Approximate Solutions for the Minimal Revision Problem of Specification Automata

Kangjin Kim and Georgios E. Fainekos

***Abstract*— As robots are being integrated into our daily lives, it becomes necessary to provide guarantees of safe and provably correct operation. Such guarantees can be provided using automata theoretic task and mission planning where the requirements are expressed as temporal logic specifications. However, in real-life scenarios, it is to be expected that not all user task requirements can be realized by the robot. In such cases, the robot must provide feedback to the user on why it cannot accomplish a given task. Moreover, the robot should indicate what tasks it can accomplish which are as "close" as possible to the initial user intent. Unfortunately, the latter problem, which is referred to as minimal specification revision problem, is NP complete. This paper presents an approximation algorithm that can compute good approximations to the minimal revision problem in polynomial time. The experimental study of the algorithm demonstrates that in most problem instances the heuristic algorithm actually returns the optimal solution. Finally, some cases where the algorithm does not return the optimal solution are presented.**

I. Introduction

As robots become mechanically more capable, they are going to be more and more integrated into our daily lives. Non-expert users will have to communicate with the robots in a natural language setting and request a robot or a team of robots to accomplish complicated tasks. Therefore, we need methods that can capture the high-level user requirements, solve the planning problem and map the solution to low level continuous control actions. In addition, such frameworks must come with mathematical guarantees of safe and correct operation for the whole system and not just the high level planning or the low level continuous control.

Linear Temporal Logic (LTL) [1] can provide the mathematical framework that can bridge the gap between (i) natural language and high-level planning algorithms [2], [3], and (ii) high-level planning algorithms and control [4]–[8]. LTL has been utilized as a specification language in a range of robotics applications (see [4]–[14]).

All the previous methods are based on the assumption that the LTL planning problem has a feasible solution. However, in real-life scenarios, it is to be expected that not all complex task requirements can be realized by a robot or a team of robots. In such failure cases, the robot needs to provide feedback to the non-expert user on why the specification failed. Furthermore, it would be desirable that the robot proposes a number of plans that can be realized by the robot and which are as "close" as possible to the initial user intent. Then, the user would be able to understand what are the limitations of the robot and, also, he/she would be able to choose among a number of possible feasible plans.

In [15], we made the first steps towards solving the debugging (i.e., why the planning failed) and revision (i.e., what the robot can actually do) problems for automata theoretic LTL planning [16]. We remark that many robotic applications, e.g., [4], [6], [10]–[12], [14], are utilizing this particular method. In the follow-up paper [17], we studied the theoretical foundations of the specification revision problem when both the system and the specification can be represented by ω-automata [18]. We focused on the Minimal Revision Problem (MRP), i.e., finding the "closest" satisfiable specification to the initial specification, and we proved that the problem is NP-complete even when severely restricting the search space. Furthermore, we presented an encoding of MRP as a satisfiability problem and we demonstrated experimentally that we can quickly get the exact solution to MRP for small problem instances.

In this paper, we revisit the MRP problem that we introduced in [17]. We present a heuristic algorithm that can approximately solve the MRP problem in polynomial time. We experimentally establish that in most cases the heuristic algorithm returns the optimal solution on random problem instances and on LTL planning scenarios from our previous work. Furthermore, we demonstrate that now we can quickly return a solution to the MRP problem on large problem instances. Finally, we provide examples where the algorithm is guaranteed not to return the optimal solution.

Related Research: The automatic specification revision problem for automata based planning techniques is a relatively new problem. Finding out why a specification is not satisfiable on a model is a problem that is very related to the problems of *vacuity* and *coverage* in model checking [19]. In the context of general planners, the problem of finding good excuses on why the planning failed has been studied in [20]. Another related problem is the detection of the causes of unrealizability in LTL games. In this case, a number of heuristics have been developed in order to localize the error and provide meaningful information to the user for debugging [21], [22]. Along these lines, LTLMop [23] was developed to debug unrealizable LTL specifications in reactive planning for robotic applications.

II. Problem Formulation

In this paper, we work with discrete abstractions (Finite State Machines) of the continuous robotic control system [4]. Each state of the Finite State Machine (FSM) $\mathcal{T}$ is labeled

This work has been partially supported by award NSF CNS 1116136.

K. Kim and G. Fainekos are with the School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281, USA {Kangjin.Kim,fainekos}@asu.edu

by a number of symbols from a set $\Pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$ that represent regions in the configuration space [24] of the robot or, more generally, actions that can be performed by the robot. The control requirements for such a system can be posed using specification automata $\mathcal{B}$ with Büchi acceptance conditions [18] also known as ω-automata.

When a specification $\mathcal{B}$ is not satisfiable on a particular system $\mathcal{T}$, then the current motion planning and control synthesis methods, e.g., [4], [10], [14], based on automata theoretic concepts [16] simply return that the specification is not satisfiable without any other user feedback. In such cases, we would like to be able to solve the following problem and provide feedback to the user.

Problem 1 (Minimal Revision Problem (MRP)): Given a system $\mathcal{T}$ and a specification automaton $\mathcal{B}$, if the specification $\mathcal{B}$ cannot be satisfied on $\mathcal{T}$, then find the "closest" specification $\mathcal{B}'$ to $\mathcal{B}$ which can be satisfied on $\mathcal{T}$.

Problem 1 was first introduced in [15] for Linear Temporal Logic (LTL) specifications. In [15], we provided solutions to the debugging and (not minimal) revision problems and we demonstrated that we can easily get a minimal revision of the specification when the discrete controller synthesis phase fails due to unreachable states in the system.

Assumption 1: All the states on $\mathcal{T}$ are reachable.

In [17], we introduced a notion of distance on a restricted space of specification automata and, then, we demonstrated that MRP is in NP-complete. Since brute force search is prohibitive for any reasonably sized problem, we presented an encoding of MRP as a satisfiability problem. Nevertheless, even when utilizing state of the art satisfiability solvers, the size of the systems that we could handle remained small (single robot scenarios in medium complexity environments).

Contributions: In this paper, we provide an approximation algorithm for MRP. The algorithm is based on Dijkstra's single-source shortest path algorithm (DSPA) [25], which can be regarded both as a greedy and a dynamic programming algorithm [26]. We demonstrate through numerical experiments that not only the algorithm returns an optimal solution in various scenarios, but also that it outperforms in computation time our satisfiability based solution.

III. Preliminaries

In this section, we review basic results from [4], [15]–[17]. In detail, we provide a review of the automata theoretic planning and the specification revision problem. Our contributions in Section IV will be founded on these results.

A. Constructing Discrete Controllers

We assume that the combined actions of the robot/team of robots and their operating environment can be represented using an FSM.

Definition 1 (FSM): A Finite State Machine is a tuple $\mathcal{T} = (Q, Q_0, \rightarrow_{\mathcal{T}}, h_{\mathcal{T}}, \Pi)$ where: Q is a set of states; $Q_0 \subseteq Q$ is the set of possible initial states; $\rightarrow_{\mathcal{T}} = E \subseteq Q \times Q$ is the transition relation; and, $h_{\mathcal{T}} : Q \rightarrow \mathcal{P}(\Pi)$ maps each state q to the set of atomic propositions that are true on q.

We define a *path* on the FSM to be a sequence of states and a *trace* to be the corresponding sequence of sets of propositions. Formally, a path is a function $p : \mathbb{N} \rightarrow Q$ such that for each $i \in \mathbb{N}$ we have $p(i) \rightarrow_{\mathcal{T}} p(i+1)$ and the corresponding trace is the function composition $\bar{p} = h_{\mathcal{T}} \circ p : \mathbb{N} \rightarrow \mathcal{P}(\Pi)$. The language $\mathcal{L}(\mathcal{T})$ of $\mathcal{T}$ consists of all possible traces.

In this work, we are interested in the ω-automata that will impose certain requirements on the traces of $\mathcal{T}$. ω-automata differ from the classic finite automata in that they accept infinite strings (traces of $\mathcal{T}$ in our case).

Definition 2: A automaton is a tuple $\mathcal{B} = (S_{\mathcal{B}}, s_0^{\mathcal{B}}, \Omega, \delta_{\mathcal{B}}, F_{\mathcal{B}})$ where: $S_{\mathcal{B}}$ is a finite set of states; $s_0^{\mathcal{B}}$ is the initial state; Ω is an input alphabet; $\delta_{\mathcal{B}} : S_{\mathcal{B}} \times \Omega \rightarrow \mathcal{P}(S_{\mathcal{B}})$ is a transition function; and $F_{\mathcal{B}} \subseteq S_{\mathcal{B}}$ is a set of final states.

When $s' \in \delta_{\mathcal{B}}(s, l)$, we also write $s \xrightarrow{l}_{\mathcal{B}} s'$ or $(s, l, s') \in \rightarrow_{\mathcal{B}}$. A *run* r of $\mathcal{B}$ is a sequence of states $r : \mathbb{N} \rightarrow S_{\mathcal{B}}$ that occurs under an input trace $\bar{p}$ taking values in Ω. That is, for $i = 0$ we have $r(0) = s_0^{\mathcal{B}}$ and for all $i \geq 0$ we have $r(i) \xrightarrow{\bar{p}(i)}_{\mathcal{B}} r(i+1)$. Let $\lim(\cdot)$ be the function that returns the set of states that are encountered infinitely often in the run r of $\mathcal{B}$. Then, a run r of an automaton $\mathcal{B}$ over an infinite trace $\bar{p}$ is *accepting* if and only if $\lim(r) \cap F_{\mathcal{B}} \neq \emptyset$. This is called a Büchi acceptance condition. Finally, we define the language $\mathcal{L}(\mathcal{B})$ of $\mathcal{B}$ to be the set of all traces $\bar{p}$ that have a run that is accepted by $\mathcal{B}$.

A *specification* automaton is an automaton with Büchi acceptance condition where the input alphabet is the powerset of the labels of the system $\mathcal{T}$, i.e., $\Omega = \mathcal{P}(\Pi)$. In order to simplify the discussion in Section III-B, we will be using the following assumptions and notation

- we define the set $E_{\mathcal{B}} \subseteq S_{\mathcal{B}} \times S_{\mathcal{B}}$, such that $(s, s') \in E_{\mathcal{B}}$ iff $\exists l \in \Omega$, $s \xrightarrow{l}_{\mathcal{B}} s'$; and,
- we define the function $\lambda_{\mathcal{B}} : S_{\mathcal{B}} \times S_{\mathcal{B}} \rightarrow \Omega$ which maps a pair of states to the label of the corresponding transition, i.e., if $s \xrightarrow{l}_{\mathcal{B}} s'$, then $\lambda_{\mathcal{B}}(s, s') = l$; and if $(s, s') \notin E_{\mathcal{B}}$, then $\lambda_{\mathcal{B}}(s, s') = \emptyset$.

In brief, our goal is to generate paths on $\mathcal{T}$ that satisfy the specification $\mathcal{B}_{\mathbf{s}}$. In automata theoretic terms, we want to find the subset of the language $\mathcal{L}(\mathcal{T})$ which also belongs to the language $\mathcal{L}(\mathcal{B}_{\mathbf{s}})$. This subset is simply the intersection of the two languages $\mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\mathcal{B}_{\mathbf{s}})$ and it can be constructed by taking the product $\mathcal{T} \times \mathcal{B}_{\mathbf{s}}$ of the FSM $\mathcal{T}$ and the specification automaton $\mathcal{B}_{\mathbf{s}}$. Informally, the automaton $\mathcal{B}_{\mathbf{s}}$ restricts the behavior of the system $\mathcal{T}$ by permitting only certain acceptable transitions. Then, given an initial state in the FSM $\mathcal{T}$, we can choose a particular trace from $\mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\mathcal{B}_{\mathbf{s}})$ according to a preferred criterion.

Definition 3: The product automaton $\mathcal{A} = \mathcal{T} \times \mathcal{B}_{\mathbf{s}}$ is the automaton $\mathcal{A} = (S_{\mathcal{A}}, s_0^{\mathcal{A}}, \mathcal{P}(\Pi), \delta_{\mathcal{A}}, F_{\mathcal{A}})$ where: $S_{\mathcal{A}} = Q \times S_{\mathcal{B}_{\mathbf{s}}}$; $s_0^{\mathcal{A}} = \{(q_0, s_0^{\mathcal{B}_{\mathbf{s}}}) \mid q_0 \in Q_0\}$; $\delta_{\mathcal{A}} : S_{\mathcal{A}} \times \mathcal{P}(\Pi) \rightarrow \mathcal{P}(S_{\mathcal{A}})$ s.t. $(q_j, s_j) \in \delta_{\mathcal{A}}((q_i, s_i), l)$ iff $q_i \rightarrow_{\mathcal{T}} q_j$ and $s_j \in \delta_{\mathcal{B}_{\mathbf{s}}}(s_i, l)$ with $l \subseteq h_{\mathcal{T}}(q_j)$; $F_{\mathcal{A}} = Q \times F$ is the set of accepting states.

Note that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\mathcal{B}_{\mathbf{s}})$. We say that $\mathcal{B}_{\mathbf{s}}$ is *satisfiable* on $\mathcal{T}$ if $\mathcal{L}(\mathcal{A}) \neq \emptyset$. Moreover, finding a satisfying path on $\mathcal{T} \times \mathcal{B}_{\mathbf{s}}$ is an easy algorithmic problem [1]. First, we convert automaton $\mathcal{T} \times \mathcal{B}_\mathbf{s}$ to a directed graph and, then, we find the strongly connected components (SCC) in that graph. If at least one SCC that contains a final state is reachable from an initial state, then there exist accepting (infinite) runs on $\mathcal{T} \times \mathcal{B}_\mathbf{s}$ that have a finite representation. Each such run consists of two parts: **prefix:** a part that is executed only once (from an initial state to a final state) and, **lasso:** a part that is repeated infinitely (from a final state back to itself). Note that if no final state is reachable from the initial or if no final state is within an SCC, then the language $\mathcal{L}(\mathcal{A})$ is empty and, hence, the high level synthesis problem does not have a solution. *Namely, the synthesis phase has failed and we cannot find a system behavior that satisfies the specification.*

B. The Specification Revision Problem

Intuitively, a revised specification is one that can be satisfied on the discrete abstraction of the workspace or the configuration space of the robot. To search for a minimal revision, we define an ordering relation on automata as well as a distance function between automata. Similar to the case of LTL formulas in [15], we do not want to consider the "space" of all possible Büchi automata, but rather the "space" of specification automata which are semantically close to the initial specification automaton $\mathcal{B}_\mathbf{s}$. The later will imply that we remain close to the initial intention of the designer. We propose that this space consists of all the automata that can be derived from $\mathcal{B}_\mathbf{s}$ by removing atomic propositions from the transition input. Our definition of the ordering relation between automata relies on the previous assumption.

Definition 4 (Relaxation): Let $\mathcal{B}_1 = (S_{\mathcal{B}_1}, s_0^{\mathcal{B}_1}, \Omega, \to_{\mathcal{B}_1}, F_{\mathcal{B}_1})$ and $\mathcal{B}_2 = (S_{\mathcal{B}_2}, s_0^{\mathcal{B}_2}, \Omega, \to_{\mathcal{B}_2}, F_{\mathcal{B}_2})$ be two Büchi automata. Then, we say that $\mathcal{B}_2$ is a relaxation of $\mathcal{B}_1$ and we write $\mathcal{B}_1 \preceq \mathcal{B}_2$ if and only if $S_{\mathcal{B}_1} = S_{\mathcal{B}_2} = S$, $s_0^{\mathcal{B}_1} = s_0^{\mathcal{B}_2}$, $F_{\mathcal{B}_1} = F_{\mathcal{B}_2}$ and

1) $\forall (s, l, s') \in \to_{\mathcal{B}_1} - \to_{\mathcal{B}_2} . \exists l'$.
   $(s, l', s') \in \to_{\mathcal{B}_2} - \to_{\mathcal{B}_1}$ and $l' \subseteq l$.
2) $\forall (s, l, s') \in \to_{\mathcal{B}_2} - \to_{\mathcal{B}_1} . \exists l'$.
   $(s, l', s') \in \to_{\mathcal{B}_1} - \to_{\mathcal{B}_2}$ and $l \subseteq l'$.

We remark that if $\mathcal{B}_1 \preceq \mathcal{B}_2$, then $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$ since the relaxed automaton allows more behaviors to occur. If two automata $\mathcal{B}_1$ and $\mathcal{B}_2$ cannot be compared under relation $\preceq$, then we write $\mathcal{B}_1 \parallel \mathcal{B}_2$.

We can now define the set of automata over which we will search for a minimal solution that has nonempty intersection with the system.

Definition 5: Given a system $\mathcal{T}$ and a specification automaton $\mathcal{B}_\mathbf{s}$, the set of *valid relaxations* of $\mathcal{B}_\mathbf{s}$ is defined as $\mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T}) = \{\mathcal{B} \mid \mathcal{B}_\mathbf{s} \preceq \mathcal{B} \text{ and } \mathcal{L}(\mathcal{T} \times \mathcal{B}) \neq \emptyset\}$.

We can now search for a minimal solution in the set $\mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})$. That is, we can search for some $\mathcal{B} \in \mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})$ such that if for any other $\mathcal{B}' \in \mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})$, we have $\mathcal{B}' \preceq \mathcal{B}$, then $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}')$. However, this does not imply that a minimal solution semantically is minimal structurally as well. In other words, it could be the case that $\mathcal{B}_1$ and $\mathcal{B}_2$ are minimal relaxations of some $\mathcal{B}_\mathbf{s}$, but $\mathcal{B}_1 \parallel \mathcal{B}_2$ and, moreover, $\mathcal{B}_1$ requires the modification of only one transition while $\mathcal{B}_2$ requires the modification of two transitions. Therefore, we must define a metric on the set $\mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})$, which accounts for the number of changes from the initial specification automaton $\mathcal{B}_\mathbf{s}$.

Definition 6: Given a system $\mathcal{T}$ and a specification automaton $\mathcal{B}_\mathbf{s}$, we define the distance of any $\mathcal{B} \in \mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})$ from $\mathcal{B}_\mathbf{s}$ to be $\mathbf{dist}_{\mathcal{B}_\mathbf{s}}(\mathcal{B}) = \sum_{(s,s') \in E_{\mathcal{B}_\mathbf{s}}} |\lambda_{\mathcal{B}_\mathbf{s}}(s, s') - \lambda_{\mathcal{B}}(s, s')|$ where $|\cdot|$ is the cardinality of the set.

Therefore, Problem 1 can be restated as:

Problem 2: Given a system $\mathcal{T}$ and a specification automaton $\mathcal{B}_\mathbf{s}$ such that $\mathcal{L}(\mathcal{T} \times \mathcal{B}_\mathbf{s}) = \emptyset$, find $\mathcal{B} \in \arg\min\{\mathbf{dist}_{\mathcal{B}_\mathbf{s}}(\mathcal{B}') \mid \mathcal{B}' \in \mathfrak{R}(\mathcal{B}_\mathbf{s}, \mathcal{T})\}$.

C. Minimal Revision as a Graph Problem

In order to solve Problem 2, we construct a directed labeled graph $G_\mathcal{A}$ from the product automaton $\mathcal{A} = \mathcal{T} \times \mathcal{B}_\mathbf{s}$. The edges of $G_\mathcal{A}$ are labeled by a set of atomic propositions which if removed from the corresponding transition on $\mathcal{B}_\mathbf{s}$, they will enable the transition on $\mathcal{A}$. The overall problem then becomes one of finding the least number of atomic propositions to be removed in order for the product graph to have an accepting run. Next, we provide the formal definition of the graph $G_\mathcal{A}$ which corresponds to a product automaton $\mathcal{A}$ while considering the effect of revisions.

Definition 7: Given a system $\mathcal{T}$ and a specification automaton $\mathcal{B}_\mathbf{s}$, we define the graph $G_\mathcal{A} = (V, E, v_s, V_f, \overline{\Pi}, \Lambda)$, which corresponds to the product $\mathcal{A} = \mathcal{T} \times \mathcal{B}_\mathbf{s}$ as follows: $V = \mathcal{S}$ is the set of nodes; $E = E_\mathcal{A} \cup E_D \subseteq \mathcal{S} \times \mathcal{S}$, where $E_\mathcal{A}$ is the set of edges that correspond to transitions on $\mathcal{A}$, i.e., $((q, s), (q', s')) \in E_\mathcal{A}$ iff $\exists l \in \mathcal{P}(\Pi) . (q, s) \xrightarrow{l}_\mathcal{A} (q', s')$; and E_D is the set of edges that correspond to disabled transitions, i.e., $((q, s), (q', s')) \in E_D$ iff $q \to_\mathcal{T} q'$ and $s \xrightarrow{l}_{\mathcal{B}_\mathbf{s}} s'$ with $l \cap (\Pi - h_\mathcal{T}(q')) \neq \emptyset$; $v_s = s_0^\mathcal{A}$ is the source node; $V_f = F_\mathcal{A}$ is the set of sinks; $\overline{\Pi} = \{\langle \pi, (s, s') \rangle \mid \pi \in \Pi, (s, s') \in E_{\mathcal{B}_\mathbf{s}}\}$; $\Lambda : E \to \mathcal{P}(\overline{\Pi})$ is the edge labeling function such that if $e = ((q, s), (q', s'))$, then $\Lambda(e) = \{\langle \pi, (s, s') \rangle \mid \pi \in (\lambda_{\mathcal{B}_\mathbf{s}}(s, s') - h_\mathcal{T}(q'))\}$.

If $\Lambda(e) \neq \emptyset$, then it specifies those atomic propositions in $\lambda_{\mathcal{B}_\mathbf{s}}(s, s')$ that need to be removed to enable the edge in the product state. Note that the labels of the edges of $G_\mathcal{A}$ are subsets of $\overline{\Pi}$ rather than Π. This is due to the fact that we are looking into removing an atomic proposition π from a specific transition (s, l, s') of $\mathcal{B}_\mathbf{s}$ rather than all π in $\mathcal{B}_\mathbf{s}$.

IV. An Approximation Algorithm for MRP

In this section, we present an approximation algorithm (AAMRP) for the Minimal Revision Problem (MRP). It is based on Dijkstra's shortest path algorithm (DSPA) [25]. The main difference from DSPA is that instead of finding the minimum weight path to reach each node, AAMRP tracks the number of atomic propositions that must be removed from each edge on the paths of the graph $G_\mathcal{A}$.

The pseudocode for the AAMRP is presented in Algorithms 1 and 2. The main algorithm (Alg. 1) divides the problem into two tasks. First, in line 5, it finds an approximation to the minimum number of atomic propositions from $\overline{\Pi}$ that must be removed to have a prefix path to each reachable sink (see Section III-A). Then, in line 10, it repeats the process

Algorithm 1 AAMRP

Inputs: a graph $G_{\mathcal{A}} = (V, E, v_s, V_f, \overline{\Pi}, \Lambda)$.
Outputs: the list L of atomic propositions form $\overline{\Pi}$ that must be removed $\mathcal{B}_\mathbf{s}$.

```
 1: procedure AAMRP($G_{\mathcal{A}}$)
 2:     $L \leftarrow \overline{\Pi}$
 3:     $\mathcal{M}[:,:] \leftarrow (\overline{\Pi}, \infty)$        ▷ Each entry is set to $(\overline{\Pi}, \infty)$
 4:     $\mathcal{M}[v_s,:] \leftarrow (\emptyset, 0)$        ▷ Initialize the source node
 5:     $\langle \mathcal{M}, \mathbf{P}, \mathcal{V} \rangle \leftarrow$ FINDMINPATH($G_{\mathcal{A}}, \mathcal{M}, 0$)
 6:     if $\mathcal{V} \cap V_f = \emptyset$ then
 7:         $L \leftarrow \emptyset$
 8:     else
 9:         for $v_f \in \mathcal{V} \cap V_f$ do
10:             $L_p \leftarrow$ GETAPFROMPATH($v_s, v_f, \mathcal{M}, \mathbf{P}$)
11:             $\mathcal{M}'[:,:] \leftarrow (\overline{\Pi}, \infty)$
12:             $\mathcal{M}'[v_f,:] \leftarrow \mathcal{M}[v_f,:]$
13:             $G'_{\mathcal{A}} \leftarrow (V, E, v_f, \{v_f\}, \overline{\Pi}, L)$
14:             $\langle \mathcal{M}', \mathbf{P}', \mathcal{V}' \rangle \leftarrow$ FINDMINPATH($G'_{\mathcal{A}}, \mathcal{M}', 1$)
15:             if $v_f \in \mathcal{V}'$ then
16:                 $L_l \leftarrow$ GETAPFROMPATH($v_f, v_f, \mathcal{M}', \mathbf{P}'$)
17:                 if $|L_p \cup L_l| \leq |L|$ then
18:                     $L \leftarrow L_p \cup L_l$
19:                 end if
20:             end if
21:         end for
22:     end if
23:     return $L$
24: end procedure
```

The function GETAPFROMPATH($(v_s, v_f, \mathcal{M}, \mathbf{P})$) returns the atomic propositions that must be removed from $\mathcal{B}_\mathbf{s}$ in order to enable a path on $\mathcal{A}$ from a starting state v_s to a final state v_f given the tables $\mathcal{M}$ and $\mathbf{P}$.

Algorithm 2 FINDMINPATH

Inputs: a graph $G_{\mathcal{A}} = (V, E, v_s, V_f, \overline{\Pi}, \Lambda)$, a table $\mathcal{M}$ and a flag $lasso$ on whether this is a lasso path search.
Variables: a queue $\mathcal{Q}$, a set $\mathcal{V}$ of visited nodes and a table $\mathbf{P}$ indicating the parent of each node on a path.
Output: the tables $\mathcal{M}$ and $\mathbf{P}$ and the visited nodes $\mathcal{V}$

```
 1: procedure FINDMINPATH($G_{\mathcal{A}}$,$\mathcal{M}$,$lasso$)
 2:     $\mathcal{V} \leftarrow \{v_s\}$; $\mathcal{Q} \leftarrow V - \{v_s\}$
 3:     $\mathbf{P}[:] \leftarrow \emptyset$        ▷ Each entry of $\mathbf{P}$ is set to $\emptyset$
 4:     for $v \in V$ such that $(v_s, v) \in E$ and $v \neq v_s$ do
 5:         $\langle \mathcal{M}, \mathbf{P} \rangle \leftarrow$ RELAX($(v_s, v), \mathcal{M}, \mathbf{P}, \Lambda$)
 6:     end for
 7:     if $lasso = 1$ then
 8:         if $(v_s, v_s) \in E$ then
 9:             $\mathcal{M}[v_s, 1] \leftarrow \mathcal{M}[v_s, 1] \cup \Lambda(v_s, v_s)$
10:             $\mathcal{M}[v_s, 2] \leftarrow |\mathcal{M}[v_s, 1] \cup \Lambda(v_s, v_s)|$
11:             $\mathbf{P}[v_s] = v_s$
12:         else
13:             $\mathcal{M}[v_s,:] \leftarrow (\overline{\Pi}, \infty)$
14:         end if
15:     end if
16:     while $\mathcal{Q} \neq \emptyset$ do
17:                 ▷ Get node $u$ with minimum $\mathcal{M}[u, 2]$
18:         $u \leftarrow$ EXTRACTMIN($\mathcal{Q}$)
19:         if $\mathcal{M}[u, 2] < \infty$ then
20:             $\mathcal{V} \leftarrow \mathcal{V} \cup \{u\}$
21:             for $v \in V$ such that $(u, v) \in E$ do
22:                 $\langle \mathcal{M}, \mathbf{P} \rangle \leftarrow$ RELAX($(u, v), \mathcal{M}, \mathbf{P}, \Lambda$)
23:             end for
24:         end if
25:     end while
26:     return $\mathcal{M}$, $\mathbf{P}$, $\mathcal{V}$
27: end procedure
```

from each reachable final state to find an approximation to the minimum number of atomic propositions that must be removed so that a lasso path is enabled. The combination of prefix/lasso that removes the minimal number of atomic propositions is returned to the user.

Algorithm 2 follows closely DSPA [25]. It maintains a list of visited nodes $\mathcal{V}$ and a table $\mathcal{M}$ indexed by the graph vertices which stores the set of atomic propositions that must be removed in order to reach a particular node on the graph. Given a node v, the size of the set $|\mathcal{M}[v, 1]|$ is an upper bound on the minimum number of atomic propositions that must be removed. That is, if we remove all $\overline{\pi} \in \mathcal{M}[v, 1]$ from $\mathcal{B}_\mathbf{s}$, then we enable a simple path (i.e., with no cycles) from a starting state to the state v. The size of $|\mathcal{M}[v, 1]|$ is stored in $\mathcal{M}[v, 2]$ which also indicates that the node v is reachable when $\mathcal{M}[v, 2] < \infty$.

The algorithm works by maintaining a queue with the unvisited nodes on the graph. Each node v in the queue has as key the number of atomic propositions that must be removed so that v becomes reachable on $\mathcal{A}$. The algorithm proceeds by choosing the node with the minimum number of atomic propositions discovered so far (line 18). Then, this node is used in order to updated the estimates for the minimum number of atomic propositions needed in order to reach its neighbors (line 22). A notable difference of Alg. 2 from DSPA is the check for lasso paths in lines 7-15. After the source node is used for updating the estimates of its neighbors, its own estimate for the minimum number of atomic propositions is updated either to the value indicated by the self loop or the maximum possible number of atomic propositions. This is required in order to compare the different paths that reach a node from itself.

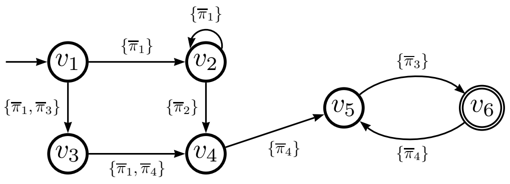

Fig. 1. The graph of Example 1. The source $v_s = v_1$ is denoted by an arrow and the sink v_6 by double circle ($V_f = \{v_6\}$).

The following example demonstrates how the algorithm

Algorithm 3 RELAX

Inputs: an edge (u,v), the tables $\mathcal{M}$ and $\mathbf{P}$ and the edge labeling function Λ

Output: the tables $\mathcal{M}$ and $\mathbf{P}$

```
1: procedure RELAX($(u,v)$,$\mathcal{M}$,$\mathbf{P}$,$\Lambda$)
2:     if $|\mathcal{M}[u,1] \cup \Lambda(u,v)| < \mathcal{M}[v,2]$ then
3:         $\mathcal{M}[v,1] \leftarrow \mathcal{M}[u,1] \cup \Lambda(u,v)$
4:         $\mathcal{M}[v,2] \leftarrow |\mathcal{M}[u,1] \cup \Lambda(u,v)|$
5:         $\mathbf{P}[v] \leftarrow u$
6:     end if
7:     return $\mathcal{M}$, $\mathbf{P}$
8: end procedure
```

works and indicates the structural conditions on the graph that make the algorithm non-optimal.

Example 1: Let us consider the graph in Fig. 1. The source node of this graph is $v_s = v_1$ and the set of sink nodes is $V_f = \{v_6\}$. The $\overline{\Pi}$ set of this graph is $\{\overline{\pi}_1, \ldots, \overline{\pi}_4\}$. Consider the first call of FINDMINPATH (line 5 of Alg. 1).

- Before the first execution of the while loop (line 16): The queue contains $\mathcal{Q} = \{v_2, \ldots, v_6\}$. The table $\mathcal{M}$ has the following entries: $\mathcal{M}[v_1,:] = \langle\emptyset, 0\rangle$, $\mathcal{M}[v_2,:] = \langle\{\overline{\pi}_1\}, 1\rangle$, $\mathcal{M}[v_3,:] = \langle\{\overline{\pi}_1, \overline{\pi}_3\}, 2\rangle$, $\mathcal{M}[v_4,:] = \ldots = \mathcal{M}[v_6,:] = \langle\overline{\Pi}, \infty\rangle$.
- Before the second execution of the while loop (line 16): The node v_2 was popped from the queue since it had $\mathcal{M}[v_2,2] = 1$. The queue now contains $\mathcal{Q} = \{v_3, \ldots, v_6\}$. The table $\mathcal{M}$ has the following rows: $\mathcal{M}[v_1,:] = \langle\emptyset, 1\rangle$, $\mathcal{M}[v_2,:] = \langle\{\overline{\pi}_1\}, 1\rangle$, $\mathcal{M}[v_3,:] = \langle\{\overline{\pi}_1, \overline{\pi}_3\}, 2\rangle$, $\mathcal{M}[v_4] = \langle\{\overline{\pi}_1, \overline{\pi}_2\}, 2\rangle$, $\mathcal{M}[v_5,:] = \mathcal{M}[v_6,:] = \langle\overline{\Pi}, \infty\rangle$.
- At the end of FINDMINPATH (line 27): The queue now is empty. The table $\mathcal{M}$ has the following rows: $\mathcal{M}[v_1,:] = \langle\emptyset, 0\rangle$, $\mathcal{M}[v_2,:] = \langle\{\overline{\pi}_1\}, 1\rangle$, $\mathcal{M}[v_3,:] = \langle\{\overline{\pi}_1, \overline{\pi}_3\}, 2\rangle$, $\mathcal{M}[v_4,:] = \langle\{\overline{\pi}_1, \overline{\pi}_2\}, 2\rangle$, $\mathcal{M}[v_5,:] = \langle\{\overline{\pi}_1, \overline{\pi}_2, \overline{\pi}_4\}, 3\rangle$, $\mathcal{M}[v_6,:] = \langle\overline{\Pi}, 4\rangle$, which corresponds to the path v_1, v_2, v_4, v_5, v_6.

Note that algorithm returns a set of atomic propositions $L_p = \overline{\Pi}$ which is not optimal ($|L_p| = 4$). The path v_1, v_3, v_4, v_5, v_6 would return $L_p = \{\overline{\pi}_1, \overline{\pi}_3, \overline{\pi}_4\}$ with $|L_p| = 3$. $\triangle$

Correctness: The correctness of AAMRP is based on the fact that a node $v \in V$ is reachable on $G_{\mathcal{A}}$ if and only if $\mathcal{M}[v,2] < \infty$. The argument for this claim is similar to the proof of correctness of DSPA in [25]. If AAMRP returns a set of atomic propositions L which are removed from $\mathcal{B}_{\mathbf{s}}$, then the language $\mathcal{L}(\mathcal{A})$ is non-empty. This is immediate by the construction of the graph $G_{\mathcal{A}}$ (Def. 7).

Running time: The running time analysis of the AAMRP is similar to that of DSPA. In the following, we will abuse notation when we use the O notation and treat each set symbol S as its cardinality $|S|$.

First, we will consider FINDMINPATH. The fundamental difference of AAMRP over DSPA is that we have set theoretic operations. We will assume that we are using a data structure for sets that supports $O(1)$ set cardinality quarries, $O(\log n)$ membership quarries and element insertions [25] and $O(n)$ set up time. Under the assumption that $\mathcal{Q}$ is implemented in such a data structure, each EXTRACTMIN takes $O(\log V)$ time. We have $O(V)$ such operations (actually $|V|-1$) for a total of $O(V \log V)$.

Setting up the data structure for $\mathcal{Q}$ will take $O(V)$ time. Furthermore, in the worst case, we have a set $\Lambda(e)$ for each edge $e \in E$ with set-up time $O(E\overline{\Pi})$. Note that the initialization of $\mathcal{M}[v,:]$ to $\langle\overline{\Pi}, \infty\rangle$ does not have to be implemented since we can have indicator variables indicating when a set is supposed to contain all the (known in advance) elements.

Assuming that E is stored in an adjacency list, the total number of calls to RELAX at lines 4 and 21 of Alg. 2 will be $O(E)$ times. Each call to RELAX will have to perform a union of two sets ($\mathcal{M}[u,1]$ and $\Lambda(u,v)$). Assuming that both sets have in the worst case $|\overline{\Pi}|$ elements, each union will take $O(\overline{\Pi} \log \overline{\Pi})$ time. Finally, each set size quarry takes $O(1)$ time and updating the keys in $\mathcal{Q}$ takes $O(\log V)$ time. Therefore, the running time of FINDMINPATH is $O(V + E\overline{\Pi} + V \log V + E(\overline{\Pi} \log \overline{\Pi} + \log V))$.

Note that under Assumption 1 all nodes of $\mathcal{T}$ are reachable ($|V| < |E|$), the same property does not hold for the product automaton. (e.g, think of an environment $\mathcal{T}$ and a specification automaton whose graphs are Directed Acyclic Graphs (DAG). However, we still have ($|V| < |E|$). FINDMINPATH takes $O(E(\overline{\Pi} \log \overline{\Pi} + \log V))$. Therefore, we observe that the running time also depends on the size of the set $\overline{\Pi}$. However, such a bound is very pessimistic since not all the edges will be disabled on $\mathcal{A}$ and, moreover, most edges will not have the whole set $\overline{\Pi}$ as candidates for removal.

Finally, we consider AAMRP. The loop at line 9 is going to be called $O(V_f)$ times. At each iteration, FINDMINPATH is called. Furthermore, each call to GETAPFROMPATH is going to take $O(V\overline{\Pi} \log \overline{\Pi})$ time (in the worst case we are going to have $|V|$ unions of sets of atomic propositions). Therefore, the running time of AAMRP is $O(V_f(V\overline{\Pi} \log \overline{\Pi} + E(\overline{\Pi} \log \overline{\Pi} + \log V))) = O(V_f E(\overline{\Pi} \log \overline{\Pi} + \log V))$ which is polynomial in the size of the input graph.

Approximation bounds: Example 1 can be modified to demonstrate that AAMRP does not have a constant approximation ratio on arbitrary graphs. It is also easy to see that AAMRP always returns the optimal solution on directed trees. There is also a special case, where AAMRP returns a solution whose size is twice the size of the optimal solution.

Theorem 1: AAMRP on planar Directed Acyclic Graphs (DAG) where all the paths merge on the same node is a 2-approximation algorithm.

V. Examples and Numerical Experiments

In this section, we present experimental results using our prototype implementation of AAMRP. The prototype implementation is in Python. Therefore, we expect the running times to substantially improve with a C implementation using state-of-the-art data structure implementations.

For the experiments, we utilized the ASU super computing center which consists of clusters of Dual 4-core processors, 16 GB Intel(R) Xeon(R) CPU X5355 @2.66 Ghz running CentOS 5.5. Our implementations do not utilize the parallel architecture. The clusters were used to run the many different test cases in parallel.

In order to experimentally demonstrate the approximation ratio of AAMRP, we compared the solutions returned by AAMRP with our Answer Set Programming (ASP) implementation of MRP that we developed in [17]. The ASP implementation is guaranteed to return a minimal solution to the MRP problem.

Example 2 (Robot Motion Planning): We revisit our example introduced in [17]. The product automaton of this example has 85 states, 910 transitions and 17 reachable final states. It takes 0.095 sec by AAMRP and 0.699 sec by ASP. AAMRP returns the set of atomic propositions $\{\langle \pi_4, (s_2, s_4)\rangle\}$ as a minimal revision to the problem, which is revision(3) among the three minimal revisions of the example. Thus, it is an optimal solution. $\triangle$

We performed a large number of experimental comparisons on random benchmark instances of various sizes. The first series of experiments involved randomly generated DAGs. Each test case consisted of two randomly generated DAGs which represented an environment and a specification. Both graphs have self-loops on their leaves so that a feasible lasso path can be found. The number of atomic propositions in each instance was equal to four times the number of nodes in each acyclic graph. For example, in the benchmark where the graph had 9 nodes, each DAG had 3 nodes, and the number of atomic propositions was 12. The sinks are chosen randomly and they represent 5%-40% of the nodes. The number of edges in most instances were 2-3 times more than the number of nodes.

Table I compares the results of the ASP solver with the results of AAMRP on test cases of different sizes (total number of nodes). For each graph size, we performed 200 tests and we report minimum, average and maximum computation times in sec. Both algorithms were able to finish the computation and return a minimal revision for all the test cases. Nevertheless, in the large problem instances, AAMRP achieved a 10 time speed-up on the average running time. An interesting observation is that the maximum approximation ratio is experimentally determined to be 2 which validates the theoretical analysis.

For the next experiment, each test case was a cross product graph of two bidirectional spanning trees. One represents the environment and the other the specification. The number of atomic propositions was equal to two times the number of nodes in each spanning tree. For example, in the instance having 9 nodes, each spanning tree had 3 nodes, and, thus, it had 6 atomic propositions. Each instance had 5%-40% of sinks. The number of edges in most instances was three times more then the number of nodes in each instance.

The results are presented in Table II. The observations on the results are similar to the previous experiments. However, we remark that in this case ASP was not able to provide an answer to all the test cases within a 2hr window. The comparison for the approximation ratio was possible only for the test cases where ASP successfully completed the computation. In this case, in the large problem instances, AAMRP achieved at least 100x speed-up on average running time.

Finally, we attempted to determine the problem sizes that our prototype implementation can handle. The results are presented in Table III. We observe that approximately 60,025 nodes would be the limit of the AAMRP implementation in Python. However, we expect that we can achieve at least a 10 times speed up with a C implementation and we plan to pursue this direction in the future.

VI. Conclusions

In this paper, we provided a polynomial time approximation algorithm for the problem of minimal revision of specification automata. The minimal revision problem is useful when automata theoretic planning fails and the modification of the environment is not possible. In such cases, it is desirable that the user receives feedback from the system on what the system can actually achieve. The challenge in proposing a new specification automaton is that the new specification should be as close as possible to the initial intent of the user. Our proposed algorithm experimentally achieves approximation ratio very close to 1. Furthermore, the running time of our prototype implementation is reasonable enough to be able to handle realistic scenarios.

Future research will proceed along several directions. Since the initial specification is ultimately provided in some form of natural language, we would like the feedback that we provide to be in a natural language setting as well. Second, we would like to study whether a constant factor approximation algorithm exists for the general case. Finally, we plan on developing a robust and efficient publicly available implementation of our approximation algorithm.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: MIT Press, 1999.

[2] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Translating structured english to robot controllers," *Advanced Robotics*, vol. 22, no. 12, pp. 1343–1359, 2008.

[3] J. Dzifcak, M. Scheutz, C. Baral, and P. Schermerhorn, "What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution," in *Proceedings of the IEEE international conference on robotics and automation*, 2009.

[4] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, no. 2, pp. 343–352, Feb. 2009.

[5] S. Karaman, R. Sanfelice, and E. Frazzoli, "Optimal control of mixed logical dynamical systems with linear temporal logic specifications," in *IEEE Conf. on Decision and Control*, 2008.

[6] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals," in *International Conference on Robotics and Automation*. IEEE, 2010, pp. 2689–2696.

[7] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon control for temporal logic specifications," in *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*. New York, NY, USA: ACM, 2010, pp. 101–110.

[8] P. Roy, P. Tabuada, and R. Majumdar, "Pessoa 2.0: a controller synthesis tool for cyber-physical systems," in *Proceedings of the 14th international conference on Hybrid systems: computation and control*, ser. HSCC '11. New York, NY, USA: ACM, 2011, pp. 315–316.

Nodes	ASP				AAMRP				RATIO		
	min	avg	max	succ	min	avg	max	succ	min	avg	max
9	0.003	0.0071	0.012	200/200	0.013	0.0157	0.025	200/200	1	1.00667	2
100	0.099	0.1954	1.405	200/200	0.027	0.06727	0.09	200/200	1	1.000625	1.125
196	0.335	1.25058	6.003	200/200	0.057	0.22372	0.289	200/200	1	1	1
324	0.869	5.3316	14.731	200/200	0.113	0.6601	0.912	200/200	1	1.001417	1.2
400	1.267	12.87	35.58	200/200	0.131	1.28913	1.351	200/200	1	1	1
529	3.086	34.1642	103.638	200/200	0.37	3.107	4.141	200/200	1	1	1

TABLE I

NUMBER OF NODES VERSUS THE RESULTS OF ASP SOLVER AND AAMRP. UNDER THE ASP AND AAMRP COLUMNS THE NUMBERS INDICATE COMPUTATION TIMES IN sec. RATIO INDICATES THE EXPERIMENTALLY OBSERVED APPROXIMATION RATIO TO THE OPTIMAL SOLUTION.

Nodes	ASP				AAMRP				RATIO		
	min	avg	max	succ	min	avg	max	succ	min	avg	max
9	0.005	0.0097	0.039	200/200	0.012	0.0153	0.0449	200/200	1	1	1
100	0.378	18.4679	3502.343	200/200	0.028	0.063	0.09	200/200	1	1.001	1.2
196	3.336	31.995	685.819	167/200	0.0439	0.203	0.249	200/200	1	1	1
306	9.801	75.524	2795.337	149/200	0.101	0.5493	0.7	200/200	1	1.000839	1.125
400	21.744	124.7486	164.5459	148/200	0.134	1.124	1.2929	200/200	1	1	1
506	58.67	241.167	1054.98	152/200	0.2329	2.0795	1.821	200/200	1	1.002193	1.333333

TABLE II

NUMBER OF NODES VERSUS THE RESULTS OF ASP SOLVER AND AAMRP. UNDER THE ASP AND AAMRP COLUMNS THE NUMBERS INDICATE COMPUTATION TIMES IN sec. RATIO INDICATES THE EXPERIMENTALLY OBSERVED APPROXIMATION RATIO TO THE OPTIMAL SOLUTION.

Nodes	ASP				AAMRP				RATIO
	min	avg	max	succ	min	avg	max	succ	
1024	24.438	168.2133	237.758	10/10	0.125	0.23	0.325	9/10	1
10000				0/10	15.723	76.164	128.471	9/10	
20164				0/10	50.325	570.737	1009.675	8/10	
50176				0/10	425.362	1993.449	4013.717	3/10	
60025				0/10	6734.133	6917.094	7100.055	2/10	

TABLE III

NUMBER OF NODES VERSUS THE RESULTS OF ASP SOLVER AND AAMRP. UNDER THE ASP AND AAMRP COLUMNS THE NUMBERS INDICATE COMPUTATION TIMES IN sec. RATIO INDICATES THE EXPERIMENTALLY OBSERVED APPROXIMATION RATIO TO THE OPTIMAL SOLUTION.

[9] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal logic based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370 – 1381, 2009.

[10] M. Kloetzer and C. Belta, "Automatic deployment of distributed teams of robots from temporal logic specifications," *IEEE Transactions on Robotics*, vol. 26, no. 1, pp. 48–61, 2010.

[11] L. Bobadilla, O. Sanchez, J. Czarnowski, K. Gossman, and S. LaValle, "Controlling wild bodies using linear temporal logic," in *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[12] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, "Optimal multi-robot path planning with temporal logic constraints," in *IEEE/RSJ International Conference on Intelligent Robots and Systems,*, 2011, pp. 3087 –3092.

[13] B. Lacerda and P. Lima, "Designing petri net supervisors from ltl specifications," in *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[14] A. LaViers, M. Egerstedt, Y. Chen, and C. Belta, "Automatic generation of balletic motions," *IEEE/ACM International Conference on Cyber-Physical Systems*, vol. 0, pp. 13–21, 2011.

[15] G. E. Fainekos, "Revising temporal logic specifications for motion planning," in *Proceedings of the ICRA*, May 2011.

[16] G. D. Giacomo and M. Y. Vardi, "Automata-theoretic approach to planning for temporally extended goals," in *European Conference on Planning*, ser. LNCS, vol. 1809. Springer, 1999, pp. 226–238.

[17] K. Kim, G. Fainekos, and S. Sankaranarayanan, "On the revision problem of specification automata," in *Proceedings of the IEEE Conference on Robotics and Automation*, May 2012.

[18] J. R. Buchi, "Weak second order arithmetic and finite automata," *Zeitschrift für Math. Logik und Grundlagen Math.*, vol. 6, pp. 66–92, 1960.

[19] O. Kupferman, W. Li, and S. A. Seshia, "A theory of mutations with applications to vacuity, coverage, and fault tolerance," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008, pp. 25:1–25:9.

[20] Moritz Göbelbecker and Thomas Keller and Patrick Eyerich and Michael Brenner and Bernhard Nebel, "Coming up with good excuses: What to do when no plan can be found," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling*. AAAI, 2010, pp. 81–88.

[21] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev, "Diagnostic information for realizability," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, F. Logozzo, D. Peled, and L. Zuck, Eds. Springer, 2008, vol. 4905, pp. 52–67.

[22] R. Konighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications using simple counterstrategies," in *Formal Methods in Computer-Aided Design*. IEEE, nov 2009, pp. 152 –159.

[23] V. Raman and H. Kress-Gazit, "Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP," in *Computer Aided Verification*, ser. LNCS, vol. 6806. Springer, 2011, pp. 663–668.

[24] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006. [Online]. Available: http://msl.cs.uiuc.edu/planning/

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press/McGraw-Hill, Sep 2001.

[26] M. Sniedovich, "Dijkstras algorithm revisited: the dynamic programming connexion," *Control and Cybernetics*, vol. 35, no. 3, pp. 599–620, 2006.