University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

January 2007

Learning Plans for Safety and Reachability Goals with Partial Observability

Wonhong Nam
University of Pennsylvania, wnam@cis.upenn.edu

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Nam, Wonhong and Alur, Rajeev, "Learning Plans for Safety and Reachability Goals with Partial Observability" (2007). *Technical Reports (CIS)*. Paper 643.
http://repository.upenn.edu/cis_reports/643

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-07-16.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/643
For more information, please contact repository@pobox.upenn.edu.

Learning Plans for Safety and Reachability Goals with Partial Observability

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 and permissive plan* to guide the planning domain so that the safety property is not violated and which can force the planning domain to eventually reach states satisfying the reachability goal no matter how the planning domain behaves. Our technique is based on *active learning of regular languages and symbolic model checking*. We describe an implementation of the proposed technique and demonstrate that the tool can efficiently construct safe and permissive plans for four examples.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-07-16.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/643

Learning Plans for Safety and Reachability Goals with Partial Observability *

Wonhong Nam and Rajeev Alur
Department of Computer and Information Science
University of Pennsylvania
{wnam, alur}@cis.upenn.edu

Tech. Report MS-CIS-07-16

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 and permissive plan* to guide the planning domain so that the safety property is not violated and which can force the planning domain to eventually reach states satisfying the reachability goal no matter how the planning domain behaves. Our technique is based on *active learning of regular languages* and *symbolic model checking*. We describe an implementation of the proposed technique and demonstrate that the tool can efficiently construct safe and permissive plans for four examples.

1 Introduction

While the classical planning problems [FN71, PW92, CPRT03] mainly concentrate on *reachability goals*, *temporally extended goals* including *safety properties* have also been recently considered in a number of studies [BK98, CM98, KD01, PT01, CdGV02]. This trend is due to practical applications that require plans which handle more general goals than reachability. On the other side, planning problem formulation typically assumes that the planning domains are *fully observable* [FN71, PW92, BK98, CM98, KD01, PT01];

*This research was partially supported by ARO grant DAAD19-01-1-0473, and NSF grants ITR/SY 0121431 and CCR0306382.

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 in order to treat common situations where the plan cannot know the exact state of the planning domain [WAS98, Rin99, BG00, BCRT01]. The combination of *extended goals* with *partial observability* is rarely studied due to the hardness of these problems.

In this paper, we introduce 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 as well as a reachability requirement as the goal, and an assumption that the plan has only partial information about the planning domain. In order to solve this problem, we first focus on the safety goal, 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 induced by P on D always stays in the safe states 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 too 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 satisfying ψ whatever the planning domain does. Finally, the planning problem we consider in this paper is, given a planning domain D and a safety requirement φ and a reachability requirement ψ, to construct a safe and permissive plan P.

Our solution first uses active learning [1] for a regular language (the L^* algorithm [Ang87, RS93]) in order to construct a safe plan. The learning phase requires us to provide a teacher who can answer membership and equivalence queries. The membership query is to check whether every execution of the planning domain D corresponding to a given input/output sequence σ always stays in the states satisfying φ. This can be solved by a model checking problem for the composed model $D||\sigma$. The equivalence checking is to test whether the current conjecture plan P is a safe plan, which also can be posed as a model checking problem for the composed model $D||P$. The important property of this step is that it always construct 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 returned by the learning

[1] Note that while *passive learning* learns a target from given sets of *positive samples* and *negative samples*, *active learning* converges to the target by asking queries to a teacher (oracle) who answers correctly for the queries. Typically, the former cannot learn the target exactly, but the latter can do.

phase is permissive with respect to a given reachability requirement ψ. This test can be performed by ATL model checking [AHM+98, AHK02], which checks whether the given plan can force the planning domain to eventually reach states described by ψ no matter how the planning domain behaves. If this test passes, the plan 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 not in the language of the current conjecture plan. Indeed, it is NP-hard to look for such a safe string, and hence we implement it only heuristically 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 (1) P is the minimal DFA accepting $L(P)$, (2) the number of queries is polynomial in the size of P and in the length of the longest counter-example obtained while constructing the plan (3) P is always safe for D with respect to φ, and (4) P may be permissive for D with respect to ψ.

We report on a prototype implementation of our solution, which uses the symbolic model checker NuSmv [CCG+02] to implement the L^* algorithm and the ATL model checker Mocha [AHM+98] to test for a permissive plan. Given a planning domain written in the NuSmv input language, and safety and reachability properties, our tool automatically construct a safe plan which may be permissive as well. We present experiments on four examples: *Stack, MCell, TLine* and *Reader/Writer problem*, where each planning domain has about 20 boolean variables. The plans are constructed by our tool within 40 minutes. More importantly, they are guaranteed to be safe, and in fact, all of them are permissive.

1.1 Related work

In the planning field, 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. [BCPT03, BP04] 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 the language of the plan.

2 Preliminaries

In this section, we formalize the notion of a planning domain and a plan, and we define a safety/reachability planning problem 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.

2.1 Planning domains

A *symbolic planning domain* with partial observability is a tuple $D = (X, X^I, X^O, \mathit{Init}, T)$ where

- X is a finite set of *variables* controlled by the planning domain D.
- X^I is a finite set of *input variables* that the planning domain reads from its environment (plan), where $X^I \cap X = \emptyset$. All the variables in X and X^I have finite domains (e.g. boolean, bounded integers and enumerated types).
- $X^O \subseteq X$ is a finite set of *output variables* that are observable to the environment (plan) of D.
- $\mathit{Init}(X)$ is an initial state predicate over X.
- $T(X, X^I, X')$ is a transition predicate over $X \cup X^I \cup X'$. For a set of variables X, we denote the set of primed variables of X as $X' = \{x' \mid x \in X\}$. X' represents a set of variables encoding the successor states. T can define a non-deterministic transition relation.

A *state* of the planning domain D is a valuation of the variables in X. 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^I and S^O denote the set of all valuations of the variables in X^I and X^O, respectively.

Example 1 (Planning domain for stack) *Consider a planning domain $D(X, X^I, X^O, \mathit{Init}, T)$ describing a stack with capacity of 5. The stack has 'push' and 'pop' as its input values. Its environment (plan) can observe*

only whether it is full or empty. A variable 'Error' represents if overflow or underflow occurs. The planning domain D for this stack can be represented with the following elements:

- $X = \{$`Ele, State, Error`$\}$ where `Ele:0..5`, `State:`$\{$`full, empty, normal`$\}$, and `Error` is boolean.
- $X^I = \{$`In`$\}$ where `In:`$\{$`push, pop`$\}$.
- $X^O = \{$`State`$\}$.
- $Init \equiv$ (`Ele=0`) $\wedge$ (`State=empty`) $\wedge$ (`Error=F`).
- $T \equiv$ ((`In=push`) $\rightarrow$ (`Ele'=(Ele+1) mod 6`))
  $\wedge$ ((`In=pop`) $\rightarrow$ (`Ele'=(Ele-1) mod 6`))
  $\wedge$ ((`Ele=4` $\wedge$ `In=push`) $\rightarrow$ (`State'=full`))
  $\wedge$ ((`Ele=1` $\wedge$ `In=pop`) $\rightarrow$ (`State'=empty`))
  $\wedge$ ($\neg$((`Ele=4` $\wedge$ `In=push`) $\vee$ (`Ele=1` $\wedge$ `In=pop`))$\rightarrow$ (`State'=normal`))
  $\wedge$ ((`Ele=0` $\wedge$ `In=pop`) $\rightarrow$ (`Error'=T`))
  $\wedge$ ((`Ele=5` $\wedge$ `In=push`) $\rightarrow$ (`Error'=T`))
  $\wedge$ ($\neg$((`Ele=0` $\wedge$ `In=pop`) $\vee$ (`Ele=5` $\wedge$ `In=push`))$\rightarrow$ (`Error'=F`)).

2.2 Plans

Given a planning domain $D(X, X^I, X^O, Init, T)$, a *plan* for the planning domain D is a DFA $P = (Q, q_0, S^O, S^I, F, \delta)$ with following components:

- Q is a finite set of *states* of the plan P.
- $q_0 \in Q$ is the *initial state*.
- S^O is the output alphabet from the planning domain D.
- S^I is the input alphabet to the planning domain D. From the plan's viewpoint, S^O is the input alphabet and S^I is the output alphabet. In this paper, however, we use the terms based on the planning domain's viewpoint.
- $F \subseteq Q$ is a set of accepting states.
- $\delta : Q \times S^O \times S^I \rightarrow Q$ is a transition function.

For a plan state $q \in Q$ and an output $o \in S^O$ from a planning domain, let the set of *legal inputs* at q on o, denoted $LI(q, o)$ be the set of $i \in S^I$ such that $\delta(q, o, i) = q'$ where $q' \in F$; that is, if a plan P is at a state q and receives an output $o \in S^O$ from a planning domain, P can choose an input $i \in LI(q, o)$ and moves to a state $q' = \delta(q, o, i)$.

Given a planning domain $D(X, X^I, X^O, Init, T)$ and a plan $P(Q, q_0, S^O, S^I, F, \delta)$ for D, a *run* of D according to P is a sequence $(s_0, o_0, i_0, q_0)(s_1, o_1, i_1, q_1) \cdots \in (S \times S^O \times S^I \times Q)^*$ such that

- $Init(s_0)$ holds,
- For every $j \geq 0$, $o_j = s_j[X^O]$,
- For every $j \geq 0$, $i_j \in LI(q_j, o_j)$,
- For every $j \geq 0$, $T(s_j, i_j, s_{j+1})$ holds, and
- For every $j \geq 0$, $q_{j+1} = \delta(q_j, o_j, i_j)$.

A safety property for a planning domain $D(X, X^I, X^O, Init, T)$ is a predicate over X. Given a planning domain D, a plan P for D and a safety property $\varphi(X)$, we define $D, P \models_{safe} \varphi$ if for every run $(s_0, o_0, i_0, q_0)(s_1, o_1, i_1, q_1) \cdots$ of D according to P, $\varphi(s_j)$ holds for every $j \geq 0$. We say that a plan P is a safe plan for D with respect to φ if $D, P \models_{safe} \varphi$.

Example 2 (Plan for stack) *Consider a plan for the planning domain D in Example 1. Figure 1 illustrates a plan $P = (Q, q_0, S^O, S^I, F, \delta)$ for the stack, where $S^O = \{\texttt{full}, \texttt{empty}, \texttt{normal}\}$ and $S^I = \{\texttt{push}, \texttt{pop}\}$. 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 output and moves to q_1, since for every output o, $LI(q_0, o) = \{\texttt{push}\}$.*

2.3 Safety/reachability planning problems

A *safety planning problem with partial observability* is, given a planning domain D and a safety property φ, to construct a plan P such that $D, P \models_{safe} \varphi$. Given a planning domain D and a safety property φ, it is clear that the safe plan is not unique. However, some of them may be too restrictive; for example, the most restrictive plan $P(Q, q_0, S^O, S^I, F, \delta)$ is a plan such that $LI(q, o) = \emptyset$ 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 since there is no run according to this plan. We call this plan a *null plan*.

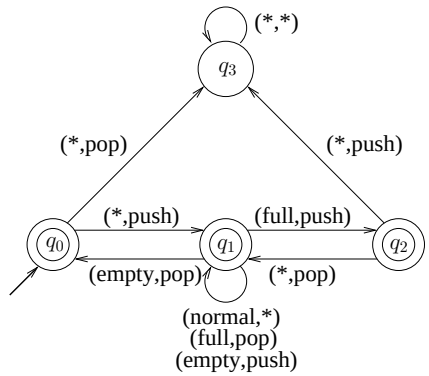

Figure 1: A plan for the stack in Example 1

Hence, we need a notion for what is a permissive plan (i.e. not too restrictive). For this purpose, we adopt strong reachability planning [CPRT03] which arises naturally in the planning problem field.

Given a planning domain $D(X, X^I, X^O, \mathit{Init}, T)$, a *strategy* for the planning domain D is a function $\theta : (S^O)^* \to S^I$ that maps every planning domain output sequence $o_0 \cdots o_n \in (S^O)^*$ to an input $i \in S^I$. Then, a plan P includes not a unique strategy but many strategies since it has many moves for a given sequence $o_0 \cdots o_n$ of outputs; that is, P can pick $i \in LI(q, o_n)$ where q is the current state of the plan. A plan P can fix its strategy by determining an input $i \in LI(q, o)$ for every state q and every output o.

Given a strategy θ, we define *plays* between D and θ, denoted by $\mathit{plays}(D, \theta)$, to be the set of infinite executions that are possible when the planning domain D follows the strategy θ; that is, an infinite sequence $(s_0, o_0, i_0)(s_1, o_1, i_1) \cdots$ is in $\mathit{plays}(D, \theta)$ if

- $\mathit{Init}(s_0)$ holds,
- For every $j \geq 0$, $o_j = s_j[X^O]$,
- For every $j \geq 0$, $i_j = \theta(o_0 \cdots o_j)$, and
- For every $j \geq 0$, $T(s_j, i_j, s_{j+1})$ holds.

Given a planning domain $D(X, X^I, X^O, \mathit{Init}, T)$ 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 $(s_0, o_0, i_0)(s_1, o_1, i_1) \cdots \in \mathit{plays}(D, \theta)$, there exists $j \geq 0$ such that $\psi(s_j)$. Finally, given a planning domain D 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 $D, P \models_{reach} \psi$.

Now, the safety/reachability planning problem with partial observability we focus on in this paper is, given a planning domain D, a safety property φ and a reachability property ψ, to construct a safe and permissive plan P; that is, P such that $D, P \models_{safe} \varphi$ and $D, P \models_{reach} \psi$.

Example 3 (Planning problem for stack) *Consider a safety/reachability planning problem for the stack in Example 1. In this example, we wish to avoid overflow as well as underflow, and we also want to fully utilize the stack. These requirements can be represented by a safety property* $\varphi(X) \equiv (\texttt{Error} = F)$ *and a reachability property* $\psi(X) \equiv (\texttt{NumOfElements} = 5)$*. Now, given the planning domain D in Example 1, the safety property* $\varphi(X) \equiv (\texttt{Error} = F)$ *and the reachability property* $\psi(X) \equiv (\texttt{NumOfElements} = 5)$*, a safety/reachability planning problem with partial observability is to synthesize a safe and permissive plan P such that* $D, P \models_{safe} \varphi$ *and* $D, P \models_{reach} \psi$*.*

2.4 L* algorithm

The L^* *algorithm* actively learns an unknown regular language U (let Σ be its alphabet) and generates a minimal DFA that accepts the regular language. This algorithm was introduced by Angluin [Ang87], but we adopt an improved version by Rivest and Schapire [RS93]. The algorithm infers the structure of the DFA by asking a teacher, who knows the unknown language, membership and equivalence queries. Membership queries ask whether a given string $\sigma \in \Sigma^*$ is in the language U, and the answer for the queries is yes or no. Equivalence queries ask whether a given conjecture DFA C represents the language U, and the answer is yes or no with a counter-example that is a symmetric difference between $L(C)$ and U.

At any given time, the L^* algorithm has, in order to construct a conjecture machine, information about a finite collection of strings over Σ, classified either as members or non-members of U based on membership queries. This information is maintained in an *observation table* (R, E, M) which represents the conjecture DFA; R is a set of representative strings for states in the DFA such that each representative string $r_q \in R$ for a state q leads from the initial state (uniquely) to the state q, and E is a set of experiment suffix strings that are used to distinguish states. M maps strings σ in $(R \cup R{\cdot}\Sigma) \cdot E$ to 1 if σ is in U, and to 0 otherwise. Once a conjecture machine C is built, the algorithm asks an equivalence query. Finally, if the answer is

'yes', it returns the current conjecture DFA C; otherwise, a counter-example $cex \in ((L(C) \setminus U) \cup (U \setminus L(C))$ is provided by the teacher. In the latter case, the algorithm updates the current conjecture using the counter-example cex.

If a teacher for two kinds of queries is provided, the L^* algorithm is guaranteed to construct a minimal DFA for the unknown regular language using only $O(|\Sigma|n^2 + n \log m)$ membership queries and at most $n-1$ equivalence queries, where n is the number of states in the final DFA and m is the length of the longest counter-example provided by the teacher for equivalence queries.

3 Learning plans

Given a planning domain D, a safety property φ and a reachability property ψ, our aim is to construct a safe permissive plan P. Figure 2 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 is able to check that a given plan P has a winning strategy for D with respect to ψ. If the plan is a permissive one, 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 in order to add more behaviors to the current plan. In fact, since it is a NP-hard problem to look for a safe string, we only search locally such a string. The trial cannot always guarantee to 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, while a safe plan always exists (e.g., a null plan is always safe). This section describes three phases in detail.

3.1 Learning safe plans

Our first step is, given a planning domain D 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 input/output sequences with respect to φ. Then, we explain our learning procedure which always constructs a safe plan P.

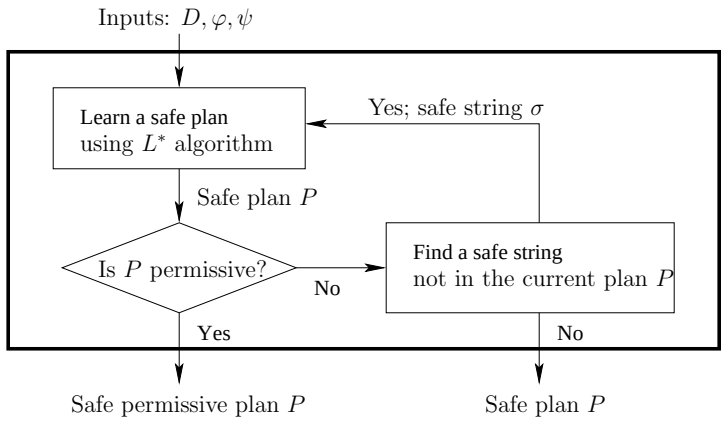

Figure 2: High-level overview

3.1.1 Most general safe plan

Given a planning domain D 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 input/output sequences $\sigma = (o_0, i_0)(o_1, i_1) \cdots \in (S^O \times S^I)^*$ such that for every run $(s_0, o_0, i_0, q_0)(s_1, o_1, i_1, q_1) \cdots$ corresponding to σ, $\varphi(s_j)$ holds for every $j \geq 0$. Recall that for an input/output sequence, there exists a set of runs corresponding to the sequence due to non-determinism and partial observability of planning domains.

To learn a particular regular language, the L^* algorithm needs a teacher who can answer membership and equivalence queries corresponding 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 a problem that we cannot determine exactly whether a given string is a member or not. Suppose that L_1 and L_2 are languages of two different safe plans, and $\sigma \in L_1$ and $\sigma \notin L_2$. Then, if we declare 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 membership queries corresponds with the language of the most general safe plan, and the teacher for equivalence queries checks whether the conjecture plan is safe or not (if not, a counter-example would be an input/output string of which a run violates φ; then, the counter-example corresponds with the most general safe plan). Then, our procedure eventually converges to the most general safe plan since membership queries and the counter-examples from equivalence queries correspond to the most

general safe plan, or it can discover a safe plan before the convergence (when the equivalence query passes).

3.1.2 Membership query

Given an input/output sequence $\sigma = (o_0, i_0)(o_1, i_1) \cdots \in (S^O \times S^I)^*$, the teacher for membership queries checks whether the sequence σ is in the language for the most general safe plan. Since the most general safe plan includes all the safe input/output sequences, the membership query is to check that every run corresponding to σ is safe. For this query, we first construct a simple plan P_σ with $|\sigma| + 2$ states that accepts precisely the sequence σ and its prefixes, and check that 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 to be checked is `AG` φ. The CTL formula means that for every path, every state along the path satisfies φ. It is easy to transform from our models to representation for symbolic CTL model checking, since we define symbolically a planning domain and the translation from an explicit plan to symbolic representation is also easy.

3.1.3 Equivalence query

Given a plan P, the equivalence query asks whether P is a safe plan. For this test, we again use the CTL model checking explained in the membership query.

Figure 3 shows a pseudo-code for the teacher which we implement using CTL model checking technique.

3.2 Checking for permissive plans

Once we are provided a safe plan from the learning step, we check that it is a permissive plan as well. Given a planning domain D, 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 an ATL model checker MOCHA. MOCHA [AHM+98] is a verification environment for modular verification against specifications written in ATL (Alternating-time Temporal Logic) [AHK02], which is a game logic extension of CTL.

```
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;
}
```

Figure 3: Implementation of the L^* teacher

Given a planning domain D, a plan P and a reachability property ψ, we first translate automatically into the modeling language *reactive modules* [AHM+98], 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 since our input language and *reactive modules* are both *finite state-transition system* level languages. The logic ATL admits a formula $\ll A \gg \mathtt{F}\ \psi$ where A is a subset of modules, and ψ is a state predicate. The formula $\ll A \gg \mathtt{F}\ \psi$ asserts that the modules in A can cooperate to reach states satisfying ψ no matter how the remaining modules behave. Considering A as the plan P, the semantics of $\ll A \gg \mathtt{F}\ \psi$ is exactly the same as $D, P \models_{reach} \psi$. Then, we use symbolic ATL model checking of MOCHA, which implements the algorithm shown in Figure 4, 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. $Init_D$ and T_D are the initial state and transition predicates for D, respectively, and $Init_P$ and T_P are the initial state and transition predicates for P, respectively.

The algorithm starts with a set of states satisfying ψ, and identifies all the states where the plan P can force the game to reach the winning area in the next step whatever the planning domain D does. It repeats the

$$
\begin{array}{l}
\textbf{Boolean}\,\mathit{CheckReachGame}(\textbf{Domain}\,D, \textbf{Plan}\,P, \textbf{Prop}\,\psi)\{ \\
\quad \tau_1 := \mathit{false}; \\
\quad \tau_2 := \psi; \\
\quad \textbf{while}\ (\tau_2 \not\rightarrow \tau_1)\ \{ \\
\qquad \tau_1 := \tau_1 \vee \tau_2; \\
\qquad \tau_2 := \exists X'_P \forall X'_D.\ T_0(X_P, X'_P) \wedge \\
\qquad\qquad\qquad (T_1(X_D, X'_D) \rightarrow \tau_1(X'_D, X'_P)); \\
\quad \} \\
\quad \textbf{if}\ ((\mathit{Init}_D(X_D) \wedge \mathit{Init}_P(X_P)) \rightarrow \tau_1)\ \textbf{then return}\ \mathit{true}; \\
\quad \textbf{else return}\ \mathit{false}; \\
\}
\end{array}
$$

Figure 4: Symbolic model checking for $\ll P \gg \mathtt{F}\ \psi$

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* since P has a winning strategy from every initial state. Otherwise, it returns *false*.

3.3 Finding safe strings

Once the test for a permissive plan fails, we search an input/output sequence, which adds more behaviors to the current plan but keeps it from violating a given safety property φ. Given a plan P, the search problem for such a sequence must find a string $\sigma = (o_0, i_0)(o_1, i_1) \cdots \notin L(P)$ such that every run of the planning domain corresponding to σ always stays in states satisfying the safety property φ. However, the search problem is NP-hard.

Proposition 1 *Given a planning domain D, 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 above is 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 find locally 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 input/output pairs (o, i) from every state, since the L^* algorithm has checked membership for every string $\sigma \in R \cdot \Sigma$ and allowed input/output (o, i) from

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

Figure 5: Local search for a safe string

a state q_r if $r \cdot (o,i)$ is a member. In the above, r is the representative string that reaches q_r. However, in our experience, there are often scenarios where a safe input/output sequence $(o,i) \cdot (o',i')$ is disallowed from a particular state q even though (o,i) guarantees that (o',i') is safe. The reason is as following; if q' is the state reached from q on (o,i), then there can be another path to q' that makes the input/output (o',i') from q' unsafe. In this case, we can provide $r \cdot (o,i) \cdot (o',i')$ to the L^* algorithm in order 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 (in the above case, $r \cdot (o,i) \cdot (o',i')$), we check whether from every state, the current conjecture plan allows all the safe input/output sequences of length 2 (by a procedure *LocalSearchSafeString*() in Figure 5); that is, we search $q_r \in Q$ and $(o,i),(o',i') \in S^O \times S^I$ such that $r(o,i)(o',i') \notin L(P)$ and $r(o,i)(o',i')$ is safe, by traversing the plan P and asking membership queries. If there exists $r(o,i)(o',i')$ satisfying the above, we use $r(o,i)(o',i')$ as a safe input/output sequence in order 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 permissive one as well.

4 Experiments

We have implemented our learning technique for the safety/reachability planning problem using a CTL model checker NuSMV [CCG+02] and a ATL model checker MOCHA [AHM+98]. For experiments, we used four planning domains. One is our artificial example *Stack* for which we already knew a simple, safe and permissive plan, and the rest are from existing studies; *MCell* [Fab06], *TLine* [Won99, ZW01] and *Reader/Writer* [Tan92]. In

domain	var	I/O	MQ	EQ	sec	states	S&P
Stack	8	3	710	10	16.2	10	yes
MCell	14	8	26476	13	1970	13	yes
TLine	9	5	16000	22	970	32	yes
R/W	8	3	1008	12	22.3	12	yes

Table 1: Experimental results

MCell, *TLine* and *Reader/Writer*, the original case studies used full observability and/or reachability goals, but we allow partial observability and safety goals natural to the examples. All experiments have been performed on a PC using a 2.4GHz Pentium processor, 2GB memory and a Linux operating system. In all cases, our tool could generate useful and safe permissive plans automatically using little computational resource. The planning domains, safety and reachability specifications, and results are available at `http://www.cis.upenn.edu/~wnam/planning/`.

Even though Bertoli et al. studied a similar problem [BCPT03, BP04] and they have a general planning tool MBP [BCP+03], the public version of MBP does not support *safety and reachability* under *partial observability* at this point. Also, their papers [BCPT03, BP04] has only one example which cannot be applied to our problem setting. Finally, it is unclear how we can compare our method with other techniques.

The results for four planning domains are shown in Table 1. It lists the total number of boolean variables and input/output boolean variables in each planning domain, and shows the number of membership and equivalence queries 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 we constructed. The last column ('S&P') indicates whether the plan is safe and permissive.

In our experiments, our prototype tool learned a safe and permissive plan in 40 minutes for every example. The fact that all the constructed plans were permissive shows that our solution can learn a safe and permissive plan in many cases even though it cannot guarantee to always construct a safe and permissive plan. The running time 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 on the execution time for a model checking problem which corresponds with each query). In each example, the number of variables is modest for current model checkers, our tool takes more time since each query (membership query and equivalence query) is

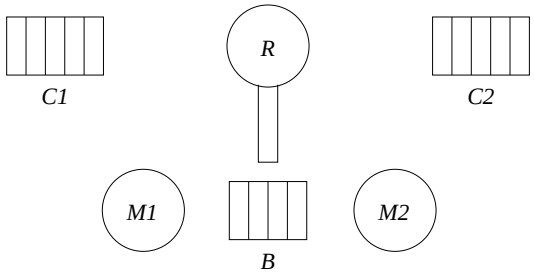

Figure 6: Manufacturing cell

encoded into one model checking problem. For this problem, especially in larger problems where the number of queries needed increases, we can easily adopt a symbolic learning technique which can construct a plan implicitly, as symbolic learning for compositional verification [AMN05, NA06].

Stack We already explained this example in Section 2. For the experiment, we use a model that has a capacity of 10.

MCell A manufacturing cell [Fab06] consists of two machines *M1* and *M2*, and a robot *R*.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, since they have unlimited capacity; the input-conveyor gets its workpieces from environment which is always able to produce a workpiece, and the output-conveyor is always emptied by 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*; from *C1* to *B*, from *B* to *M1* or *M2*, and from *M1* or *M2* to *C2*. The machine *M1* takes nondeterministically 1–3 time units to perform its work, and *M2* takes 1–2 time units nondeterministically. A plan for this planning domain has to control the robot, and a safety property of this example is to avoid *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 be able to force *M1* to eventually run as the reachability requirement.

Transfer line A transfer line [Won99, ZW01] consists of two machines *M1* and *M2*, followed by a test unit *TU*, linked by buffers *B1* and *B2*. Every

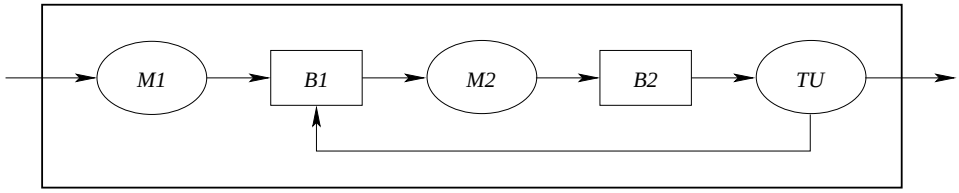

Figure 7: Transfer line

machine, *M1, M2* and *TU*, has two states: *idle* and *busy*. The buffers *B1* and *B2* have a capacity of 3 and 1, respectively.

The machine *M1* takes a raw workpiece from outside to the buffer *B1*, and the machine *M2* shifts a workpiece from *B1* to the buffer *B2*. Similarly, *TU* takes a workpiece from *B2* and checks it. A workpiece 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 we adopt is whether a plan can force a workpiece to be shifted to *TU*.

Reader/writer In a reader/writer system [Tan92], we have one writer-process *W* and two reader-processes, *R1* and *R2*. All of these utilize a mutual critical region. The processes respectively and continuously acquire a 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 read simultaneously, provided that no one is writing. The critical region must therefore be subject to mutual exclusion; the writer-process or only reader-processes at a time.

Once the reader *R1* occupies the critical region, it consumes 3 time units in the region. On the other hand, *R2* takes 2 time units. Both readers are idle for 1 time unit after releasing the region. A plan has to control the writer-process *W* which takes only 1 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.

5 Conclusions

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 learned by our tool may be also a permissive one with respect to a given reachability requirement. To construct such a safe plan, the technique is based on regular language learning with CTL model checking as well as ATL model checking. Our experiments present promising results where the tool learns safe and permissive plans with 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 use our tool more broadly, it is needed to support more expressive goal representation, e.g. LTL and CTL. This extension requires modification of the teacher for the L^* algorithm. Second, it is also worth while to support a standard input language PDDL (Planning Domain Definition Language) [McD89] for planning domains in order to easily compare with other tools and to experiment on more benchmarks.

References

[AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.

[AHM+98] R. Alur, T.A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 516–520, 1998.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of the 17th International Conference of Computer Aided Verification*, pages 548–562, 2005.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[BCP+03] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: A model based planner. In *Proceedings of IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[BCPT03] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A framework for planning with extended goals under partial observability. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, pages 215–224, 2003.

[BCRT01] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 473–478, 2001.

[BG00] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of Artificial Intelligence Planning Systems*, pages 52–61, 2000.

[BK98] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[BP04] P. Bertoli and M. Pistore. Planning with extended goals and partial observability. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, pages 270–278, 2004.

[CCG+02] 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 *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages 359–364, 2002.

[CdGV02] D. Calvanese, G. de Giacomo, and M. Vardi. Reasoning about action and planning in LTL action theories. In *Proceedings of the 8th International Conference on the Principles of Knowledge Representation and Reasoning*, pages 593–602, 2002.

[CM98] S. Cerrito and M. Mayer. Bounded model search in linear temporal logic and its application to planning. In *Proceedings of the 7th International Conference on Analytic Tableaux and Related Methods*, pages 124–140, 1998.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1):35–84, 2003.

[Fab06] M. Fabian. *Discrete Event Systems*. Chalmers University of Technology, 2006.

[FN71] R.E. Fikes and N.J. Nilsson. STRIP: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[KD01] J. Kvarnstrom and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[McD89] D. McDermott. PDDL - The planning domain definition language. Technical report, CVC TR-98-003, Yale Center for Computational Vision and Control, 1989.

[NA06] W. Nam and R. Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis*, pages 170–185, 2006.

[PT01] M. Postore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 479–486, 2001.

[PW92] J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on the Principles of Knowledge Representation and Reasoning*, pages 103–114, 1992.

[Rin99] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[RS93] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 1992.

[WAS98] D. Weld, C. Anderson, and D. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of the*

15th National Conference on Artificial Intelligence, pages 897–904, 1998.

[Won99] W.M. Wonham. *Notes on Control of Discrete-Event Systems*. University of Toronto, 1999.

[ZW01] Z. Zhang and W. Wonham. STCT: An efficient algorithm for supervisory control design. In *Proceedings of Symposium on Supervisory Control of Discrete Event Systems*, pages 82–93, 2001.