CENTRO PER LA RICERCA
SCIENTIFICA E TECNOLOGICA

38050 Povo (Trento), Italy
Tel.: +39 0461 314312
Fax: +39 0461 302040
e−mail: prdoc@itc.it − url: http://www.itc.it

# PLANNING WITH A LANGUAGE FOR EXTENDED GOALS

Dal Lago U., Pistore M., Traverso P.

May 2002

Technical Report # 0205−01

# Planning with a Language for Extended Goals

**Ugo Dal Lago** and **Marco Pistore** and **Paolo Traverso**
ITC-IRST
Via Sommarive 18, 38050 Povo (Trento), Italy
{dallago,pistore,traverso}@irst.itc.it

**Abstract**

Planning for extended goals in non-deterministic domains is one of the most significant and challenging planning problems. In spite of the recent results in this field, no work has proposed a language designed specifically for planning. As a consequence, it is still impossible to specify and plan for several classes of goals that are typical of real world applications, like for instance "try to achieve a goal whenever possible", or "if you fail to achieve a goal, recover by trying to achieve something else".

We propose a new goal language that allows for capturing the intended meaning of these goals. We give a semantics to this language that is radically different from the usual semantics for extended goals, e.g., the semantics for LTL or CTL. Finally, we implement an algorithm for planning for extended goals expressed in this language, and experiment with it on a parametric domain.

## Introduction

Several applications (e.g., space and robotics) often require planning in non-deterministic domains and for extended goals: actions are modeled with different outcomes that cannot be predicted at planning time, and goals are not simply states to be reached ("reachability goals"), but also conditions on the whole plan execution paths ("extended goals"). This planning problem is challenging. It requires planning algorithms that take into account the possibly different action outcomes, and that deal in practice with the explosion of the search space. Moreover, goals should take into account non-determinism and express behaviors that should hold for all paths and behaviors that should hold only for some paths.

The work described in (Pistore & Traverso 2001) and (Pistore, Bettin, & Traverso 2001) proposes an approach to this problem, and shows that the approach is theoretically founded (a formal notion of when a plan satisfies an extended goal is provided) and practical (an implementation of the planning algorithm is experimentally evaluated on a parametric domain). In those papers, CTL is used as the language for expressing goals. CTL (Emerson 1990) is a well-known and widely used language for the formal verification of properties by model checking. It provides the ability to express temporal behaviors that take into account the non-determinism of the domain. In CTL, this is done by means of a universal quantifier ($\mathrm{A}$) and of an existential quantifier ($\mathrm{E}$). The former requires a property to hold in all the execution paths, the latter in at least one. However, in many practical cases, we found CTL inadequate for planning, since it cannot express different kinds of goals that are relevant for applications in non-deterministic domains. Consider, for instance, the classical goal of reaching a state that satisfies a given condition $p$. If the goal should be reached in spite of non-determinism ("strong reachability"), we can use the CTL formula $\mathrm{AF}\,p$, intuitively meaning that for all its execution paths ($\mathrm{A}$), the plan will finally (F) reach a state where $p$ holds. As an example, we could strongly require that a mobile robot reaches a given room. However, in many applications, strong reachability cannot be satisfied. In this case, a reasonable requirement for a plan is that *"it should do everything that is possible to achieve a given condition"*. For instance, the robot should actually try – "do its best" – to reach the room. Unfortunately, in CTL it is impossible to express such goal: the weaker condition imposed by the existential path quantifier $\mathrm{EF}\,p$ only requires that the plan has one of the many possible executions that reach a state where $p$ holds. Moreover, if we have plans that actually try to achieve given goals, their executions should raise a failure on the paths in which the goals are not reachable anymore. Goals should be able to express conditions for controlling and recovering from failure. For instance, a reasonable goal for a robot can be "try to reach a room and, in case of failure, do reach a different room". Again, CTL offers the possibility to compose goals with disjunctions, like $\mathrm{EF}\,p \vee \mathrm{AF}\,q$. However, this formula does not capture the idea of failure recovery: there is no preference in which is the main goal to be achieved (e.g., $\mathrm{EF}\,p$), and which is the recovery goal (e.g., $\mathrm{AF}\,q$) to be achieved just in case of failure.

In this paper we define a new goal language designed to express extended goals for planning in non-deterministic domains. The language has been motivated from a set of applications we have been involved in (see, e.g., (Aiello *et al.* 2001)). The language provides basic goals for expressing conditions that the system should *guarantee* to reach or maintain, and conditions that the system should *try* to reach or maintain. Basic goals can be concatenated by taking into account possible failures. For instance, it is possible to specify goals that we should achieve as a reaction to failure, and goals that we should repeat to achieve until a failure occurs.

We extend the planning algorithm described in (Pistore

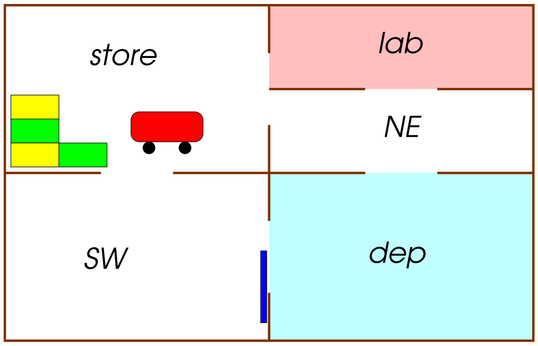



Figure 1: A simple navigation domain

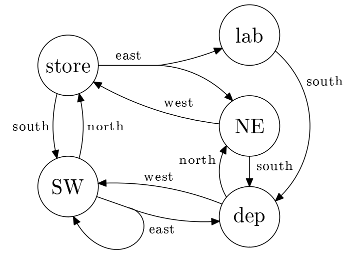



Figure 2: The transition graph of the navigation domain

& Traverso 2001; Pistore, Bettin, & Traverso 2001) to deal with the new language and implement it in the planner MBP (Bertoli *et al.* 2001). We evaluate the algorithm on the domain described in (Pistore, Bettin, & Traverso 2001). Given the new goal language, the algorithm generates better plans than those generated for CTL goals, maintaining a comparable performance.

A full version of this paper, with the details of the planning algorithm and of the experimental evaluation, is available at URL `http://sra.itc.it/tools/mbp`.

## Background on Planning for Extended Goals

We review some basic definitions presented in (Pistore & Traverso 2001; Pistore, Bettin, & Traverso 2001).

**Definition 1** *A (non-deterministic)* planning domain $\mathcal{D}$ *is a tuple* $(\mathcal{B}, \mathcal{Q}, \mathcal{A}, \rightarrow)$*, where* $\mathcal{B}$ *is the finite set of (basic) propositions,* $\mathcal{Q} \subseteq 2^{\mathcal{B}}$ *is the set of states,* $\mathcal{A}$ *is the finite set of actions, and* $\rightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ *is the transition relation, that describes how an action leads from one state to possibly many different states. We write* $q \xrightarrow{a} q'$ *for* $(q, a, q') \in \rightarrow$.

We require that relation $\rightarrow$ is total, i.e., for every $q \in \mathcal{Q}$ there is some $a \in \mathcal{A}$ and $q' \in \mathcal{Q}$ such that $q \xrightarrow{a} q'$. We denote with $\mathrm{Act}(q) \triangleq \{a : \exists q'. \, q \xrightarrow{a} q'\}$ the set of the actions that can be performed in state $q$, and with $\mathrm{Exec}(q, a) \triangleq \{q' : q \xrightarrow{a} q'\}$ the set of the states that can be reached from $q$ performing action $a \in \mathrm{Act}(q)$.

A simple domain is shown in Fig. 1. It consists of a building of five rooms, namely a $store$, a department $dep$, a laboratory $lab$, and two passage rooms $SW$ and $NE$. A robot can move between the rooms. The intended task of the robot is to deliver objects from the store to the department, avoiding the laboratory, that is a dangerous room. For the sake of simplicity, we do not model explicitly the objects, but only the movements of the robot. Between rooms $SW$ and $dep$, there is a door that the robot cannot control. Therefore, an $east$ action from room $SW$ successfully leads to room $dep$ only if the door is open. Another non-deterministic outcome occurs when the robot tries to move $east$ from the $store$: in this case, the robot may end non-deterministically either in room $NE$ or in room $lab$. The transition graph for the domain is represented in Fig. 2. For all states in the domain, we assume to have an action *wait* (not represented in the figure) that leaves the state unchanged.

In order to deal with extended goals, plans allow for specifying actions to be executed that depend not only on the current state of the domain, but also on the *execution context*, i.e., on an "internal state" of the executor, which can take into account, e.g., previous execution steps.

**Definition 2** *A* plan *for a domain* $\mathcal{D}$ *is a tuple* $\pi = (C, c_0, act, ctxt)$*, where* $C$ *is a set of contexts,* $c_0 \in C$ *is the initial context,* $act : \mathcal{Q} \times C \rightharpoonup \mathcal{A}$ *is the action function, and* $ctxt : \mathcal{Q} \times C \times \mathcal{Q} \rightharpoonup C$ *is the context function.*

If we are in state $q$ and in execution context $c$, then $act(q, c)$ returns the action to be executed by the plan, while $ctxt(q, c, q')$ associates to each reached state $q'$ 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. For instance, consider the plan $\pi_1$:

$$
\begin{array}{ll}
act(store, c_0) = south & ctxt(store, c_0, SW) = c_0 \\
act(SW, c_0) = east & ctxt(SW, c_0, dep) = c_0 \\
 & ctxt(SW, c_0, SW) = c_1 \\
act(dep, c_0) = wait & ctxt(dep, c_0, dep) = c_0 \\
act(SW, c_1) = north & ctxt(SW, c_1, store) = c_1 \\
act(store, c_1) = south & ctxt(store, c_1, SW) = c_1
\end{array}
$$

According to this plan, the robot moves south, then it moves east; if $dep$ is reached, it waits there forever; otherwise it continues to move north and south forever.

The executions of a plan can be represented by an *execution structure*, i.e, a Kripke Structure (Emerson 1990) that describes all the possible transitions from $(q, c)$ to $(q', c')$ that can be triggered by executing the plan.

**Definition 3** *The* execution structure *of plan* $\pi$ *in a domain* $\mathcal{D}$ *from state* $q_0$ *is the structure* $K = \langle S, R, L \rangle$*, where:*

- $S = \{(q, c) : act(q, c) \text{ is defined}\}$,
- $R = \{((q, c), (q', c')) \in S \times S : q \xrightarrow{a} q \text{ with } a = act(q, c) \text{ and } c' = ctxt(q, c, q')\}$
- $L(q, c) = \{b : b \in q\}$.

The execution structure $K_{\pi_1}$ of plan $\pi_1$ is shown in Fig. 3.

Extended goals in (Pistore & Traverso 2001; Pistore, Bettin, & Traverso 2001) are expressed with CTL formulas, and the standard semantics for CTL (Emerson 1990) is used to define when a goal $g$ is true in a state $(q, c)$ of the execution structure $K$ (written $(q, c) \models g$).

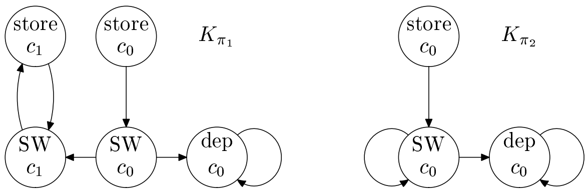



Figure 3: Two examples of execution structures

**Definition 4** *Let $\pi$ be a plan for domain $\mathcal{D}$ and $K$ be the corresponding execution structure. Plan $\pi$ satisfies goal $g$ from initial state $q_0$, written $\pi, q_0 \models g$, if $(q_0, c_0) \models g$. Plan $\pi$ satisfies goal $g$ from the set of initial states $Q_0$ if $\pi, q_0 \models g$ for each $q_0 \in Q_0$.*

For instance, goal $\mathrm{AG}\,\neg lab \wedge \mathrm{EF}\,dep$ (“the robot should never enter the $lab$ and should have a possibility of reaching the $dep$”) is satisfied by plan $\pi_1$ from the initial state $store$.

## The Goal Language

We propose the following language that overcomes some main limitations of CTL as a goal language. Let $\mathcal{B}$ be the set of *basic propositions*. The *propositional formulas* $p \in \mathcal{P}rop$ and the *extended goals* $g \in \mathcal{G}$ over $\mathcal{B}$ are defined as follows:

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

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

We now provide some intuitions and motivations for this language (and illustrate the intended meaning of the constructs on the example in Fig. 2). We often need a plan that is guaranteed to achieve a given goal. This can be done with operators **DoReach** (that specifies a property that should be reached) and **DoMaint** (that specifies a property that should be maintained true). However, in several domains, no plans exist that satisfy these strong goals. This is the case, for instance, for the goal **DoMaint** $\neg lab$ **And DoReach** $dep$, if the robot is initially in $store$. We need therefore to weaken the goal, but at the same time we want to capture its *intentional aspects*, i.e., we require that the plan *“does everything that is possible”* to achieve it. This is the intended meaning of **TryReach** and **TryMaint**. These operators are very different from operators EF and EG in CTL, which require plans that have just a possibility to achieve the goal. In the example, consider the goal **DoMaint** $\neg lab$ **And TryReach** $dep$. According to our intended meaning, this goal is not satisfied by plan $\pi_1$ (see previous section), that just tries one time to go east. Instead, the goal is satisfied by a plan $\pi_2$ that first moves the robot $south$ to room $SW$, and then keeps trying to go $east$, until this action succeeds and the $dep$ is reached. CTL goal $\mathrm{AG}\,\neg lab \wedge \mathrm{EF}\,dep$ is satisfied by both plans $\pi_1$ and $\pi_2$: indeed, the execution structures corresponding to these plans (see Fig. 3) have a path that reaches $dep$. Plan $\pi_1$ is not valid if we specify CTL goal $\mathrm{AG}\,\neg lab \wedge \mathrm{AG}\,\mathrm{EF}\,dep$, that requires that there is always a path that leads to $dep$ (this is a “strong cyclic” reachability goal). Also in this case, however, the intentional aspect is lost; an acceptable plan for the CTL goal (but not for the original goal) is the one that tries one time to go $east$ and, if this move fails, goes back to the $store$ before trying again.

In several applications, goals should specify reactions to failures. For instance, a main goal for a mobile robot might be to continue to deliver objects to a given room, but when this is impossible, the robot should not give up, but recover from failure by delivering the current object to a next room. Consider, for instance, the goal (**TryMaint** $\neg lab$ **Fail DoReach** $store$) **And DoReach** $dep$. In order to satisfy it from the $store$, the robot should first go $east$. If room $lab$ is entered after this move, then goal **TryMaint** $\neg lab$ fails, and the robot has to go back to the $store$ to satisfy recovery goal **DoReach** $store$. This is very different from CTL goal $\mathrm{AG}\,(lab \rightarrow \mathrm{AF}\,store)$ that requires that if the robot ends in the lab it goes to the store. Indeed, this goal does not specify the fact that the $lab$ should be avoided and that going back to the $store$ is only a recovery goal. The CTL goal is satisfied as well by a plan that “intentionally” leads the robot into the $lab$, and then takes it back to the $store$.

Construct **Fail** is used to model recovery from failure. Despite its simplicity, this construct is rather flexible. In combination with the other operators of the language, it allows for representing both failures that can be detected at planning time and failures that occurs at execution time. Consider, for instance, goal **DoReach** $dep$ **Fail DoReach** $store$. Sub-goal **DoReach** $dep$ requires to find a plan that is guaranteed to reach room $dep$. If such a plan exists from a given state, then sub-goal **DoReach** $dep$ cannot fail, and the recovery goal **DoReach** $store$ is never considered. If there is no such plan, then sub-goal **DoReach** $dep$ cannot be satisfied from the given state and the recovery goal **DoReach** $store$ is tried instead. In this case, the failure of the primary goal **DoReach** $dep$ in a given state can be decided *at planning time*. Consider now goal **TryReach** $dep$ **Fail DoReach** $store$. In this case, sub-goal **TryReach** $dep$ requires to find a plan that *tries* to reach room $dep$. During the execution of the plan, a state may be reached from which it is not possible to reach the $dep$. When such a state is reached, goal **TryReach** $dep$ fails and the recovery goal **DoReach** $store$ is considered. That is, in this case, the failure of the primary goal **TryReach** $dep$ is decided *at execution time*.

We remark that the goal language does not allow for arbitrary nesting of temporal operators, e.g., it is not possible to express goals like **DoReach** (**TryMaint** $p$). However, it provides goal **Repeat** $g$, that specifies that sub-goal $g$ should be achieved cyclically, until it fails. In our experience, the **Repeat** construct replaces most of the usages of nesting.

We now provide a formal semantics for the new goal language that captures the desired intended meaning. Due to the features of the new language, the semantics of goals cannot be defined following the approach used for CTL (Emerson 1990), that associates to each formula the set of states that satisfy it. Instead, for each goal $g$, we associate to each state $s$ of the execution structure two sets $\mathcal{S}_g(s)$ and $\mathcal{F}_g(s)$ of finite paths in the execution structure. They represent the paths that lead to a success or to a failure in the achievement of goal $g$ from state $s$. We say that an execution structure satisfies a goal $g$ from state $s_0$ if $\mathcal{F}_g(s_0) = \emptyset$, that is, if no failure path exists for the goal. Let us consider, for instance, the case of goal $g$ = **TryReach** $dep$. Set $\mathcal{S}_g(s)$ contains the paths that lead from $s$ to states where $dep$ holds, while set $\mathcal{F}_g(s)$ contains the paths that lead from $s$ to states where $dep$ is not reachable anymore. In the case of execution structure $K_{\pi_1}$ of Fig. 3, we have $\mathcal{S}_g(store, c_0) = \{((store, c_0), (SW, c_0), (dep, c_0))\}$ and $\mathcal{F}_g(store, c_0) = \{((store, c_0), (SW, c_0), (SW, c_1))\}$. We remark that we do not consider $((store, c_0), (SW, c_0), (dep, c_0), (dep, c_0))$ as a success path, since its prefix $((store, c_0), (SW, c_0), (dep, c_0))$ is already a success path. In the case of execution structure $K_{\pi_2}$ of Fig. 3, $\mathcal{F}_g(store, c_0) = \emptyset$, while $\mathcal{S}_g(store, c_0)$ contains all paths $((store, c_0), (SW, c_0), \ldots, (SW, c_0), (dep, c_0))$; that is, we take into account that a success path may stay in $(SW, c_0)$ for an arbitrarily number of steps before the $dep$ is eventually reached. Paths $((store, c_0), (SW, c_0), \ldots, (SW, c_0))$ are neither success paths nor failure paths: indeed, condition $dep$ is never satisfied along these paths, but they can be extended to success paths. According to this semantics, goal **TryReach** $dep$ is not satisfied by execution structure $K_{\pi_1}$, while it is satisfied by execution structure $K_{\pi_2}$.

Some notations on paths are now in order. We use $\rho$ to denote infinite paths and $\sigma$ to denote finite paths in the execution structure. We represent with $\text{first}(\sigma)$ the first state of path $\sigma$ and with $\text{last}(\sigma)$ the last state of the path. We write $s \in \sigma$ if state $s$ appears in $\sigma$. We represent with $\sigma;\sigma'$ the path obtained by the concatenation of $\sigma$ and $\sigma'$; concatenation is defined only if $\text{last}(\sigma) = \text{first}(\sigma')$. We write $\sigma \leq \sigma'$ if $\sigma$ is a prefix of $\sigma'$, i.e., if there is some path $\sigma''$ such that $\sigma;\sigma'' = \sigma'$. As usual, we write $\sigma < \sigma'$ if $\sigma \leq \sigma'$ and $\sigma \neq \sigma'$. Finally, given a set $\Sigma$ of finite paths, we define the set of minimal paths in $\Sigma$ as follows: $\min(\Sigma) \equiv \{\sigma \in \Sigma : \forall \sigma'. \sigma' < \sigma \implies \sigma' \notin \Sigma\}$.

We now define sets $\mathcal{S}_g(s)$ and $\mathcal{F}_g(s)$. The definition is by induction on the structure of $g$.

**Goal** $g = p$ is satisfied if condition $p$ holds in the current state, while it fails if condition $p$ does not hold. Formally, if $s \models p$, then $\mathcal{S}_g(s) = \{(s)\}$ and $\mathcal{F}_g(s) = \emptyset$. Otherwise, $\mathcal{S}_g(s) = \emptyset$ and $\mathcal{F}_g(s) = \{(s)\}$.

**Goal** $g$ = **TryReach** $p$ has success when a state is reached that satisfies $p$. It fails if condition $p$ cannot be satisfied in any of the future states. Formally, $\mathcal{S}_g(s) = \min\{\sigma : \text{first}(\sigma) = s \wedge \text{last}(\sigma) \models p\}$ and $\mathcal{F}_g(s) = \min\{\sigma : \text{first}(\sigma) = s \wedge \forall s' \in \sigma. s' \not\models p \wedge \forall \sigma' \geq \sigma. \text{last}(\sigma') \not\models p\}$.

**Goal** $g$ = **DoReach** $p$ requires that condition $p$ is eventually achieved despite non-determinism. If this is the case, then the successful paths are those that end in a state satisfying $p$. If there is some possible future computation from a state $s$ along which $p$ is never achieved, then goal $g$ fails immediately in $s$. Formally, if there is some infinite path $\rho$ from $s$ such that $s' \not\models p$ for each $s' \in \rho$, then $\mathcal{S}_g(s) = \emptyset$ and $\mathcal{F}_g(s) = \{(s)\}$. Otherwise, $\mathcal{S}_g(s) = \min\{\sigma : \text{first}(\sigma) = s \wedge \text{last}(\sigma) \models p\}$ and $\mathcal{F}_g(s) = \emptyset$.

**Goal** $g$ = **TryMaint** $p$. Maintainability goals express conditions that should hold forever. No finite success path can thus be associated to maintainability goals. On the other hand, these goals may have failure paths. Goal $g$ = **TryMaint** $p$ fails in all those states where condition $p$ does not hold. Formally, $\mathcal{S}_g(s) = \emptyset$ and $\mathcal{F}_g(s) = \min\{\sigma : \text{first}(\sigma) = s \wedge \text{last}(\sigma) \not\models p\}$.

**Goal** $g$ = **DoMaint** $p$ fails immediately in all those states that do not guarantee that condition $p$ can be maintained forever. Formally, if $s' \models p$ holds for all states $s'$ reachable from $s$, then $\mathcal{F}_g(s) = \emptyset$. Otherwise $\mathcal{F}_g(s) = \{(s)\}$. In both cases, $\mathcal{S}_g(s) = \emptyset$.

**Goal** $g = g_1$ **Then** $g_2$ requires to satisfy sub-goal $g_1$ first and, once $g_1$ succeeds, to satisfy sub-goal $g_2$. Goal $g_1$ **Then** $g_2$ succeeds when also goal $g_2$ succeeds. It fails when either goal $g_1$ or goal $g_2$ fail. Formally, $\mathcal{S}_g(s) = \{\sigma_1;\sigma_2 : \sigma_1 \in \mathcal{S}_{g_1}(s) \wedge \sigma_2 \in \mathcal{S}_{g_2}(\text{last}(\sigma_1))\}$ and $\mathcal{F}_g(s) = \{\sigma_1 : \sigma_1 \in \mathcal{F}_{g_1}(s)\} \cup \{\sigma_1;\sigma_2 : \sigma_1 \in \mathcal{S}_{g_1}(s) \wedge \sigma_2 \in \mathcal{F}_{g_2}(\text{last}(\sigma_1))\}$.

**Goal** $g = g_1$ **Fail** $g_2$ permits to recover from failure. Sub-goal $g_1$ is tried first. If it succeeds, then the whole goal $g_1$ **Fail** $g_2$ succeeds. If sub-goal $g_1$ fails, then $g_2$ is taken into account. Formally, $\mathcal{S}_g(s) = \{\sigma_1 : \sigma_1 \in \mathcal{S}_{g_1}(s)\} \cup \{\sigma_1;\sigma_2 : \sigma_1 \in \mathcal{F}_{g_1}(s) \wedge \sigma_2 \in \mathcal{S}_{g_2}(\text{last}(\sigma_1))\}$ and $\mathcal{F}_g(s) = \{\sigma_1;\sigma_2 : \sigma_1 \in \mathcal{F}_{g_1}(s) \wedge \sigma_2 \in \mathcal{F}_{g_2}(\text{last}(\sigma_1))\}$.

**Goal** $g = g_1$ **And** $g_2$ succeeds if both sub-goals succeed, and fails if one of the sub-goals fails. Formally, $\mathcal{S}_g(s) = \min\{\sigma : \exists \sigma_1 \leq \sigma. \sigma_1 \in \mathcal{S}_{g_1}(s) \wedge \exists \sigma_2 \leq \sigma. \sigma_2 \in \mathcal{S}_{g_2}(s)\}$ and $\mathcal{F}_g(s) = \min(\mathcal{F}_{g_1}(s) \cup \mathcal{F}_{g_2}(s))$.

**Goal** $g$ = **Repeat** $g'$ requires to achieve goal $g$ in a cyclic way. That is, as soon as goal $g'$ has success, a new instance of goal $g'$ is activated. Goal **Repeat** $g'$ fails as soon as one of the instances of goal $g'$ fails. Formally, $\mathcal{S}_g(s) = \emptyset$ and $\mathcal{F}_g(s) = \{\sigma_0;\sigma_1;\ldots;\sigma_n;\sigma' : \sigma' \in \mathcal{F}_{g'}(\text{first}(\sigma')) \wedge \exists \sigma'_i \in \mathcal{S}_{g'}(\text{first}(\sigma'_i)).\ \sigma_i = \sigma'_i; (\text{last}(\sigma'_i), \text{last}(\sigma_i))\}$.

We can now define the notion of a plan that satisfies a goal expressed in the new language.

**Definition 5** *Let $\pi$ be a plan for domain $\mathcal{D}$ and $K$ be the corresponding execution structure. Plan $\pi$ satisfies goal $g \in \mathcal{G}$ from initial state $q_0$, written $\pi, q_0 \models g$, if $\mathcal{F}_g(q_0, c_0) = \emptyset$ for execution structure $K$. Plan $\pi$ satisfies goal $g$ from the set of initial states $Q_0$ if $\pi, q_0 \models g$ for each $q_0 \in Q_0$.*

## Outline of the algorithm

In this section we extend to the new goal language the symbolic planning algorithm defined in (Pistore, Bettin, & Traverso 2001) for CTL. The algorithm is based on control automata. They are used to encode the requirements expressed by the goal, and are then exploited to guide the search of the plan. The states of a control automaton define the different goals that the plan intends to achieve in different instants; they correspond to the contexts of the plan under construction. The transitions of the control automaton define constraints on the behaviors that the states of the domain should exhibit in order to be *compatible* with the control states, i.e., to satisfy the corresponding goals. In this section, we first define a mapping from the new goal language into (an enriched version of) control automata. The mapping we propose departs substantially from the construction defined in (Pistore, Bettin, & Traverso 2001). Then we show how control automata can be used to guide the search of the plan. In the search phase, the algorithms of (Pistore, Bettin, & Traverso 2001) can be re-used with minor changes.

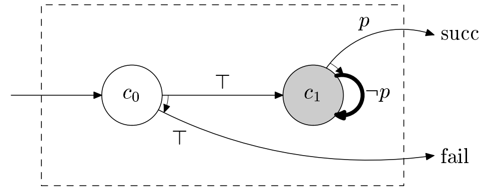
## Construction of the control automaton

**Definition 6** *A* control automaton *is a tuple* $(C, c_0, T, RB)$*, where:*

- $C$ *is the set of* control states*;*
- $c_0 \in C$ *is the* initial control state*;*
- $T(c) = \langle t_1, t_2, \ldots, t_m \rangle$ *is the list of* transitions *for control state* $c$*; each transition* $t_i$ *is either* normal*, in which case* $t_i \in \mathcal{P}rop \times (C \times \{\circ, \bullet\})^*$*; or* immediate*, in which case* $t_i \in \mathcal{P}rop \times (C \cup \{\mathrm{succ}, \mathrm{fail}\})$.
- $RB = \{rb_1, \ldots, rb_n\}$*, with* $rb_i \subseteq C$*, is the set of* red blocks.

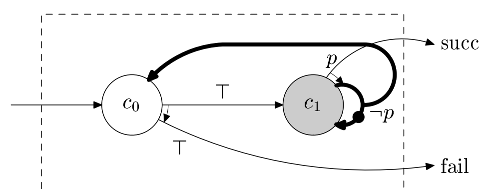
A list of transitions $T(c)$ is associated to each control state $c$: each transition is a possible behavior that a state can satisfy in order to be compatible with the control state $c$. The order of the list represents the preference among these transitions. The transitions of a control automaton are either *normal* or *immediate*. The former transitions correspond to the execution of an action in the plan. The latter ones describe updates in the control state that do not correspond to the execution of an action; they resemble $\epsilon$-transitions of classical automata theory. The *normal transitions* are defined by a condition (that is, a formula in $\mathcal{P}rop$) and by a list of target control states. Each target control state is marked either by a $\circ$ or by a $\bullet$. State $s$ satisfies a normal transition $(p, \langle(c'_1, k'_1), \ldots, (c'_n, k'_n)\rangle)$ (with $k'_i \in \{\circ, \bullet\}$) if it satisfies condition $p$ and if there is some action $a$ from $s$ such that: $(i)$ all the next states reachable from $s$ doing action $a$ are compatible with some of the target control states; and $(ii)$ some next state is compatible with each target state marked as $\bullet$. The *immediate transitions* are defined by a condition and by a target control state. A state satisfies an immediate transition $(p, c')$ if it satisfies condition $p$ and if it is compatible with the target control state $c'$. Special target control states $\mathrm{succ}$ and $\mathrm{fail}$ are used to represent success and failure: all states are compatible with success, while no state is compatible with failure. The *red blocks* of a control automaton represent sets of control states where the execution cannot stay forever. Typically, a red block consists of the set of control states in which the execution is trying to achieve a given condition, as in the case of a reachability goal. If an execution persists inside such a set of control states, then the condition is never reached, which is not acceptable for a reachability goal. In the control automaton, a red block is used to represent the fact that any valid execution should eventually leave these control states.

We now describe the control automata corresponding to the goal language. Rather than providing the formal definition of the control automata, we represent them using a graphical notation. We start with goal **DoReach** $p$:




The control automaton has two control states: $c_0$ (the initial control state) and $c_1$. There are two transitions leaving control state $c_1$. The first one, guarded by condition $p$, is a success transition that corresponds to the cases where $p$ holds in the current domain state. The second transition, guarded by condition $\neg p$, 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. We remark that the second transition is a normal transition since it requires the execution of an action in the plan; the first transition, instead, is immediate. In the diagrams, we distinguish the two kinds of transitions by using thin arrows for the immediate ones and thick arrows for the normal ones. A domain state is compatible with control state $c_1$ only if it satisfies goal **DoReach** $p$, that is, if condition $p$ holds in the current state (first transition from $c_1$), or if goal **DoReach** $p$ holds in all the next states (second transition from $c_1$). According to the semantics of **DoReach** $p$, it is not possible for an execution to stay in state $c_1$ forever, as this corresponds to the case where condition $p$ is never reached. That is, set $\{c_1\}$ is a red block of the control automaton. In the diagrams, states that are in a red block are marked in gray. Control state $c_0$ takes into account that it is not always possible to assure that condition $p$ will be eventually reached, and that if this is not the case, then goal **DoReach** $p$ fails. The precedence order among the two transitions from control state $c_0$, represented by the small circular arrow between them, guarantees that the transition leading to a failure is followed only if it is not possible to satisfy the constraints of control state $c_1$.

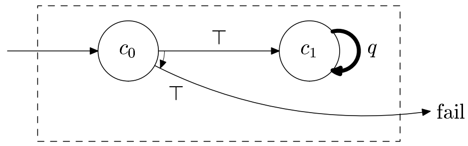
The control automaton for **TryReach** $p$ is the following:




The difference with respect to the control automaton for **DoReach** $p$ is in the transition from $c_1$ guarded by condition $\neg p$. In this case we do not require that goal **TryReach** $p$ holds for *all* the 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

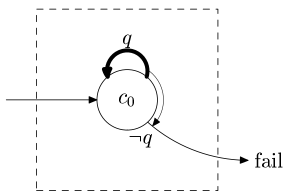
**TryReach** $p$; that is, target $c_1$ of the transition is marked by • in the control automaton. This "non-emptiness" requirement is represented in the diagram with the • on the arrow leading back to $c_1$. The preferred transition from control state $c_0$ is the one that leads to $c_1$. This ensures that the algorithm will try to achieve goal **TryReach** $p$ whenever possible.

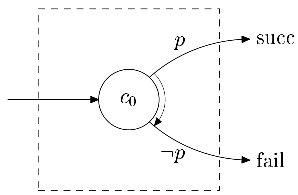
The control automaton for **DoMaint** $q$ is the following:



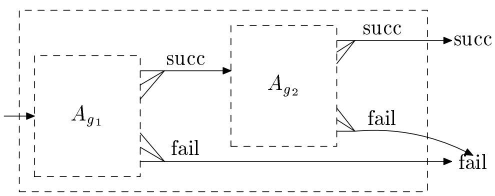

The transition from control state $c_1$ guarantees that a state $s$ is compatible with control state $c_1$ only if $s$ satisfies condition $q$ and all the next states of $s$ are also compatible with $c_1$. Control state $c_1$ is not gray. Indeed, a valid computation that satisfies **DoMaint** $q$ remains in control state $c_1$ forever.

The control automaton for the remaining basic goals **TryMaint** $q$ and $p$ are, respectively:

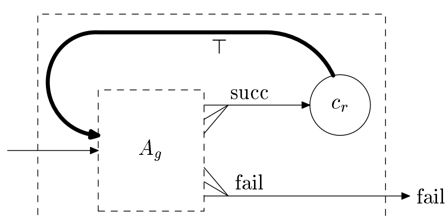



The control automata for the compound goals are defined compositionally, by combining the control automata of their sub-goal. The control automaton for goal $g_1$ **Then** $g_2$ is the following:

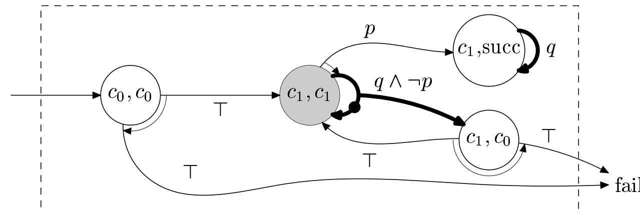



The initial state of the compound automaton coincides with the initial state of automaton $A_{g_1}$, and the transitions that leave the $A_{g_1}$ with success are redirected to the initial state of $A_{g_2}$. The control automaton for goal $g_1$ **Fail** $g_2$ is defined similarly. The difference is that in this case the transitions that leave $A_{g_1}$ with *failure* are redirected to the initial state of $A_{g_2}$. The control automaton for goal **Repeat** $g$ is as follows:




A new control state $c_r$ is added to the automaton. This guarantees that, if goal $g$ has been successfully fulfilled in state $s$, then the achievement of a new instance of goal $g$ starts from the next states, not directly in $s$.

The control automaton for goal $g_1$ **And** $g_2$ has to perform a parallel simulation of the control automata for the two sub-goals, and to accept only those behaviors that are accepted separately by $A_{g_1}$ and by $A_{g_2}$. Technically, this corresponds to take the *synchronous product* $A_{g_1} \otimes A_{g_2}$ of the control automata for the sub-goals. We skip the formal definition of synchronous product for lack of space. It can be found in the full version of this paper. It is simple, but is long and rather technical, since it has to take into account several special cases. Here, we present a simple example of synchronous product. The control automaton for goal **DoMaint** $q$ **And TryReach** $p$ (i.e., the synchronous product of the control automata $A_g$ for goal **DoMaint** $q$ and $A_{g'}$ for **TryReach** $p$) is as follows:




The control states of the product automaton are couples $(c, c')$, where $c$ is a control state of $A_g$ and $c'$ is a control state of $A_{g'}$. The extra control state $(c_1, succ)$ is added to take into account the case where goal **TryReach** $p$ has been successfully fulfilled, but goal **DoMaint** $q$ is still active. The transitions from control states $(c, c')$ are obtained by combining in a suitable way the transitions of $A_g$ from $c$ and the transitions of $A_{g'}$ from $c'$. For instance, in our example, the normal transition with condition $q \wedge \neg p$ from control state $(c_1, c_1)$ corresponds to the combination of the two normal transitions of the control automata for **DoMaint** $q$ and **TryReach** $p$.

## Plan search

Once the control automaton corresponding to a goal has been built, the planning algorithm performs a search in the domain, guided by the strategy encoded in the control automaton. During this search, a set of domain states is associated to each state of the control automaton. Intuitively, these are the states that are compatible with the control state, i.e., for which a plan exists for the goal encoded in the control state. Initially, all the 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. Once a fix-point is reached, the information gathered during the search is exploited in order to extract a plan. We remark that iterative refinement and plan extraction are independent from the goal language. In fact, for them we have been able to re-use the algorithms of (Pistore, Bettin, & Traverso 2001) with minor changes.

In the iterative refinement algorithm, the following conditions are enforced: **(C1)** a domain state $s$ is associated to a control state $c$ only if $s$ can satisfy the behavior described by some transition of $c$; and **(C2)** executions from a given state $s$ are not acceptable if they stay forever inside a red block. In each step of the iterative refinement, either a control state is selected and the corresponding set of domain states is refined according to **(C1)**; or a red block is selected and all the sets of domain states associated to its control states are refined according to **(C2)**. The refinement algorithm terminates when no further refinement step is possible, that is, when a fix-point is reached.

The core of the refinement step resides is function *ctxt-assoc*$_A(c, \gamma)$. It takes as input the control automaton $A = (C, c_0, T, RB)$, a control state $c \in C$, and the current association $\gamma : C \to 2^{\mathcal{Q}}$, and returns the new set of domain states to be associated to $c$. It is defined as follows:

$$\textit{ctxt-assoc}_A(c,\gamma) \triangleq \big\{q \in \mathcal{Q} : \exists t \in T(c).\ q \in \textit{trans-assoc}_A(t,\gamma)\big\}.$$

According to this definition, a state $q$ is compatible with a control state $c$ if it satisfies the conditions of some transition $t$ from that control state. If $t = (p, c')$ is an immediate transition, then:

$$\textit{trans-assoc}_A(t,\gamma) \triangleq \big\{q \in \mathcal{Q} :\ q \models p \ \wedge\ q \in \gamma(c')\big\}.$$

where we assume that $\gamma(\mathrm{fail}) = \emptyset$ and $\gamma(\mathrm{succ}) = \mathcal{Q}$. That is, in the case of an immediate transition, we require that $q$ satisfies property $p$ and that it is compatible with the new control state $c'$ according to the current association $\gamma$. If $t = (p, \langle(c'_1, k'_1), \ldots, (c'_n, k'_n)\rangle)$ is a normal transition, then:

$$\textit{trans-assoc}_A(t,\gamma) \triangleq \big\{q \in \mathcal{Q} :\ q \models p \ \wedge\ \exists a \in \mathrm{Act}(q).\ (q,a) \in \textit{gen-pre-image}\big((\gamma(c'_1),k'_1),\ldots,(\gamma(c'_n),k'_n)\big)\big\}$$

where:

$$\textit{gen-pre-image}\big((Q_1,k_1),\ldots(Q_n,k'_n)\big) \triangleq \big\{(q,a) : \exists Q'_1 \subseteq Q_1 \ldots Q'_k \subseteq Q_k.\ \mathrm{Exec}(q,a) = Q'_1 \cup \cdots \cup Q'_k \ \wedge\ Q'_i \cap Q'_j \text{ if } i \neq j \wedge Q'_i \neq \emptyset \text{ if } k_i = \bullet\big\}.$$

Also in the case of normal transitions, *trans-assoc*$_A$ requires that $q$ satisfies property $p$. Moreover, it requires that there is some action $a$ such that the next states $\mathrm{Exec}(q, a)$ satisfy the following conditions: $(i)$ all the next states are compatible with some of the target control states, according to association $\gamma$; and $(ii)$ some next state is compatible with each target control state marked as $\bullet$. These two conditions are enforced by requiring that the state-action pair $(q, a)$ appears in the *generalized pre-image* of the sets of states $\gamma(c'_i)$ associated by $\gamma$ to the target control states $c'_i$. The generalized pre-image, that is computed by function *gen-pre-image*, can be seen as a combination and generalization of the *strong-pre-image* and *multi-weak-pre-image* used in (Pistore, Bettin, & Traverso 2001).

Function *ctxt-assoc* is used in the refinement steps corresponding to **(C1)** as well as in the refinement steps corresponding to **(C2)**. In the former case, the refinement step simply updates $\gamma(c)$ to the value of *ctxt-assoc*$_A(c, \gamma)$. In the latter case, the refinement should guarantee that any valid execution eventually leaves the control states in the selected red block $rb$. To this purpose, the empty set of domain states is initially associated to the control states in the red block; then, iteratively, one of the control states $c \in rb$ is chosen, and its association $\gamma(c)$ is updated to *ctxt-assoc*$_A(c, \gamma)$; these updates terminate when a fix-point is reached, that is, when $\gamma(c) = \textit{ctxt-assoc}_A(c, \gamma)$ for each $c \in rb$. In this way, a least fix-point is computed, which guarantees that a domain state is associated to a control state in the red block only if there is a plan from that domain state that leaves the red block in a finite number of actions.

Once a stable association $\gamma$ from control states to sets of domain states is built for a control automaton, a plan can be easily obtained. The contexts for the plan correspond to the states of the control automaton. The information necessary to define functions $act$ and $ctxt$ is implicitly computed during the execution of the refinement steps. Indeed, function *trans-assoc* defines the possible actions $a = act(q, c)$ to be executed in the state-context pair $(q, c)$, namely the actions that satisfy the constraints of one of the normal transitions of the control automaton. Moreover, function *gen-pre-image* defines the next execution context $ctxt(q, c, q')$ for any possible next state $q' \in \mathrm{Exec}(q, a)$. The preference order among the transitions associated to each control state is exploited in this phase, in order to guarantee that the extracted plan satisfies the goal in the best possible way. A detailed description of the plan extraction phase appears in the full version of this paper.

## Implementation and experimental evaluation

We have implemented the algorithm in the MBP planner (Bertoli *et al.* 2001). In the plan search phase, we exploit the BDD-based symbolic techniques (Burch *et al.* 1992) provided by MBP, thus allowing for a very efficient exploration of large domains. We have done some preliminary experiments for testing the expressiveness of the new goal language and for comparing the performance of the new algorithm w.r.t. the one presented in (Pistore, Bettin, & Traverso 2001) for CTL goals. For the experiments, we have used the test suite for extended goals made available in the MBP package (see `http://sra.itc.it/tools/mbp`). The tests are based on the robot navigation domain proposed by (Kabanza, Barbeau, & St-Denis 1997). The small domain described in Fig. 1 is a very trivial instance of this class of domains. Some of the tests in the suite correspond to domains with more than $10^8$ states.

The results of the tests (reported in the full version of this paper) are very positive. We have been able to express all the CTL goals in the test suite using the new goal language. Moreover, in spite of the fact that the domain was proposed for CTL goals, the quality of the generated plans is much higher in the case of the new goal language. For instance, in the case of **TryReach** goals, the plans generated by the new algorithm lead the robot to the goal room avoiding the useless moves that are present in the plans generated from the corresponding CTL goals. The new algorithm compares well with the CTL planning algorithm also in terms of performance: despite the more difficult task, in all the experiments the time required by the new algorithm for building the plan is comparable with the time required for CTL goals. In the worst cases, the new algorithm requires about twice the time of the algorithm for CTL, while in other cases the new algorithm behaves slightly better than the old one. The experiments show that the overhead for building a more complex control automaton is irrelevant w.r.t. the symbolic search phase. The slight differences in performances are mainly due to the use of different sequences of BDD operations in the search phase.

## Concluding remarks

We have described a new language for extended goals for planning in non-deterministic domains. This language builds on and overcomes the limits of the work presented in (Pistore & Traverso 2001), where extended goals are expressed in CTL. The main new features w.r.t. CTL are the capability of representing the "intentional" aspects of goals and the possibility of dealing with failure. We have defined a formal semantics for the new language that is very different from the standard CTL semantics. We have extended the planning algorithm described in (Pistore, Bettin, & Traverso 2001) to the new goal language. Our experimental evaluation shows that the new algorithm generates better plans than those generated from CTL goals, maintaining a comparable performance.

As far as we know, this language has never been proposed before, neither in the AI planning literature, nor in the field of automatic synthesis (see, e.g., (Vardi 1995)). The language is somehow related to languages for tactics in theorem proving. In deductive planning, tactics have been used to express plans, rather than extended goals (see (Stephan & Biundo 1993; Spalazzi & Traverso 2000)). The aim of these works is significantly different from ours. Besides (Pistore & Traverso 2001), other works in planning have dealt with the problem of extended goals. Most of these works are based on extensions of LTL, see, e.g., (Kabanza, Barbeau, & St-Denis 1997). Similarly to CTL, LTL cannot capture the goals defined in our language. Several other works only consider the case of deterministic domains (see, e.g., (de Giacomo & Vardi 1999; Bacchus & Kabanza 2000)). In (Bacchus, Boutilier, & Grove 1996) a past version of LTL is used to define temporally extended rewarding functions that can be used to provide "preferences" among temporally extended goals for MDP planning. This work shares with us the motivations for dealing with the problem of generating acceptable plans when it is impossible to satisfy a "most preferred" goal. In the framework of (Bacchus, Boutilier, & Grove 1996), preferences are expressed with "quantitative measures", like utility functions. We propose a complementary approach where the goal language is enriched with logical operators (e.g., **TryReach** and **Fail**) that can be used to specify sort of "qualitative preferences". The comparison of the expressive power and of the practical applicability of the two approaches is an interesting topic for future evaluation. Moreover, the combination of the two approaches is a compelling topic for future research.

Further possible directions for future work are the extension of the language with arbitrary nesting of temporal operators (in the style of CTL or LTL), an extension to planning under partial observability (Bertoli *et al.* 2002) and to adversarial planning (Jensen, Veloso, & Bowling 2001). More important, we plan to evaluate in depth the proposed approach in the real applications that originally motivated our work.

## References

Aiello, L. C.; Cesta, A.; Giunchiglia, E.; Pistore, M.; and Traverso, P. 2001. Planning and verification techniques for the high level programming and monitoring of autonomous robotic devices. In *Proceedings of the ESA Workshop on On Board Autonoy*. ESA Press.

Bacchus, F., and Kabanza, F. 2000. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bacchus, F.; Boutilier, C.; and Grove, A. 1996. Rewarding Behaviors. In *Proc. of AAAI'96*. AAAI-Press.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001. MBP: a Model Based Planner. In *Proc. of IJCAI'01 workshop on Planning under Uncertainty and Incomplete Information*.

Bertoli, P.; Cimatti, A.; Pistore, M.; and Traverso., P. 2002. Plan validation for extended goals under partial observability (preliminary report). In *Proc. of the AIPS'02 Workshop on Planning via Model Checking*.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic Model Checking: $10^{20}$ States and Beyond. *Information and Computation* 98(2):142–170.

de Giacomo, G., and Vardi, M. 1999. Automata-theoretic approach to planning with temporally extended goals. In *Proc. of ECP'99.*

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.

Jensen, R. M.; Veloso, M. M.; and Bowling, M. H. 2001. OBDD-based optimistic and strong cyclic adversarial planning. In *Proc. of ECP'01.*

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

Pistore, M., and Traverso, P. 2001. Planning as Model Checking for Extended Goals in Non-deterministic Domains. In *Proc. of IJCAI'01.* AAAI Press.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Proc. of ECP'01.*

Spalazzi, L., and Traverso, P. 2000. A Dynamic Logic for Acting, Sensing and Planning. *Journal of Logic and Computation* 10(6):787–821.

Stephan, W., and Biundo, S. 1993. A New Logical Framework for Deductive Planning. In *Proc. of IJCAI'93*.

Vardi, M. Y. 1995. An automata-theoretic approach to fair realizability and synthesis. In *Proc. of CAV'95.*