# Adapting Specifications for Reactive Controllers

all names
*dept. name of organization (of Aff.)*
*name of organization (of Aff.)*
City, Country
email address

***Abstract*—For systems to respond to scenarios that were unforeseen at design time, they must be capable of safely adapting, at runtime, the assumptions they make about the environment, the goals they are expected to achieve, and the strategy that guarantees the goals are fulfilled if the assumptions hold. Such adaptation often involves the system degrading its functionality, by weakening its environment assumptions and/or the goals it aims to meet, ideally in a graceful manner. However, finding weaker assumptions that account for the unanticipated behaviour and of goals that are achievable in the new environment in a systematic and safe way remains an open challenge.**

**In this paper, we propose a novel framework that supports assumption and, if necessary, goal degradation to allow systems to cope with runtime assumption violations. The framework, which integrates into the MORPH reference architecture, combines symbolic learning and reactive synthesis to compute implementable controllers that may be deployed safely. We describe and implement an algorithm that illustrates the working of this framework. We further demonstrate in our evaluation its effectiveness and applicability to a series of benchmarks from the literature. The results show that the algorithm successfully learns realizable specifications that accommodate previously violating environment behaviour in almost all cases. Exceptions are discussed in the evaluation.**

## I. Introduction

For software systems to be truly capable of self-adaptation at runtime, they must be capable of responding to scenarios that were unforeseen at design time. To do this, they must be endowed with the ability to reason, at runtime, about their system capabilities, their goals, and the assumptions made about the behaviour of their environment [1]. Such reasoning is also the essence of requirements engineering [2]. Indeed, self-adaptation is akin to having a system perform requirements engineering at runtime. In terms of Jackson and Zave's machine-world model [3], the system must be capable of exploring *specifications* (i.e., pairs of assumptions and guarantees $\langle \mathcal{A}, \mathcal{G} \rangle$) that are realizable [4] with respect to the system's capabilities or *interface*. We say a specification $\langle \mathcal{A}, \mathcal{G} \rangle$ is realizable if the system has a way of guaranteeing the goal $\mathcal{G}$ through its interface (by monitoring the environment using its sensors and reacting through its actuators) as long as the environment's behaviour is within the assumptions $\mathcal{A}$.

The system must therefore be capable of automatically synthesizing an operational strategy (c.f., operationalization [5]) that adheres to the adapted specification. That is, it must not only find $\langle \mathcal{A}, \mathcal{G} \rangle$ but must also compute a strategy to achieve $\mathcal{G}$ assuming $\mathcal{A}$. It must be able to execute that strategy while continuing to be aware of new unexpected violations of $\mathcal{A}$ (c.f., runtime monitoring). Upon detecting further assumption violations, it must be capable of evolving the specification, if necessary, in such a way that a new strategy can be synthesized.

In this paper, we propose an approach that supports assumption and goal graceful degradation so that software systems can cope with runtime assumption violations. We focus on system's whose specifications are expressible in a special class of temporal logic language called GR(1)[6]. In particular, the form of runtime adaptation that we envisage is as follows. Consider an unexpected scenario in which the original (design-time) assumptions for which the system was developed are violated during runtime. A system monitors its assumptions and, upon discovering a violation, must first weaken these assumptions to make them consistent with the observed behaviour and then identify what specification it can achieve that continues to guarantee the current goals under the updated assumptions. Should guaranteeing the goals be impossible, the system must be capable of weakening the goals (i.e., degrading the services provided) and identifying a new specification that can guarantee the weakened goals under the updated assumptions. The specification must then be operationalized and executed. Of course, revising goals and assumptions and re-synthesizing a strategy for them may take time and thus pre-defined fail-safe, fall-back, or quiescent operational modes are still required during the change. However, given the potential automatic recovery mechanism that learning and synthesis supports, remaining in such modes would only be temporary.

As argued by many, self-adaptation requires an explicit representation at runtime of goals and assumptions using models. These runtime models can be used to monitor, verify and anticipate runtime behaviour. There is a significant body of work on runtime monitoring [7] and verification [8]. However, the challenging problem of how to evolve goals and assumptions at runtime has been studied to a lesser extent. Existing work proposes defining rules at design time on how the models may be evolved, e.g., see [9]. Thus, requirement specification adaptation is constrained by how well these possible evolutions were anticipated at design time. Work that identifies quantitative parameters that can be adjusted to adapt non-functional requirements [10] also require identifying these parameters at design time.

*We are interested in the challenge of developing general automated reasoning techniques that can evolve a system's goals and assumptions without requiring these evolutions to be anticipated at design time.*

In summary, the contributions of this paper are: $i$) introduction of a novel framework that supports assumption weakening and goal degradation to allow software systems to cope with runtime assumption violations. $ii$) An algorithm for implementing the proposed framework using symbolic learning and automated synthesis of reactive controllers. $iii$) A report on experimentation with the approach that shows the algorithm successfully adapting specifications to environment violations.

## II. A Framework for Adaptation based on Learning and Controller Synthesis

We propose a framework for supporting specification adaptation using symbolic learning and controller synthesis. As an execution environment, we build on the Morph [11] reference architecture and refine its adaptation layer explaining how learning and synthesis can work together to adapt to unexpected assumption violations.

To illustrate the framework, we consider the classic mine pump example originally introduced in [12]. We use this example given its simple to understand on one end, and it also allows us to demonstrate the various issues that such a framework must be capable of handling.

The mine pump case study describes a simple control system for preventing flooding in a mineshaft. The controller is responsible for turning the pump on so that the water level in the mine is kept within safe bounds. There are also safety requirements that the controller must meet when methane is present in the mine. The goals $\mathcal{G}$ the system must achieve are as $g_1$: "The pump shall be off when methane is present in the mine." and $g_2$: "The pump shall be on when the water reaches high levels in the mine". The assumptions $\mathcal{A}$ are $a_1$: "The water level decreases gradually when the pump is on." and for the sake of illustration $a_2$: "Methane presence and high water levels never occur at the same time."

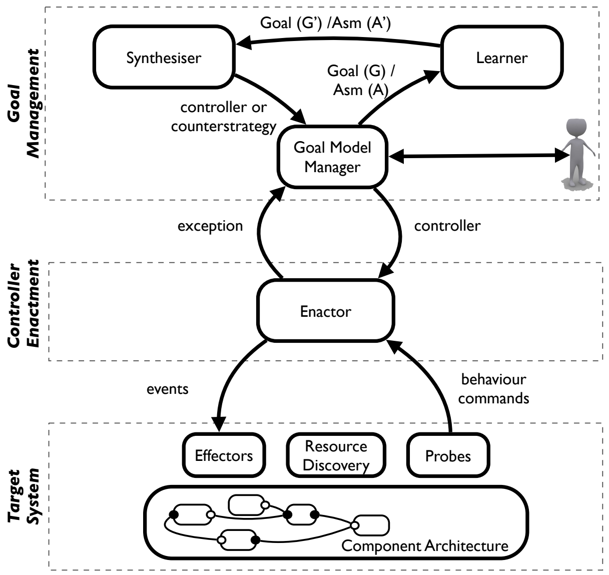


Fig. 1. The Morph Reference Architecture for Adaptive Systems.

We assume that the architecture of a software with adaptive capabilities (depicted in Fig. 1) has a core component, the Enactor, which is loaded with a reactive controller. The Enactor monitors the states of the various system components and their environment (e.g., high water levels are reached) and triggers actions controlled by the software components based on the system's current state (e.g., turns the pump on).

While the system is running, if all assumptions are valid, the controller is guaranteed to achieve its goal. When an unexpected event is received (e.g., high water and methane detected at the same time), the Enactor raises an exception to the Goal Management layer whose main responsibility is to evolve the knowledge it has about the system's environment and goals, and to use the updated knowledge to produce a new controller deployable in the Enactor. Our main contribution lies

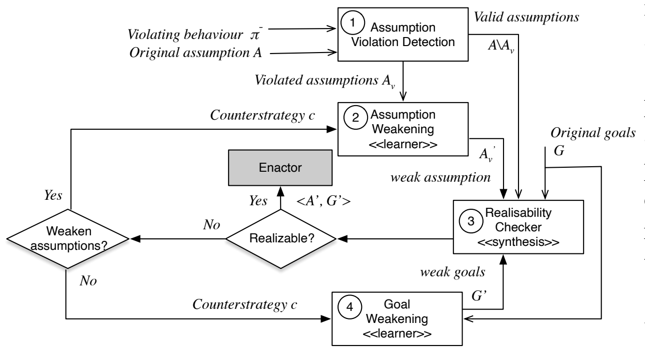


Fig. 2. Goal Management Workflow Sketch

in orchestrating and implementing the functions of the Goal Management layer.

The Goal Management comprises three main components. The *goal manager component* in the Goal Management must deal with the fact that the system is running a controller built for a specification $\langle \mathcal{A}, G \rangle$ which is now known to be invalid as behaviour $\bar{\pi}$ has been observed which is inconsistent with the assumptions. To do so, it manages interactions between two other components—*specification learner* and *synthesizer*. The three components collectively aim to produce a weakened specification $\langle \mathcal{A}', \mathcal{G}' \rangle$ that is consistent with $\bar{\pi}$ and is realizable, i.e., there exists a controller that can achieve $G'$. Human intervention can play a role in the weakening process to indicate preference between different realizable weakened specifications. This may be particularly relevant for selecting between alternative weakened system goals.

Fig. 2 shows an abstract representation of how the learner and synthesizer interact. The learner is responsible for finding candidate weakenings to the assumptions and/or goals (as instructed by the goal manager). The assumption weakening step takes as input the original invalid specification $\langle \mathcal{A}, \mathcal{G} \rangle$ and the violating behaviour $\bar{\pi}$ observed from the environment. The learner outputs a new specification $\langle \mathcal{A}', \mathcal{G} \rangle$ in which the assumptions now are consistent with $\bar{\pi}$. Nonetheless, the specification may still be unrealisable. The synthesizer checks if the new specification is relalizable. If not, it provides a counterexample (technically referred to as a *counterstrategy*).

Counterstrategies are then used in consecutive iterations by the learner to find alternative weakenings to the assumptions or goals. The learner and synthesizer interactions form a loop where the synthesizer checks realizability and the learner attempts to modify the specification to achieve realisability when counterstrategies are identified. Once a realizable specification is found (e.g., where the original assumption ($a_2$) about joint presence of high water and methane is weakened to a trivially valid assumption, e.g., allowing methane to present when there is highwater, and the goal ($g_2$) is degraded to "The pump shall be on when the water reaches high levels and there is no methane present in the mine" , a new controller is deployed. This process of specification weakening is initiated by the goal manager whenever a new assumption violation(s) is detected.

## III. Background

We introduce the notations and terminologies we use to describe our specification adaptation approach.

### A. LTL over Infinite and Finite Sequences

The syntax of Linear Temporal Logic (LTL) is defined over a finite non-empty set of propositional variables *AP*, Boolean connectives $\neg, \wedge$ and $\rightarrow$ and temporal operators $\circ$ (next), $\mathbf{G}$ (always), and $\mathbf{U}$ (strong until). A well-formed LTL formula is constructed as follows.

$$\phi := \top \mid \bot \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \circ\,\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi_1 \mathbf{U} \phi_2$$

A formula $p$ or its negation is called a *literal*. The semantics of LTL is traditionally given in terms of infinite sequences $s_0 s_1 ...$ over the alphabet *AP*. An infinite sequence $\sigma$ is lasso-shaped if it has the form $w_1(w_2)^\omega$, where $w_1$ and $w_2$ are finite sequences. Given two sequences of states $w_1$ and $w_2$, their concatenation is denoted as $w_1.w_2$. A sequence $\pi \in \Sigma^\omega$, where $\Sigma^\omega$ denotes the set of infinite sequences over *AP*, *satisfies* $\phi$ at position $i$ if and only if one the following holds:

- $\pi, i \models p$ iff $p \in \pi_i$
- $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$
- $\pi, i \models \phi \wedge \psi$ iff $\pi, i \models \phi$ and $\pi, i \models \psi$
- $\pi, i \models \circ\phi$ iff $\pi, i+1 \models \phi$
- $\pi, i \models \phi_1 \mathbf{U} \phi_2$ iff $\exists k \geq i : \pi, k \models \phi_2$ and $\forall j, i \leq j < k : \pi, j \models \phi_1$.

$\pi \models \phi$ iff $\pi, 0 \models \phi$ Other Boolean connectives are defined in the standard way. The temporal operators $\mathbf{F}\phi$ and $\mathbf{G}\phi$ are defined as $\mathbf{F}\phi \equiv \top \mathbf{U} \phi$ and $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$ respectively. The the goal "The pump shall be off when methane is detected in the mine" can be formalized as $\mathbf{G}\ (\mathsf{Methane} \rightarrow \circ\neg\mathsf{PumpOn})$. The language of an LTL formula $\phi$, denoted $\mathcal{L}(\phi)$ is the set of infinite state sequences $w$ such that $w \models \phi$. In this work, define weakness of a formula relative to another in terms of implication (rather than using other notions like [13]).

A variant of LTL was introduced recently in [14], LTL$_f$, whose semantics are defined in terms of finite traces instead. Given a finite sequence $\pi$, the satisfaction of an LTL$_f$ formula is inductively defined as follows:

- $\pi, i \models \circ\phi$ iff $i < \mathit{length}(\pi)$ and $\pi, i+1 \models \phi$
- $\pi, i \models \bullet\phi$ iff $i < \mathit{length}(\pi)$ implies $\pi, i+1 \models \phi$
- $\pi, i \models \mathbf{G}\phi$ iff for all $i \leq j \leq \mathit{length}(\pi)$ $\pi, j \models \phi$

The temporal operator $\bullet$ is called weak next which abbreviates $\neg \circ \neg$. (Note that on finite traces, $\bullet\phi$ is not equivalent to $\circ\phi$.) Past temporal operators are sometimes used to keep specifications compact and easy to understand [15]. We use the past operator yesterday ($\mathbf{Y}$) which is the past analogue of $\circ$, as well as historically ($\mathbf{H}$), to mean "it has always been the case that". These are introduced to enable our use of specifications found the literature for benchmarking. The semantics are defined as $\phi, i \models \mathbf{Y}\phi$ iff $0 < i$ and $\pi, i-1 \models \phi$, and $\phi, i \models \mathbf{H}\phi$ iff $0 \leq j < i$ and $\pi, j \models \phi$. Note that any LTL formula with past operators can be rewritten by only future-time operators [16]. We will refer literals preceded by a $\circ$ or a $\mathbf{Y}$ as a *temporal literal*.

## B. Synthesis of Open Controllers

A specification is represented as a tuple $\langle \mathcal{A}, \mathcal{G} \rangle$ of two disjoint sets—a set of assumptions $\mathcal{A}$ and a set of goals $\mathcal{G}$. Intuitively, assumptions are those constraints to which an environment is expected to conform and goals are those constraints which the software must satisfy. The environment's state is characterized by a subset of variables $\mathcal{X} \in \mathcal{V}$ called *input variables* (e.g., $\{\mathsf{Methane}, \mathsf{HighWater}\}$), while the controller's state is determined by $\mathcal{Y} = \mathcal{V} \backslash \mathcal{X}$ of *output variables* (e.g., $\{\mathsf{PumpOn}\}$).

In this context, we are interested in a subset of LTL called *GR (1)* [17]. This includes formulae of the kind $\Phi^{\mathcal{A}} \rightarrow \Phi^{\mathcal{G}}$, where both $\Phi^{\mathcal{A}}$ and $\Phi^{\mathcal{G}}$ are conjunctions of three kinds of LTL formulae: (1) *initial conditions*, $\theta$, purely Boolean formulae $B(\mathcal{V})$ over the variables in $\mathcal{V}$ not containing any temporal operator, constraining the initial state of a system; (2) *invariants*, $\rho$, of the form $\mathbf{G}B(\mathcal{V} \cup \circ\mathcal{V} \cup\ \mathbf{Y}\mathcal{V})$, temporal formulae containing only an outer $\mathbf{G}$ operator and $\circ$ (resp. $\mathbf{Y}$) operators such that no two $\circ$ (resp. $\mathbf{Y}$) are nested; these formulae encode one-step transitions allowed in the system; (3) *fairness conditions* of the form $\mathbf{GF}B(\mathcal{V})$, expressing a Boolean condition $B(\mathcal{V})$ holding infinitely many times in a system execution. We consider an extended version of the GR(1) language that allows for specifications expressed according to the response pattern $\mathbf{G}(\psi \rightarrow \mathbf{F}\phi)$ where $\psi$ and $\phi$ are Boolean expressions defined over $\mathcal{V} \cup \circ\mathcal{V} \cup\ \mathbf{Y}\mathcal{V}$. Each conjunct in $\Phi^{\mathcal{A}}$ is called an *assumption* (**Asm**), and each one in $\Phi^{\mathcal{G}}$ is called a *guarantee* (**Gar**). Revisiting the example introduced in Section II, This can now be formalized as follows.

| | |
|---|---|
| **Gar** | PumpOffWhenMethane |
| **FormalDef** | $\mathbf{G}\ (\mathsf{Methane} \rightarrow \circ\neg\mathsf{PumpOn})$ |
| **Gar** | PumpOnWhenHighWater |
| **FormalDef** | $\mathbf{G}\ (\mathsf{HighWater} \rightarrow \circ\mathsf{PumpOn})$ |
| **Asm** | AvoidUnsafeLevels |
| **FormalDef** | $\mathbf{G}\ \neg(\mathsf{Methane} \wedge \mathsf{HighWater})$ |
| **Asm** | WaterLoweredWhenPumpOn |
| **FormalDef** | $\mathbf{G}\ (\mathbf{Y}\mathsf{PumpOn} \wedge \mathsf{PumpOn} \rightarrow \circ\neg\mathsf{HighWater})$ |

We further assume an initial goal $\neg\mathsf{PumpOn}$ and an initial assumption $\neg\mathsf{Methane} \wedge \neg\mathsf{HighWater}$. The semantics of GR(1) specifications are typically defined over two-player game structures. A *GR(1) game structure* $\mathcal{G}$ is a tuple $\langle \mathcal{X}, \mathcal{Y}, \mathcal{A}_{\mathit{init}}, \mathcal{G}_{\mathit{init}}, \mathcal{A}_{\mathit{inv}}, \mathcal{G}_{\mathit{inv}}, \gamma \rangle$, where $\mathcal{X}, \mathcal{Y}$ are the sets of input and output variables respectively; $\mathcal{A}_{\mathit{init}}, \mathcal{G}_{\mathit{init}}, \mathcal{A}_{\mathit{inv}}$ and $\mathcal{G}_{\mathit{inv}}$ are the initial conditions and invariants of the GR(1) formula; $\gamma = \bigwedge_{i=1..m} \mathcal{A}_{\mathit{fair}}(i) \rightarrow \bigwedge_{j=1..n} \mathcal{G}_{\mathit{fair}}(j)$ is a formula representing the game's *winning condition*.

A game structure can be depicted as a directed graph such that every state corresponds to some valuation of the system variables and each edge corresponds to a pair of actions available to the two players. The first player, called *environment*, assigns a value to the input variables in order to satisfy the assumptions, and the second player, the *controller*, responds by setting the output variables in compliance with the guarantees. The environment's goal is to force the controller to violate the guarantees while satisfying its assumptions.

The controller synthesis problem is formalized as identifying a winning strategy for the controller in this game: a *strategy* is a mapping from the history of past valuations in the game and the current input choice by the environment, to an output choice ensuring that the controller satisfies the guarantees. A specification $\langle \mathcal{A}, \mathcal{G} \rangle$ is realizable if a strategy exists such that for all initial environment choices, the initial states are winning states, i.e., states from which the system can realize the formula.

$$\begin{aligned}(\mathcal{A}_{\mathit{init}} \rightarrow \mathcal{G}_{\mathit{init}}) \wedge (\mathcal{A}_{\mathit{init}} \rightarrow \mathbf{G}((\mathbf{H}\mathcal{A}_{\mathit{inv}}) \rightarrow \mathcal{G}_{\mathit{inv}}))\wedge \\ (\mathcal{A}_{\mathit{init}} \rightarrow \mathbf{G}(\mathcal{A}_{\mathit{inv}} \rightarrow \gamma))\end{aligned} \tag{1}$$

Efficient algorithms for GR(1) realizability checking and controller synthesis exist, .e.g., [18], [17]. Formal definitions of games and strategies, and of the algorithm to compute controller strategies, are given in [18].

If such a controller strategy does not exist, the specification is said to be *unrealizable*. In this case there is a strategy, called a *counterstrategy*, for the environment that forces the violation of at least one guarantee [19]. Counterstrategies and, symmetrically to strategies, map histories of executions and the current state onto the next input choice by the environment. There are several formal definitions of counterstrategies [19], [20], [21], all having in common its representation as a transition system whose states and/or transitions are labeled with valuations of variables. We consider the definition of "concrete" counterstrategies given in [21]. A run through a counterstrategy is a sequence of states (possibly infinite) from the initial state.

Given an unrealizable specification $\langle \mathcal{A}, \mathcal{G} \rangle$, we say that $\mathcal{G}^* \subseteq \mathcal{G}$ is *minimally unfulfillable w.r.t. to* $\mathcal{A}$ iff the removal of any goal $g \in \mathcal{G}^*$ makes $\langle \mathcal{A}, \mathcal{G}^* \backslash \{g\} \rangle$ realizable [22]. $\langle \mathcal{A}, \mathcal{G}^* \rangle$ is called an unrealizable core.

## IV. Specification Adaptation Approach

In this section, we describe a specific implementation (shown in Algorithm 1) of the Goal Manager component introduced in Section II. The algorithm calls three main procedures to guide the weakening process: `Synthesize`, `WeakenGoal` and `WeakenAsm`. We explain these in detail in the remainder of this section, for now we simply anticipate that `Synthesize` returns a set of control strategies $Ctl$ or a counterstrategy $c$ witnessing that the specification is unrealizable, while `WeakenGoal` and `WeakenAsm` return new sets of goals and assumptions respectively.

At the start, the approach assumes a realizable specification $\langle \mathcal{A}, \mathcal{G} \rangle$ is given in which one or more assumptions are violated by a given system observation $\bar{\pi}$. The first three lines of the algorithm are straightforward: goals, assumptions, counterstrategies and the flag to degrade the guarantees are initialized (lines 1–2). The assumptions violated by $\bar{\pi}$ in the initial assumptions $\mathcal{A}$, we denote as $\mathcal{A}_v$, are weakened to $\mathcal{A}_v^{'}$ so as to be consistent with the violation trace (line 4). Section IV-A describes the working of this step. The updated specification $\langle \mathcal{A}', \mathcal{G} \rangle$ is checked for realizability (line 5). If it produces an empty controller, the specification $\langle \mathcal{A}', \mathcal{G} \rangle$ is unrealizable, and counterstrategy $c$ is returned. (The approach does not use information about the minimally unfulfillable goals $\mathcal{G}'^*$ in assumption weakening.) The procedure `WeakenAsm` is then iteratively called to search for a weakening of the original assumptions $\mathcal{A}$ which yields a realizable specification using the violation trace $\bar{\pi}$ and the computed counterstrategy $c$ (and all counterstrategies computed thus far in $C_a$). The counterstrategies are used to exclude assumption weakenings that yield unrealizable specifications. Details of this are given in Section IV-B.

The decision to switch to goal weakening is triggered by the *degrade* flag once set to true. This occurs when no weakening $\mathcal{A}_v'$ of the violated assumptions in $\mathcal{A}$ exists that is consistent with the violation traces observed and that excludes all counterstrategies computed thus far. It may also occur according to some other predefined, domain-specific heuristic (line 10).

Once the goal weakening process is instigated, the last counterstrategy $c$ computed (from the assumption weakening) is passed to `WeakenGoal` along with the set $\mathcal{G}_{core}$, which corresponds to the set of goals that cannot be fulfilled collectively in $c$, and the set of original goals $\mathcal{G}$. A process similar to assumption weakening is followed in which any goal or its weakening that appeared in $\mathcal{G}_{core}$ (in any iteration) is updated. We use the expression in the original goals when weakening goals to remain as (syntacticly) similar as possible to the original specification (lines 17–20). The main difference here is that any goal weakening computed must be satisfied by runs from the counterstrategies in $C_g$.

Note that in our design, the iterative procedure only explores goals weakening after the assumption weakening exploration has terminated. The algorithm could be modified to allow interleaving between assumption and goal weakening and thus discover other combinations of realizable assumptions and goals. Furthermore, we assume the goal weakening uses the final assumption weakening and its computed counterstrategy. This, as discussed in Section V, can be modified to consider any weakening computed from previous iterations along with their counterstrategy.

We discuss in what follows our implementation of the `WeakenAsm` and `WeakenGoal` procedures. For both procedures, we consider an LTL learning task to be defined as a tuple $\langle AP, \Phi, \Sigma, \Sigma^- \rangle$ where $\Phi$ a conjunction of LTL formulae, $\Sigma^+$ (resp. $\Sigma^-$) is a set of finite or infinite traces where for some $\sigma \in \Sigma^+$, $\sigma \not\models \Phi$, and for some $\bar{\sigma} \in \Sigma^-$, $\bar{\sigma} \models \Phi$, according to the semantics described in Section III-A. A weakening $\Phi'$ of $\Phi$ is a solution if for all $\sigma \in \Sigma^+$, $\sigma \models \Phi'$ and for all $\bar{\sigma} \in \Sigma^-$, $\bar{\sigma} \not\models \Phi$. Several sound logic-based learners for LTL exist such as [23], [24], [25].

### *A. Assumption Weakening for Consistency*

Once the goal manager identifies the set of assumptions violated in $\mathcal{A}'$ (let us denote this as $A_v$), the assumptions are passed to the `WeakenAsm` procedure. This procedure is responsible for identifying a weakened version of the violated assumptions in $\mathcal{A}'$ that is satisfied in the observed violation trace, i.e., $\bar{\pi} \models A^{'}$, but in none of the counterstrategies found, i.e., $c \not\models A^{'}$ for all $c \in C_a$. At the start, the set of counterstrategies $C_a$ is empty. Note that the inclusion of the violation trace

**Algorithm 1:** Degradation of a realizable specification

```
Data: Violation trace $\bar{\pi}$
Data: realizable specification $\langle \mathcal{A}, \mathcal{G} \rangle$ such that
      $\bar{\pi} \not\models \mathcal{A}$
Result: realizable specification $\langle \mathcal{A}', \mathcal{G}' \rangle$ such that
      $\bar{\pi} \models \mathcal{A}'$
1  $\mathcal{G}' := \emptyset$;   $\mathcal{A}' := \emptyset$;
2  $degrade$:= $false$;
3  $C_a := \emptyset$;   $C_g := \emptyset$;
4  $\mathcal{A}' \leftarrow$ WeakenAsm($\mathcal{A}, \bar{\pi}, C_a$);
5  $Ctl, c, \mathcal{G}'^* \leftarrow$ Synthesize($\mathcal{A}', \mathcal{G}'$);
6  while $Ctl = \emptyset$ and not degrade do
7      $C_a := c \cup C_a$;
8      $\mathcal{A}' \leftarrow$ WeakenAsm($\mathcal{A}, \bar{\pi}, C_a$);
9      if $\mathcal{A}'_v = \emptyset$ and not heuristic($\mathcal{A}', \mathcal{G}$) then
10         $degrade$:= $true$;
11     end
12     else
13         $Ctl, c, \mathcal{G}'^* \leftarrow$ Synthesize($\mathcal{A}', \mathcal{G}'$);
14     end
15 end
16 $\mathcal{G}_{core} := \mathcal{G}'^*$;
17 while $Ctl = \emptyset$ do
18     $C_g := c \cup C_g$;
19     $\mathcal{G}' \leftarrow$ WeakenGoal($\mathcal{G}, \mathcal{G}_{core}, C_g$);
20     $Ctl, c, \mathcal{G}'^* \leftarrow$ Synthesize($\mathcal{A}', \mathcal{G}'$);
21     $\mathcal{G}_{core} := \mathcal{G}_{core} \cup \mathcal{G}'^*$;
22 end
23 return $\langle \mathcal{A}', \mathcal{G}' \rangle$;
```

requires that the language of the weakened assumptions includes at least one infinite trace with $\bar{\pi}$ as its prefix.

In our context, the only forms of assumptions violations that are observed are those invalidating invariants. (Fairness violations cannot be observed in finite time.) We assume an assumption invariant has the general form $\mathbf{G}(\phi \rightarrow \psi)$ where $\phi$ and $\psi$ are Boolean formulas over $\mathcal{V} \cup \circ\mathcal{X} \cup \mathbf{Y}\mathcal{V}$. (Any Boolean formula over $\mathcal{V} \cup \circ\mathcal{X} \cup \mathbf{Y}\mathcal{V}$ can be rewritten to conform to this form.) This provides our procedure with a standard means for weakening assumptions, since any formula of this form can be weakened by adding a conjunct to the antecedent of the implication $\phi$. An important consideration we make when weakening assumptions from finite violation traces is that we interpret any $\circ$ appearing in the assumptions as $\bullet$. This is necessary given the nature of the trace that will be forced to be consistent with the weakened specification. Our assumption weakening procedure is tasked with finding a minimal negated conjunction of temporal literals:

$$\neg(\bigwedge [\theta][\neg] v_j) \tag{2}$$

where $\theta \in \{\circ, \mathbf{Y}\}$ and $v_j \in \mathcal{X}$ if $\theta = \circ$, to conjoin with the antecedent $\phi$ of a violated assumption that makes the weakened version of the assumption satisfied in $\bar{\pi}$ (i.e., $\bar{\pi} \models \mathbf{G}(\phi \wedge \neg(\bigwedge[\theta][\neg]v_j) \rightarrow \psi)$. (In the context of a learning task, $\bar{\pi}$ is added to $\Sigma^+$ and $\Sigma^-$ is initially empty.) The reason for this representation is best illustrated by an example.

Let us consider our motivating example. Having observed the finite trace $\bar{\pi}_1 = s_0 s_1$ with:

$$s_0 = \{\neg\mathsf{HighWater}, \neg\mathsf{Methane}, \neg\mathsf{PumpOn}\}$$
$$s_1 = \{\mathsf{HighWater}, \mathsf{Methane}, \neg\mathsf{PumpOn}\}$$

the goal manager detects a violation to the assumption AvoidUnsafeLevels, formally expressed as $\mathbf{G}\,\neg(\mathsf{Methane} \wedge \mathsf{HighWater})$ at $s_1$. This triggers an assumption weakening process to ensure the assumption AvoidUnsafeLevels accounts for this observed behaviour. For implementation purposes, our procedure rewrites the violated assumption to the equivalent formula $\mathbf{G}\,(\top \rightarrow \neg\mathsf{Methane} \vee \neg\mathsf{HighWater})$. It adds $\bar{\pi}_1$ to $\Sigma^+$ then seeks a negated conjunction that, if added to the antecedent of the implication $\top$, makes the updated assumption satisfied in $\bar{\pi}_1$. Our learning procedure searches for a conjunction $\bigwedge[\theta][\neg]v_j$ that is satisfied in $s_1$, whose negation will be conjoined to $\top$ thus making the weakened assumption true at state $s_1$. In this instance, an example of a negated conjunction computed by the learner is $\neg(\mathsf{Methane})$.

To compute such weakening, WeakenAsm defines a candidate solution space using all (temporal) literals in the language. To reduce the solution space, our procedure implements a set of syntactic constraints that excludes (temporal) literals that are not true in the state at which the assumption is violated from being added as a possible conjunct in $\bigwedge[\theta][\neg]v_j$ . This is justified on semantics grounds. To illustrate this, consider our violated assumption $\mathbf{G}\,(\top \rightarrow \neg\mathsf{Methane} \vee \neg\mathsf{HighWater})$s. Our procedure excludes temporal literals such as $\neg\mathsf{Methane}$ from being added as a conjunct since its inclusion would result in an assumption of the form $\mathbf{G}\,(\top \wedge \neg(\mathsf{Methane}) \rightarrow \neg\mathsf{Methane} \vee \neg\mathsf{HighWater})$ which is also violated in the trace $\bar{\pi}_1$ and in fact in this case unsatisfiable.

With the restriction above imposed, still multiple subformulas of the form in Equation (2) remain candidate solutions for weakening the violated assumption including: $\neg(\mathsf{HighWater})$, $\neg(\mathsf{Methane})$ and $\neg(\neg\mathsf{PumpOn})$ as well as $\neg(\mathsf{HighWater} \wedge \mathsf{Methane})$, $\neg(\mathsf{HighWater} \wedge \neg\mathsf{PumpOn})$,

$\neg$(Methane$\wedge\neg$PumpOn) and $\neg$(HighWater$\wedge$Methane$\wedge\neg$PumpOn). The learning procedure selects the sub-formula that has the fewest number of conjuncts that satisfies the condition $\bar{\pi} \models \mathcal{A}^{'}$. In the case of our running example, the assumption AvoidUnsafeLevels becomes **G** ($\neg$Methane $\rightarrow$ $\neg$Methane $\vee$ $\neg$HighWater) which is a tautology.

The learned assumptions is guaranteed, by construction, to be weaker than the violated assumptions, and to be consistent with the assumptions in the original assumption of $\mathcal{A}$, that are not violated in $\bar{\pi}$.

## *B. Assumption Weakening for Realizability*

The above process ensures that the updated assumptions $\mathcal{A}'$ are consistent with the new observed behaviour. However, it does not guarantee that the updated specification $\langle \mathcal{A}', \mathcal{G} \rangle$ is realizable. The reason for this is that our initial application of `WeakenAsm` is only given prefixes of traces to include and may "over" weaken, i.e. a typical problem when learning from positive examples only. The learner may compute a weaker assumption that is too general or one that is too specific to the violation trace encountered, which in either case may force the system to violate its goals. Without sufficient examples that demonstrate the language of an adequate assumption, the procedure is not guaranteed to compute assumptions that yield a realizable specification.

To address this, our algorithm first checks if a controller can be synthesized from the updated specification $\langle \mathcal{A}', \mathcal{G} \rangle$. If it cannot, the realizability checker returns a counterstrategy $c$ demonstrating how an environment compliant with its new assumption $\mathcal{A}'$ may force a violation to a goal in $\mathcal{G}$. This counterstrategy records the sequences of states that the environment can traverse to force such violation. Fig. 3 shows the counterstrategy graph for the updated mine pump specification. Each edge is labelled with the truth value assignment to input choices and output choices made by the environment and controller respectively. States $s_1$ and $s_2$ are deadlock states since no successor state exists which can satisfy both the goals PumpOnWhenHighWater, which requires the pump should be on in the next state, and PumpOffWhenMethane, which requires the pump to be off.

Having found a counterstrategy, hence *Ctl* $= \emptyset$, the approach enters the assumption weakening loop (lines 6–15 in Alg. 1). If a heuristic condition is defined and met for switching to goal weakening, then the corresponding flag is set, and the loop terminates. Otherwise, the counterstrategy is added to the set $C_a$ and `WeakenAsm` is invoked to find an alternative weakening $\mathcal{A}_2^{'}$ to the violating assumption in $\mathcal{A}$ that is consistent with the trace $\bar{\pi}$, such that $c_i \not\models \mathcal{A}_2^{'}$ for all $c_i \in C_a$. To guide

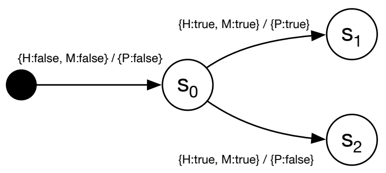

{H:true, M:true} / {P:true}
{H:false, M:false} / {P:false}
{H:true, M:true} / {P:false}
s0
s1
s2

Fig. 3. Counterstrategy graph computed for $\langle \mathcal{A}', G \rangle$. H, M and P correspond to the variables HighWater, Methane and PumpOn respectively.

the search toward a realizable specification, our implementation extracts a single (lasso-shaped) run $r_c$ from the counterstrategy and uses it as a negative example in $\Sigma^-$ of the learning task to prune out any assumption weakening that may prescribe such a counterstrategy. Note that a single run is sufficient to eliminate the counterstrategy as shown in [20]. The learning procedure is instigated again in search of an alternative minimal negated conjunction of temporal literals $\neg(\bigwedge[\theta][\neg]v_j)$. (The procedure is guaranteed to produce a different assumption by the soundness of the learner.) Note that our implementation only attempts to weaken the violated subset of assumptions in $\mathcal{A}$. We assume these are detected and flagged by the goal manager.

The assumption weakening loop terminates successfully once a weakening $\mathcal{A}_n^{'}$ of $\mathcal{A}$ is found that makes the specification realizable, where $\mathcal{A}_n^{'}$ denotes $n^{\text{th}}$ weakening of the violating assumption(s) in $\mathcal{A}$. In this case, the adaptation procedure also terminates. The approach may fail to find a weakening of $\mathcal{A}$, for instance owing to restrictions over the language space of assumptions, or because no weaker assumption exists to the learning task defined (i.e., that is both satisfied by the violating trace and unsatisfied by the counterstrategies). As in the first case, $\mathcal{A}_n^{'} \rightarrow \mathcal{A}$ is guaranteed to be valid by construction. If it fails, our procedure initiates a search for a weakening $\mathcal{G}'$ of the goals $\mathcal{G}$ instead, using the specification $\langle \mathcal{A}_n^{'}, \mathcal{G} \rangle$ the obtained prior to the strengthening attempts, such that the specification $\langle \mathcal{A}', \mathcal{G}_m^{'} \rangle$ is realizable, where $\mathcal{G}^m$ denotes $m^{\text{th}}$ weakening of $\mathcal{G}$.

Let us return to the mine pump example. Since the weakened specification $\langle \mathcal{A}^{'}, \mathcal{G} \rangle$ is not realizable, `WeakenAsm` is invoked given the violation trace and the counterstrategy in Fig. 3. From the counterstrategy, a run $r$ is extracted from the counterstrategy graph. In this case $r = s_0 s_1$ with $s_0 = \{\neg$HighWater, $\neg$Methane, $\neg$PumpOn$\}$ and $s_1 = \{$HighWater, Methane, PumpOn$\}$. An alternative weakening is computed: **G** (PumpOn $\rightarrow$ $\neg$Methane $\vee$ $\neg$HighWater) in $\mathcal{A}_2^{'}$ which is also unrealizable. Its counterstrategy graph $c_2$ is shown in Fig. 4 which instigates another

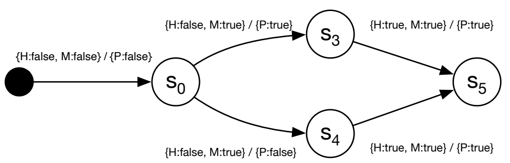

{H:false, M:false} / {P:false}
{H:false, M:true} / {P:true}
{H:true, M:true} / {P:true}
{H:false, M:true} / {P:false}
{H:true, M:true} / {P:true}
s0
s3
s4
s5

Fig. 4. Counterstrategy graph computed for $\langle \mathcal{A}'_2, G\rangle$.

assumption weakening cycle. This time however, no alternative weakening to the violating assumption in $\mathcal{A}$ is found. Hence the algorithm sets *degrade* to *true* and proceeds to goal weakening.

## C. Goal Weakening

The goal weakening loop starts with an unrealizable specification. The aim is to find a goal $\mathcal{G}'$ that is a weakening of the original goals $\mathcal{G}$, i.e., $\mathcal{G}' \rightarrow \mathcal{G}$ such that $\langle \mathcal{A}', \mathcal{G}'\rangle$ is realizable.

In the first instance, the counterstrategy $c$ computed for the assumption weakening case (line 13) is used. As with assumption weakening, a run $r_c$ from the counterstrategy is extracted. If the run is finite, as discussed in Section IV-A, we interpret every $\circ$ operator as a weak next $\bullet$. Otherwise, we assume the semantics over infinite traces. As we will illustrate in our example below, with finite LTL semantics, the run $r_c$ would not violate the guarantees at the last state. (This is because no next state exists.) For a finite run $r_c$ to be used for weakening the goals, it needs to demonstrate a behaviour that is inconsistent with the current goal(s) but in the language of the weakened goals. To generate such a trace, a successor state $s'$ is forced in the trace. A natural question that then arises is what assignment should such state be given. In our approach, we force a state that satisfies $A'$ and violates a goal in $\mathcal{G}$. Given that $r_c$ ends in a deadlock, any forced successor satisfying $\mathcal{A}'$ is guaranteed to violate one or more goals in $\mathcal{G}'*$ but none of the goals $\mathcal{G}\backslash\mathcal{G}'*$. In our work, the choice of violated goal from $\mathcal{G}'*$ is at random. However, it can be easily extended to incorporate knowledge about risks and priorities, e.g., [26], to determine which goal to violate or not. The implementation uses a SAT solving-based approach [27] to compute a successor state such that $r_c.s' \models \mathcal{A}' \wedge \neg\mathcal{G}'^* \wedge \bigwedge \mathcal{G}\backslash\mathcal{G}'^*$. If the run extracted from the counterstrategy is instead infinite, then no successor state is generated. The lasso-shaped trace would illustrate instead a violation to a fairness goal in $\mathcal{G}\backslash\mathcal{G}'^*$.

In the case of the mine pump example, a run extracted from Fig. 4 is extracted at random, in this case, as $r_{c_2} = s_0 s_3 s_5$ with:

$$\begin{aligned} s_0 &= \{\neg\mathsf{HighWater}, \neg\mathsf{Methane}, \neg\mathsf{PumpOn}\} \\ s_3 &= \{\neg\mathsf{HighWater}, \mathsf{Methane}, \mathsf{PumpOn}\} \\ s_5 &= \{\mathsf{HighWater}, \mathsf{Methane}, \neg\mathsf{PumpOn}\} \end{aligned}$$

The trace does not violate any of the goals up until $s_5$. The minimally unfulfillable goals in $\mathcal{G}'^*$ are PumpOffWhenMethane:$\mathbf{G}$ $(\mathsf{Methane} \rightarrow \circ\mathsf{PumpOn})$ and PumpOnWhenHighWater: $\mathbf{G}$ $(\mathsf{HighWater} \rightarrow \circ\neg\mathsf{PumpOn})$. Our procedure automatically populates a successor state $s' = \{\neg\mathsf{HighWater}, \neg\mathsf{Methane}, \neg\mathsf{PumpOn}\}$ to $s_5$ that satisfies both the updated assumption AvoidUnsafeLevels and the original WaterLoweredWhenPumpOn but violate PumpOnWhenHighWater.

Contrary to the assumption weakening procedure, in which runs from counterstrategies were used as examples of (prefixes of) traces that should be inconsistent with specification, the runs from the counterstrategy for goal weakening are used as examples of prefixes of infinite traces that should be included in the language of the updated specification, such that $r_{c_j} \models \mathcal{G}'$ where $r_{c_j}$ is infinite, or $r_{c_j}.s' \models \mathcal{G}'$, where $r_{c_j}$ is finite, for each $c_j \in C_g$. Hence, $r_{c_j}$ is added to $\Sigma^+$ rather than $\Sigma^-$. This forces the learning engine to search for a weakening to $\mathcal{G}$ that makes every $c_j \in C_g$ no longer a counterstrategy (since the elimination of a run from a representing a play in the counterstrategy for the environment, excludes the whole counterstrategy). As with assumption invariants, goal invariants are of the form $\mathbf{G}(\phi \rightarrow \psi)$, whilst fairness goals, have the general form $\mathbf{GF}(\phi)$ where $\phi$ is a disjunction of conjunction of temporal literals.

When weakening a goal fairness (resp. invariant), our implementation seeks a minimal disjunction of conjunction of temporal literals (i.e., $\bigvee \bigwedge [\theta][\neg]v_j$) to disjoin to the formula (resp. consequent) $\psi$. Returning to the mine pump example, our learning procedure computes the disjunct Methane to add to the consequent of PumpOnWhenHighWater to become $\mathbf{G}$ $(\mathsf{HighWater} \rightarrow \circ\neg\mathsf{PumpOn} \vee \mathsf{Methane})$.

Once updated goals $\mathcal{G}'$ are computed, the specification $\langle \mathcal{A}', \mathcal{G}'\rangle$ is checked for realizability. If unrealizable, then the approach proceeds to weaken the goals again using the new counterstrategy and unrealizable core computed. Note that in every iteration, the set of goals whose formalization may be potentially updated is expanded (line 21 in Alg. 1). The process iterates until a realizable specification is reached. This is ensured since our weakening procedure can, in the worst case, produce trivial weakening to violated goals which is equivalent to *true*. In the case of the mine pump example, the update goals yields a realizable specification.

## V. Evaluation

In order to evaluate our approach to specification adaptation, we must reproduce the adaptation scenario described in Section II. That is, we start with a system running a strategy that guarantees some realizable specification as long as the system's environment conforms to the specification's assumptions. The specification adaptation must occur when the system observes environment behaviour that is inconsistent with the assumptions. At this point, the adaptation algorithm in Section IV should be capable of finding a new specification that is realizable and that is consistent with the environment behaviour that triggered the adaptation. Such a specification can then be used to synthesise and deploy a new controller that continues system execution.

Thus, our research questions focus on the algorithm's ability to adapt specifications using as input the assumption violation trace and the specification used to synthesise the controller that was running at assumption violation time. Our research questions are:

RQ1 Can the algorithm find a realizable specification that accommodates for violating traces?

RQ2 Can the algorithm find the *ideal* specification, used to simulate the environment?

To answer these questions we: $(i)$ Start with realizable specifications from the literature, which we refer to as *ideal specifications* because we take their assumptions to describe the true behaviour of the environment in which the system is currently deployed. $(ii)$ For each ideal specification we computed *mutated specifications* by randomly applying assumption strengthening patterns. Each realizable mutation represents our starting point. That is, we assume a system running a strategy synthesised from the mutated realizable specification. $(iii)$ For each realizable mutation we compute a control *strategy*. $(iv)$ We compute *execution traces* for each control strategy in an environment that conforms to the ideal specification, and in which a violation of the mutated assumptions occurs. (Violating trace).

The result is a set of mutated realizable specifications and a set of corresponding violating traces that demonstrate how the specification assumptions are invalid.

### A. Implementation

The algorithm was implemented in Python on a Windows PC with Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz Processor and 16GB RAM. The learning software used for assumption and goal weakening was FastLAS [23] which is a publicly available, scalable symbolic learning tool. We use spectra [28] for synthesis and the answer set solver clingo [29] for violation generation and detection. For this implementation, before the algorithm moves from assumption to goal weakening, we chose to use the first weakened assumption $A'_1$ rather than $A'_n$. This is because in practise, $A'_n$ tends to be more complex, in terms of number of (temporal) literals added. Restoring the first weakened assumption helps preserve syntactic similarity to the original specification.

TABLE I
Summary of mutated specifications and corresponding violating traces.

| | | Trace | | | | In $A_v$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Length | $\#\mathcal{A}$ | $\#\mathcal{G}$ | $\#A_v$ | Operators | | | Variables | | |
| Case-Study | Count | Mean | Mean | Mean | Mean | Mean | Min | Max | Mean | Min | Max |
| Arbiter | 10 | 1.6 | 1.0 | 4.0 | 1.0 | 1.0 | 1 | 1 | 1.0 | 1 | 1 |
| Genbuf | 5 | 1.4 | 27.2 | 79.4 | 1.4 | 3.5 | 1 | 26 | 2.9 | 1 | 20 |
| Lift | 10 | 2.1 | 7.0 | 10.0 | 1.0 | 2.6 | 1 | 4 | 2.2 | 1 | 3 |
| Minepump | 25 | 2.1 | 1.0 | 2.0 | 1.0 | 2.2 | 1 | 4 | 1.5 | 1 | 2 |
| Traffic | 50 | 1.8 | 4.4 | 3.4 | 2.4 | 2.6 | 1 | 5 | 2.2 | 1 | 4 |
| Traffic Single | 50 | 2.3 | 2.8 | 1.4 | 1.0 | 3.4 | 2 | 4 | 2.8 | 2 | 3 |
| All | 150 | 2.0 | 4.0 | 5.5 | 1.5 | 2.7 | 1 | 26 | 2.3 | 1 | 20 |

### B. Research Question 1

For RQ1, up to 10 unique violating traces were generated. The algorithm was then run on each violating trace of each *mutated* specification and the results recorded (see Tables III for results and II for performance). Almost all runs found a realizable specification with assumptions that are not violated by the violating trace. In 3 of 150 runs, the timeout of 100 seconds was exceeded and the specification at that point was still unrealizable.

### C. Research Question 2

For RQ2, the *final* weakened, realizable specifications produced by the algorithm in addressing question one, were compared, using spot [30], with the *ideal* specifications used to simulate the environments. The results can be seen in Table III. Unless the timeout was reached, the algorithm always produced a realizable specification that did not violate the violating trace. A success indicates that the algorithm weakened the assumptions and or guarantees until both assumptions and guarantees were identical to those of the *ideal* specification. It can be seen that the algorithm weakened the *mutated* specification to the *ideal* specification only 1 out of 150 runs. It is worth noting that in none of the runs did the algorithm weaken a goal to trivial—equivalent to dropping the goal.

The results demonstrate that the algorithm can find a realizable specification that accommodates a violating trace for several specifications in reasonable time for less complex case studies. Although the algorithm can learn the *ideal* specification used to simulate the environment, it does so rarely. This is because the algorithm is being

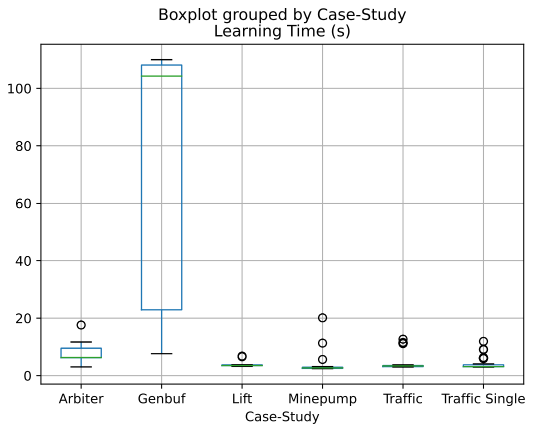
TABLE II
ALGORITHM PERFORMANCE FOR RQ1.

| | | Realizability Checks | | | | | | No. Weakened | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Assumption | | | Guarantee | | | Asm | Gar | Learning Time (s) | | |
| Case-Study | Count | Mean | Min | Max | Mean | Min | Max | Mean | Mean | Mean | Min | Max |
| Arbiter | 10 | 2.0 | 1 | 3 | 0.2 | 0 | 1 | 0.2 | 0.6 | 7.9 | 3.0 | 17.6 |
| Genbuf | 5 | 1.2 | 1 | 2 | 3.2 | 0 | 6 | 0.8 | 1.8 | 70.6 | 7.6 | 110.0 |
| Lift | 10 | 1.2 | 1 | 2 | 0.0 | 0 | 0 | 0.0 | 0.0 | 4.1 | 3.2 | 6.8 |
| Minepump | 25 | 1.4 | 1 | 7 | 0.0 | 0 | 0 | 0.0 | 0.0 | 3.8 | 2.4 | 20.1 |
| Traffic | 50 | 1.0 | 1 | 1 | 0.2 | 0 | 1 | 0.2 | 0.4 | 4.9 | 3.0 | 12.7 |
| Traffic Single | 50 | 1.3 | 1 | 4 | 0.0 | 0 | 0 | 0.0 | 0.0 | 4.0 | 3.0 | 11.9 |
| All | 150 | 1.3 | 1 | 7 | 0.2 | 0 | 6 | 0.1 | 0.2 | 6.8 | 2.4 | 110.0 |

TABLE III
RESULTS OF ALGORITHM FOR RQ2.

| Case-Study | realizable | Unrealizable | Success | All |
|---|---|---|---|---|
| Arbiter | 10 | 0 | 0 | 10 |
| Genbuf | 2 | 3 | 0 | 5 |
| Lift | 10 | 0 | 0 | 10 |
| Minepump | 25 | 0 | 0 | 25 |
| Traffic | 50 | 0 | 0 | 50 |
| Traffic Single | 49 | 0 | 1 | 50 |
| All | 146 | 3 | 1 | 150 |

run on a single violating trace, which is only one example of environment behaviour. If the subsequent controller were re-deployed in the environment, and further violating traces gathered, this could help the algorithm find the *ideal* specification over time.



## VI. RELATED WORK

Our work is related to Goal Oriented Requirements Engineering (GORE) [31] and the Machine-World model [3] in which focus is shifted from software requirements to system goals and assumptions to support reasoning about alternative realization strategies, pertinence, and completeness. As others (e.g., [32]), we bring requirements engineering activities to runtime. More specifically, we perform at runtime tasks (i.e., identification, assessment and resolution) related to GORE obstacle analysis. More specifically, we perform obstacle resolution at runtime, reacting to obstacles (environment assumption violations) and resolving them with a new specification that accounts for their occurrence. Predicting assumption violations is beyond this work, but is considered in MORPH and has been addressed using runtime verification, model checking and machine learning (e.g., [33], [34], [35]). Patterns to support manual obstacle resolution have been studied [36] and more recently more automated using general techniques based on logic-based learning such as [37]. They focus only on goal satisfiability rather than realisability [4], thus ignoring controllability of actions and events.

We envisage the application of our approach in MORPH-based architectures. However, it could also be applied to other architectures (e.g., [38], [39]) if extended with runtime planning capabilities.

Runtime reasoning for adaptation has been studied significantly. Most approaches consider a predefined set of adaptation options or strategies which are selected at runtime ([40], [41], [42], [43], [44]) or require manual or semi-automatic procedures for adaptation [45]. Unanticipated adaptations require dynamic update/reconfiguration strategies [46], [47], [48].

Runtime adaptation of robotic missions synthesised from temporal specifications has been studied before (e.g.,[49], [50], [51]). Focus is on motion plans and adaptation is triggered by the impossibility of moving between two locations that were assumed to be connected. It these works, it is the strategy that is adapted to achieve the same motion goals (e.g., move round the obstacle). Assumption learning is also studied in [52], [53]. In none of these, goal weakening is considered.

## VII. CONCLUSION

This paper proposed a framework for adapting specification of open controllers when their assumptions about the environment are violated, which combines reactive synthesis and symbolic learning to produce a realizable degredation of the initial specification. Our evaluation demonstrates the ability of our implementation to produce realizable specifications for benchmark case studies in the literature. The paper discusses several avenues for future work including deployment and the integration of notions of risk when selecting which goal to weaken. Approaches like [54] may also be considered to support the learning explore the solution space more efficiently. In addition, our implementation seeks the closest adaptation syntactically to the original. We will extend our approach to consider semantic similarity too.

## References

[1] S. Uchitel, V. Braberman, and N. D'Ippolito, "Runtime controller synthesis for self-adaptation: Be discrete!" in *11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 16-17, 2016, Austin, Texas*. ACM, 2016.

[2] A. van Lamsweerde, *Requirements Engineering: From system goals to UML models to software specifications*. John Wiley & Sons, 2009.

[3] M. Jackson, "The world and the machine," in *Proceedings of the 17th international conference on Software engineering*, ser. ICSE '95. ACM, 1995, pp. 283–292.

[4] M. Abadi, L. Lamport, and P. Wolper, "Realizable and unrealizable specifications of reactive systems," in *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, ser. ICALP '89. Berlin, Heidelberg: Springer-Verlag, 1989, p. 1–17.

[5] E. Letier and A. van Lamsweerde, "Deriving operational software specifications from system goals," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, p. 119–128, nov 2002. [Online]. Available: https://doi.org/10.1145/605466.605485

[6] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," in *Intl. Conf. on Verification, Model Checking and Abstract Interpretation*. Springer, 2006, pp. 364–380.

[7] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Software & Systems Modeling*, vol. 18, 10 2019.

[8] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstic, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss, "A survey of challenges for runtime verification from advanced application domains (beyond software)," *Formal Methods Syst. Des.*, vol. 54, no. 3, pp. 279–335, 2019. [Online]. Available: https://doi.org/10.1007/s10703-019-00337-w

[9] R. Ali, F. Dalpiaz, P. Giorgini, and V. E. S. Souza, "Requirements evolution: From assumptions to reality," in *Proceedings of the 12th International Conference Enterprise, Business-Process and Information Systems Modeling*, 2011, pp. 372–382.

[10] C. Ghezzi and A. Molzam Sharifloo, *Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 191–213.

[11] V. A. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "An extended description of MORPH: A reference architecture for configuration and behaviour self-adaptation," in *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds., vol. 9640. Springer, 2013, pp. 377–408. [Online]. Available: https://doi.org/10.1007/978-3-319-74183-3_13

[12] J. Kramer and J. Magee, "Dynamic configuration for distributed systems," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 424–436, 1985. [Online]. Available: https://doi.org/10.1109/TSE.1985.232231

[13] D. G. Cavezza, D. Alrajeh, and A. György, "A weakness measure for GR(1) formulae," *Formal Aspects Comput.*, vol. 33, no. 1, pp. 27–63, 2021. [Online]. Available: https://doi.org/10.1007/s00165-020-00519-y

[14] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, p. 854–860.

[15] A. Cimatti, M. Roveri, and D. Sheridan, "Bounded verification of past ltl," in *Formal Methods in Computer-Aided Design*, A. J. Hu and A. K. Martin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 245–259.

[16] D. Gabbay, "The declarative past and imperative future," in *Temporal Logic in Specification*, B. Banieqbal, H. Barringer, and A. Pnueli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 409–448.

[17] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," in *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006, pp. 364–380.

[18] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar, "Synthesis of Reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012.

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

[20] D. G. Cavezza and D. Alrajeh, "Interpolation-based GR(1) assumptions refinement," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10205, 2017, pp. 281–297. [Online]. Available: https://doi.org/10.1007/978-3-662-54577-5_16

[21] A. Kuvent, S. Maoz, and J. O. Ringert, "A symbolic justice violations transition system for unrealizable gr(1) specifications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 362–372. [Online]. Available: https://doi.org/10.1145/3106237.3106240

[22] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev, "Diagnostic information for realizability," in *9th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2008, pp. 52–67.

[23] M. Law, A. Russo, E. Bertino, K. Broda, and J. Lobo, "Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 03, pp. 2877–2885, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5678

[24] D. Neider and I. Gavran, "Learning linear temporal properties," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. S. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–10. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603016

[25] D. Athakravi, D. Alrajeh, K. Broda, A. Russo, and K. Satoh, "Inductive learning using constraint-driven bias," in *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, 2014, pp. 16–32.

[26] A. Cailliau and A. van Lamsweerde, "Runtime monitoring and resolution of probabilistic obstacles to system goals," *ACM Trans. Auton. Adapt. Syst.*, vol. 14, no. 1, pp. 3:1–3:40, 2019. [Online]. Available: https://doi.org/10.1145/3337800

[27] V. Lifschitz, *Answer Set Programming*, 1st ed. Springer Publishing Company, Incorporated, 2019.

[28] S. Maoz and J. O. Ringert, "Reactive synthesis with spectra: A tutorial," in *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '21. IEEE Press, 2021, p. 320–321. [Online]. Available: https://doi.org/10.1109/ICSE-Companion52605.2021.00136

[29] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, "Theory solving made easy with clingo 5," in *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, ser. OASIcs, M. Carro, A. King, N. Saeedloei, and M. D. Vos, Eds., vol. 52. Schloss Dagstuhl

- Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:15. [Online]. Available: https://doi.org/10.4230/OASIcs.ICLP.2016.2

[30] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for LTL and $\omega$-automata manipulation," in *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, ser. Lecture Notes in Computer Science, vol. 9938. Springer, Oct. 2016, pp. 122–129.

[31] A. van Lamsweerde, "Goal-oriented requirements engineering: a guided tour," in *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, 2001, pp. 249–262.

[32] N. Bencomo, G. S. Blair, F. Fleurey, and C. Jeanneret, "Summary of the 5th international workshop on models@run.time," in *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Dingel and A. Solberg, Eds., vol. 6627. Springer, 2010, pp. 204–208. [Online]. Available: https://doi.org/10.1007/978-3-642-21210-9_20

[33] S. Zudaire, F. Gorostiaga, C. Sánchez, G. Schneider, and S. Uchitel, "Assumption monitoring using runtime verification for uav temporal task plan executions," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 6824–6830.

[34] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm, "Runtime verification of real-time event streams under non-synchronized arrival," *Softw. Qual. J.*, vol. 28, no. 2, pp. 745–787, 2020. [Online]. Available: https://doi.org/10.1007/s11219-019-09493-y

[35] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting self-adaptation via quantitative verification and sensitivity analysis at run time," *IEEE Transactions on Software Engineering*, vol. 42, no. 1, pp. 75–99, 2016.

[36] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 978–1005, 2000.

[37] D. Alrajeh, A. Cailliau, and A. van Lamsweerde, "Adapting requirements models to varying environments," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 50–61. [Online]. Available: https://doi.org/10.1145/3377811.3380927

[38] E. Zavala, X. Franch, J. Marco, and C. Berger, "Hafloop: An architecture for supporting highly adaptive feedback loops in self-adaptive systems," *Future Generation Computer Systems*, vol. 105, pp. 607–630, 2020.

[39] S. Garcıa, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab, "An architecture for decentralized, collaborative, and autonomous robots," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 75–7509.

[40] P. Jamshidi, J. Cámara, B. Schmerl, C. Käestner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019, pp. 39–50.

[41] A. Nooruldeen and K. W. Schmidt, "State attraction under language specification for the reconfiguration of discrete event systems," *IEEE Trans. on Automatic Control*, vol. 60, no. 6, pp. 1630–1634, June 2015.

[42] H. E. Garcia and A. Ray, "State-space supervisory control of reconfigurable discrete event systems," *Int. Journal of Control*, vol. 63, no. 4, pp. 767–797, 1996.

[43] J. Cámara, B. Schmerl, and D. Garlan, "Software architecture and task plan co-adaptation for mobile service robots," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 125–136. [Online]. Available: https://doi.org/10.1145/3387939.3391591

[44] U. Topcu, N. Ozay, J. Liu, and R. M. Murray, "On synthesizing robust discrete controllers under modeling uncertainty," in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 85–94. [Online]. Available: https://doi.org/10.1145/2185632.2185648

[45] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proc. of the 4th Int. Conf. on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 278–293.

[46] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Assured and correct dynamic update of controllers," in *Proc. of the 11th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2016, pp. 96–107.

[47] A. Anderson and J. Rathke, "Migrating protocols in multi-threaded message-passing systems," in *Proc. of the 2Nd Int. Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '09. New York, NY, USA: ACM, 2009, pp. 8:1–8:5.

[48] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. on Software engineering*, vol. 22, no. 2, 1996.

[49] M. Guo, K. H. Johansson, and D. V. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 5025–5032.

[50] S. C. Livingston, R. M. Murray, and J. W. Burdick, "Backtracking temporal logic synthesis for uncertain environments," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 5163–5170.

[51] S. C. Livingston, P. Prabhakar, A. B. Jose, and R. M. Murray, "Patching task-level robot controllers based on a local $\mu$-calculus formula," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 4588–4595.

[52] J. Fu, H. G. Tanner, and J. Heinz, "Adaptive planning in unknown environments using grammatical inference," in *52nd IEEE Conference on Decision and Control*, 2013, pp. 5357–5363.

[53] Y. Chen, J. Tůmová, and C. Belta, "Ltl robot motion control based on automata learning of environmental dynamics," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 5177–5182.

[54] D. Alrajeh, P. Benjamin, and S. Uchitel, "Adaptation$^2$: Adapting specification learners in assured adaptive systems," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 1347–1352. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678919