Active Learning of Plans for Safety and Reachability Goals With Partial Observability

Wonhong Nam and Rajeev Alur, *Fellow, IEEE*

***Abstract*—Traditional planning assumes reachability goals and/or full observability. In this paper, we propose a novel solution for *safety and reachability planning* with *partial observability*. Given a planning domain, a safety property, and a reachability goal, we automatically learn a *safe permissive plan* to guide the planning domain so that the safety property is not violated and that can force the planning domain to eventually reach states that satisfy the reachability goal, regardless of how the planning domain behaves. Our technique is based on the *active learning of regular languages* and *symbolic model checking*. The planning method first learns a safe plan using the L^* algorithm, which is an efficient active learning algorithm for regular languages. We then check whether the safe plan learned is also permissive by Alternating-time Temporal Logic (ATL) model checking. If the plan is permissive, it is indeed a safe permissive plan. Otherwise, we identify and add a safe string to converge a safe permissive plan. We describe an implementation of the proposed technique and demonstrate that our tool can efficiently construct safe permissive plans for four sets of examples.**

***Index Terms*—Active learning, automated planning, partial observability, symbolic model checking.**

I. Introduction

ALTHOUGH the classical planning problems [1]–[3] mainly concentrate on *reachability goals*, *temporally extended goals*, including *safety properties*, have also recently been considered in a number of studies [4]–[8]. This trend is due to practical applications that require plans that handle more general goals than reachability. On the other hand, planning problem formulation typically assumes that the planning domains are *fully observable* [1], [2], [4]–[7]; that is, the plan knows the exact state of the planning domain, and it can take its action based on this information. However, the assumption of full observability is relaxed to treat common situations where the plan cannot know the exact state of the planning domain [9]–[12]. The combination of *extended goals* with *partial observability* is rarely studied due to the hardness of these problems.

Manuscript received December 1, 2008; revised February 27, 2009. First published August 4, 2009; current version published March 17, 2010. This work was supported in part by the Army Research Office under Grant DAAD19-01-1-0473 and the National Science Foundation under Grant ITR/SY 0121431 and Grant CCR0306382. This paper was recommended by Associate Editor X. Zeng.

W. Nam is with the College of Information & Communication, Konkuk University, Seoul 143-701, Korea (e-mail: wnam@konkuk.ac.kr).

R. Alur is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: alur@cis.upenn.edu).

Digital Object Identifier 10.1109/TSMCB.2009.2025657

In this paper, we propose a novel technique to learn a *safe plan with partial observability* for a given planning domain, which is also a *permissive plan*; that is, we adopt a safety requirement and a reachability requirement as the goal, and we assume that the plan has only partial information about the planning domain. To our best knowledge, this paper is the first attempt to employ regular language learning for planning problems. In addition, there is no study to apply this language learning technique in controller synthesis, which has a strong connection with automated planning. To solve this problem, we first focus on the safety goal and then describe the issues related to permissive plans that ensure reachability. Given a *planning domain* D and a set φ of *safe* states for the planning domain, a plan P for D is *safe* if every run that was induced by P on D always stays in the safe states that were described by φ. Then, it is clear that there exist many safe plans for D with respect to φ rather than a unique safe plan. To avoid very restrictive plans, we define a notion of *permissive* plans; given a reachability goal ψ, a plan P is a permissive plan with respect to ψ if the plan P has a *winning strategy* to make the planning domain eventually reach states that satisfy ψ, regardless of what the planning domain does. Finally, the planning problem that we consider in this paper is, given a planning domain D, a safety requirement φ, and a reachability requirement ψ, to construct a safe permissive plan P.

For this problem, our solution consists of the following three steps: 1) *learning a safe plan*, i.e., synthesizing a safe plan using the L^* algorithm [13], [14]; 2) *checking for a permissive plan*, i.e., testing whether a safe plan that the learning phase returned is also permissive; and 3) *finding a safe string*, i.e., searching a safe string that makes the current plan permissive if the safe plan is not a permissive one. Our learning-based planning first uses active learning for a regular language (the L^* algorithm [13], [14]) to construct a safe plan. Note that, whereas *passive learning* learns a target language from given sets of *positive samples* and *negative samples*, *active learning* (*supervised learning*) converges to the target language by asking queries to a teacher (oracle) who correctly answers for the queries. Typically, the passive learning cannot exactly learn the target language, but active learning can. The learning phase requires us to provide a teacher who can answer membership queries (MQs) and equivalence queries (EQs). Given an observation/action (O/A) sequence σ, the MQ checks whether every execution of the planning domain D that corresponds to σ always stays in the states that satisfy φ. This condition can be solved by a model checking problem for the composed model $D\|\sigma$. Given a conjecture plan P, the EQ tests whether the current conjecture plan P is a safe plan, which can also be posed

as a model-checking problem for the composed model $D\|P$. The important property of this step is that it always constructs a safe plan, and the number of queries required is polynomial in the size of the output plan P.

The second step is to check whether the safe plan that was returned by the learning phase is permissive with respect to a given reachability requirement ψ. This test can be performed by Alternating-time Temporal Logic (ATL) model checking [15], [16], which checks whether the given plan can force the planning domain to eventually reach states that were described by ψ, regardless of how the planning domain behaves. If this test passes, the plan that we construct is safe and permissive. Otherwise, we attempt to make it permissive by adding behaviors.

In the third step, we try to find a safe string that is not in the language of the current conjecture plan. Indeed, it is NP-hard to look for such a safe string, and hence, we only heuristically implement it using local search. If the search finds a safe string, we update the current plan to add it to its language and proceed to the first step. Otherwise, our procedure terminates with a safe plan.

In summary, given a planning domain D, a safety requirement φ, and a reachability requirement ψ, our approach automatically synthesizes a plan P such that the following four conditions are met.

1) P is the minimal Deterministic Finite Automaton (DFA) that accepts $L(P)$.
2) The number of queries is polynomial in the size of P and in the length of the longest counterexample that was obtained while constructing the plan.
3) P is always safe for D with respect to φ.
4) P may be permissive for D with respect to ψ.

The general techniques for this problem [17], [18] reduce this partial observation problem into a full observation problem, which is exponential in the original problem. However, our technique does not require this process, and the number of queries that will be needed is only polynomial in the number of states in a solution plan that will be constructed.

We report on a prototype implementation of our solution, which uses the symbolic model checker NUSMV [19] to implement the L^* algorithm and the ATL model checker MOCHA [15] to test for a permissive plan. Given a planning domain that was written in the NUSMV input language, a safety property, and a reachability property, our tool automatically constructs a safe plan that may also be permissive. We present experiments on the following four sets of examples, where each planning domain has about 20 Boolean variables:

1) *Stack*;
2) *MCell*;
3) *TLine*;
4) *Reader/Writer problem*.

The plans are constructed by our tool in 5 min on the average. More importantly, they are guaranteed to be safe, and in fact, all of them are permissive.

II. Related Work

In automated planning, a study for temporally extended goals with partial information is one of the most challenging problems. To the best of our knowledge, only Bertoli *et al.* [17], [18] have addressed this problem, where they have proposed an algorithm that builds plans in the general setting of extended goals and partial information. However, in their algorithm, each context (state) of a plan corresponds to a pair of *belief-desire* and *belief-intention*, which are associated to a set of states of a planning domain. That is, they have to manipulate, as a set of plan's states, a set of sets of states in the planning domain, which seems unlikely to scale. On the other hand, in this paper, we directly construct a set of states for the plan by active learning of the language of the plan.

A fundamental assumption that underlies classical planning [1], [2], [4], [20] is that the planning problem is deterministic; that is, the execution of planning domains always has a unique successor state, and the initial state is completely specified as a unique state. However, research in the planning problem much more focuses on nondeterministic planning domains [9], [10], [21]–[23].

For another view, whereas the classical planning research [1]–[3], [24] mostly concentrates on reachability properties as its goal, many studies [4]–[8], [20], [22] have recently been performed on temporally extended goals to represent more expressive goal conditions. However, some of them in this direction are restricted to deterministic planning domains [4], [20]. Extended goals make the planning problem close to the problem of automatic synthesis of controllers [25], [26]. However, most of the work in this area focuses on the theoretical foundations without providing practical implementations. For extended goals, there are several studies that adopt temporal logics. Most of these approaches [4]–[6], [8], [20], [27] adopt Linear Temporal Logic (LTL) as the goal language and cannot represent conditions on the degree in which the goal should be satisfied with respect to the nondeterminism in the execution. On the other hand, some studies [7], [28] use Computation Tree Logic (CTL) as the goal language, and they can represent the degree. In addition, Liu and Darabi [29] have proposed a framework for the reconfiguration of a discrete-event system controller, which has a dynamic event observation set. Alpan and Jafari [30] have studied, based on Petri nets, a synthesis technique to obtain a combined plant-and-supervisor model that describes the closed-loop controlled behavior of the overall system.

Partially observable planning domains have been tackled either using a probabilistic Markov-based approach [11], [31] or within a framework of possible-world semantics [9], [10], [12]. These research works do not go beyond the possibility of expressing more than simple reachability goals. One exception is Karlsson's work [32], where an LTL with a knowledge operator is used to define search control strategies in a progressive probabilistic planner. Another research direction related on partial observation is the partially observable Markov decision process (POMDP) [33]. A POMDP models an agent decision process where each action has a probability and a reward, and the goal of this problem is to maximize the expected reward of the agent over a possibly infinite horizon. Reinforcement learning, including Q-learning, is one of the prominent solutions for this problem, where we learn an action-value function that represents the optimal action for each state [34]. However, the problem in this paper is to guarantee the safety and reachability

without reward. In addition, our learning technique, based on the L^* algorithm, learns a regular language, which is a direct solution for the problem, by asking MQs/EQs to the oracle.

III. Preliminaries

In this section, we formalize the notion of a planning domain and a plan, and we define a safety/reachability planning problem that we consider in this paper. In addition, we explain the L^* algorithm for learning an unknown regular language, which we use to construct a safe plan.

A. Planning Domains

Definition 1—Planing Domain: A *symbolic planning domain* with partial observability is a tuple $D = (X, X^A, X^O, T)$ with following components.

1) X is a finite set of *variables* controlled by the planning domain D.
2) X^A is a finite set of *action variables* that the planning domain reads from its environment (e.g., a plan), where $X^A \cap X = \emptyset$. All the variables in X and X^A have finite domains (e.g., Boolean, bounded integers, and enumerated types).
3) $X^O \subseteq X$ is a finite set of *observation variables* that the environment (plan) of D can access.
4) $T(X, X^A, X')$ is a transition predicate over $X \cup X^A \cup X'$. For a set of variables X, we denote the set of primed variables of X as $X' = \{x' | x \in X\}$, which represents a set of variables that encode the successor states. T can define a nondeterministic transition relation.

A *state* of the planning domain D is a valuation of the variables in X. However, the environment (e.g., a plan) that provides an action to the domain in run time cannot observe all the values of variables in X. It can observe only the values of variables in X^O. That is, with *partial observation* for the state of the domain in runtime, the plan should force the planning domain to keep safe and to reach a goal. For a state s over a set X of variables, let $s[Y]$, where $Y \subseteq X$, denote the valuation over Y obtained by restricting s to Y. Let S denote the set of all planning domain states. Similarly, S^A and S^O denote the set of all valuations of the variables in X^A and X^O, respectively.

Example 1—Planning Domain for a Stack: Consider a planning domain $D(X, X^A, X^O, T)$ that describes a stack with a capacity of n. The stack has "*push*" and "*pop*" as its actions. Its environment (plan) can observe only whether it is full or empty. A Boolean variable "*Error*" represents if overflow or underflow occurs. One reasonable safety requirement is to avoid overflow and underflow, and a reachability requirement is to fully utilize the stack. These requirements can be represented by a safety property, i.e., $\varphi(X) \equiv (Error = F)$, and a reachability property, i.e., $\psi(X) \equiv (\texttt{NumOfElements} = n)$. The planning domain $D(X, X^A, X^O, T)$ for this stack can be represented with the following elements.

- $X = \{\texttt{Ele}, \texttt{State}, \texttt{Error}\}$, where $\texttt{Ele} : \texttt{0..n}$, $\texttt{State} : \{\texttt{full}, \texttt{empty}, \texttt{normal}\}$, and $\texttt{Error}$ is Boolean.
- $X^A = \{\texttt{Action}\}$, where $\texttt{Action} : \{\texttt{push}, \texttt{pop}\}$.
- $X^O = \{\texttt{State}\}$.

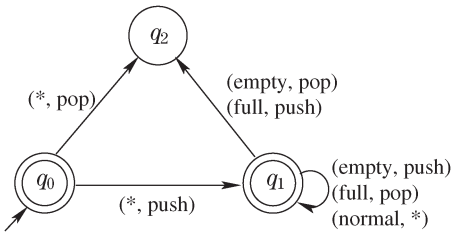


Fig. 1. Plan for the stack in Example 1.

- $T \equiv$
$$\begin{aligned}
&((\texttt{Action} = \texttt{push}) \rightarrow (\texttt{Ele}' = (\texttt{Ele} + 1)\texttt{mod}(\texttt{n} + 1))) \\
&\wedge ((\texttt{Action} = \texttt{pop}) \rightarrow (\texttt{Ele}' = (\texttt{Ele} - 1)\texttt{mod}(\texttt{n} + 1))) \\
&\wedge ((\texttt{Ele} = (\texttt{n} - 1) \wedge \texttt{Action} = \texttt{push}) \\
&\quad \rightarrow (\texttt{State}' = \texttt{full})) \\
&\wedge ((\texttt{Ele} = 1 \wedge \texttt{Action} = \texttt{pop}) \rightarrow (\texttt{State}' = \texttt{empty})) \\
&\wedge (\neg((\texttt{Ele} = (\texttt{n} - 1) \wedge \texttt{Action} = \texttt{push}) \\
&\qquad \vee (\texttt{Ele} = 1 \wedge \texttt{Action} = \texttt{pop})) \\
&\quad \rightarrow (\texttt{State}' = \texttt{normal})) \\
&\wedge ((\texttt{Ele} = 0 \wedge \texttt{Action} = \texttt{pop}) \rightarrow (\texttt{Error}' = \texttt{T})) \\
&\wedge ((\texttt{Ele} = \texttt{n} \wedge \texttt{Action} = \texttt{push}) \rightarrow (\texttt{Error}' = \texttt{T})) \\
&\wedge (\neg((\texttt{Ele} = 0 \wedge \texttt{Action} = \texttt{pop}) \\
&\qquad \vee (\texttt{Ele} = \texttt{n} \wedge \texttt{Action} = \texttt{push})) \rightarrow (\texttt{Error}' = \texttt{F})).
\end{aligned}$$

□

B. Plans

Definition 2—Plan: Given a planning domain $D(X, X^A, X^O, T)$, a *plan* for the planning domain D is a DFA $P = (Q, q_0, S^O, S^A, F, \delta)$, where

1) Q is finite set of *states* of the plan P.
2) $q_0 \in Q$ is *initial state*.
3) S^O is observation alphabet that the plan P can access for the planning domain D.
4) S^A is action alphabet that the plan P feeds to the planning domain D.
5) $F \subseteq Q$ is set of accepting states.
6) $\delta : Q \times S^O \times S^A \rightarrow Q$ is transition function.

For a plan state $q \in Q$ and an observation $o \in S^O$ on a planning domain, let the set of *legal actions* at q on o, denoted $LA(q, o)$, be the set of $a \in S^A$ such that $\delta(q, o, a) = q'$, where $q' \in F$; that is, if a plan P is at a state q and makes an observation $o \in S^O$ from a planning domain D, P can choose an action $a \in LA(q, o)$ and move to a state $q' = \delta(q, o, a)$.

Example 2—Plan for the Stack: Consider a plan for the planning domain D in Example 1. Fig. 1 illustrates a plan $P = (Q, q_0, S^O, S^A, F, \delta)$ for the stack, where $S^O = \{\texttt{full}, \texttt{empty}, \texttt{normal}\}$, and $S^A = \{\texttt{push}, \texttt{pop}\}$. In the figure, "$*$" matches every alphabet symbol as a wild card. This plan is initially in the state q_0. At this state, it allows only "*push*" for every observation and moves to q_1, because for every observation o, $LA(q_0, o) = \{\texttt{push}\}$. At the state q_1, if this plan observes "*full*" ("*empty*"), then it allows only "*pop*" (respectively "*push*"). Otherwise (i.e., when the plan observes "*normal*"), it allows both "*pop*" and "*push*". q_2 in this plan is a sink state, where the plan does not allow any legal action.

C. Safety/Reachability Planning Problems

Given a planning domain $D(X, X^A, X^O, T)$, a set of initial states is represented by $I(X)$, which is an initial state predicate over X.

Definition 3—Run: Given a planning domain $D(X, X^A, X^O, T)$, an initial state predicate $I(X)$, and a plan $P(Q, q_0, S^O, S^A, F, \delta)$ for D, a *run* of D according to P is a sequence $(s_0, o_0, a_0, q_0)(s_1, o_1, a_1, q_1) \cdots \in (S \times S^O \times S^A \times Q)^*$ such that the following conditions are met.

1) $I(s_0)$ holds.
2) For every $i \geq 0$, $o_i = s_i[X^O]$.
3) For every $i \geq 0$, $a_i \in LA(q_i, o_i)$.
4) For every $i \geq 0$, $T(s_i, a_i, s_{i+1})$ holds.
5) For every $i \geq 0$, $q_{i+1} = \delta(q_i, o_i, a_i)$.

A safety property for a planning domain $D(X, X^A, X^O, T)$ is a predicate over X. Given a planning domain D, an initial state predicate $I(X)$, a plan P for D, and a safety property $\varphi(X)$, we define $D, P \models_{safe} \varphi$ if, for every run $(s_0, o_0, a_0, q_0)(s_1, o_1, a_1, q_1) \cdots$ of D according to P, $\varphi(s_i)$ holds for every $i \geq 0$. We say that a plan P is a safe plan for D with respect to φ if $D, P \models_{safe} \varphi$.

Definition 4—Safety Planning Problem: A *safety planning problem with partial observability* is, given a planning domain D, an initial state predicate $I(X)$, and a safety property φ, to construct a plan P such that $D, P \models_{safe} \varphi$.

Given a planning domain D with an initial state predicate $I(X)$ and a safety property φ, it is clear that the safe plan is not unique. However, some of them may be very restrictive; for example, the most restrictive plan $P(Q, q_0, S^O, S^A, F, \delta)$ is a plan such that $LA(q, o) = \emptyset$ for every state $q \in Q$ and every $o \in S^O$. It is always a safe plan for all planning domains and all safety properties, because there is no run according to this plan.

Hence, we need a notion for what is a permissive plan (i.e., not very restrictive). For this purpose, we adopt strong reachability planning [3], which naturally arises in automated planning.

Given a planning domain $D(X, X^A, X^O, T)$, a *strategy* for the planning domain D is a function $\theta : (S^O)^* \rightarrow S^A$ that maps every planning-domain observation sequence $o_0 \cdots o_n \in (S^O)^*$ to an action $a \in S^A$. Then, a plan P includes many strategies rather than a unique strategy, since it has many moves for a given sequence $o_0 \cdots o_n$ of observations; that is, P can pick $a \in LA(q, o_n)$, where q is the current state of the plan. A plan P can fix its strategy by determining an action $a \in LA(q, o)$ for every state q and every observation o.

Given a strategy θ, we define *plays* between D with an initial state predicate $I(X)$ and θ, denoted by $plays(D, \theta)$, to be the set of infinite executions that are possible when the planning domain D follows the strategy θ; that is, an infinite sequence $(s_0, o_0, a_0)(s_1, o_1, a_1) \cdots$ is in $plays(D, \theta)$ if the following conditions are met.

1) $I(s_0)$ holds.
2) For every $i \geq 0$, $o_i = s_i[X^O]$.
3) For every $i \geq 0$, $a_i = \theta(o_0 \cdots o_i)$.
4) For every $i \geq 0$, $T(s_i, a_i, s_{i+1})$ holds.

Given a planning domain $D(X, X^A, X^O, T)$, an initial state predicate $I(X)$, and a reachability property $\psi(X)$ that is a predicate over X, a strategy θ is a *winning strategy* with respect to ψ if, for every play $(s_0, o_0, a_0)(s_1, o_1, a_1) \cdots \in plays(D, \theta)$, there exists $i \geq 0$ such that $\psi(s_i)$. Finally, given a planning domain D with an initial state predicate $I(X)$ and a reachability property ψ, if a plan P has a winning strategy, then P is a permissive plan for D with respect to ψ. We denote this relationship with $D, P \models_{reach} \psi$.

Definition 5—Safety/Reachability Planning Problem: The safety/reachability planning problem with partial observability that we focus on in this paper is, given a planning domain D, an initial state predicate $I(X)$, a safety property φ, and a reachability property ψ, to construct a safe permissive plan P, i.e., P such that $D, P \models_{safe} \varphi$ and $D, P \models_{reach} \psi$.

Example 3—Planning Problem for the Stack: Consider a safety/reachability planning problem for the stack in Example 1. Let us suppose that its capacity is 5 (i.e., $\mathtt{n} = 5$) and that the initial state predicate is $I \equiv (\mathtt{Ele} = 0) \wedge (\mathtt{State} = \mathtt{empty}) \wedge (\mathtt{Error} = \mathtt{F})$. In this example, we wish to avoid overflow and underflow, and we also want to fully utilize the stack. These requirements can be represented by a safety property $\varphi(X) \equiv (\mathtt{Error} = F)$ and a reachability property $\psi(X) \equiv (\mathtt{NumOfElements} = 5)$. Now, given the planning domain D in Example 1 with the aforementioned initial state predicate I, the safety property $\varphi(X) \equiv (\mathtt{Error} = F)$, and the reachability property $\psi(X) \equiv (\mathtt{NumOfElements} = 5)$, a safety/reachability planning problem with partial observability is to synthesize a safe permissive plan P such that $D, P \models_{safe} \varphi$ and $D, P \models_{reach} \psi$.

IV. L^* Algorithm

The active learning techniques for regular languages can be classified according to the teachers that they use: 1) a teacher who answers MQs and EQs [13], [14] and 2) a teacher who answers only EQs but always provides the lexicographically first counterexample [35], [36]. To the best of our knowledge, the best result of the first techniques requires $O(|\Sigma|n^2 + n \log m)$ MQs and at most $n - 1$ EQs, where n is the number of states in the target DFA, and m is the length of the longest counterexample that was provided by the teacher [14]. On the other hand, the second one needs $O(|\Sigma|n^2)$ EQs [36]. The total number of queries needed is similar in both techniques, but we believe that in our problem, EQs require more computation than MQs. Therefore, we employ the learning technique with a teacher for MQs and EQs, which requires fewer EQs.

The L^* algorithm learns an unknown regular language and generates a minimal DFA that accepts the language by asking MQs and EQs to a teacher. This algorithm had been introduced by Angluin [13], but later on, its efficiency was improved by Rivest and Schapire [14] (see [37] for a brief introduction to these algorithms). In this paper, we employ the improved version. The algorithm infers the structure of the DFA by asking a teacher MQs and EQs.

Fig. 2 illustrates the L^* algorithm [14]. Let U be the unknown regular language and Σ be its alphabet. At any given time, the L^* algorithm has, to construct a conjecture DFA, information about a finite collection of strings over Σ, which are classified either as members or nonmembers of U. This

```
1:  R := {ε};
2:  E := {ε};
3:  G[ε, ε] := member(ε·ε);
4:  foreach (a ∈ Σ) { G[ε·a, ε] := member(ε·a·ε); }
5:  repeat:
6:      while ((r_new := closed(R, E, G)) ≠ null) {
7:          add(R, r_new);
8:          foreach (a ∈ Σ), (e ∈ E)
9:              G[r_new·a, e] := member(r_new·a·e);
10:     }
11:     C := makeConjectureDFA(R, E, G);
12:     if ((cex := equivalent(C)) = null) then
13:         return C;
14:     else {
15:         e_new := findSuffix(cex);
16:         add(E, e_new);
17:         foreach (r ∈ R), (a ∈ Σ) {
18:             G[r, e_new] := member(r·e_new);
19:             G[r·a, e_new] := member(r·a·e_new);
20:         }
21:     }
```

Fig. 2. L^* algorithm.

information is maintained in an *observation table* (R, E, G), where R and E are sets of strings over Σ, and G is a function from $(R \cup R \cdot \Sigma) \times E$ to $\{0, 1\}$. More precisely, R is a set of representative strings for states in the conjecture DFA such that each representative string $r_q \in R$ for a state q leads from the initial state (uniquely) to the state q. E is a set of experiment suffix strings that are used to distinguish states; i.e., for any two states q_1 and q_2 of the DFA, there is a string $e \in E$ such that $member(r_{q_1} \cdot e) \neq member(r_{q_2} \cdot e)$, where r_{q_1} and r_{q_2} are representative strings for q_1 and q_2, respectively. G maps strings $\sigma = \sigma_1 \cdot \sigma_2$ (where $\sigma_1 \in R \cup R \cdot \Sigma$, and $\sigma_2 \in E$) to 1 if σ is in U, otherwise, it strings to 0. Initially, R and E are set to $\{\varepsilon\}$, and G is initialized using MQs for every string in $(R \cup R \cdot \Sigma) \times E$ (Lines 3–4). In Line 6, the algorithm checks whether the observation table is *closed*. The function $closed(R, E, G)$ returns *null* (meaning true) if, for every $r \in R$ and $a \in \Sigma$, there exists $r' \in R$ such that $G[r \cdot a, e] = G[r', e]$ for every $e \in E$; otherwise, it returns $r \cdot a$ such that there is no r' that satisfies the aforementioned condition. If the table is not closed, such a string $r \cdot a$ (e.g., denoted by r_{new} in Line 7) is simply added to R as a new state (i.e., a representative string). The algorithm again updates G with regard to $r \cdot a$ (Lines 8 and 9). Once the table is closed, it constructs a conjecture DFA $C = (Q, q_0, F, \delta)$ as follows (Line 11): $Q = R$, $q_0 = \varepsilon$, $F = \{r \in R \mid G[r, \varepsilon] = 1\}$, and for every $r \in R$ and $a \in \Sigma$, $\delta(r, a) = r'$ such that $G[r \cdot a, e] = G[r', e]$ for every $e \in E$. Finally, if the answer for the EQ is "yes", it returns the current conjecture machine C; otherwise, a counterexample $cex \in ((L(C) \setminus U) \cup (U \setminus L(C)))$ is provided by the teacher. The algorithm analyzes the counterexample cex to find the longest suffix e_{new} of cex that witnesses a difference between U and $L(C)$ (Line 15). Adding e_{new} to E reflects the difference in the next conjecture by splitting states in C. It then updates G with respect to e_{new} (Lines 17–20).

The L^* algorithm is guaranteed to construct a minimal DFA for the unknown regular language using only $O(|\Sigma|n^2 + n \log m)$ MQs and at most $n - 1$ EQs, where n is the number of states in the final DFA, and m is the length of the longest counterexample that was provided by the teacher when answering EQs. Moreover, the algorithm never constructs any intermediate automaton with more than n states.

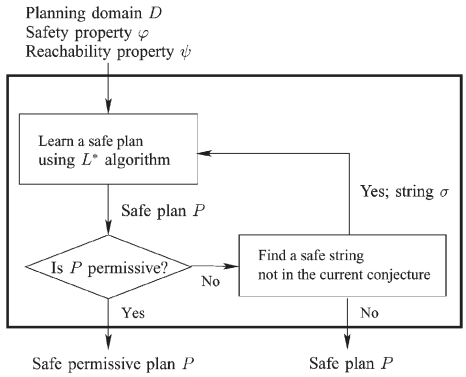


Fig. 3. High-level overview.

V. Learning Plans

Given a planning domain D with an initial state predicate I, a safety property φ, and a reachability property ψ, our aim is to construct a safe permissive plan P. Fig. 3 illustrates the high-level overview of our solution. The first step is to learn a safe plan using regular language learning. Once we obtain a safe plan, we check whether the safe plan is a permissive plan with respect to ψ. For this phase, we use an ATL model checker that can check that a given plan P has a winning strategy for D with respect to ψ. If the plan is permissive, it is the output of our solution. In this case, the result plan is safe and permissive. Otherwise, we try to find a safe string that is fed to the L^* algorithm to add more behaviors to the current plan. Looking for a safe string is an NP-hard problem; thus, we only locally search such a string. The trial cannot always guarantee that we will find a safe string. If we fail in the search, our procedure terminates and returns a safe plan. Note that, given a planning domain, a permissive plan does not always exist, whereas a safe plan always exists. This section describes three phases in detail.

A. Learning Safe Plans

Our first step is, given a planning domain D with an initial state predicate I, and a safety property φ, to learn a safe plan P using the L^* algorithm. We first define the *most general safe plan* that has all of the safe O/A sequences with respect to φ. Then, we explain our learning procedure, which always constructs a safe plan P.

1) Most General Safe Plan: Given a planning domain D, an initial state predicate I, and a safety property φ, a plan P is the *most general safe plan* if $L(P) \supseteq L(P')$ for every safe plan P'. Note that the most general safe plan P for D contains all O/A sequences $\sigma = (o_0, a_0)(o_1, a_1) \cdots \in (S^O \times S^A)^*$ such that, for every run $(s_0, o_0, a_0, q_0)(s_1, o_1, a_1, q_1) \cdots$ that corresponds to σ, $\varphi(s_i)$ holds for every $i \geq 0$. Recall that, for an O/A sequence, there exists a set of runs that correspond to the sequence due to nondeterminism and partial observability of planning domains.

To learn a particular regular language, the L^* algorithm needs a teacher who can answer MQs and EQs that correspond

to the language. However, our target language is the language of any plan among all the safe ones rather than a fixed language, and it causes the problem that we cannot exactly determine whether a given string is a member. Suppose that L_1 and L_2 are languages of two different safe plans, $\sigma \in L_1$, and $\sigma \notin L_2$. Then, if we declare that σ is a member, L_2 is ruled out from our target languages. Otherwise, L_1 is eliminated. Our solution for this problem is that the teacher for MQs corresponds with the language of the most general safe plan, and the teacher for EQs checks whether the conjecture plan is safe (if not, a counterexample would be an O/A string of which a run violates φ, and then, the counterexample corresponds with the most general safe plan). Then, our procedure eventually converges to the most general safe plan, because MQs and the counterexamples from EQs correspond to the most general safe plan, or it can discover a safe plan before the convergence (when the EQ passes).

2) MQ: Given an O/A sequence $\sigma = (o_0, a_0)(o_1, a_1) \cdots \in (S^O \times S^A)^*$, the teacher for MQs checks whether the sequence σ is in the language for the most general safe plan. The most general safe plan includes all the safe O/A sequences; thus, the MQ checks that every run that corresponds to σ is safe. For this query, we first construct a simple plan P_σ that precisely accepts the sequence σ and its prefixes and check whether the plan P_σ is safe for D with respect to φ.

Finally, we can check that a plan P_σ is safe for a planning domain D using an ordinary symbolic CTL model checking, where the model is the synchronous composition of D and P_σ, and the specification that will be checked is $\mathtt{AG}\,\varphi$. This CTL specification means that, for every path, every state along the path satisfies φ. It is easy to transform our models into representation for symbolic CTL model checking, since we symbolically define a planning domain, and the translation from an explicit plan to symbolic representation is also easy.

3) EQ: Given a plan P, the EQ asks whether P is a safe plan. For this test, we again use the CTL model checking as explained in the MQ.

Fig. 4 shows a pseudocode for the teacher, which we implement using the CTL model checking technique.

```
Boolean Membership(String σ) {
    P_σ := ConstructPlan(σ);
    if (SafePlan(D, P_σ, φ) = null) then return true;
    else return false;
}

String SafePlan(Domain D, Plan P, Prop φ) {
    D||P := Composition(D, P);
    CTLProperty := AG φ;
    cex := CTLModelChecking(D||P, CTLProperty);
    return cex;
}

String Equivalence(Plan P) {
    cex := SafePlan(D, P, φ);
    return cex;
}
```

Fig. 4. Implementation of the L^* teacher.

```
Boolean CheckReachGame(Domain D, Plan P, Prop ψ){
    τ1 := false;
    τ2 := ψ;
    while (τ2 ≠ τ1) {
        τ1 := τ1 ∨ τ2;
        τ2 := ∃X'_P∨X'_D. (T0(X_P, X'_P) ∧
                           (T1(X_D, X'_D) → τ1(X'_D, X'_P)));
    }
    if ((I_D(X_D) ∧ I_P(X_P)) → τ1) then return true;
    else return false;
}
```

Fig. 5. Symbolic model checking for $\ll M \gg \mathtt{F}\,\psi$.

B. Checking for Permissive Plans

Once we are provided a safe plan from the learning step, we check that it is also a permissive plan. Given a planning domain D, an initial state predicate I, a plan P, and a reachability property ψ, the plan P is a permissive plan if P has a winning strategy for D with respect to ψ. To check that a given plan is a permissive plan, we use the ATL model checker MOCHA. MOCHA [15] is a verification environment for modular verification against specifications that were written in ATL [16], which is a game logic extension of CTL.

Given a planning domain D with an initial state predicate I, a plan P, and a reachability property ψ, we first automatically translate into the modeling language *reactive modules* [15], where D and P are described as separate modules, and the property ψ is specified as an ATL formula. In this process, we can convert without any information loss, because our input language and reactive modules are both *finite state transition system* level languages. The logic ATL admits a formula $\ll M \gg \mathtt{F}\,\psi$, where M is a subset of modules, and ψ is a state predicate. The formula $\ll M \gg \mathtt{F}\,\psi$ asserts that the modules in M can cooperate to reach states that satisfy ψ, regardless of how the remaining modules behave. Considering M as the plan P, the semantics of $\ll M \gg \mathtt{F}\,\psi$ is exactly the same as $D, P \models_{reach} \psi$. Then, we use the symbolic ATL model checking of MOCHA, which implements the algorithm in Fig. 5, to check that the plan has a winning strategy. In the procedure, X_D is a set of variables controlled by the planning domain D, and X_P is a set of variables of the plan P. I_D and T_D are the initial state and transition predicates for D, respectively, and I_P and T_P are the initial state and transition predicates for P, respectively.

The algorithm starts with a set of states that satisfy ψ and identifies all the states where the plan P can force the game to reach the winning area in the next step, regardless of whatever the planning domain D does. It repeats the computation until the fix point. Finally, if all the initial states are included in the set of states computed, then the algorithm terminates with *true*, because P has a winning strategy from every initial state. Otherwise, it returns *false*.

C. Finding Safe Strings

Once the test for a permissive plan fails, we search an O/A sequence, which adds more behavior to the current plan but keeps the planning domain from violating a given safety property φ. Given a plan P, the search problem for such a sequence must find a string $\sigma = (o_0, a_0)(o_1, a_1) \cdots \notin L(P)$ such that every run of the planning domain that corresponds to σ always stays in states that satisfy the safety property φ. However, the search problem is NP-hard.

```
String LocalSearchSafeString(Plan P) {
    foreach (q_r ∈ Q) and (o,a),(o',a') ∈ (S^O × S^A) {
        if ¬Accept(P, r(o,a)(o',a')) then
            if MQ(r(o,a)(o',a')) then return r(o,a)(o',a');
    }
    return null;
}
```

Fig. 6. Local search for a safe string.

Proposition 1: Given a planning domain D, an initial state predicate I, a safe plan P, and a safety property φ, checking whether there is no string $\sigma \notin L(P)$ such that $L(P) \cup \{\sigma\}$ is safe is NP-hard.

The proof of the aforementioned proposition is given by a reduction from 3-SAT and crucially uses the fact that the problem can be reduced into a partial information game; we omit the proof.

We hence turn to a heuristic way to locally find such a safe string that the current plan may have missed. By a property of the observation table of the L^* algorithm, a current conjecture plan always allows all the safe O/A pairs (o, a) from every state, because the L^* algorithm has checked the membership for every string $\sigma \in R \cdot \Sigma$, and it has allowed O/A (o, a) from a state q_r if σ (i.e., $r \cdot (o, a)$) is a member. In the aforementioned discussion, r is the representative string that reaches q_r. However, in our experience, there are often scenarios where a safe O/A sequence $(o, a) \cdot (o', a')$ is disallowed from a particular state q, although (o, a) guarantees that (o', a') is safe. The reason is given as follows. q' is the state that was reached from q on (o, a), but there exists another path to q', which makes the O/A (o', a') from q' unsafe. In this case, we can provide the safe sequence, i.e., $r \cdot (o, a) \cdot (o', a')$, where r is the representative string for the state q, to the L^* algorithm to add it to the language of the conjecture plan. Then, the L^* algorithm splits the state q' into two states in the next iteration to distinguish those paths.

To find these safe strings, i.e., in the aforementioned case, $r \cdot (o, a) \cdot (o', a')$, we check whether, from every state, the current conjecture plan allows all the safe O/A sequences of length 2 (by a procedure *LocalSearchSafeString*() in Fig. 6); that is, we search $q_r \in Q$ and $(o, a), (o', a') \in S^O \times S^A$ such that $r(o, a)(o', a') \notin L(P)$ and $r(o, a)(o', a')$ is safe by traversing the plan P and asking MQs. If there exists $r(o, a)(o', a')$ that satisfies the aforementioned condition, we use $r(o, a)(o', a')$ as a safe O/A sequence to make the current plan permissive.

In summary, our learning technique always constructs a safe plan for a given planning domain and may provide the assurance that the constructed plan is also permissive.

VI. Experiments

We have implemented our learning technique for the safety/reachability planning problem using the CTL model checker NUSMV [19] and the ATL model checker MOCHA [15]. For experiments, we used four sets of planning domains. One set is our artificial example *Stack*, for which we have already known a simple safe permissive plan, and the rest of the sets are from existing studies, i.e., *MCell* [38], *TLine* [39], [40], and *Reader/Writer* [41]. In *MCell*, *TLine*, and *Reader/Writer*, the original case studies used full observability and/or reachability goals, but we adopt partial observability and safety goals, which are natural to the examples. All experiments have been performed on a PC using a 2.4-GHz Pentium processor, 2-GB memory, and a Linux operating system. In all cases, our tool could automatically generate useful safe permissive plans by using little computational resource.

TABLE I
Experimental Results

Domain	TV	O/A	MQ	EQ	time	states	S&P
Stack1	7	3	350	7	6.3	7	yes
Stack2	7	3	456	8	8.3	8	yes
Stack3	8	3	576	9	11.0	9	yes
Stack4	8	3	710	10	14.1	10	yes
MCell1	10	5	562	5	9.6	5	yes
MCell2	14	8	19011	11	1217.1	11	yes
TLine1	8	5	848	9	16.1	9	yes
TLine2	9	5	6494	17	180.8	23	yes
TLine3	9	5	16000	22	726.0	32	yes
R/W1	5	2	31	3	0.6	3	yes
R/W2	8	3	1008	12	19.3	12	yes
R/W3	11	4	13627	30	738.0	31	yes

Although Bertoli *et al.* studied a similar problem [17], [18], and they have the general planning tool Model Based Planner (MBP) [42], the public version of MBP does not support *safety* and *reachability* under *partial observability* at this point. In addition, their papers [17], [18] have only one example that cannot be applied to our problem setting.

The results for the planning domains are shown in Table I. It lists the total number of Boolean variables (TV) and O/A Boolean variables (O/A) in each planning domain and shows the number of MQs and EQs that the learner asked during the plan-learning phase. The table also shows the total execution time (in seconds) and the number of states in the plan that we constructed. The last column ("S&P") indicates whether the plan is safe and permissive.

In our experiments, our prototype tool learned a safe permissive plan in 5 min on the average. The fact that all the constructed plans were permissive shows that our solution can learn a safe permissive plan in many cases, although it cannot guarantee to always construct a safe permissive plan. The runtime of our tool is affected by the number of queries and the number of variables in the planning domain (the number of the variables is strongly related to the execution time for a model-checking problem, which corresponds with each query). For this problem, particularly in larger problems where the number of states in a plan or the number of queries needed increases, we can easily adopt a symbolic learning technique that can implicitly construct a plan, as our previous work, symbolic learning [43], [44].

A. Stack

We have already explained this example in Section II. For the experiment, we use four models with a capacity of 7–10 (Stacks 1–4 in Table I).

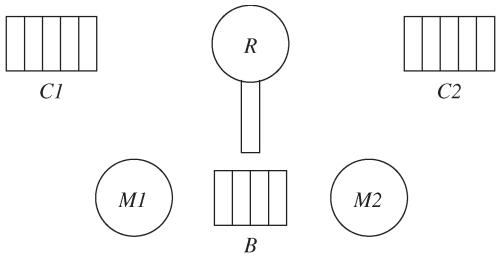


Fig. 7. Manufacturing cell.

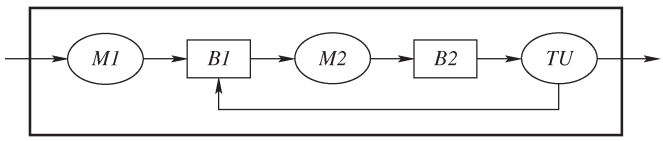


Fig. 8. Transfer line.

B. MCell

A manufacturing cell [38] consists of two machines $M1$ and $M2$ and a robot R (see Fig. 7). Workpieces arrive at the cell on an input conveyor $C1$ and leave the cell through an output conveyor $C2$. In between the machines $M1$ and $M2$, there is a buffer B with the capacity to hold four workpieces. The robot can transport the workpiece between components in the cell. In this experiment, we disregard the input and output conveyors, because they have unlimited capacity; the input conveyor gets its workpieces from an environment that can always produce a workpiece, and the output conveyor is always emptied by the environment.

The robot can get workpieces from components $C1$, $M1$, $M2$, and B, and it can load workpieces to components $C2$, $M1$, $M2$, and B, i.e., from $C1$ to B, from B to $M1$ or $M2$, and from $M1$ or $M2$ to $C2$. The machine $M1$ nondeterministically takes one to three time units to perform its work, and $M2$ nondeterministically takes one or two time units. A plan for this planning domain has to control the robot, and a safety property of this example avoids *underflow* and *overflow* for the buffer B. A plan can observe only whether each machine is idle, busy, or completes its work. We want a plan to force $M1$ to eventually run as the reachability requirement. There are two models for this example according to the number of machines: 1) MCell1 and 2) MCell2.

C. Transfer Line

A transfer line [39], [40] consists of two machines $M1$ and $M2$, followed by a test unit TU, which were linked by buffers $B1$ and $B2$. Fig. 8 illustrates this model. Each of the machines $M1$, $M2$, and TU has two states: 1) *idle* and 2) *busy*. The buffers $B1$ and $B2$ have a capacity of 3 and 1, respectively.

Machine $M1$ takes a raw workpiece from the outside to the buffer $B1$, and Machine $M2$ shifts a workpiece from $B1$ to the buffer $B2$. Similarly, TU takes a workpiece from $B2$ and checks it. A workpiece that was tested by TU may be accepted or rejected; if it is accepted, it is released from the system. Otherwise, it is returned to $B1$. A plan for the transfer line has to schedule the order for machines to work, but it knows only whether the buffers are full or empty. The safety specification is that $B1$ and $B2$ must be protected against *underflow* and *overflow*. In addition, the reachability property that we adopt is whether a plan can force a workpiece to be shifted to TU. According to the capacity of $B1$, we have three versions of this example: 1) TLine1; 2) TLine2; and 3) TLine3.

D. Reader/Writer

In a reader/writer system [41], we have one writer process W and two reader processes $R1$ and $R2$. All of these processes utilize a mutual critical region. The processes respectively and continuously acquire permission for the region, use it, and release it. Only one writer process can be allowed in the region at a time, provided that no reader is there. The reader processes, on the other hand, can simultaneously read, provided that no one writes. The critical region must therefore be subject to mutual exclusion, i.e., the writer process or only reader processes at a time.

Once the reader $R1$ occupies the critical region, it consumes three time units in the region. On the other hand, $R2$ takes two time units. Both readers are idle for one time unit after releasing the region. A plan has to control the writer process W, which takes only one time unit in the region, but it knows only whether each reader is in the final time unit in the region. The reachability specification for the permissive test is that a plan can make the writer process active. According to the number of readers, we have three models in this set: 1) R/W1; 2) R/W2; and 3) R/W3.

VII. Conclusion

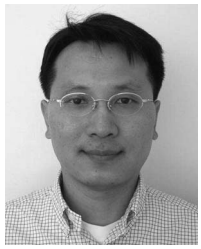
For the safety/reachability planning problem with partial observability, we have proposed a novel solution to automatically construct a safe plan for a given planning domain with respect to a given safety property. The plan that was learned by our tool may also be permissive with respect to a given reachability requirement. To construct such a safe permissive plan, the technique is based on regular language learning with the CTL model checking and the ATL model checking. Our experiments present promising results, where the tool learns safe permissive plans with a polynomial number of queries in reasonable time. There are several directions for future work. First, the current version of our technique supports safety and reachability properties as the target goal. To more broadly use our tool, it is needed to support more expressive goal representation, e.g., temporal logic representations in LTL or CTL. This extension requires modification of the teacher for the L^* algorithm. Second, it is also worthwhile to support a standard input language Planning-Domain Definition Language [45] for planning domains to easily compare with other tools and to experiment on more benchmarks.

References

[1] R. Fikes and N. Nilsson, "STRIP: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.

[2] J. Penberthy and D. Weld, "UCPOP: A sound, complete, partial-order planner for ADL," in *Proc. 3rd Int. Conf. KR*, 1992, pp. 103–114.

[3] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "Weak, strong, and strong cyclic planning via symbolic model checking," *Artif. Intell.*, vol. 147, no. 1/2, pp. 35–84, Jul. 2003.
[4] F. Bacchus and F. Kabanza, "Planning for temporally extended goals," *Ann. Math. Artif. Intell.*, vol. 22, no. 1/2, pp. 5–27, 1998.
[5] S. Cerrito and M. Mayer, "Bounded model search in linear temporal logic and its application to planning," in *Proc. 7th Int. Conf. Analytic Tableaux Related Methods*, 1998, pp. 124–140.
[6] J. Kvarnstrom and P. Doherty, "TALplanner: A temporal-logic-based forward chaining planner," *Ann. Math. Artif. Intell.*, vol. 30, no. 1–4, pp. 119–169, 2001.
[7] M. Postore and P. Traverso, "Planning as model checking for extended goals in nondeterministic domains," in *Proc. 17th IJCAI*, 2001, pp. 479–486.
[8] D. Calvanese, G. de Giacomo, and M. Vardi, "Reasoning about action and planning in LTL action theories," in *Proc. 8th Int. Conf. KR*, 2002, pp. 593–602.
[9] D. Weld, C. Anderson, and D. Smith, "Extending Graphplan to handle uncertainty and sensing actions," in *Proc. 15th Nat. Conf. Artif. Intell. (AAAI)*, 1998, pp. 897–904.
[10] J. Rintanen, "Constructing conditional plans by a theorem prover," *J. Artif. Intell. Res.*, vol. 10, pp. 323–352, 1999.
[11] B. Bonet and H. Geffner, "Planning with incomplete information as heuristic search in belief space," in *Proc. AIPS*, 2000, pp. 52–61.
[12] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, "Planning in nondeterministic domains under partial observability via symbolic model checking," in *Proc. 17th IJCAI*, 2001, pp. 473–478.
[13] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.
[14] R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," *Inf. Comput.*, vol. 103, no. 2, pp. 299–347, Apr. 1993.
[15] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran, "MOCHA: Modularity in model checking," in *Proc. 10th Int. Conf. CAV*, 1998, pp. 516–520.
[16] R. Alur, T. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *J. ACM*, vol. 49, no. 5, pp. 1–42, Sep. 2002.
[17] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso, "A framework for planning with extended goals under partial observability," in *Proc. 13th ICAPS*, 2003, pp. 215–224.
[18] P. Bertoli and M. Pistore, "Planning with extended goals and partial observability," in *Proc. 14th ICAPS*, 2004, pp. 270–278.
[19] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An OpenSource tool for symbolic model checking," in *Proc. 14th Int. Conf. CAV*, 2002, pp. 359–364.
[20] G. de Giacomo and M. Vardi, "Automata-theoretic approach to planning with temporally extended goals," in *Proc. 5th ECP*, 1999, pp. 35–48.
[21] L. Pryor and G. Collins, "Planning for contingency: A decision-based approach," *J. Artif. Intell. Res.*, vol. 4, pp. 81–120, 1996.
[22] F. Kabanza, M. Barbeau, and R. St-Denis, "Planning control rules for reactive agents," *Artif. Intell.*, vol. 95, no. 1, pp. 67–113, Aug. 1997.
[23] A. Cimatti, M. Roveri, and P. Traverso, "Automatic OBDD-based generation of universal plans in nondeterministic domains," in *Proc. 15th Nat. Conf. Artif. Intell. (AAAI)*, 1998, pp. 875–881.
[24] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso, "Planning via model checking: A decision procedure for AR," in *Proc. 4th ECP*, 1997, pp. 130–142.
[25] E. Asarin, O. Maler, and A. Pnueli, "Symbolic controller synthesis for discrete and timed systems," in *Hybrid Systems II*. New York: Springer-Verlag, 1995, pp. 1–20.
[26] O. Kupferman and M. Vardi, "Synthesis with incomplete information," in *Proc. 2nd Int. Conf. Temporal Logic*, 1997, pp. 91–106.
[27] F. Bacchus and F. Kabanza, "Using temporal logic to express search control knowledge for planning," *Artif. Intell.*, vol. 116, no. 1/2, pp. 123–191, Jan. 2000.
[28] M. Postore, R. Bettin, and P. Traverso, "Symbolic techniques for planning with extended goals in nondeterministic domains," in *Proc. 6th ECP*, 2001, pp. 253–264.
[29] J. Liu and H. Darabi, "Control reconfiguration of discrete event systems controllers with partial observation," *IEEE Trans. Syst., Man, Cybern. B: Cybern.*, vol. 34, no. 6, pp. 2262–2272, Dec. 2004.
[30] G. Alpan and M. A. Jafari, "Synthesis of a closed-loop combined plant and controller model," *IEEE Trans. Syst., Man, Cybern. B: Cybern.*, vol. 32, no. 2, pp. 163–175, Apr. 2002.
[31] P. Vrancx, K. Verbeeck, and A. Nowé, "Decentralized learning in Markov games," *IEEE Trans. Syst., Man, Cybern. B: Cybern.*, vol. 38, no. 4, pp. 976–981, Aug. 2008.
[32] L. Karlsson, "Conditional progressive planning under uncertainty," in *Proc. 17th IJCAI*, 2001, pp. 431–438.
[33] J. Pineau and G. J. Gordon, "POMDP planning for robust robot control," in *Proc. 12th ISRR*, 2005, pp. 69–82.
[34] T. Jaakkola, S. P. Singh, and M. I. Jordan, "Reinforcement learning algorithm for partially observable Markov decision problems," in *Proc. NIPS 7*, 1994, pp. 345–352.
[35] O. Ibarra and T. Jiang, "Learning regular languages from counterexamples," *J. Comput. Syst. Sci.*, vol. 43, no. 2, pp. 299–316, Oct. 1991.
[36] A. Birkendorf, A. Böker, and H.-U. Simon, "Learning deterministic finite automata from smallest counterexamples," *SIAM J. Discrete Math.*, vol. 13, no. 4, pp. 465–491, Oct. 2000.
[37] M. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. Cambridge, MA: MIT Press, 1994.
[38] M. Fabian, *Discrete Event Systems*. Gothenburg, Sweden: Chalmers Univ. Technol., 2006.
[39] W. Wonham, *Notes on Control of Discrete-Event Systems*. Toronto, ON, Canada: Univ. Toronto, 1999.
[40] Z. Zhang and W. Wonham, "STCT: An efficient algorithm for supervisory control design," in *Proc. SCODES*, 2001, pp. 82–93.
[41] A. Tanenbaum, *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
[42] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "MBP: A model-based planner," in *Proc. IJCAI Workshop Planning Under Uncertainty Incomplete Inf.*, 2001, pp. 93–97.
[43] R. Alur, P. Madhusudan, and W. Nam, "Symbolic compositional verification by learning assumptions," in *Proc. 17th Int. Conf. CAV*, 2005, pp. 548–562.
[44] W. Nam, P. Madhusudan, and R. Alur, "Automatic symbolic compositional verification by learning assumptions," *Form. Methods Syst. Des.*, vol. 32, no. 3, pp. 207–234, Jun. 2008.
[45] D. McDermott, "PDDL: The planning domain definition language," Yale Center Comput. Vis. Control, New Haven, CT, Tech. Rep., CVC TR-98-003, 1989.



Wonhong Nam received the B.S. and M.Sc. degrees from Korea University, Seoul, Korea, in 1998 and in 2001, respectively, and the Ph.D. degree from the University of Pennsylvania, Philadelphia, in 2007.

From 2007 to 2009, he was a Postdoctoral Researcher with the College of Information Sciences and Technology, Pennsylvania State University, University Park. He is currently an Assistant Professor of the College of Information & Communication, Konkuk University, Seoul, Korea. His research interests include formal methods, formal verification, model checking, automated planning, and web services composition.

Rajeev Alur (F'07) received the B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur, India, in 1987 and the Ph.D. degree in computer science from Stanford University, Stanford, CA, in 1991.

In 1997, he was with the Computing Science Research Center, Bell Laboratories. He is currently the Zisman Family Professor and Graduate Group Chair with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia. His research interests include formal modeling and analysis of reactive systems, hybrid systems, model checking, software verification, logics and automata, and design automation for embedded software.

Prof. Alur is a Fellow of the Association for Computing Machinery. He has been the Chair or a Cochair of scientific meetings such as the International Conference on Computer-Aided Verification (CAV), ACM Symposium on Embedded Software (EMSOFT), International Conference on Hybrid Systems: Computation and Control (HSCC), and IEEE Symposium on Logic in Computer Science (LICS). He was the Chair of the ACM Special Interest Group on Embedded Systems (SIGBED). He received the CAV Award in 2008, the President of India's Gold Medal for Academic Excellence in 1987, the National Science Foundation Faculty Early Career Development (CAREER) Award in 1997, the ITR Award in 2001, and the Alfred P. Sloan Faculty Fellowship in 1999. He was also a highly cited Scientist by the Institute for Scientific Information in 2005.