Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

Fusing Procedural and Declarative Planning Goals for Nondeterministic Domains

Dzmitry Shaparau
FBK-IRST, Trento, Italy
shaparau@fbk.eu

Marco Pistore
FBK-IRST, Trento, Italy
pistore@fbk.eu

Paolo Traverso
FBK-IRST, Trento, Italy
traverso@fbk.eu

Abstract

While in most planning approaches goals and plans are different objects, it is often useful to specify goals that combine declarative conditions with procedural plans.

In this paper, 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. We implement the planning framework and perform a set of experimental evaluations that show the potentialities of our approach.

Introduction

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. (Kabanza, Barbeau, and St-Denis 1997; Pistore and Traverso 2001; Dal Lago, Pistore, and Traverso 2002; Kabanza and Thiébaux 2005).

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 directly specify 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, consider the goal for a robot that has to visit periodically some locations in a building. This goal 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 paper, 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 temporal goals of the EAGLE language (Dal Lago, Pistore, and Traverso 2002) 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 (Dal Lago, Pistore, and Traverso 2002). We implement the proposed approach and perform a preliminary set of experimental evaluations with some examples taken from the robot navigation domain. We evaluate the 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.

The paper is structured as follows. We first give a short background on planning in nondeterministic domains. We then define the goal language and give its semantics. Next we show how we can construct control automata that can guide the search for a plan. We finally report the results of our experimental evaluation and discuss a comparison with related work.

Background

The aim of this section is to review the basic definitions of planning in nondeterministic domains which we use in the rest of the paper. All of them are taken, with minor modifications, from (Dal Lago, Pistore, and Traverso 2002; Cimatti et al. 2003).

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.*

Intuitively, temporally extended goals are the goals that express the conditions on the whole execution of plans, not just on the final states. Alternatively, they can be considered as goals that dynamically change during execution. Traditionally, such evolving goals are modeled through 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 defines the action to be executed depending on the current domain state and execution context, and one which defines the evolution of the execution context depending on the outcome of the action execution.

Definition 2 (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 we are in a state s and in an execution context c, then $act(s, c)$ returns the action to be executed by the plan, while $ctxt(s, c, s')$ associates to each reached state s' a new execution context. The tuple $\langle s, c \rangle \in \mathcal{S} \times \mathcal{C}$ defines the plan execution state.

We use σ to denote a finite *plan execution path*, which is a sequence of state-actions $\langle s_1, c_i \rangle \xrightarrow{a_1} \ldots \xrightarrow{a_n} \langle s_n, c_k \rangle$. We denote with $first(\sigma)$ and $last(\sigma)$ the first and the last plan states in the execution paths. We write $s \in \sigma$ if state s appears in the path σ. We use a notation $\sigma = \sigma_1; \sigma_2$ if the path σ is obtained by the concatenation of σ_1 and σ_2. We write $\sigma_1 \leq \sigma_2$ if σ_1 is a prefix for σ_2, i.e $\exists \sigma' : \sigma_2 = \sigma_1; \sigma'$.

To define when a goal is satisfied we assume that the "semantics" of the goal language defines when a goal "g" is satisfied in a plan execution state $\langle s, c \rangle$, and we require that the goal is satisfied in the initial plan execution state $\langle s_0, c_0 \rangle$

The definition of $\langle s, c \rangle \models g$ depends on the specific goal language. For instance, in the case of CTL goals (Pistore and Traverso 2001), the standard semantics can be used (Emerson 1990), while ad-hoc semantics have been defined for goal languages specifically designed for planning, e.g., EAGLE goals (Dal Lago, Pistore, and Traverso 2002).

The Goal Language

We propose an extension of the existing approaches for extended goal language which combines the benefits of temporal logic formulas with a procedural goal definition approach. Let $\mathcal{B}$ be a set of *basic propositions* and p be a *propositional formula* over $\mathcal{B}$. Let $\mathcal{A}$ be a set of actions, and $\mathcal{G}$ be a set of temporally extended goals. The *extended goal tasks* 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 the intuition and motivation for this language. First, the language defines the operator **goal** g, where g is a temporally extended goal. Currently, we support CTL (Pistore and Traverso 2001) and EAGLE (Dal Lago, Pistore, and Traverso 2002) goals, but our approach can be generalized to support any kind of temporally extended goals. In order to make this paper self-contained we describe the management of two basic types of goals expressed in EAGLE language: **DoReach** p and **TryReach** p. **DoReach** represents a reachability goal which has to be satisfied in spite of nondeterminism. **TryReach** is a weak version of **DoReach**. It requires that at least one execution path has to end up in a successful state, but it also requires that the plan has to do "everything possible" to satisfy the goal.

The other constructs of the language extend temporally extended goals with procedural declarations.

The primitive action call operator **doAction** 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 effectively 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 some intermediate sub-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.

The sequence of goal tasks $t_1; t_2$ 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 ideas for plan search improvements is to split a complex goal into a sequence of simpler goals. For example, a goal *"Deliver the box from the room A to the 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 the different search strategies (pathfinding and scanning), therefore the right goal split in a sequences can significantly reduce planning search space.

The conditional goal task **if-else** provides the possibility to switch between different goals based on the reached do-

main state. Moreover, we often need to define a cyclic goal task **while** 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**"*.

The goal task **check** verifies whether the current domain state is allowed according to the check-point condition. For example, if we have a task 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.

Construction **try-catch** is used to model recovery from failure due to domain nondeterminism. This operator requires a plan that does everything possible to achieve the 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.

The last construction of the proposed language is **policy** which can be defined for any goal task t. The policy is a formula on the current states, actions, and next states (operator **next**). Intuitively, policy is an additional restriction to the domain transition function, which defines the relation between the current state and the next state of basic propositions. Therefore we allow the 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) $\Rightarrow$ **action**=*look-around*". It means that we restrict the planner to consider only the actions which change the robot position or the 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 complex goals. The reason is that simpler goals can be restricted by stronger policies.

We now show key benefits of the proposed goal language by the example of a simple robot navigation domain.

Example 1 (Experimental domain) *Consider the domain represented in Figure 1. It consists of a sequence of N rooms connected by doors. 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 of two states {"clean", "damaged" }. The actions are* `goRight`, `goLeft`, `scan`, `investigate`, `disarm`, `destroy`. *Actions* `goRight`, `goLeft` *deterministically change the robot position. Action* `scan` *nondeterministically changes the room state from "unknown" to "empty" or "withBox". Action* `investigate` *nondeterministically changes the room state from "withBox" to "safe" or "withBomb". Action* `disarm` *deterministically changes the room state from "withBomb" to "safe" and can be applied only 2 times. Action* `destroy` *deterministically changes the room state to "safe" and sets the room content to "damaged".*

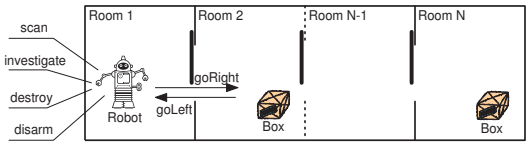


Figure 1: A robot navigation domain

Intuitively, `disarm` and `destroy` actions can be used to remove the threat of the box explosion, but `destroy` has to be applied only if `disarm` is not allowed. Based on these actions we will see how the planner can capture intentions of the planning goal and generate the best possible plan, i.e. to prevent the content of room to be damaged, the planner has to propose to perform `destroy` only if `disarm` can not be applied.

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):

Example 2 (Procedural goal task)

```
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 the robot movements between the rooms are hardcoded, but the robot activities in the room require a combination of reachability goals. Moreover, we prefer to reach a state "room[position]=empty", and only if it is not possible we require to achieve "room[position]=safe". In order to reduce the search space for planning robot activities in the room we define a policy "**next**(position)=position" that does not allow for considering robot movement actions.

We note that the proposed procedural goal language does not force the user to hardcode the complete solution in the goal definition. It provides a toolkit for embedding domain specific knowledge in the goal definition and to guide the plan search process. In the example above we hardcoded robot movements because it is quite intuitive and easy. Consider another goal example where, for instance, we need to remove the threat of the box explosion in the room 3 only:

```
policy (next(position)!=position) do
  goal DoReach position=3
end
doAction investigate
if (room[3]==box) do
  doAction destroy
end
```

In this example, the procedure of the robot activities in the room is fixed but the robot movement to the room is defined as a reachability goal. In this example we encode the strategy that if the robot detects a box it has to be destroyed.

In comparison with CTL goals, the proposed goal language is more expressive because it provides such constructions as **try-catch** or **while** that can not be expressed in CTL. Moreover, the operator **goal** provides a native support for the goals expressed in CTL. In comparison with the EaGLe goals, the proposed goal language can be considered as a wide extension that contains a lot of new procedural constructs such as **doAction**, **if-else**, **while**, **policy**, and **goal**. Moreover, the syntax of our language allows the usage of plan fragments in the definition of planning goals, i.e. previously generated or written manually plans can be used as parts of new goals.

We now provide a formal semantics for the proposed language. In order to formalize whether the plan π satisfies the goal task t, we assign to each plan execution state $\langle s, c\rangle$ two sets: $\mathcal{S}_t^\pi(\langle s, c\rangle)$ and $\mathcal{F}_t^\pi(\langle s, c\rangle)$. These sets contain finite plan execution paths which satisfy or fail the goal task t from the plan state $\langle s, c\rangle$ 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 3 *A plan π satisfies a goal task t in a state $\langle s, c\rangle$ if and only if $\mathcal{F}_t^\pi(\langle s, c\rangle) = \emptyset$.*

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

$t =$ **goal** g is satisfied by π if π satisfies the goal g written either in CTL (Pistore and Traverso 2001) or EaGLe (Dal Lago, Pistore, and Traverso 2002) goal languages.

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

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

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

$t =$ **if** p **do** t_1 **else** t_2 **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 $\mathcal{S}_t^\pi(\langle s, c\rangle) = \mathcal{S}_{t_1}^\pi(\langle s, c\rangle)$ and $\mathcal{F}_t^\pi(\langle s, c\rangle) = \mathcal{F}_{t_1}^\pi(\langle s, c\rangle)$. Otherwise, $\mathcal{S}_t^\pi(\langle s, c\rangle) = \mathcal{S}_{t_2}^\pi(\langle s, c\rangle)$ and $\mathcal{F}_t^\pi(\langle s, c\rangle) = \mathcal{F}_{t_2}^\pi(\langle s, c\rangle)$.

$t =$ **while** p **do** t_1 **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 $\mathcal{S}_t^\pi(\langle s, c\rangle) = \{\langle s, c\rangle\}$ and $\mathcal{F}_t^\pi(\langle s, c\rangle) = \emptyset$. Otherwise, $\mathcal{S}_t^\pi(\langle s, c\rangle) = \{\sigma : last(\sigma) \not\models p \wedge \exists\sigma_1; ...; \sigma_n = \sigma.\forall i = 1..n : first(\sigma_i) \models p \wedge \sigma_i \in \mathcal{S}_{t_1}^\pi(first(\sigma_i))\}$ and $\mathcal{F}_t^\pi(s) = \{\sigma : \exists\sigma_1; ...; \sigma_n = \sigma.\forall i = 1..n-1 : first(\sigma_i) \models p \wedge \sigma_i \in \mathcal{S}_{t_1}^\pi(first(\sigma_i)) \wedge \sigma_n \in \mathcal{F}_{t_1}^\pi(last(\sigma_{n-1}))\}$.

$t =$ **try** t_0 **catch** p_1 **do** t_1 ... **catch** p_n **do** t_n **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 fails. Formally,
$\mathcal{S}_t^\pi(\langle s, c\rangle) = \{\sigma : \sigma \in \mathcal{S}_{t_0}^\pi(\langle s, c\rangle)\} \cup \{\sigma = \sigma_0;\sigma' : \sigma_0 \in \mathcal{F}_{t_0}^\pi(\langle s, c\rangle) \wedge \exists i.last(\sigma_0) \models p_i \wedge \sigma' \in \mathcal{S}_{t_i}^\pi(last(\sigma_0))\}$ and
$\mathcal{F}_t^\pi(\langle s, c\rangle) = \{\sigma : \sigma \in \mathcal{F}_{t_0}^\pi(\langle s, c\rangle) \wedge \forall i = 1..n : last(\sigma) \not\models p_i\} \cup \{\sigma = \sigma_0;\sigma' : \sigma_0 \in \mathcal{F}_{t_0}^\pi(\langle s, c\rangle) \wedge \exists i.last(\sigma_0) \models p_i \wedge \sigma' \in \mathcal{F}_{t_i}^\pi(last(\sigma_0))\}$.

$t =$ **policy** f **do** t_1 **end** requires to satisfy the 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, $\mathcal{S}_t^\pi(\langle s, c\rangle) = \{\sigma : \sigma \in \mathcal{S}_{t_1}^\pi(\langle s, c\rangle) \wedge \forall s_i, a, s_{i+1}.f\mid_{s_i,a,s_{i+1}} = \top\}$ and $\mathcal{F}_t^\pi(\langle s, c\rangle) = \mathcal{F}_{t_1}^\pi(\langle s, c\rangle)$, where $s_i, s_{i+1} \in \sigma$ are states connected by the action a.

Planning Approach

In this work we re-use the planning algorithm that was proposed in (Dal Lago, Pistore, and Traverso 2002) with minor improvements. It is based on the control automata and performed in three steps:

- Initially, we transform the goal task into a control automaton that encodes the goal requirements. Each control state describes some intermediate sub-goal that the plan intends to achieve. The transitions between control states define constraints on the domain states to satisfy the *source* sub-goal and to start satisfiability of the *target* sub-goal.
- Once the control automaton has been built we associate to each control state a set of domain states for which a plan exists with respect to the sub-goal encoded in this control state. The plan search process starts from assumption that all domain states are considered compatible with all the control states, and this initial assignment is then iteratively refined by discarding those domain states that are recognized incompatible with a given control state until a fix-point is reached.
- Finally, we consider each control state as a plan execution context and extract the resulting plan based on the set of domain states associated to the control states.

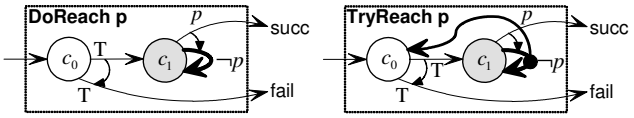
We re-use this approach because it is very scalable to support new goal languages. In fact, last two steps of the planning algorithm are independent from the goal language. Therefore, we only need to define the control automaton construction process for the proposed goal language.

Definition 4 (Control Automaton) *Let F be a policy formula defined over domain states $\mathcal{S}$, next states **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.*

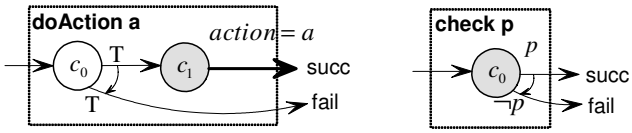
The order of the transition list represents the preference among these transitions. The *normal* transitions correspond to the action execution in the plan. Domain state s satisfies the *normal* transition guarded by proposition formula p if it satisfies p and there is an action a from s such that all possible action outcomes are compatible with some of the target control states. Moreover, a *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 cannot be guarded by the policy formulas. The *red block* states indicate 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 (Pistore, Bettin, and Traverso 2001) for control automata definitions. We start from the **goal** operator. As we mentioned above, for simplicity we consider only **TryReach** and **DoReach** temporally extended goals:



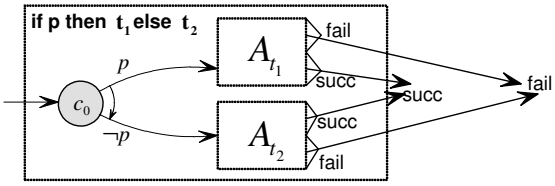
Both automata consist of two states: c_0 (initial control state) and c_1 (reachability *red block* state). There are two transitions outgoing from the red block state c_1 in **DoReach** automaton. The first one is guarded by the condition p. It is a success transition that corresponds to the domain states 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* transition 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 the goal **TryReach** p holds for all the next states, but only for some of them. Therefore, the transition has two possible targets, namely the control state 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 the goal **TryReach** p requires that there should always be 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.

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



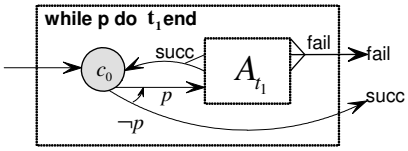
We notice that, in **doAction** a, the transition from the control state c_1 guarantees that a domain state is acceptable only if the next state is achieved by the execution of the action a. In **check** p all transitions are immediate, because action performing is not required. The automaton only checks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

The control automata for compound goal tasks are modeled as a composition of automata designed for sub-tasks. The conditional goal task **if-else** is modeled as follows:



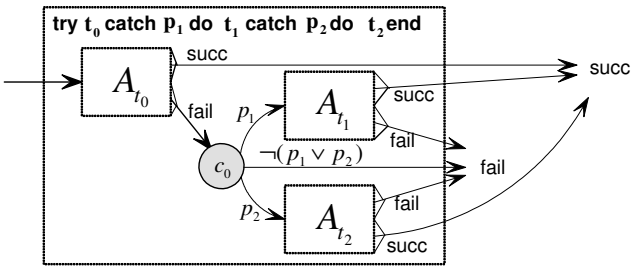
The context c_0 immediately moves the plan execution to the initial context of one of the control automata for the goal tasks t_1 or t_2 according to the current domain state, i.e. whether the property p holds in the current domain state or not. We notice that if **else** part of the **if-else** construction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

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



The context c_0 has two immediate transitions guarded by the 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 context c_0, but the failure transition for t_1 falsifies the compound automaton. We notice that c_0 is marked as a *red block*. It guaranties that the loop is finite.

The control automaton for the **try-catch** task with two **catch** operators is the 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 an immediate transition to each t_i automaton that is guarded by the property p_i. If there exists a domain state s which satisfies more than one **catch** property p_i then the recovery goal is selected nondeterministically. The last transition from c_0 to failure is guarded by the property $\neg(p_1 \vee \ldots \vee p_n)$. It manages the domain states that are not catched by the 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 the 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.

Experimental Evaluation

We implemented the proposed planning approach on top of the MBP planner (Bertoli et al. 2001). In order to test the scalability of the proposed technique, we conducted a set of tests in some experimental domains. All experiments have been executed on a 1.6GHz Intel Centrino machine with 512MB memory and running a Linux operating system. We consider the robot navigation domain defined in Example 1 and evaluate the planning time for different kind of planning goals, i.e. sequential, conditional, cyclic, recovering and reachablity goals.

We tested the performance of the planning algorithm for goal tasks similar to the one on Example 2 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". We encode the reachability goal using the EaGLe operator **DoReach**, therefore the planner generates the plan in both cases using the same planning approach based on the control automata. It allows us to claim that our approach is useful due to the procedural goal language but not due to the especial plan search procedure. The average experimental results are shown on Figure 2.

The experiments showed that for simple domains (less than 6 rooms) planning for a reachability goal is faster, because the memory management and the operations on the control automaton take more time than planning for reachability goal. But in more complex domains our planning technique solves the problem much faster, especially with the systematic usage of policies.

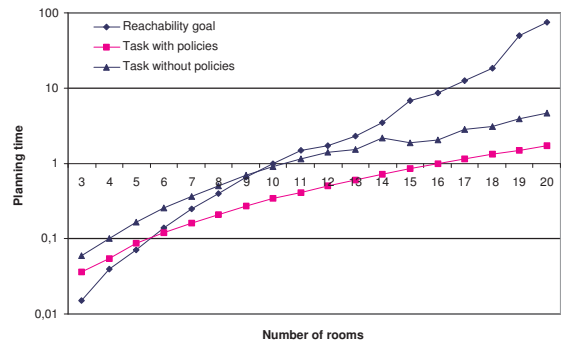


Figure 2: Evaluation results for the robot domain

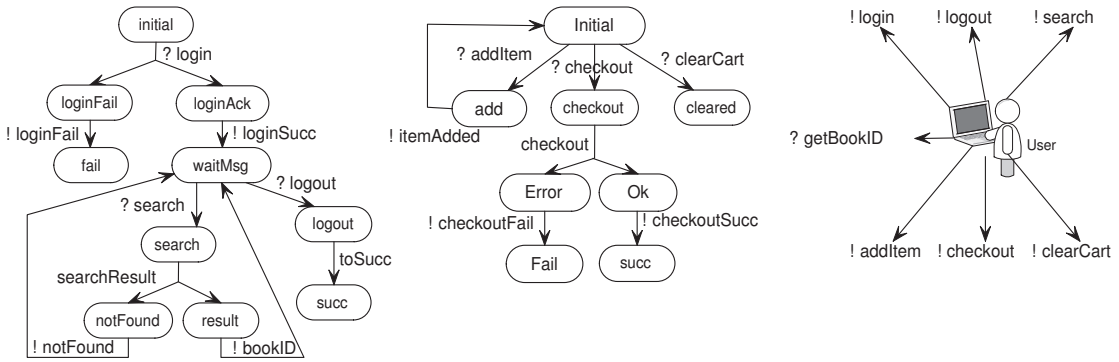
We also tested our approach with the goal that does not contains the last operator "**check**(content[3]=clean)". In this case the robot activities in the room are not linked with the robot position. It means that we can resolve the goal

```
try
   goal TryReach (room[position]=empty)
catch (room[position]=withBomb) do
   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 a plan verification problem which is incomparably easier. For instance, planning for such kind of goals in the domain with 50 rooms takes less than 1 second. This experiment shows that the procedural decomposition of the planning goal and re-usage of the previously generated plans can significantly simplify the plan generation process.

The second experimental domain is inspired by a real application, namely *automatic web service composition* (Pistore, Traverso, and Bertoli 2005; Pistore et al. 2005).

Example 3 *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 3(a,b). BookSearch describes a workflow to find a book in the bookstore. BookCart describes a workflow to add the obtained book to the user cart and, finally, to checkout it. All possible User activities for communications with the bookstore are shown on Figure 3(c). We use a fluent "message" to emulate the message exchange 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*



a) **BookSearch Web Service** b) **BookCart Web Service** c) **Possible User Actions**

Figure 3: Experimental domain for WS composition

action "! bookID" of BookSearch. Due to 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 (STS) and the planning domain is obtained as a product of these STSs. N pairs of BookSeach and BookCart services emulate N bookstores and our goal is 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 the book is bought. In each bookstore we can try to find the book and buy it. If it is impossible due to domain nondeterminism we need to perform some recovering actions to leave the bookstore in a consistent state. Therefore, the planning goal can be written as follows:

```
while ('book is not bought' &&
       '∃ available bookstore') do
 doAction 'choose available bookstore'
 try
  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 Figure 4. As in the robot

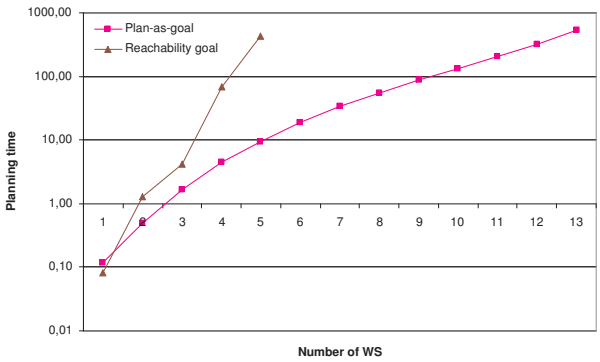


Figure 4: Evaluation results for the WS composition domain

navigation domain we can see that our approach is effective in large domains.

We note that all simple reachability goals used in the experiments can not be significantly improved using EaGLe goal language. It confirms the fact that the expressiveness and performance of the proposed procedural goal language is better in comparison with EaGLe. It is very difficult to compare our approach fairly with CTL goals because CTL can not express **try-catch** goals that are almost obligatory for the real-life problems in nondeterministic domains.

All experiments show that fusing procedural and declarative approaches for the goal definition allows the user to define a planning goal more clearly and detailed. Moreover, such detailed goal can be solved by the planner in a very efficient way in comparison with the more general goal such as a simple reachability goal. It is especially effective in large domains.

Conclusions and Related Work

In this paper, we have 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 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.

Several approaches have been proposed to deal with the problem of planning in nondeterministic domains, where actions are modeled with different possible outcomes, see, e.g., (Rintanen 1999; Cimatti et al. 2003; Hoffmann and Brafman 2005; Kabanza and Thiébaux 2005; Kuter et al. 2005; Kuter and Nau 2004). Some of them deal also with the problem of planning for temporally extended goals (Bacchus and Kabanza 2000; Pistore and Traverso 2001; Dal Lago, Pistore, and Traverso 2002; Kabanza and Thiébaux 2005). However, none of these works can combine procedural and declarative goals like the planning language that is proposed in this paper. This is the case also of the recent work that extends HTN planning to nondeterministic domains (Kuter et al. 2005; Kuter and Nau 2004). In this work, it is possible to plan in a weak, strong, or strong cyclic way for HTN specifications, but the combination of HTNs with declarative temporal specifications like those expressed in EAGLE are not allowed. This is indeed an interesting topic for future investigation, since goals that could combine HTNs with temporal formulas could exploit the integration of HTNs techniques with the techniques for planning based on control automata.

The idea of abstract task specification using procedural constructs is widely used in model-based programming, see, e.g., (Williams et al. 2004). Typically, the model-based program is represented by a control automaton that generates reachability goals according to the program source. Each reachability goal is resolved by reactive planning. In our approach we do not consider the goal as a program to execute, therefore the goal interpretation and planning approach is different.

Finally, our work has some similarities in spirit with the work based on Golog, see, e.g., (Levesque et al. 1997; McIlraith and Fadel 2002; Baier, Fritz, and McIlraith 2007; Claßen et al. 2007). In Golog, sketchy plans with nondeterministic constructs can be written as logic programs. Sequences of actions can be inferred by deduction in situation calculus on the basis of user supplied axioms about preconditions and postconditions. In spite of these similarities, both the approach and the techniques that we propose are very different from Golog.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence* 16(1–2):123–191.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001. MBP: a Model Based Planner. In *IJCAI01*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence* 147(1-2):35–84.

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

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In *AAAI*, 447–454. Edmonton, Alberta, Canada: AAAI-Press/The MIT Press.

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

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *ICAPS*, 71–80.

Kabanza, F., and Thiébaux, S. 2005. Search control in planning for temporally extended goals. In *ICAPS*, 130–139.

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

Kuter, U., and Nau, D. S. 2004. Forward-chaining planning in nondeterministic domains. In *AAAI*, 513–518.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2005. A hierarchical task-network planner based on symbolic model checking. In *ICAPS*, 300–309.

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–83.

McIlraith, S., and Fadel, R. 2002. Planning with complex actions. In *NMR*.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In Nebel, B., ed., *IJCAI-01*, 479–486. Morgan Kaufmann Publisher.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in nondeterministic domains. In *ECP'01, LNAI. Springer Verlag*.

Pistore, M.; Marconi, A.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning at the Knowledge Level. In *IJCAI'05*.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning in Asynchronous Domains. In *ICAPS'05*.

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

Williams, B. C.; Ingham, M. D.; Chung, S.; Elliott, P.; Hofbaur, M.; and Sullivan, G. T. 2004. Model-based programming of fault-aware systems. *AI Mag.* 24(4):61–75.