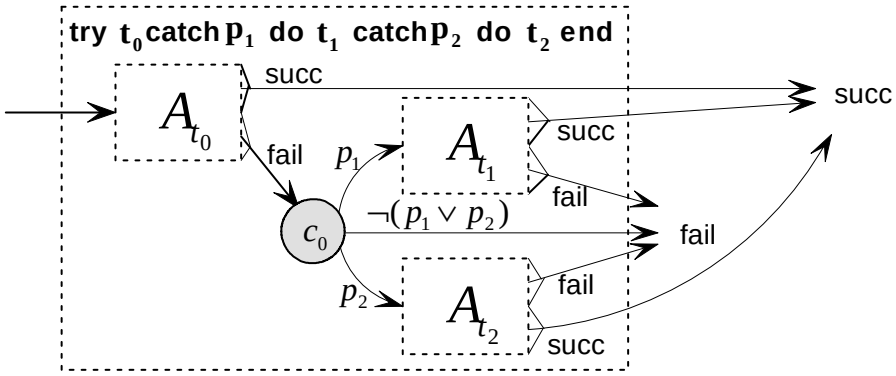
The control automaton for the cyclic goal task **while** is following:



The context c_0 has two immediate transitions guarded by conditions p and $\neg p$. The former leads to the initial context of the automaton for t_1, i.e. the body of the cycle, and the later leads to the success of the compound automaton. The successful transitions of the automaton for t_1 return back

to the c_0, but failure transitions for t_1 falsify compound automaton. We notice that c_0 is marked as a *red block*. It guaranties that loop is finite.

The control automaton for the **try-catch** task with two **catch** operators is following:



All successful transitions of t_0 lead to the success of the compound automaton, but all failure transitions of t_0 lead to the context c_0 which is responsible for management of the "recovery" goals. c_0 has immediate transition to each t_i automaton that guarded by property p_i. In case if there exists a domain state s which satisfies more than one catch property p_i then recovery tasks is selected nondeterministically. The last transition from c_0 to failure is guarded by property $\neg(p_1 \vee \cdots \vee p_n)$. It manages domain states that are not catched by recovery goals.

The operator **policy** causes the refinement of the control automaton of the goal task to which the policy is applied. In order to build the automaton for a goal task **policy** f **do** t_1 **end** we do following:

- construct the automaton for the goal task t_1
- consider all *normal* transitions of the control automaton built for t_1 and conjunctively add f to their guarding policy formulas.

6.4.2 Planning algorithm

To build plans for the goals described by the new goal language we re-use the technique described in Section 5.3 with a minor modification to support control automata with policies. In particular, for each normal transition t, we extend the function *trans-assoc(t, assoc)* by adding a constraint expressed by the policy formula assigned to the transition t.

Chapter 7

Implementation and Evaluation

The planning technique for goals with preferences is implemented as a component of the toolkit , namely WSynth, that was realized as a part of the project Astro [2]. Within this project, we developed tools to support the design and execution of distributed applications obtained by combining existing "services" made available on the Web. The planning algorithm for planning with preferences described in this work is successfully exploited in that context to automatically generate the composition of the existing services, given a description of the requirements that the composition should satisfy.

The novel goal language for fusing procedural and declarative goals is implemented as an extension of the MBP planner [8], which is one of the top planners for extended goals in nondeterministic domains [33, 73, 9, 10, 30].

In this chapter we describe the architectural design of the software components that implement proposed planning techniques and approaches. In fact, the MBP planner and the planner component of WSynth tool are implemented based on the same conceptual model of the architectural design. We also evaluate obtained framework based on the range of experiments applied to the case studies described in the thesis and to the case studies motivated by real-life problems for automated Web Service composition.

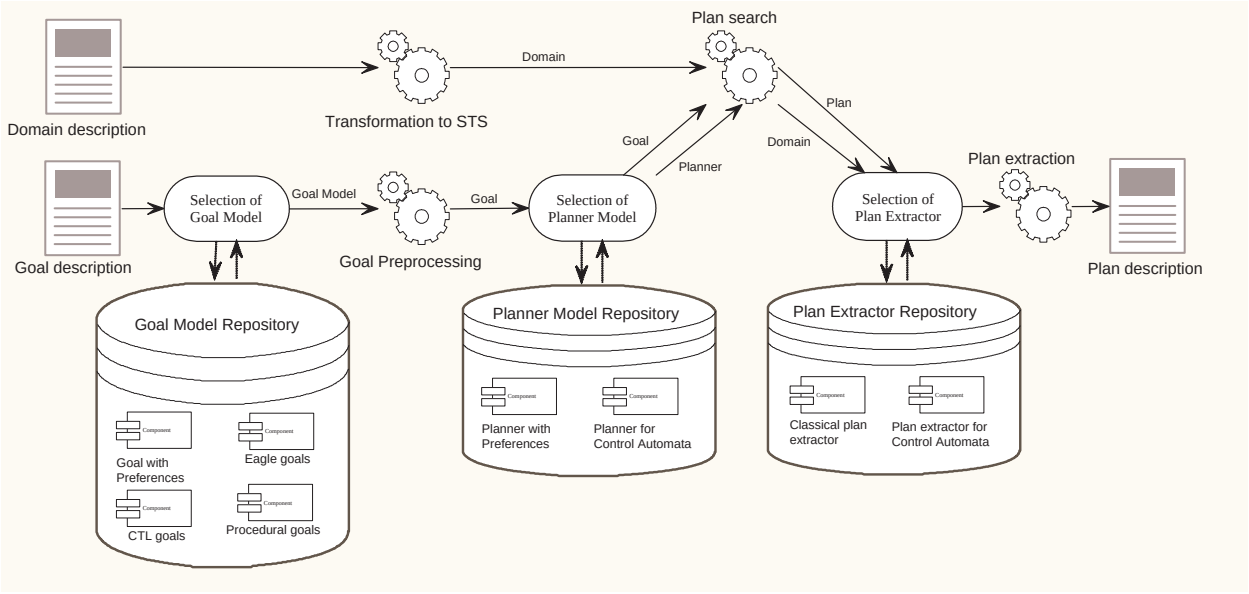


Figure 7.1: The conceptual model of the functional architecture of the MBP planner.

7.1 Planner Architecture

The conceptual model of the functional architecture of the MBP planner is depicted on Figure 7.1. The planning core of WSynth tool is implemented as implication of MBP, therefore, it has identical architecture.

Input specification

The planner accepts two input parameters:

- **Domain description**. The current version assumes that the domain is described using SMV domain definition specification [27]. We note that NuPDDL domain definition language [7], which is an extension for one of the most popular planning domain definition language PDDL [42, 41], can be translated to the SMV notation using nupddl2smv tool [7].

- **Goal description**. The goal is written using the syntax of one of the

goal languages proposed in this dissertation, i.e. goals with preferences and procedural goals. Moreover, the MBP planner supports another languages for extended goals such as CTL or EaGLe.

Plan search phase

The plans search phase is performed as follows:

1. The framework translates the domain description to a state-transition system that encodes the planning domain according to Definition 1.

2. The framework lookups the *goal model* according to the goal description.

3. The framework preprocesses the given goal description based on the specified *goal model*. The obtained object encodes a goal either as a control automaton according to Definition 27 or as a flatten goal with preferences according to Definition 15.

4. The framework lookups a *planner model* that can handle the plan search process for obtained planning goal, i.e. the goal with preferences, or the goal as control automaton.

5. On the last steps, the framework performs planning based on the selected *planner model* and generates a plan description in the human readable format according to the generated plan structure.

Discussion

We note that in this thesis we developed additional goal and plan models to support goals with preferences described in Section 4. We also provide an implementation of the goal model that translates the novel procedural goal language proposed in Section 6 into the control automaton.

7.2 Evaluation and Experimental Results

In order to evaluate proposed techniques, we conducted a set of tests in several experimental domains. All experiments have been done by the 1.6GHz Intel Centrino machine with 512MB memory and running a Linux operating system.

7.2.1 Planning with Preferences

To show effectiveness of the proposed technique for planning with preferences we evaluate it in respect to:

- the size of the planning domain
- the complexity of the planning goal

We consider two domains. The first one is a robot navigation domain, which is defined as follows.

Experimental domain 1. *Consider the domain represented in Figure 7.2. It consists of N rooms connected by doors and a corridor. Each room contains a box. A robot may move between adjacent rooms if the door between these rooms is not blocked, pick up a box in a room and put down it in the corridor. The robot can carry only one box at the same time. A state of the domain is defined in terms of fluent* `room` *that ranges from 0 to N and describes the robot position, of boolean fluent* `busy` *that describes whether the robot is carrying a box at the moment, of boolean fluents* `door[i][j]`*, that describe whether the door between rooms* `i` *and* `j` *is blocked, and of boolean fluents* `box[i]` *describing whether the box in room* `i` *is on its place in the room. The actions are* `pass-i-j, pick-up, put-down`*. Actions* `pass-i-j`*, which changes the robot position from* `i` *to* `j`*, can nondeterministically block* `door[i][j]`*.*

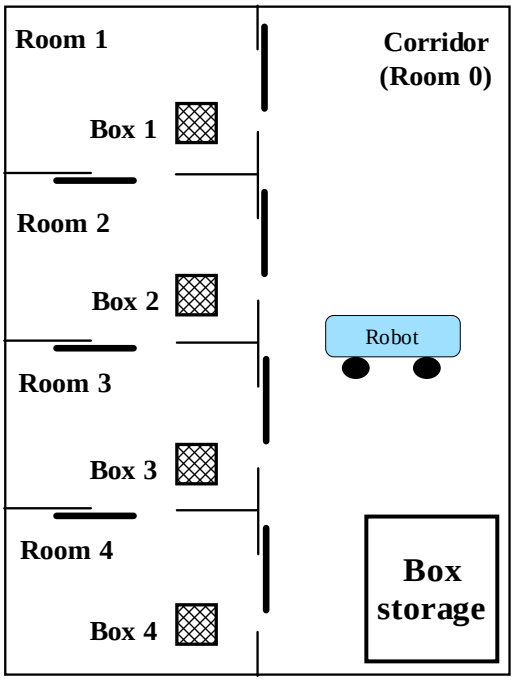


Figure 7.2: A robot navigation domain

We choose this domain for experiments because its size can be easily scaled up, i.e. the number of the domain states depends on the number of rooms with boxes. Moreover, it allows for scaling a complexity for planning goals. For instance, we assume that the planning goal expresses different preferences on how boxes are supposed to be delivered to the box storage in the corridor. The most preferable goal is "deliver all boxes". The set of boxes to be delivered is gradually reduced for goals of intermediate preference. The worst goal is "reach the corridor". It means that the robot has to avoid situation where all doors of the room in which the robot is currently in are blocked. Notice that the door used by the robot to enter the room can become blocked, so it may become necessary to follow a different path to leave the room, which may lead to more blocked doors and unreachable boxes.

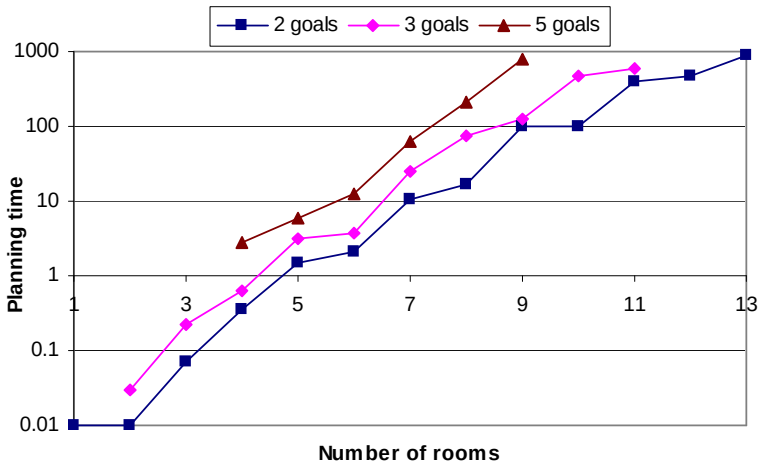


Figure 7.3: Experiments with strong planning with preferences in the robot navigation domain

We fixed number of goals and tested the performance of strong planning algorithm with preferences with respect to the number of rooms in the domain. The average results for 2, 3 and 5 goals are shown on the Figure 7.3. It shows that the proposed technique is able to manage domains of large size: it takes less than 600 seconds to plan for 5 goals in the domain with 9 rooms (i.e. more than 2^{29} states).

The second experimental domain is inspired by a real application, namely *automatic web service composition* [75, 72]. Within the Astro project, we are developing tools to support the design and execution of distributed applications obtained by combining existing "services" made available on the web. The planning algorithms described in this dissertation are successfully exploited in that context to automatically generate the composition of the existing services, given a description of the requirements that the composition should satisfy.

Experimental domain 2. *The planning domain describes a set of "component services", where each service has the structure depicted on Fig-*

ure 7.4. In particular, Figure 7.4.a and Figure 7.4.b describe the "component service" that we use for experiments with strong and strong cyclic planning with preferences respectively.

The initial action "isReady" forces the component service to nondeterministically decide whether it is able to deliver the requested item (i.e., booking an hotel or a flight, renting a car, etc.) or not. In the latter case, the only possibility is to cancel the request. If the service is available, it is still possible to cancel the request, but it is also possible to execute the nondeterministic action "getData", which allows us to acquire information on the service (i.e., the name of the booked hotel, or the id for the car rental). We note that, in case of experiments with strong cyclic planning, the action "getData" can nondeterministically lead back to the state "Yes". The idea is that sometimes data requests have to be repeated several times to be proceeded. In this case we need strong cyclic planning to find a plan to reach the successful state.

The size of the domain search space is defined by the number of the "component services" that have to be composed. The planning domain is represented by the product of all its "component services". In order to scale the complexity of the planning goal we assume that the planning goal expresses different preferences on the task that the web service composition is supposed to deliver. For instance, the most preferred goal g_1 could be "book hotel and flight, and rent a car", the second preference g_2 could be "book hotel and flight without car", and the last preference g_3 could be "book hotel and train" — similar goals are very frequent in the domain of web service composition.

We considered two sets of experiments. In the first set, we tested the performance of the planning algorithm with respect to the size of the planning domain. We considered goals with 1, 7, and 15 preferences, and for all these cases we considered domains with an increasing number of services.

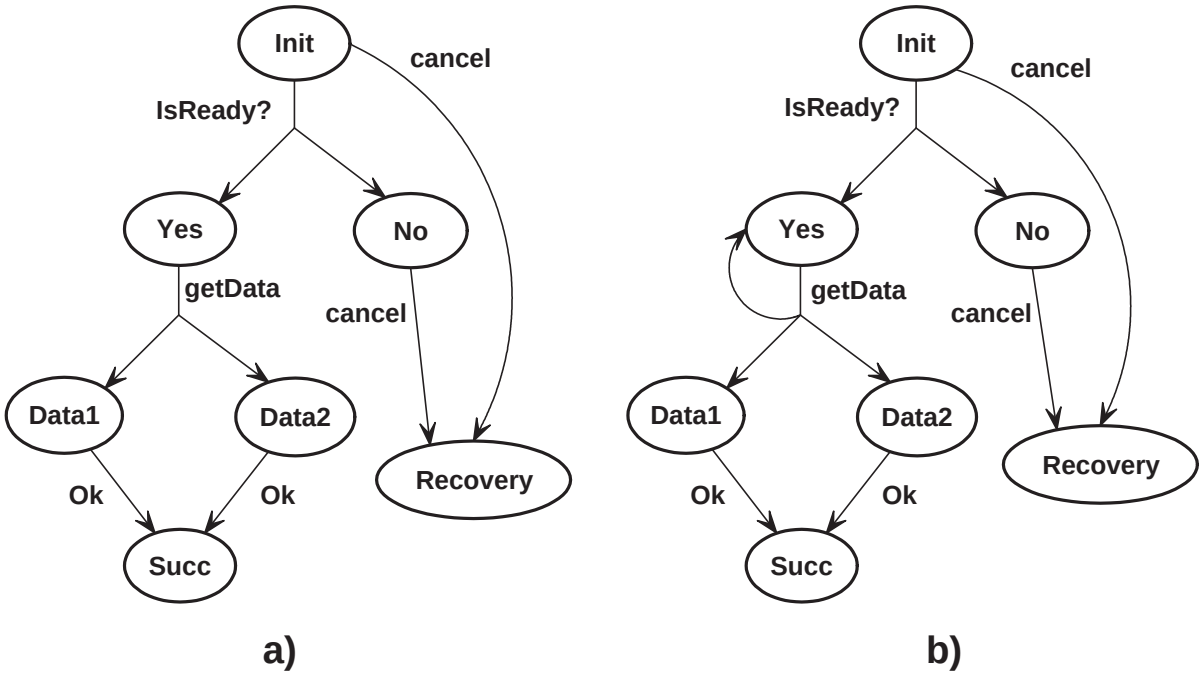


Figure 7.4: The structure of the "component service": a) for experiments with strong planning; b) for experiments with strong cyclic planning

The results for strong and strong cyclic planning algorithms are shown in the left side of Figure 7.5 and Figure 7.6 respectively. The horizontal axis refers to the number of services composing the domain (notice that n services means 7^n states in the domain). In the vertical axis, we report the planning time in seconds. In the second set of experiments, we test the performance with respect to the size of goals. The results for strong and strong cyclic planning algorithms are shown in the right side of Figure 7.5 and Figure 7.6 respectively, where we fixed the size of the domain to 10, 20 and 30 services, and increase the number of preferences in the goal $\mathcal{G}_{list}$ (horizontal axis).

Both sets of experiments show that algorithms are able to manage domains of large size: strong planning for 25 goals and strong cyclic planning for 13 goals in a domain composed from 30 services (i.e., more than 2^{82}

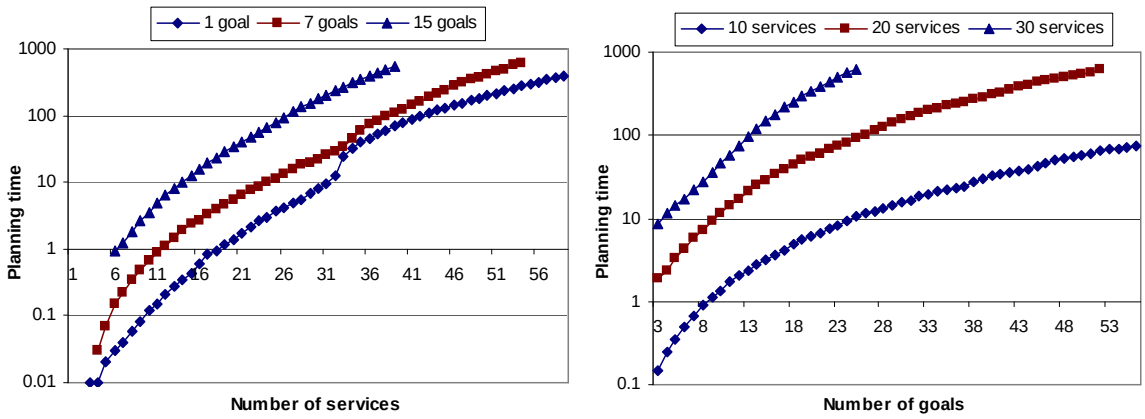


Figure 7.5: Experiments with strong planning with preferences in the service composition domain

states) take about 600 seconds.

7.2.2 Procedural and Declarative Extended Goals

We consider two domains and evaluate the planning time for different kind of planning goals, i.e. sequential, conditional, cyclic, recovering and reachablity goals. The first test domain is a robot navigation domain, which is defined as follows.

Experimental domain 3. *Consider the domain represented in Figure 7.7. It consists of N rooms connected by doors one by one. Each room can contain a box, which can be a bomb or not. A robot may move between adjacent rooms, scan the room to check whether there is a box or not, investigate the box to check whether it is a bomb or not, disarm a bomb, and destroy the box. A state of the domain is defined in terms of fluent* `position` *that ranges from 1 to N and describes the robot position, of fluent* `room[i]` *that can be in one of five states {"unknown", "empty", "withBox", "withBomb", or "safe"}, of fluent* `room_content` *that can be in one*

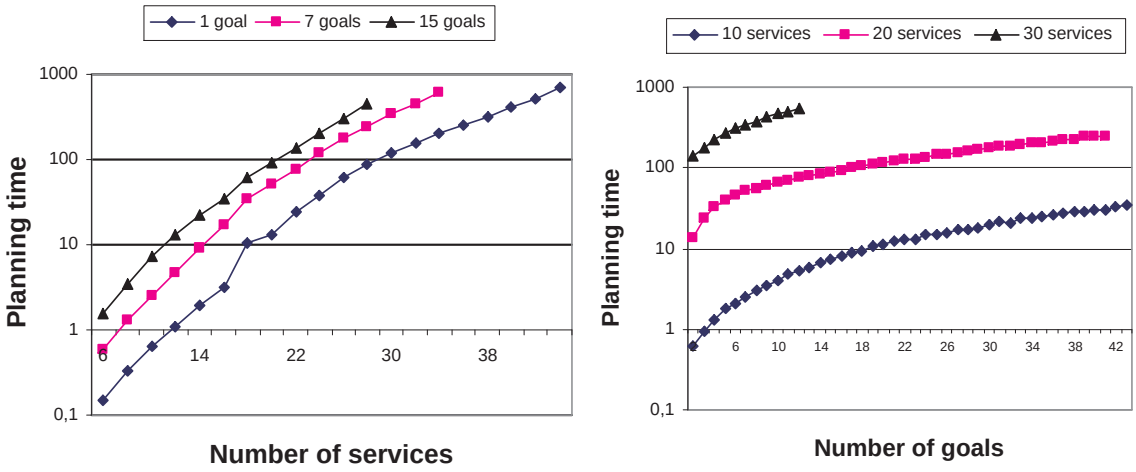


Figure 7.6: Experiments with strong cyclic planning with preferences in the service composition domain

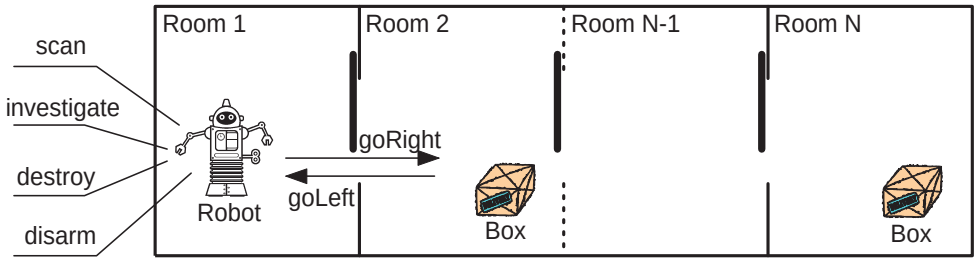


Figure 7.7: A robot navigation domain

of two states {"clean", "damaged" }. The actions are **`goRight, goLeft, scan, investigate, disarm, destroy`**. *Actions* **`goRight, goLeft`** *deterministically change robot position. Action* **`scan`** *nondeterministically changes room state from "unknown" to "empty" or "withBox". Action* **`investigate`** *nondeterministically changes room state from "withBox" to "safe" or "withBomb". Action* **`disarm`** *deterministically changes room state from "withBomb" to "safe" and can be applied only 2 times. Action* **`destroy`** *deterministically changes room state from "withBox" or "withBomb" to "safe" and sets room content to "damaged".*

Intuitively, `disarm` and `destroy` actions can be used to solve the prob-

lem with bomb in the box, but later has to be applied only if former is not allowed. Based on these actions we can see how the planner can capture intentions of the planning goal and generate as best plan as possible. The planning goal is to reach a state where all rooms are "empty" or "safe" and the content of some (very important) rooms is not damaged. We can define such goal in our new goal language as follows (as example we consider a domain with 3 rooms):

```
while (room[position]=unknown ) do
  policy (next(position)=position) do
    try
      goal TryReach (room[position]=empty)
    catch (room[position]=withBomb) do
      goal DoReach (room[position]=safe)
    end
  end
  doAction goRight
end
check(content[3]=clean)
```

In this goal robot movements between rooms are hardcoded, but robot activities in the room requires a combination of reachability goals. Moreover, we prefer to reach a state "room[position]=empty", and only if it is not possible to achieve "room[position]=safe". In order to reduce search space for planning robot activities in the room we defined a policy "**next**(position)= position" that does not allow for considering robot movement actions. Such goal is more expressive than temporal logic formula and much more preferable for human operator than hard-coding the plan by hand. It demonstrates the "happy medium" of the fusing procedural and declarative approaches for the goal definition.

We tested the performance of the planning algorithm with respect to

the number of rooms in the domain and compared with planning for a simple reachability goal that can be defined as "Reach the state where all rooms in the domain are safe or empty" The average results are shown on the Figure 7.8.

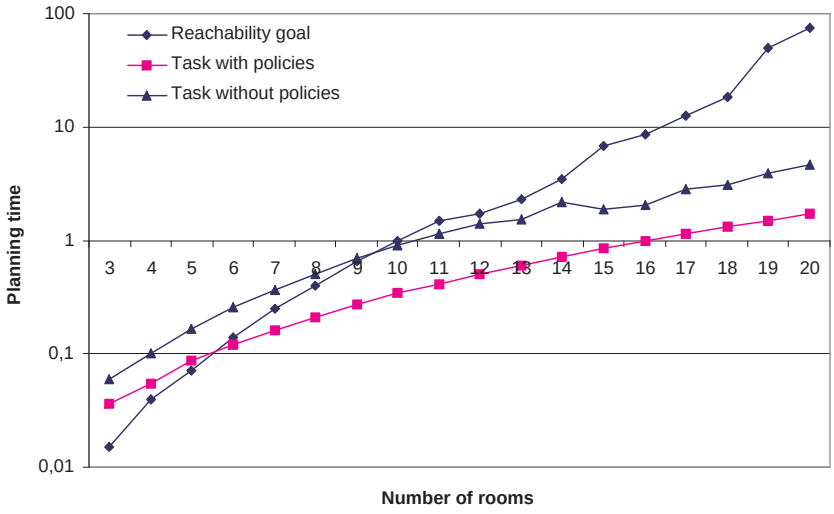


Figure 7.8: Evaluation results for the robot domain

Experiments showed that for simple domains (less than 6 rooms) planning for a reachability goal is faster, because the memory management and operations on the control automaton take more time than planning for reachability goal. But in more complex domain our planning technique solves the problem much faster, especially with systematic usage of policies. If we make our goal weaker by removing the last operator "**check**(content[3]=clean)" then we get a goal where robot activities in the room are not linked with robot position. It means that we can resolve the goal

```
try
   goal TryReach (room[position]=empty)
catch (room[position]=withBomb) do
```

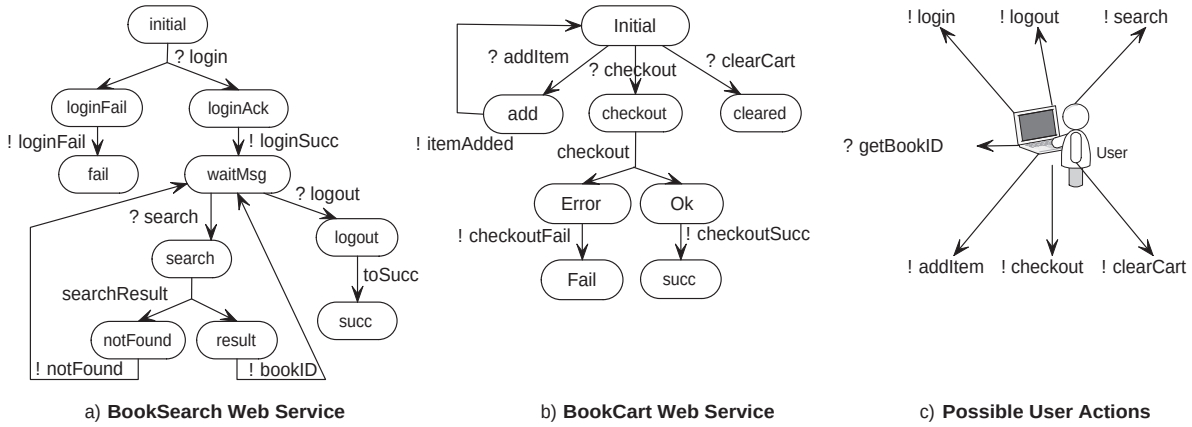


Figure 7.9: Experimental domain for WS composition

goal **DoReach** (room[position]=safe)
end;

in the domain with one room, i.e. in a very small domain. Such planning takes about 0.01 second. After that we can re-use generated plan in the previous goal and obtain a goal task which does not contain **goal** operators. Hence, all *normal* transitions correspond only to **doAction** operators. In fact, we transform the plan search problem to the plan verification problem which is incomparably easier. For instance, planning for such kind of goal in the domain with 50 rooms takes less than 1 second.

The second experimental domain is inspired by the *automatic web service composition* application [75, 72].

Experimental domain 4. *For testing reasons we model a simplified version of the bookstore Web Service application. In particular we consider only 2 Web Services: BookSearch and BookCart which are shown on Figure 7.9(a,b). BookSearch describes a workflow to find a book in the bookstore. BookCart describes a workflow to add obtained book to the user cart and, finally, to checkout it. All possible User activities for communications with bookstore is shown on Figure 7.9(c). We use a fluent "message" to emulate message exchanging between Web Services and User. We write*

"?" before the action name if this action requires (as a precondition) specific message to be sent, and "!" if this action initiates (as a postcondition) the message sending. For instance, the User action "? getBookID" can be performed only if current message is "bookID" that can be set by the action "! bookID" of BookSearch. Due to the lack of space we do not provide a formal definition for the planning domain and the planning problem. Intuitively, each actor in the Web Service composition is represented by a state-transition system and the planning domain is obtained as a product of these STS. N pairs of BookSeach and BookCart services emulate N bookstores and our goal to plan user activities to find and buy at least one book in at least one bookstore.

We tested the performance of the planning algorithm with respect to the number of bookstores in the domain and compared with planning for a simple reachability goal that can be defined as "Reach the state where user found and bought a book in exactly one bookstore". Using the proposed goal language we can encode some search knowledge in the goal definition. For instance, to satisfy such kind of goal we can:

- consider bookstores one by one iteratively until a book is bought. In each bookstore we can:
- try to find the book and buy it. If it is impossible:
- perform some recovering actions to leave bookstore in a consistent state.

Therefore, the planning goal can be written in following way:

```
while ('book is not bought' &&
       '∃ available bookstore') do
  doAction 'choose next bookstore'
  try
```

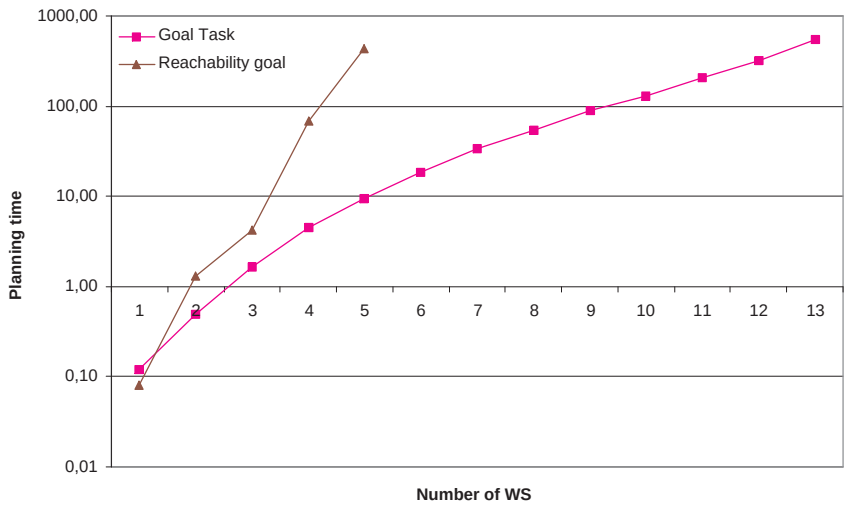


Figure 7.10: Evaluation results for the WS composition domain

```
    goal TryReach 'achieve search results'
    if ('book is found') do
      goal TryReach 'cart is checked out'
    else
      goal DoReach 'user is logged out'
    end;
  catch ('login is failed') {
    goal DoReach 'BookSerch is in fail' &&
                 'BookCart is in initial'
  catch ('cart checkout is failed') {
    goal DoReach 'user is logged out' &&
                 'BookCart is in fail'
  end
end
```

The average results are shown on the Figure 7.10. As in robot domain we can see that our approach is extremely effective in large domains.

7.3 Discussion

In this chapter we reported practical results obtained as a result of application proposed planning techniques to different case studies. The results of the experimental evaluation demonstrates the vitality of novel classes of extended goals proposed in this work. Each planning approach proposed in this work we evaluated in experimental domains inspired by the problem of automated web service composition [68, 75, 72, 82, 57, 58]. Indeed, the Experimental Domain 2 demonstrated the approach to build the composition of abstracted services typical for travel agency domains. The Experimental Domain 4 demonstrated the approach to build a complex bookstore application as a composition of distributed services based on the procedural planning goals. We also showed that all proposed planning algorithm can deal in large space domains.

We remark that results of this work has been successfully applied in the real-life application Astro [2].

Chapter 8

Conclusions

In this thesis we presented novel techniques for planning with extended goals in nondeterministic domains. We considered two classes of extended goals: *goals with preferences*, and *procedural goals* intermixed with *declarative goals*. In both cases we presented a solution in terms of conditional planning. We provided a theoretical framework that includes (*i*) the novel goal language for each class of extended goals we discussed, (*ii*) the formal model for this goal language, and (*iii*) the planning algorithm that can be used to resolve such kind of planning goals. We developed an implementation of the proposed framework and integrated it with MBP planner, which is one of the state of the art planners that can deal with conditioning planning in nondeterministic domains. To evaluate obtained framework we conducted a set of tests and showed that our approaches are promising, since they allow for expressing complex goals and solve such goals efficiently in large space domains.

Planning with preferences

We showed that *planning with preferences* is extremely important for nondeterministic domains, since ever very simple reachability goals in most cases can not be satisfied by *strong* plans. Moreover, *weak* plans can not

be considered as a problem solution because of possibility to terminate in unpredicted fail state.

We proposed the intermediate approach based on the idea that each disputable sub-goal can be defined as a set of alternative sub-goals with different priorities expressing the user preferences. In this case, the basic requirement to the solution is that it has to be a strong plan for the disjunctive combination of all possible sub-goals. It means, that some of sub-goals can be satisfied weakly, or is not satisfied at all. To complete the model, we require that resulting plan has to be optimal according to the user preferences.

Formally, we modeled a *goal with preferences* by the combination of two operators: **OneOf** and **All**. The operator **All** is used to conjunct sub-goals when all of them have to be satisfied. The alternative operator **OneOf** is used for the goal composition when only one of them has to be satisfied. The **OneOf** construction provides a possibility to specify a secondary (less preferable, compensation or recovery) goals to manage the situation when the focused goal became unreachable as result of the nondeterministic action outcome. We proposed a quantitative approach to express user preferences between disjunctive sub-goals of **OneOf** operator, i.e. a natural number is assigned to each disjunctive sub-goal to formalize its "preference value".

We developed a formal model for proposed goal language with preferences. The core element of this model is a *preference function* that assigns a natural number to any domain state according to the specified planning goal. This function evaluates each domain state according to the user preferences expressed in the planning goal. Based on the idea of preference function, we formalized the concept of *optimal plan*. To model plan optimality criteria we used an *optimistic behavior assumption*, i.e. we compare the plans according to the goals of best preferences reached by the plans. In

case the maximum possible goals are equal, we apply a *pessimistic behavior assumption*, i.e., we compare the plans according to the goals of worst precedence. In comparison with other possible models of preferences, our approach has the advantage of being simpler, and the experiments show that it is sufficient in practice to get the expected plans.

To evaluate the proposed framework we tested its performance in two dimensions, i.e. we analyzed planning time in respect to complexity of the planning goal and size of the planning domain. All experiments showed that our technique allows for efficient resolving complex goals with preferences in the large space planning domains.

Procedural and declarative goals

In this work we presented a novel approach to planning in nondeterministic domains, where temporally extended goals can be expressed as arbitrary combinations of declarative specifications and of procedural plan fragments. The planning framework was developed on the basis of idea that procedural constructions typical for the plan definition can interleave with declarative goals in order to handle extended planning goals in a procedural way. This framework provides an opportunity to the human domain expert to express his knowledge about the planning domain in the goal. Moreover, using a procedural approach in the goal definition gives a possibility to help the planner in reasoning by dividing a complex goal into smaller ones and resolving them separately. The advantage of the approach is twofold. First, the expressiveness of the language allows for easier specifications of complex goals. Second, rather simple and natural combinations of procedural and declarative goals open up the possibility to improve performances significantly, as shown by our preliminary experimental evaluation

We built our framework on top of a formal model, where a planning goal

is represented by a control automaton. This formalism was developed with aim to uniform the planning goal structure and to separate the plan search techniques from the concrete planning goal definition. In this formalism the planning framework is modeled as a 3-layer structure, where the top layer represents a planning goal defined in some goal language, the middle layer represents a control automaton built according to the given planning goal, and the low-level layer represents a plan search algorithm that is guided by the control automaton. The middle layer makes the plan search techniques independent from the goal language. It makes the framework scalable, i.e. we can easily introduce the new goal language constructs as a plug-in component that describes a translation to a control automaton. This idea for the planning framework structure was successfully applied in MBP planner, and in this work we reuse and extend it.

We conducted a set of experiments, which showed how our new language can express complex planning goals in a procedural way and demonstrated that such goals can be solved efficiently in the large space planning domains. We showed how the procedural constructs can be used to make the goal more precise and how the goal can be clarified using the domain specific search knowledge information. We demonstrated that such kind of goal clarification can significantly improve and optimize the plan search process.

8.1 Directions for Future Work

This work deals with planning for new kinds of extended goals that were not addressed in the literature before. It opens a wide range of directions for future work.

The goals with preferences and goals with procedural constructions represent two orthogonal dimensions for the extended goals. Intuitively, the

user preference is a qualitative property of the planning goal that does not assume any restrictions to the goal internal structure. The idea of fusing procedural and declarative planning goals consider only the internal goal structure and does not cover the user intentions such as preferences between different goals. In this work we dealt with both goal dimensions separately, therefore one of the most important open issues is to investigate a problem of planning for goals that union both approaches. This can have a big impact on the expressiveness of the resulting goal language and effectiveness of the plan search techniques. Moreover, both classes of planning goals inspired a lot of personal open issues to be investigated.

There are two key points for planning with preferences: (*i*) methodology for formalization of different types of preferences between goals, and (*ii*) the concept of the optimal plan that meets goals preferences. In this work we addressed only core types of preferences that assume satisfiability either all sub-goals or at least one sub-goal with maximal possible preference. One of the possible extension would be to introduce new kinds of preferences, such as satisfiability of a maximal possible number of sub-goals or satisfiability of a sub-set of sub-goals with maximal possible conjunctive preference. Concerning to the plan optimality concept, a set of approaches should be investigated. Additionally to the maximal and minimal preferable goals that can be satisfied by the plan, it is necessary to investigate the impact of the intermediate preferable goals to the plan quality.

We also work on the technical part of the proposed approach, i.e. the presented algorithm based on the backward search idea where we start planning from the less preferable sub-goal. We plan to adopt some strategies to optimize the plan search process which start planning from the most preferable sub-goal and consider less preferable sub-goals only if it is really necessary.

Another set of extensions would target a procedural goal language. One

of the most important problems is to apply object oriented approach to the planning domain model, where the planning domain is represented by a set of objects that interact with each other. In this setting, we can introduce a concept of local variables and procedure parameters based on the object types defined by the domain. Moreover, the quantifies $\forall$ and $\exists$ applied to domain objects can significantly improve expressiveness of the goal language. We plan to add additional concepts to the goal languages to make it more procedural, i.e. the concept of a procedure and procedure call.

With respect to the efficiency of the proposed techniques, we plan to work on improvement of both theoretical and practical part of the proposed planning approaches. We are interested in development of the universal approach for the plan search process based on the control automata.

Bibliography

[1] Mitchell Ai-Chang, John Bresina, Len Charest, Adam Chase, Jennifer Cheng jung Hsu, Ari Jonsson, Bob Kanefsky, Paul Morris, Kanna Rajan, Jeffrey Yglesias, Brian G. Chafin, William C. Dias, and Pierre F. Maldague. Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.

[2] ASTRO. Project ASTRO: Supporting the Composition of Distributed Business Processes. [http://astroproject.org].

[3] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Ann. Math. Artif. Intell.*, 22(1-2):5–27, 1998.

[4] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191, 2000.

[5] Ruzena Bajcsy. Active perception. *PIEEE*, 76(8):996–1005, August 1988.

[6] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[7] Piergiorgio Bertoli, Alessandro Cimatti, Ugo Dal Lago, and Marco Pistore. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. In *ICAPS Workshop on PDDL, Informal Proceedings*, pages 15–24, 2003.

[8] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a Model Based Planner. In *Proc. of IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[9] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, and Paolo Traverso. A framework for planning with extended goals under partial observability. In *ICAPS*, pages 215–225, 2003.

[10] Piergiorgio Bertoli and Marco Pistore. Planning with extended goals and partial observability. In *ICAPS*, pages 270–278, 2004.

[11] Meghyn Bienvenu and Sheila McIlraith. Qualitative Dynamical Preferences in the Situation Calculus. In *Multidisciplinary IJCAI'05 Workshop on Advances in Preference Handling*, 2005.

[12] Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors. *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*. AAAI, 2005.

[13] Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *ECP*, pages 319–332, 1999.

[14] Craig Boutilier, Ronen Brafman, and Christopher Geib. Structured reachability analysis for markov decision processes. In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 24–32, San Francisco, CA, 1998. Morgan Kaufmann.

[15] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelliegence Research*, 11:1–94, 1999.

[16] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362, 2000.

[17] R. Brafman and U. Junker, editors. *Multidisciplinary IJCAI-05 Workshop on Advances in Preference Handling*, 2005.

[18] Ronen I. Brafman and Yuri Chernyavsky. Planning with Goal Preferences and Constraints. In *Proc. ICAPS'05*, 2005.

[19] John L. Bresina, Ari K. Jónsson, Paul H. Morris, and Kanna Rajan. Activity planning for the mars exploration rovers. In Biundo et al. [12], pages 40–49.

[20] Gerhard Brewka. A rank based description language for qualitative preferences. In *Proc. ECAI'04*, 2004.

[21] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[22] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[23] W. Burgard, A.B. Cremers, Dieter Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and Sebastian Thrun. The interactive museum tour-guide robot. In *Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998. Outstanding paper award.

[24] Reviewed by Thomas Dean. Ai scheduling methods. *IEEE Expert: Intelligent Systems and Their Applications*, 10(1):76–77, 1995.

[25] D. Calvanese, G. de Giacomo, and M. Vardi. Reasoning about actions and planning in ltl action theories, 2002.

[26] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. Sat-based planning in complex domains: concurrency, constraints and nondeterminism. *Artif. Intell.*, 147(1-2):85–117, 2003.

[27] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. Nusmv 2.1 user manual.

[28] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, 2005.

[29] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.

[30] A. Cimatti and M. Roveri. Conformant planning via model checking. In S. Biundo, editor, *Proceeding of the Fifth European Conference on Planning*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 21–34, Durham, United Kingdom, September 1999. Springer-Verlag.

[31] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.

[32] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating BDD-based and SAT-based symbolic model checking. In *FroCos*, pages 49–56, 2002.

[33] Alessandro Cimatti, Fausto Giunchiglia, Enrico Giunchiglia, and Paolo Traverso. Planning via model checking: A decision procedure for *r*. In *ECP*, pages 130–142, 1997.

[34] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[35] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[36] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[37] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 447–454, Edmonton, Alberta, Canada, July 2002. AAAI-Press/The MIT Press.

[38] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Programming Under Action Costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2002.

[39] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier, 1990.

[40] Kutluhan Erol, James A. Hendler, and Dana S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Artificial Intelligence Planning Systems*, pages 249–254, 1994.

[41] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[42] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—the planning domain definition language. In *AIPS-98*, 1998.

[43] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[44] Enrico Giunchiglia and Marco Maratea. Planning as satisfiability with preferences. In *AAAI*, pages 987–992, 2007.

[45] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *AAAI/IAAI*, pages 948–953, 1998.

[46] Henrik Grosskreutz and Gerhard Lakemeyer. cc-Golog - An Action Language with Continuous Change. *Logic Jnl IGPL*, 11(2):179–221, 2003.

[47] J. Hoffmann and R. Brafman. Contigent Planning via Heuristic Forward Search with Implicit Belief States. In *Proc. ICAPS'05*, 2005.

[48] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[49] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[50] Froduald Kabanza, Michel Barbeau, and Richard St.-Denis. A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. *IEEE Transactions on Automatic Control*, 43(11):1543–1559, November 1998.

[51] Froduald Kabanza and Sylvie Thiébaux. Search control in planning for temporally extended goals. In Biundo et al. [12], pages 130–139.

[52] Ugur Kuter and Dana S. Nau. Forward-chaining planning in nondeterministic domains. In *AAAI*, pages 513–518, 2004.

[53] Ugur Kuter, Dana S. Nau, Marco Pistore, and Paolo Traverso. A hierarchical task-network planner based on symbolic model checking. In Biundo et al. [12], pages 300–309.

[54] Jonas Kvarnstro"m and Martin Magnusson. Talplanner in the third international planning competition: Extensions and control rules. *J. Artif. Intell. Res. (JAIR)*, 20:343–377, 2003.

[55] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *AAAI/IAAI*, pages 447–454, 2002.

[56] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[57] Alexander Lazovik. *Interacting with Service Compositions*. PhD thesis, International Doctorate School in Information and Communication Technologies (ICT), Trento University, 2006.

[58] Alexander Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2006.

[59] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[60] I. Little, D. Aberdeen, and S. Thiébaux. Prottle: A probabilistic temporal planner. In *Proc. AAAI'05*, 2005.

[61] Stephen M. Majercik. Planning under uncertainty via stochastic satisfiability. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, page 950, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[62] Stephen M. Majercik and Michael L. Littman. MAXPLAN: A new approach to probabilistic planning. In *Artificial Intelligence Planning Systems*, pages 86–93, 1998.

[63] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artif. Intell.*, 147(1-2):119–162, 2003.

[64] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. pages 26–45, 1987.

[65] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[66] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[67] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *J. Artif. Intell. Res. (JAIR)*, 20:379–404, 2003.

[68] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, Dmitry Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[69] Marco Pistore, Pistore Bertoli, Fabio Barbon, Dmitry Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[70] Marco Pistore, Renato Bettin, and Paolo Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *ECP'01, LNAI. Springer Verlag*, 2001.

[71] Marco Pistore, Renato Bettin, and Paolo Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Proceedings of the Sixth European Conference on Planning (ECP'01)*, pages 253–264, 2001.

[72] Marco Pistore, Annapaola Marconi, Paolo Traverso, and Piergiorgio Bertoli. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

[73] Marco Pistore and Paolo Traverso. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, pages 479–486, 2001.

[74] Marco Pistore and Paolo Traverso. Planning as model checking for extended goals in non-deterministic domains. In B. Nebel, editor, *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 479–486. Morgan Kaufmann Publisher, August 2001.

[75] Marco Pistore, Paolo Traverso, and Piergiorgio Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[76] I. Refanidis and I. Vlahavas. Multiobjective heuristic state space planning. *Artificial Intelligence*, 145(1-2):1–32, 2003.

[77] Raymond Reiter. *Knowledge in action : logical foundations for specifying and implementing dynamical systems*. MIT Press, Cambridge, Mass., 2001. The frame problem and the situation calculus.

[78] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research*, 10:323–352, 1999.

[79] María Dolores Rodríguez-Moreno, Daniel Borrajo, Amedeo Cesta, and Angelo Oddi. Integrating planning and scheduling in workflow domains. *Expert Syst. Appl.*, 33(2):389–406, 2007.

[80] D. Smith. Choosing Objectives in Over-Subscription Planning. In *Proc. ICAPS'04*, 2004.

[81] T. Son and E. Pontelli. Planning with Preferences using Logic Programming. In *Proc. LPNMR'04*, 2004.

[82] B. Srivastava and J. Koehler. Web service composition — current solutions and open problems. In *ICAPS 2003*, 2003.

[83] Austin Tate, Jeff Dalton, and John Levine. O-plan: A web-based ai planning agent. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 1131–1132. AAAI Press / The MIT Press, 2000.

[84] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In *ICCAD*, pages 130–133, 1990.

[85] Menkes van den Briel, Romeo Sanchez, Minh B. Do, and Subbarao Kambhampati. Effective approaches for Partial Satisfaction (Over-Subscription) Planning. In *Proc. AAAI'04*, 2004.

[86] D. Weld, C. Anderson, and D. Smith. Extending graphplan to handle uncertainty and sensing actions, 1998.

[87] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 897–904, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[88] D.E. Wilkins and M. desJardins. A call for knowldedge-based planning. *AI Magazine*, 22(1):99–115, 2001. Overview of knowledge-based planning.