# Learning in Planning with Temporally Extended Goals and Uncontrollable Events

**André A. Ciré**[1] and **Adi Botea**[2]

**Abstract.** Recent contributions to advancing planning from the classical model to more realistic problems include using temporal logic such as LTL to express desired properties of a solution plan. This paper introduces a planning model that combines temporally extended goals and uncontrollable events. The planning task is to reach a state such that all event sequences generated from that state satisfy the problem's temporally extended goal. A real-life application that motivates this work is to use planning to configure a system in such a way that its subsequent, non-deterministic internal evolution (nominal behavior) is guaranteed to satisfy a condition expressed in temporal logic.

A solving architecture is presented that combines planning, model checking and learning. An online learning process incrementally discovers information about the problem instance at hand. The learned information is useful both to guide the search in planning and to safely avoid unnecessary calls to the model checking module. A detailed experimental analysis of the approach presented in this paper is included. The new method for online learning is shown to greatly improve the system performance.

## 1 Introduction

Recent years have seen an increased interest in advancing planning from the classical model to extensions such as using temporal logic to express desired features of a correct plan. Search in a classical planning problem can be guided with control rules expressed in temporal logic [1]. The international planning competition IPC-5 [6] has introduced hard and soft constraints, expressed in temporal logic, that finite plans should satisfy. Computing cyclic solutions to problems with temporally extended goals is presented in [10].

Previous contributions to planning such as the above ones apply temporal logic reasoning along a (candidate) solution plan that is either a finite or a cyclic sequence of actions. In contrast, this paper addresses a problem where temporal logic is applied to the future behavior of a system after a goal state is reached. Specifically, the temporal goal of a problem must be satisfied by *all* sequences of events that originate in a goal state. Events are transitions in the problem state space that are not under the control of the planning agent.

A real-life application that motivates this research is automated configuration of a composite system such as a power grid or a network of water pipes. A composite system is a collection of interacting components. Assume it has a *nominal behavior*, a non-deterministic evolution in the state space where all transitions are uncontrollable events. Even though planning cannot control the events directly, it can impact the nominal behavior by configuring elements of the sys-

tem structure such as the connections between components. Configuring the system in a specific way doesn't necessarily imply that the subsequent nominal behaviour is fully determined. Generally, many event trajectories can originate from a given configuration. The planning task is to configure the system in such a way that its subsequent nominal behavior satisfies the goal condition on every possible event sequence. The configuration step is useful in a number of scenarios such as the initial configuration of a system, a reconfiguration to recover from a failure, a reconfiguration to grow or reduce the size of a system, and a reconfiguration to adapt to a new goal condition. As soon as a solution is found, the planning agent interferes no longer with the system unless a reconfiguration process becomes necessary at some point in the future.

**Contributions.** This paper introduces a new planning model that combines temporally extended goals and uncontrollable events. A solving approach is presented that incrementally learns new information about a problem instance and uses it to improve the performance. The architecture contains a planning component, a model checking component and an online learning component. Planning explores the problem space where transitions are actions and enumerates candidate goal states. A model checking round tests if all event sequences that originate in a candidate goal state satisfy the temporally extended goal. If the test succeeds, a solution has been found. Otherwise, at least one event sequence exists for which the goal formula does not hold. The learning step analyzes such event sequences. New information is extracted, which will be used to both guide the planning and avoid unnecessary model checking rounds.

The performance of a system that implements the ideas presented in this paper is analyzed empirically in detail. The new method for incrementally learning information about a problem instance is shown to greatly improve both the planning effort and the total number of model checking rounds.

## 2 Related Work

Planning systems such as TLPLAN [1] and TALPLANNER [13] are capable of handling a large problem space by using search control rules formulated in temporal logic. MIPS [16], SGPLAN [9] and HPLAN-P [2] are examples of systems that can handle hard and soft constraints (*preferences*) related to a planning goal. This research direction was mainly encouraged by a track added to the 2006 International Planning Competition (IPC-5), in conjunction with PDDL3 [6]. A method able to generate cyclic plans that satisfy a temporally extended goal can be found in [10]. In path planning, temporal logic can encode constraints that a trajectory computed for a mobile unit (e.g., robot) should satisfy [5]. As in previous work

---

such as [7, 10, 16], we convert LTL formulas into Büchi automata. Two major features that distinguish our work from all contributions mentioned earlier are: (1) our system is capable of learning from trajectories where an extended goal does not hold; and (2) we apply our ideas to a new planning problem, where a deterministic planning component is followed by a non-deterministic evolution generated with uncontrollable events. In particular, we reason about LTL goals in the presence of events, whereas the IPC-5 domains with extended goals and preferences are deterministic.

In reactive planning, actions are executed to respond to event occurrences. Reactive planning in problems with extended goals expressed in Metric Temporal Logic (MTL) is the topic of [3, 4]. There is an important distinction between the problem that we address and fields such as reactive planning and controller synthesis. In the latter cases no goal state is defined, whereas we need to reach a goal state where the planning (configuration) is completed and the subsequent system evolution (nominal behaviour) respects the temporal goal.

Generating a control strategy consistent with an LTL formula in a non-deterministic environment is the topic in [12]. The value of this contribution seems to be more theoretical. It provides a translation of the original problem into an LTL game but indicates no heuristics or other enhancements that will be necessary to scale up the performance of a solver. It reports neither experiments nor an actual implementation of the theoretical ideas.

A high-level theme that our learning approach shares with explanation based learning (EBL) is learning from counter examples. Our work differs significantly from previous work on EBL in the planning problem addressed and in the ways that new information is extracted and subsequently used. E.g., the topic in [11] is learning from Graphplan dead-ends in classical planning whilst we focus on learning from bad event sequences in planning with temporal goals and uncontrollable events. Model-based self-configuration, a problem related to our work, is addressed in [17]. That work does not consider temporally extended goals. It can be seen as a form of EBL, since it attempts to make a search more informed as more conditions conflicting with goal states are discovered.

## 3  Problem Definition and Background

The planning model addressed in this work is a structure $\langle S, s_0, \varphi, \gamma, A, E \rangle$ with $S$ a finite state space, $s_0 \in S$ an initial state, and $\varphi$ a temporal logic formula that describes the goal. The function $\gamma : S \times (A \cup E) \to S$ models deterministic transitions in the state space. The transitions are partitioned into a set of actions $A$ (i.e., transitions under the control of the planner), and a set of uncontrollable events $E$ that define the nominal behavior of a system. The search space that has the initial problem state as a root node and uses only actions as transitions is called the problem *planning space*. The space that is rooted in a given state $s$ and uses only events for transitions is called the *event space* of state $s$. The state space associated with a problem is defined using a fixed collection of boolean variables called atoms. Each state is a complete assignment to the atoms defined for that problem. Equivalently, a state $s$ can be defined as the set of all atoms that are true in $s$ (closed world assumption).

Following the STRIPS representation, each action (event) $a$ has a set of preconditions $\text{pre}(a)$, a set of positive effects $\text{add}(a)$ and a set of negative effects $\text{del}(a)$. An action (or event) $a$ is *applicable* in a state $s$ if $s \models \text{pre}(a)$. In such a case, $\gamma(s, a) = (s \setminus \text{del}(a)) \cup \text{add}(a)$. Otherwise, $\gamma(a, s)$ is undefined. A sequence of actions (events) $a_1, a_2, \ldots, a_k$, is applicable in a state $s$ if $a_1$ is applicable in $s$, $a_2$ is applicable in $\gamma(s, a_1)$ and so on. For a sequence of actions

(events) $\pi = a_1, \ldots, a_k$ that is applicable in a state, the precondition of the entire sequence $\text{pre}(\pi)$ is the union of all atoms $p$ such that $(\exists i \in \{1 \ldots k\}) : (p \in \text{pre}(a_i) \land (\forall j < i)p \notin \text{add}(a_j))$.

The planning task is to find a finite sequence of actions that can be applied in $s_0$ and that reaches a goal state. A state $s \in S$ is a goal if every event sequence applicable in $s$ satisfies the temporal goal $\varphi$. A sequence that does not satisfy $\varphi$ is called a *bad event sequence*.

## 4  Solving Approach

The architecture outlined in Algorithm 1 contains three main modules. *Planning* explores the planning space and enumerates candidate goal states. *Model checking* explores the event space of a candidate goal state $s$ to check if it satisfies the temporally extended goal of the problem $\varphi$. If the test returns a positive answer, a solution has been found. Otherwise, the *online learning* component attempts to extract a sufficient condition that explains the negative result of the most recent model checking round.

The system incrementally learns information about a problem instance that is used to speed up the solving process. The learned information $I$ is represented as an atemporal boolean formula. A state $s$ with the property $s \models I$ is guaranteed *not* to satisfy the goal formula $\varphi$. The boolean formula $I$ is used in two parts of the algorithm, each with a great contribution to the system performance. Firstly, no model checking rounds need to be performed in states $s$ with $s \models I$. Secondly, $\neg I$ can be used as a reachability goal in the planning component, allowing the computation of relaxed plans that steer the search away from states that are guaranteed not to be goals. As a problem definition contains no explicit reachability goals, no other information besides $\neg I$ is used as a goal when building relaxed plans.

Standard algorithms that compute relaxed plans such as the one implemented in the FF planning system [8] work only with conjunctive reachability goals. As in Rintanen's work [15], FF's method is extended to handle goals such as $\neg I$, which can be an arbitrary boolean formula. In general, a relaxed plan could be used to compute a heuristic distance from a current state to a goal state, and to partition the successors of a node into *helpful* nodes (i.e., nodes obtained from applicable actions that are also part of the parent's relaxed plan) and *rescue* nodes (all other valid successors). In this paper, two open queues are used, one for helpful and another for rescue nodes. A rescue node is expanded only when the helpful open queue is empty. No heuristic values are associated with nodes. The reason is that, in this problem, the reachability goal $\neg I$ varies in time. Nodes evaluated early might have better heuristic values just because these were computed when the reachability goal was more relaxed.

When $\neg I$ is used as a reachability goal in planning, the lines 6 and 7 in Algorithm 1 are redundant, since $s_g \models \neg I$ holds for every candidate goal state $s_g$. The lines are added to the pseudocode to emphasize more clearly that model checking is triggered only for a small fraction of the states visited in planning.

The next discussion assumes that Linear Temporal Logic (LTL) goals are used. Model checking is implemented as a breadth-first search in order to discover bad event sequences of minimal length. Shorter bad event sequences can allow to learn information that has fewer conjunctive conditions and hence is more generally applicable. See details about learning later in this section. For the sake of clarity, assume that each application of an event in model checking search is performed together with both a normal (usual) progression of $\varphi$ and a progression in the Büchi automaton corresponding to $\varphi$. Büchi progression is a standard approach also adopted, for example in [10]. Other model checking methods (e.g., SAT based [14]) can

**Algorithm 1** Architecture overview.

1: $I \leftarrow$ false {initialize learned info}
2: **while** true **do**
3:    $(s_g, \pi) \leftarrow$ SearchForNextCandidateGoalState() {planning; $\pi$ is the action sequence from $s_0$ to $s_g$}
4:    **if** no state $s_g$ is found **then**
5:       return no solution
6:    **if** $s_g \models I$ **then**
7:       continue {no need for a costly model checking round}
8:    ModelChecking($s_g$) {run a model checking round}
9:    **if** model checking succeeds **then**
10:      return $\pi$
11:    **else**
12:      $I \leftarrow I \vee$ ExtractInfo() {learning}

---

be used but the actual choice is not a major point of this research. As explained in this section and demonstrated empirically in the next section, we improve the model checking component of the algorithm by reducing dramatically the *total number* of model checking rounds, not the effort spent in one individual round.

In the model checking component, the event sequences that originate in a candidate goal state $s_g$ are split into four categories, one corresponding to paths that satisfy $\varphi$ and three corresponding to bad event sequences. Bad event sequences are: *L-paths*, sequences that end with a leaf node (i.e., a node where no events can be applied) before the normal progression reduces $\varphi$ to either true or false; *F-paths*, sequences along which the normal progression reduces $\varphi$ to false; and *C-paths*, where a cycle is created and $\varphi$ is never satisfied. As soon as one bad event sequence is discovered, the corresponding round of model checking returns. If desired, the procedure could attempt to discover several bad event sequences, allowing to learn more information from one round.

The rest of this section focuses on the learning method. This is triggered each time when model checking has discovered an event sequence $\pi_e$ that is either an F-path or a C-path. No information is extracted from L-paths. Information extracted from an L-path might be too specific to $s_g$, since it would have to explain why none out of potentially many events is applicable in the leaf node.

The information extraction aims at detecting a boolean formula $c$ such that $s_g \models c$ and $c$ is sufficient to explain the failure of $\varphi$ along the sequence $\pi_e$. More specifically, $c$ should imply *both* the following conditions: (1) $\pi_e$ is applicable in $s_g$; and (2) $\varphi$ does not hold along the sequence $\pi_e$.

As indicated in Algorithm 2, the formula $c$ is initialized to $\text{pre}(\pi_e)$ to ensure that $c$ implies condition (1). To imply condition (2), $c$ is extended with zero or more conjunctive literals $l$. It is desirable to minimize the number of added literals, as a smaller formula $c$ is more generally applicable and thus more model checking rounds could be avoided in the future.

To compute a set of literals to be added to $c$, a variation of progression called *event-specific progression* is introduced. Consider a state $s_i$ obtained after applying the first $i \geq 1$ steps of $\pi_e$. The event-specific progression to $s_i$ from the previous step is equivalent to the normal progression, except that it postpones the instantiation of certain atoms, as explained next.

The normal progression can be defined recursively starting from atoms and moving to more and more complicated formulas. For the complete set of rules, see for example [1]. Only the case of atomic formulas needs to be discussed here. At the atomic level,

$\text{prog}(p, s_i) =$ true if $s_i \models p$ and $\text{prog}(p, s_i) =$ false if $s_i \models \neg p$. In other words, all occurrences of atoms in the progressed formula that are not inside a temporal operator are replaced by their actual truth values in the corresponding state.

The event-specific formula progression applies different rules at the atomic level. For each atom $p$ in the initial problem definition, define a new variable $p_0$. Define a set of atoms $Z_i$ as $\text{pre}(\pi_e) \cup \text{eff}(e_1) \cup \text{del}(e_1) \cup \cdots \cup \text{eff}(e_i) \cup \text{del}(e_i)$. Being independent from the first $i$ steps of $\pi_e$, atoms $q \notin Z_i$ preserve their value all the way from $s_g$ to $s_i$. For an atomic formula $p$, the event-specific progression is defined as $\text{eprog}(p, s_i) = p_0$ if $p \notin Z_i$, and $\text{eprog}(p, s_i) = \text{prog}(p, s_i) \in \{\text{true}, \text{false}\}$ if $p \in Z_i$. The progression rules for more complicated, non-atomic formulas are the same as in normal progression. Usual simplifications such as true $\vee \alpha =$ true are useful to eliminate irrelevant occurrences of new variables $p_0$ that might exist in $\alpha$.

Event-specific progression of $\varphi$ along $\pi_e$ is performed step-by-step for $t$ times, the same number of steps that normal progression was performed before detecting that $\pi_e$ was a bad event sequence. The resulting formula is denoted by $\text{eprog}(\varphi, \pi_e, t)$. Consider that $P$ is the set of all new boolean variables $p_0$ added during event-specific progression. Each element $p_0 \in P$ generates one literal to be added to $c$ as a new conjunction. If $p$ is true in $s_g$, then $p$ is added to $c$. Otherwise, $\neg p$ is the newly created literal.

It can be shown that the condition $c$ computed as before implies both conditions (1) and (2). Implying condition (1) is obvious from the way $c$ is initialized. A formal proof for condition (2) is skipped to save space. The intuition is that the only atoms that could possibly impact the normal formula progression of $\varphi$ along $\pi_e$ are those determined by $\text{pre}(\pi_e)$ (i.e., atoms in $Z_t$) and atoms $p$ with $p_0 \in P$. The condition $c$ is the assignment in $s_g$ of the atoms in $\text{pre}(\pi_e) \cup \{p | p_0 \in P\}$.

Before creating the literals to be added to $c$, $P$ can be reduced with a greedy procedure that is linear in the size of $P$. The correctness of the extracted information $c$ is preserved in the sense that it still implies conditions (1) and (2). A formula $\beta$ is initialized to $\text{eprog}(\pi_e, \varphi, t)$. The procedure iteratively selects one variable $p_0$ from $P$ and instantiates it in $\beta$ with the value of $p$ in $s_g$. This is repeated until $\beta$ becomes equivalent to $\text{prog}(s_g, \pi_e, \varphi, t)$, the formula obtained by normal progression from $s_g$ along $\pi_e$ for $t$ steps. The variables in $P$ that were not instantiated in this loop can safely be skipped when the literals are generated. The condition on line 7 of Algorithm 2 is easy to check for F-paths, since $\text{prog}(s_g, \pi_e, \varphi, t) =$ false. The implemented system skips the greedy reduction of $P$ for C-paths. It is possible to address this, but the experiments reported next did not indicate a performance bottleneck caused by this choice.

As a simple example, if $\text{eprog}(\varphi, \pi_e, t)$ is false, then no additional information is added to $c$ besides the existing part $\text{pre}(\pi_e)$. In such a case, regardless of the values of other variables in $s_g$, the preconditions and the effects of the event sequence alone are enough to progress $\varphi$ to false.

## 5 Experimental Results

This first part of this section introduces a new benchmark domain. Our experiments are described next. The last part of the section contains the results and their analysis.

**Benchmark and Setup.** Among the many available planning benchmarks, we are not aware of the existence of an encoding that is suitable to the model presented in Section 3, which includes

**Algorithm 2** Learning step in pseudocode.
```
 1:  c ← pre(π_e)
 2:  P ← all new variables p_0 in eprog(π_e, φ, t)
 3:  if perform greedy reduction of P (optional) then
 4:      β ← eprog(π_e, φ, t)
 5:      P_N ← P
 6:      P ← ∅
 7:      while not (β ≡ prog(s_g, π_e, φ, t)) do
 8:          select p_0 ∈ P_N
 9:          instantiate p_0 in β with p's value in s_g
10:          remove p_0 from P_N and add it to P
11:  for each p_0 ∈ P do
12:      l ← (s_g ⊨ p)?(p) : (¬p)
13:      c ← c ∧ l
14:  return c
```

a deterministic planning stage (configuration) followed by a non-deterministic evolution in the event space (nominal behaviour). A new domain has been designed to carry out the experimental evaluation presented in this section. Because of lack of space, only a brief description is included here. The website `http://abotea.rsise.anu.edu.au/factory-benchmark/` contains a detailed presentation and the source code of a problem generator.

Each problem instance contains a collection of components split into two categories: machines and repositories. At most two repositories can be connected to a machine at a time. A repository cannot simultaneously be connected to more than one machine. Each repository stores raw material of a certain type and can transfer batches of it to a connected machine. A machine can combine two types of raw material to generate a final product.

Planning actions consist of both changing connections between repositories and machines, and component-specific operations such as cleaning a machine. The nominal behavior of a system includes transferring raw products from a repository to a machine, and creating final products from combinations of raw materials. Furthermore, certain combinations of raw products can break a machine that is not clean. In experiments, a temporally extended goal, expressed in LTL, is a conjunction of conditions such as never break a machine and eventually generate certain products.

The code is implemented in Java 1.6. Büchi automata are built using the LTL2BA package, available at `http://www-i2.informatik.rwth-aachen.de/Research/RV/ltl2ba4j/index.html`. The experiments are carried out on a 3.4 Ghz machine, with 1.8 GB allocated to the heap memory and 1.8 GB assigned to the stack memory. The time limit is 15 minutes per problem. We are not aware of any existing system designed for the problem addressed in this paper. In the current experiments, the new solver is compared against a basic version where the learning component is switched off.

A set of 350 problem instances is created as follows. The number of repositories $r$ is fixed to 4 and the number of machines $m$ varies from 4 to 10. For each combination $(r, m)$, 50 problems are generated. The LTL goal formulae range in size from 5 to 15 conjunctive conditions. The parameters $r$ and $m$ are chosen in such a way that the problems gradually scale up until the basic solver reaches its limits within the given time and memory constraints. The problem collection contains both instances with solutions and instances that can be proven unsolvable within the allocated resource limits. The latter category is useful to evaluate the impact of learning on reducing the number of model checking rounds. When no goal state exists,

both system versions have to visit all states in the planning space and the difference in the overall performance is mostly explained by the number of model checking rounds.

**Results.** Figure 1 shows the total running time for instances that are proven unsolvable. Each data point in a curve corresponds to one problem instance. The problems are ordered to obtain a monotonically increasing curve for the basic solver. Learning improves the number of model checking rounds. As explained before, the number of nodes in planning search is not affected in such problems. Processing one node in informed planning (i.e., in the system with learning enabled) is more expensive, since a relaxed plan has to be computed. The overall improvement achieved by learning in this subset appears to be almost constant across the problem range.

In instances where a solution is found (Figure 2), learning improves not only the number of model checking rounds but also the number of nodes expanded in planning. As compared to Figure 1, the speed-up factor increases as the problems gets larger. The largest improvement in this set reaches two orders of magnitude.

Given a problem instance, assume that $(P, M, L)$ tells the percentage that each system module (i.e., planning, model checking, learning) contributes to the total running time. $(P(m), M(m), L(m))$ is the average over the problems with $m$ machines. When $m$ varies from 4 to 10, $L(m)$ is stable around a value of 3 to 4%. $P(m)$ slightly increases from 70% to 80%. When learning is switched off, the only modules that contribute to the total running time are planning and model checking. The average weight of the planning time slightly increases from 55% when $m = 4$ to 60% when $m = 10$.
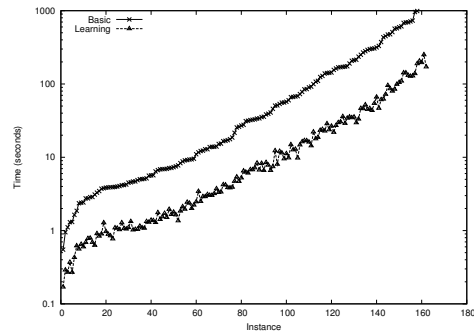


**Figure 1.** Time for instances with no solution. Note the logarithmic scale.

Learning keeps the number of model checking rounds to very small values, whereas the basic system faces an exponential growth as problems increase in difficulty. Figure 3 illustrates this for problems with solutions. The situation is very similar for problems with no solution. The corresponding chart is skipped to save space.

When learning is switched off, planning search is equivalent to breadth-first search, which is guaranteed to find solutions of optimal length. Figure 4 presents the quality of solutions computed by the system with learning enabled. The problems with solution solved by both systems are included in this summary. The sub-optimality of a solution is measured as $\frac{l-o}{o} \times 100$, where $l$ is the actual length and $o$ is the optimal length found with breadth-first search. In Figure 4, each bar counts how many problems fit into the corresponding sub-optimality range. The data indicate that a majority of the solutions found by the learning system are optimal.
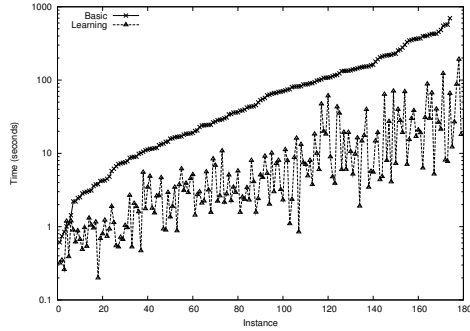
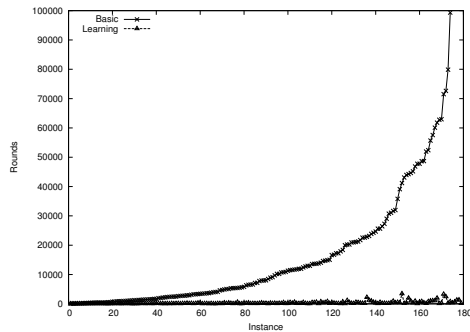**Figure 2.** Time for instances with solutions on a logarithmic scale.



**Figure 3.** Model checking rounds for instances with solution.

## 6 Conclusion and Future Work

Advancing recent contributions that extend classical planning with temporal logic, this paper focuses on a planning model that combines temporally extended goals with uncontrollable events. The model is a generic encoding of a real-life application where a system should automatically be configured such that its future nominal behavior respects a given condition expressed in temporal logic.
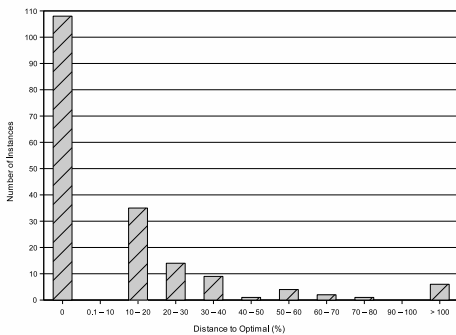


**Figure 4.** Solution quality when learning is used.

A solving architecture that combines elements of planning, model checking and learning is presented and analyzed in detail. An on-line learning procedure builds up information that is used both as a reachability goal in planning search and as a condition to safely skip unnecessary model checking rounds. In experiments, the incrementally learned information has a great contribution to speeding up the solving process.

Future work includes integrating the planning method presented in this paper with monitoring and diagnosis algorithms. The latter monitor a system to decide whether the nominal behavior is the desired one. When faults are detected, the planning method changes the system into a correct configuration.

## 7 Acknowledgment

## REFERENCES

[1] F. Bacchus and F. Kabanza, 'Using Temporal Logics to Express Search Control Knowledge for Planning', *Artificial Intelligence*, **16**, 123–191, (2000).

[2] J. Baier, F. Bacchus, and S. McIlraith, 'A Heuristic Search Approach to Planning with Temporally Extended Preferences', in *Proceedings of IJCAI-07*, pp. 1808–1815, (2007).

[3] M. Barbeau, F. Kabanza, and R. St-Denis, 'Synthesizing Plant Controllers Using Real-Time Goals', in *IJCAI-95*, pp. 791–798, (1995).

[4] M. Barbeau, F. Kabanza, and R. St-Denis, 'A Method for the Synthesis of Controllers to Handle Safety, Liveness, and Real-Time Constraints', *IEEE Transactions on Automatic Control*, **43**(11), 1453–1559, (1998).

[5] G.E. Fainekos, H. Kress-Gazit, and G.J. Pappas, 'Hybrid Controllers for Path Planning: A Temporal Logic Approach', *Decision and Control, and European Control Conference CDC-ECC-05*, 4885–4890, (2005).

[6] A. Gerevini and D. Long, 'Plan Constraints and Preferences for PDDL3', Technical report, University of Brescia, (2005).

[7] G. De Giacomo and M. Y. Vardi, 'Automata-Theoretic Approach to Planning for Temporally Extended Goals', in *Proceedings of ECP-99*, pp. 226–238, (1999).

[8] J. Hoffmann and B. Nebel, 'The FF Planning System: Fast Plan Generation Through Heuristic Search', *JAIR*, **14**, 253–302, (2001).

[9] C. W. Hsu, B. W. Wah, R. Huang, and Y. X. Chen, 'Handling Soft Constraints and Preferences in SGPlan', in *ICAPS Workshop on Preferences and Soft Constraints in Planning*, pp. 54–57, (2006).

[10] F. Kabanza and S. Thiébaux, 'Search Control in Planning for Temporally Extended Goals', in *Proceedings of ICAPS-05*, pp. 130–139, (2005).

[11] S. Kambhampati, 'Improving Graphplan's Search with EBL and DDB Techniques', in *Proceedings of IJCAI*, pp. 982–987, (1999).

[12] M. Kloetzer and C. Belta, 'Managing non-determinism in symbolic robot motion planning and control', in *Robotics and Automation-07*, pp. 3110–3115, (2007).

[13] J. Kvarnström and M. Magnusson, 'TALplanner in IPC-2002: Extensions and Control Rules', *JAIR*, **20**, 343–377, (2002).

[14] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, 'Simple Bounded LTL Model Checking', in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'2004)*, pp. 186–200, (2004).

[15] J. Rintanen, 'Unified Definition of Heuristics for Classical Planning', in *Proceedings ECAI-06*, pp. 600–604, (2006).

[16] S. Jabbar S. Edelkamp and M. Nazih, 'Large-Scale Optimal PDDL3 Planning with MIPS-XXL', in *Proceedings of the International Planning Competition IPC-05*, (2006).

[17] B. C. Williams and P. P. Nayak, 'A Model-based Approach to Reactive Self-Configuring Systems', in *Proceedings AAAI-96*, pp. 971–978, (1996).