IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Learning policies for specification updates at runtime

Author:
Patrick Benjamin

Supervisor:
Dalal Alrajeh

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

4 September 2020

Abstract

Controllers for autonomous systems are commonly synthesised from specifications written in GR(1) form, a subset of Linear Temporal Logic. These specifications have an assume-guarantee structure, where the controller must satisfy its guarantees if the environment satisfies the assumptions. At runtime, the assumptions written by a designer may not exactly reflect the environment's actual behaviour, meaning that the controller is not required to accomplish its tasks. Limited automated support exists for updating erroneous assumptions at runtime, meaning that specifications usually must be modified by hand. Nonmonotonic inductive logic programming (ILP) systems provide a complete and consistent method for modifying formulae such as assumptions, but by themselves may not produce an appropriate output. We desire that revised specification be *acceptable* - a new controller can be synthesised from it and the assumptions correctly reflect the environment's behaviour - and if possible *optimal* - meaning that the assumptions describe as great a range as possible of environment behaviours while remaining close to the designer's original intent. The search space for such solutions is large, and while we can set parameters for nonmonotonic ILP systems to restrict this search space, it is not always clear what these parameters should be. Settings that are too restrictive may mean a solution cannot be found, while those that are too broad may make the search intractable; the optimal setting is also likely to vary by context.

Our key contribution is a reinforcement learning (RL) agent that learns a domain-dependent policy for setting the ILP system's parameters to guide the search towards revised specifications that have our desired qualities. By allowing the agent to explore the search space at design time, we enable it to converge on a policy for finding appropriate revisions in few attempts upon deployment; the settings it applies should also be sufficiently narrow that the ILP revision system does not take too long to compute the revision at runtime. In this way, our contribution also demonstrates more generally the usefulness of RL agents in restricting the hypothesis space of ILP systems in a context-specific fashion.

The code from the implementation of our framework can be cloned from our GitLab repository if desired: https://gitlab.doc.ic.ac.uk/phb19/raspal-test.git.

Acknowledgments

I am indebted to my supervisor Dr Dalal Alrajeh, who has unfailingly given me the most generous and encouraging help and guidance on all aspects of this project.

My sincere thanks also go to my second marker Dr Sebastian Uchitel, as well as to Sebastián Zudaire, Davide Cavezza and Daniel Furelos-Blanco, all of whom have patiently provided invaluable insights and support with various elements of my problem.

I am also ever grateful to my long-suffering partner Alexia Kossifos, who nursed me through two broken limbs during this project, and who never wants to hear the phrase “realisable specification” again.

Contents

Chapter 1

Introduction

1.1 Motivation

Controller synthesis is the construction of a model of an autonomous software component's desired behaviour [1]. The controller and the environment in which it operates are together often known as the system. Various techniques have been proposed for reactive synthesis, the task of automatically building a provably correct controller from a formal logical specification [2, 3, 4]. A controller can be found, which we refer to as the specification being realisable, if there are no ways the environment can act which prevent the controller from fulfilling its goals (often called its guarantees).

In order to cut out the obstructive environment behaviours and thereby make the specification realisable, designers must describe the range of environment behaviours within which the controller is expected to fulfil its guarantees. Only when these descriptions - known as assumptions - hold, must the controller work towards its objectives. If a greater range of behaviours falls within the descriptions, we say the assumptions are weaker. We usually want to find the weakest set of assumptions that leaves the specification realisable, compelling the controller to satisfy its guarantees in as many situations as possible. A designer normally presumes that the assumptions will hold the majority of the time, and they serve to rule out the rarer obstructive behaviours.

At runtime, we may discover that the environment is acting differently than anticipated by the designer (or its behaviour may evolve over time), such that the assumptions are not always satisfied [5, 6]. If this is the case, we say that the environment is *violating* one or more of the assumptions, which we refer to as being *erroneous*, leading to the undesirable situation of the controller not being obliged to carry out its tasks. We would like to update the assumptions to be more reflective of the environment's behaviour, while leaving the specification realisable, and preferably also as weak and similar to the designer's original specification as possible.

Techniques have been developed that can, to a certain extent, gracefully handle violated assumptions, which we discuss further in Chapter 7 [7, 8, 9, 10]. However, very limited support exists for automatically updating assumptions at runtime; usually the system must be shut down and the specification modified manually by a designer. We propose a framework that attempts to correct violated assumptions at runtime, in such a way that the updated specification is not just realisable and reflective of the environment, but also exhibits the preferences mentioned above; our framework additionally seeks to find this optimal solution as quickly as possible.

Revision of assumptions can be achieved using nonmonotonic inductive logic programming (ILP), a field of symbolic artificial intelligence that learns general rules from observations, or revises rules with respect to observations. We use this method in the implementation of our framework to enjoy the benefits of nonmonotonic ILP theory revision systems, which include their completeness, consistency, capacity to enact complex semantic changes, and ability to be augmented with various other methods for ensuring the preferred solution is found [11, 12]. Indeed, the space of possible revised assumptions is large, so it may be expensive at runtime for a revision engine alone to search for solutions; moreover, since revision systems usually seek to enact as few changes as possible, the returned specification may not exhibit the qualities we require, such as realisability. The search space for the revision can be restricted and guided by setting parameters for the ILP system, such as the number and length of the rules it includes in its hypotheses. However, the appropriate parameters to be applied are not always known, and it may be especially difficult to find them when trying to trade off various requirements and preferences like weakness and realisability.

Our key contribution is our framework's exploitation of reinforcement learning (RL), a machine learning paradigm that sees an agent seeking to learn a policy for the best actions to perform in given situations. Our RL agent selects an array of parameter values to be provided to the theory revision system, a task for which RL is particularly suitable for several reasons. Firstly, as already mentioned, it is not always clear to humans how to find within the space of possible revised assumptions those that we consider *acceptable* (reflective of the environment and leading to a realisable specification), and of these, the ones that we consider *optimal*. Fortunately, RL agents do not need to be explicitly instructed how to achieve their task; we simply give them an indication of the quality of their performance via a reward/penalty, and they learn through training which actions lead to the highest reward, and consequently the best solution.

Secondly, different initial sets of assumptions may need updating in different ways to maintain realisability. Different environments may also necessitate different types of updates, to ensure that the new assumptions are indeed reflective of all of that environment's behaviours, rather than only the particular violating behaviour that was observed. The policy that a RL agent learns is domain-dependent, allowing us to find the acceptable and optimal solutions for different contexts. Thirdly, the agent converges on its policy by conducting extensive trial-and-error during the many training 'episodes' to which we subject it before deployment. At runtime, the policy is ready to enact, meaning that the agent can perform the optimal actions for correcting the specification with as little disruption as possible to the controller's progress.

1.2 Approach and contributions

Our proposed approach involves a RL agent receiving information about the current set of assumptions, such as their weakness and quantity, as well as about the violated ones in particular. Given this information, it selects a parameter combination with which to guide the revision system's search. We also provide some domain-dependent meta-constraints to ensure the semantic correctness of the revised assumptions. We conduct various checks on the revised specification produced by the revision engine, to see whether it is realisable, reflective of the environment, and the extent to which it adheres to our other quality preferences. We then provide rewards and penalties to the RL agent to inform it about the suitability of its selected parameters with respect to the resulting assumptions. After a certain number of training episodes, the RL agent converges on a domain-dependent policy for selecting, in as few attempts as possible, parameter combinations that revise different starting sets of assumptions in appropriate ways.

Our key contributions are the following:

- The demonstration of the use of RL for selecting appropriate parameters for ILP systems.
- The application of RL to guiding the search for acceptable and optimal sets of revised assumptions.
- The specification of domain-dependent meta-constraints to ensure the semantic correctness of revised assumptions.

1.3 Scope

The formal specifications from which controllers are synthesised are most commonly represented in temporal logic due to their expressiveness, of which Linear Temporal Logic (LTL) is considered to be the most popular [1, 2]. Synthesis of a controller from an LTL specification has doubly exponential complexity, whereas synthesis with a subset of LTL known as Generalized Reactivity of rank 1 (GR(1)) has lower, polynomial time complexity [2, 4]. Work on synthesis problems has therefore increasingly been restricted to GR(1) specifications, which will consequently also be the focus of this work, so that it can be applied to real-world controllers as they are commonly found and therefore be a useful contribution to ongoing efforts in the field [1, 13, 14, 15, 16, 17]. Most specifications used by controllers can be translated into GR(1) form [4].

As explained in 2.1.2, GR(1) assumptions are split into initial conditions, single-state and transition invariants, and fairness conditions. We focus on the revision of single-state and transition invariants. While our framework is capable of modifying initial conditions, we assume that these are easier for an analyst to specify correctly at design time, and for a user to update if desired upon deployment. Violations to fairness conditions are harder to detect at runtime and seeking to do so might be time-consuming, which might outweigh the benefits of conducting the check. Nevertheless, our framework could be extended in future work to allow revisions to fairness conditions for which the violation checks are conducted less frequently than those for violations to invariants, reducing the cost.

The framework is aimed at assumptions the correctness of which is in doubt, meaning that the assumptions are rarely likely to be satisfied and are therefore considered deserving of revision. In other words, we are bringing erroneous assumptions in line with the actual, enduring environment behaviour. Users are unlikely to want to modify acceptable assumptions to reflect anomalous environment behaviour, and doing so could lead to flip-flopping once the normal environment behaviour resumes, which may itself violate the revised assumptions. We therefore assume that our system is only executed in response to a user-defined number or pattern of repeated violations, which we propose to detect by conducting a satisfiability check at each timestep over the assumptions and the observed values of the variables $\mathcal{V}$ that appear in the specification. Our framework also allows the user to distinguish between assumptions that may be updated and those that are considered correct and should therefore not be revised if violated.

1.4 Ethical considerations

In consultation with the provided Ethics Checklist [18], we have concluded that our work does not present any serious ethical, legal or professional concerns. Controllers for autonomous systems have a very wide range of applications, and are likely to be increasingly present in industrial, commercial, domestic and other civilian applications. Autonomous systems can also have military uses, and controllers are notably used in drones [19], though these can also provide a range of essential nonmilitary functions, such as search-and-rescue missions.

While most autonomous systems can have military or other controversial uses, our work does not directly or in the short- to medium-term have any such function.

Chapter 2

Background

We provide here the preliminaries for our framework, which can be grouped into the representations and qualities of specifications; logic-based learning; and reinforcement learning.

2.1 Specifications

2.1.1 Linear Temporal Logic

LTL is an extension of propositional logic with temporal operators [20]. Its syntax is defined over a countable set $\mathcal{V}$ of propositional variables, the logical constants *true* and *false*, Boolean connectives and several temporal operators, by the following grammar shown in Backus–Naur form [4, 13]:

$$\phi ::= \textit{true} \mid \textit{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{X}\phi \mid \phi\mathrm{U}\phi.$$

where $p \in \mathcal{V}$. This can be expanded with the following [2]:

- $\neg(\neg\phi \wedge \neg\phi)$ is represented by $\phi \vee \phi$.
- $\neg\phi \vee \phi$ is represented by $\phi \rightarrow \phi$.
- $\textit{true}\mathbf{U}\phi$ is equivalent to $\mathbf{F}\phi$.
- $\neg\mathbf{F}\neg\phi$ is equivalent to $\mathbf{G}\phi$.

Satisfaction of LTL formulae is usually defined with respect to ω-words, which are infinite sequences of truth assignments of the variables in $\mathcal{V}$, where $\omega = \omega_0, \omega_1, \omega_2...$ In the context of controllers, each sequence describes a possible evolution of the controller and its environment, with each position in the sequence indicating what is true at a particular discretised timestep or state; we usually refer to them as execution traces [13]. In our problem we consider finite traces π, also known as finite words, which do not satisfy a given assumption formula; we often refer to this as the trace *violating* the assumption. The following is an example of a finite trace in the representation we use in our implementation. *fardistance, liftcommand, dropcommand, gocommand* and *idlecommand* are Boolean variables in $\mathcal{V}$, borrowed from our case study in Chapter 5. The header $\rightarrow$ *State: 1.1* $\leftarrow$ introduces the truth values of the variables in position 1 of trace 1, and *End* indicates the end of the trace:

```
→ State: 1.1 ←
fardistance = TRUE
liftcommand = TRUE
dropcommand = FALSE
gocommand = FALSE
idlecommand = FALSE
→ State: 1.2 ←
fardistance = TRUE
liftcommand = FALSE
dropcommand = FALSE
gocommand = FALSE
idlecommand = TRUE
End
```

The satisfaction of an LTL formula by a finite word is defined over positions *i* in the trace, with $0 \leq i \leq$ *last*, where *last* is the final position in the word. The word is said to satisfy a formula ϕ at position i (represented by $\pi, i \models \phi$) according to the following rules [21]:

$\pi, i \models true$ always
$\pi, i \models false$ never
$\pi, i \models p$ iff $p \in \pi(i)$, meaning p is true at the first position in the sequence
$\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$
$\pi, i \models \phi \wedge \psi$ iff $\pi, i \models \phi$ and $\pi, i \models \psi$
$\pi, i \models \mathbf{X}\phi$ iff $i <$ *last* and $\pi, i+1 \models \phi$
$\pi, i \models \phi \mathrm{U} \psi$ iff $\exists j$ where $i \leq j \leq last$ such that $\pi, j \models \psi$ and $\forall k$ where $i \leq k \leq j$ $\pi, k \models \phi$
$\pi, i \models \mathbf{F}\phi$ iff $\exists j$ where $i \leq j \leq last$ such that $\pi, j \models \phi$
$\pi, i \models \mathbf{G}\phi$ iff $\forall j$ where $i \leq j \leq last \quad \pi, j \models \phi$

As such:

- $\mathbf{X}\phi$ means that ϕ is true at the next valuation in the sequence;
- $\phi\mathbf{U}\psi$ means that ϕ remains true until ψ becomes true;
- $\mathbf{F}\phi$ means that ϕ eventually becomes true; and,
- $\mathbf{G}\phi$ means that ϕ remains true until the end of the word.

In our implementation we define the satisfaction of a formula by a finite trace slightly differently, as explained in the section below.

2.1.2 Generalized Reactivity of rank 1

The GR(1) subset of LTL is defined by its restricted syntactic structure [13]. The set of variables $\mathcal{V}$ is divided into the set of input variables $\mathcal{X}$ controlled by the environment, and the set of output variables $\mathcal{Y}$ controlled by the controller [4, 13]. A GR(1) formula has the form $\phi^{\mathcal{E}} \rightarrow \phi^{\mathcal{S}}$, where $\phi^{\mathcal{E}}$ represents a conjunction of subformulae called the assumptions, and $\phi^{\mathcal{S}}$ represents a conjunction of subformulae called the guarantees. The assumptions include one or more of the following [13]:

- **Initial conditions** - a formula $\varphi^{\mathcal{E}}_{init}$ in the form $B(\mathcal{X})$, where this signifies a Boolean formula over the variables appearing in $\mathcal{X}$;
- **Invariants** - a set of LTL formulae $\varphi^{\mathcal{E}}_{inv}$ of the form $\mathbf{G}B(\mathcal{V} \cup \mathbf{X}\mathcal{X})$; and,
- **Fairness conditions** - a set of LTL formulae $\varphi^{\mathcal{E}}_{fair}$ of the form $\mathbf{GF}B(\mathcal{V})$.

The guarantees include one or more of the following [13]:

- **Initial conditions** - a Boolean formula $\varphi^{\mathcal{S}}_{init}$ in the form $B(\mathcal{V})$;
- **Invariants** - a set of LTL formulae $\varphi^{\mathcal{S}}_{inv}$ of the form $\mathbf{G}B(\mathcal{V} \cup \mathbf{X}\mathcal{V})$; and,
- **Fairness conditions** - a set of LTL formulae $\varphi^{\mathcal{S}}_{fair}$ of the form $\mathbf{GF}B(\mathcal{V})$.

The initial conditions describe the starting state of the environment and system; single-state invariants (those without the $\mathbf{X}$ operator) describe what must always be true at any given timestep; transition invariants (those with the $\mathbf{X}$ operator) describe what must be true at the next timestep given what is true at the current timestep; and fairness conditions describe what must become true infinitely many times in an execution trace [2]. Traces satisfy GR(1) formulae according to the same rules as general LTL [13].

We focus on revising invariants in our framework, and we restrict these to the form $\mathbf{G}$ ($\bigwedge a_i \rightarrow$ b) and $\mathbf{G}$ ($\bigwedge a_i \rightarrow \mathbf{X}$b). In our implementation, we define a finite trace as satisfying a transition invariant if the antecedent holds at the last timepoint. In the definition given in the previous section, the antecedent must hold at the penultimate timepoint and the consequent must hold at the last timepoint.

2.1.3 GR(1) games and realisability

The implication structure ($\phi^{\mathcal{E}} \rightarrow \phi^{\mathcal{S}}$) of GR(1) specifications means that in order to be satisfied, whenever the assumptions are satisfied by the environment, the controller must satisfy the guarantees. As such, we can conceive of a two-agent game, whereby the environment chooses truth valuations for the input variables that satisfy the assumptions but tries to force a violation of the guarantees. In turn, the controller chooses assignments for the output variables to ensure that the guarantees remain satisfied [2, 22]. A GR(1) specification is said to be *realisable* if a controller can be synthesised with a winning *strategy* that allows it to continue satisfying the guarantees for any of the environment's chosen inputs. If no such controller exists, the specification is *unrealisable*, in which case the environment has a *counterstrategy* by which it can satisfy the assumptions and force a violation of the guarantees [13, 17, 23]. A GR(1) specification is strictly realisable if and only if the following LTL formula is realisable [24]:

$$\varphi^{sr} = (\varphi^{\mathcal{E}}_{init} \rightarrow \varphi^{\mathcal{S}}_{init}) \wedge (\varphi^{\mathcal{E}}_{init} \rightarrow \mathbf{G}((\mathbf{H}\varphi^{\mathcal{E}}_{inv}) \rightarrow \varphi^{\mathcal{S}}_{inv})) \wedge \\ (\varphi^{\mathcal{E}}_{init} \wedge \mathbf{G}\ \varphi^{\mathcal{E}}_{inv} \rightarrow (\bigwedge_{i\in 1..n}\mathrm{GF}\varphi^{\mathcal{E}}_{fair\ i} \rightarrow \bigwedge_{j\in 1..m}\mathrm{GF}\varphi^{\mathcal{S}}_{fair\ j}))$$

A counterstrategy can be represented by a labelled transition system (LTS) that, for every state, chooses environment input that from its winning state for all output choices by the controller lead to computations satisfying $\neg\varphi^{sr}$. The counterstategy LTS ensures $\neg\varphi^{sr}$ is satisfied either through forcing the controller to a deadlock by violating a system initial condition or system invariant, or through satisfying all the environment fairness conditions $\varphi^{\mathcal{E}}_{fair}$ but preventing at least one system fairness condition from ever being satisfied [24].

In many cases, a designer will be able to intuit at least some of the types of assumption that are required for their specification to be realisable, and will add them themselves. Techniques also exist for making realisable a specification that remains unrealisable; these involve adding assumptions that preclude the environment behaviour exhibited by the counterstrategy. The updated GR(1) specification then requires the controller to satisfy the guarantees only when the more restrictive assumptions are satisfied. The assumptions are added iteratively: a counterstrategy is found; a new assumption is generated in response to the counterstrategy; the specification is checked again and a new counterstrategy will be found if the specification is still unrealisable. Each iteration can be called a strengthening step, in that the restrictions on the environment are strengthened, and the approaches are often referred to as counterstrategy-guided assumption refinement [15, 25, 26, 24, 13].

Of these techniques, the most recent and automated is [13], which uses Craig interpolants to derive from the counterstrategy an assumption that directly targets a so-called unrealisable core. This approach of iteratively added assumptions can be seen as forming a tree of possible refinements. The root of the tree is the empty set of assumptions, the branches represent different possible sequences of iterative refinements (at each point an unrealisable specification might be able to be refined in several different ways, so the branches split further), and realisable specifications form the leaves. Figure 2.1 illustrates how a simple refinement tree with three levels of strengthening might look. Analysis of the realisable leaves of such a tree contributed to our definition of how the RL agent should restrict the search space in 3.3.

2.1.4 Weakness

As described in the section above, a certain number of assumptions are usually required for a given specification to be realisable. On the other hand, if the environ-

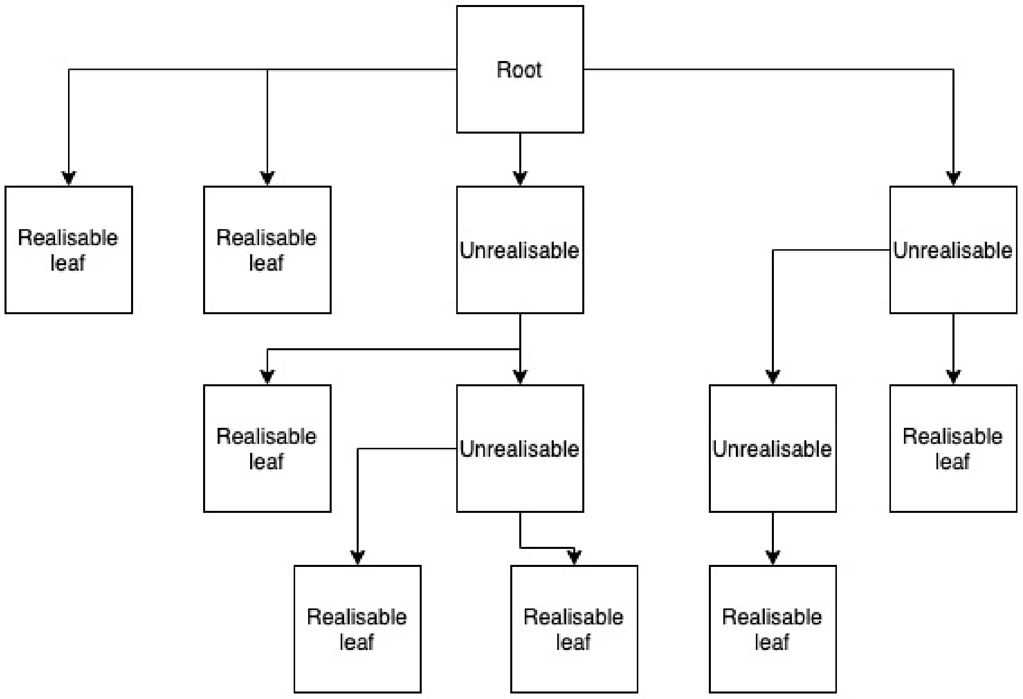

Figure 2.1: A refinement tree with three levels of refinement

ment's permitted behaviour is unnecessarily limited, the structure of GR(1) specifications is such that there are fewer situations in which the controller is required to satisfy the guarantees, whereas the user would intuitively want the goals to be fulfilled in as many circumstances as possible. Consequently for a realisable specification, we desire that the assumptions are as *weak* as possible, meaning that an environment satisfying the assumptions has a greater degree of freedom over its behaviours [14, 27, 28]. As such there is a trade-off to be found whereby assumptions are strong enough for the specification to be realisable, without being any stronger than necessary.

A number of definitions have been proposed for the weakness of assumptions. Earlier understandings centred on logical implication, whereby "a formula ϕ_1 is weaker than a formula ϕ_2 if $\phi_2 \rightarrow \phi_1$ is valid" ([14]) [25, 28]. However, [14] highlights examples of assumptions that refer to different subsets of variables such that one assumption does not imply the other, and yet one intuitively permits more behaviours than the other: consider the fairness conditions $\mathbf{GF}(r_1)$ and $\mathbf{GF}(r_2 \wedge r_3)$.

[14] instead proposes a quantitative measure for weakness that is more closely aligned with permissiveness than the earlier definitions, and which we outline here. An ω-language is a set of ω-words; a regular ω-language is one that is accepted by a deterministic Muller automaton, a type of ω-of which the acceptance condition is that the set of all state visited infinitely often is one of the sets in the acceptance collection. L(φ) denotes the regular ω-langauge satisfying a formula φ. The *Hausdorff dimension* is a measurement of the degrees of freedom of an ω-language,

conducted by quantifying the number of different evolutions that are permitted to an ω-word once its run remains in a set within the Muller automaton's acceptance collection [29]. The weakness of a GR(1) formula $\varphi = (\varphi_{init} \wedge \varphi_{inv} \bigwedge_{i \in 1..n} \varphi^i_{fair})$ is a pair $(d_1(\varphi)$,$d_2(\varphi))$, where $d_1(\varphi)$ is the Hausdorff dimension of L(φ), and $d_2(\varphi)$ is the Hausdorff dimension of L($\varphi_{init} \wedge \varphi_{inv} \wedge \bigvee_{i \in 1..n} \neg\ \varphi^i_{fair}$). We extract $d_1(\varphi)$ as our value for the weakness of a set of assumptions in our implementation [14].

Minimality of assumptions

A concept closely related to weakness is that of the minimality of a set of assumptions, which we define as being when the set contains no more assumptions than are needed for realisability [2]. When iteratively adding assumptions as per the counterstrategy-guided approaches described in 2.1.3, assumptions added in later iterations may make those generated earlier redundant, that is, they may be sufficient for realisability without needing all of the assumptions added earlier. This means that the final set of assumptions is not minimal, and the unnecessary assumptions mean that the set is not as weak as possible. [2] proposes an algorithm that removes redundant assumptions during the refinement process.

2.1.5 Similarity and coverage

When bringing erroneous assumptions in line with the environment's true behaviour, we wish to do so in a way that does not change their syntax more than is necessary, and which respects as far as possible the behaviours permitted by the original assumptions. Even if the original assumptions are not exactly correct for the given environment, a designer may still desire that the overall patterns of their assumptions remain the same, as they may have had in mind the general circumstances in which they want their system to fulfil its goals. Equally, whether the original assumptions were specified by a designer or added in a refinement procedure or some of both, it is undesirable for the revised assumptions to permit a significantly different set of behaviours. This is especially true if other assumptions are later found to be erroneous or the environment's behaviour subsequently changes. We want the situations in which the controller respects its guarantees to remain as constant as possible, to prevent it from flip-flopping between doing so and not doing so and from other undesirable behaviours.

Previous works have used so-called witness traces as a way to measure and enforce coverage of desired behaviour [30, 31, 32, 33]. Model checkers are able to generate execution traces that satisfy a given temporal formula and are as such a witness to its satisfaction. We can use the number of witness traces of a given set of assumptions which also satisfy another set of assumptions as a measure of the extent to which the sets permit similar behaviour.

2.2 Logic-based learning

2.2.1 Inductive logic programming

Inductive logic programming (ILP) is a field of symbolic artificial intelligence that uses logic programming to represent existing knowledge, and seeks to learn general rules from observations [34, 35, 36, 37]. ILP systems are presented with a background knowledge, and their goal is to find hypotheses that entail all of a given set of positive examples and none of a set of negative examples (logical entailment of consequences from premises occurs if and only if every truth valuation of variables that satisfies the premises also satisfies the consequences).

Monotonic ILP systems explore the solution space in the form of a lattice of hypotheses ordered by generality, where a general hypothesis explains more observations than a specific one [38, 39]. Such a system traverses the lattice by taking generalising steps to explain more of the positive examples, or restricting steps to entail fewer negative examples. Some implementations generalise from a more specific starting point in a bottom-up approach, while others search from the more general to the more specific in a top-down manner [38]. Nonmonotonic ILP introduces *negation as failure* represented by the symbol *not*, allowing reasoning with incomplete knowledge. Nonmonotonic ILP systems therefore use 'normal clauses' which are of the form h $\leftarrow$ p_1, ... p_m, not n_1, ..., not n_m, where each p_i is a positive literal and each n_i is a negative literal [11].

A nonmonotonic ILP task can be formalised as the tuple $\langle B, L_H, E\rangle$. *B* is the background theory consisting of a set of normal logic clauses, and *E* is a set of ground literals that comprise the observations $\{e_1$, ..., e_m, not e_{m+1},..., not $e_n\}$, including the positive examples $\{e_1$, ..., $e_m\}$ and the negative examples $\{e_{m+1}$,..., $e_n\}$ [38]. The language bias L_H reduces the search space by defining the types of normal clauses that will constitute the hypotheses, and is most commonly expressed as a set of mode declarations [11]. Mode declarations refer either to the head or body literal of a rule, and are respectively of the forms *modeh(s)* and *modeb(s)*. The schema *s* is a literal with argument placeholders of the form '+*type*', '-*type*' or '#*type*', where *type* is the argument's type, '+' denotes an input variable, '-' denotes an output variable, and '#' denotes a constant. An output variable is a free variable in a body literal, while an input variable in a body literal must have either appeared as an input variable in the head of the clause or as an output variable in a preceding body literal in the clause [38]. A clause is compatible with the mode declarations if [12]:

- its head is compatible with the schema of a head mode declaration;
- every body literal is compatible with the schema of a body mode declaration; and,
- the variables obey the link constraints indicated by the input and output placeholders.

The set of clauses compatible with these mode declarations is $\mathcal{R}_M$. A hypothesis *H* is a solution to the ILP task if it consists of normal clauses from $\mathcal{R}_M$, *B*∪*H* is consistent, and

- *B*∪*H* $\models e_i$ for every positive example e_i; and,
- *B*∪*H* $\not\models e_j$ for every negative example e_j [38].

Here $\models$ denotes brave induction [40], meaning that if *B*∪*H* is consistent, there is at least one minimal model of *B*∪*H* that covers the example [38].

In addition to the language bias we can set parameters for the ILP task, expressing restrictions on the nature of the hypothesis. Common parameters include the number of rules that are present in the hypothesis, the number of times a body literal is used in a rule, and the total number of body conditions appearing in each rule; we can even specify different numbers of conditions for rules depending on their head literal.

2.2.2 ILP tasks as abductive search

[41] proposes a nonmonotonic ILP system called the top-directed abductive learning approach (TAL). It improves on monotonic approaches by translating the ILP task into a semantically equivalent abductive (ALP) task [42], which seeks to explain observations by assuming ground facts called *abducibles* [38]. An ALP task is defined by the tuple $\langle B, A, IC, O\rangle$, where *B* is the set of normal clauses constituting the background knowledge, *A* is the set of literals that can be abduced, *IC* is a set of integrity constraints expressed as normal denials, and *O* is the observation. A solution Δ to this task is a subset of *A* such that Δ is consistent with *B*, *B* $\cup\Delta \models$ *O* and *B* $\cup\Delta \models$ *IC*. TAL creates a meta-level encoding of the clause space $\mathcal{R}_M$ of the ILP task by flattening each clause into an atom. These atoms comprise the abducible set *A* of the ALP task, such that the abductive solutions of the ALP correspond to the inductive hypotheses of the ILP task [38]. (In fact, the meta-level encoding uses R^r_M, the canonical representation of $\mathcal{R}_M$, where each clause in R^r_M represents the set of clauses in $\mathcal{R}_M$ that are equivalent except for the ordering of body literals that are not affecting by the link constraint over input and output variables - this allows TAL to avoid replication of hypotheses [38, 11]).

Answer set programming for ILP

Answer set programming (ASP) is a knowledge representation technique based on the stable model semantics and used for declarative problem solving [43]. Solutions to ASP problems are in the form of models of the program, and several fast ASP solvers have been designed based on satisfiability solving [11].

[44] presents ASPAL, a method for solving ILP tasks by encoding them as an ALP task in ASP. ASPAL encodes the nonmonotonic ILP task $\langle B, M, E\rangle$ (where *M* is the

mode declarations) as follows. The function *id* assigns a unique identifier to each clause r_i in R^r_M $h_i \leftarrow \overline{b_i}$, where $\overline{b_i}$ is the list of body literals for the rule r_i. For every r_i in R^r_M, the so-called *top theory* $\top$ includes $h_i \leftarrow \overline{b_i}, rule(id(h_i \leftarrow \overline{b_i}), \overline{C})$, where $\overline{C}$ is a list of the constant arguments in r_i. The set of abducibles $A^\top$ includes $rule(id(h_i \leftarrow \overline{b_i}), \overline{C})$ for each r_i. The ILP task $\langle B, M, E\rangle$ is translated into the ALP task $\langle B', A^\top, IC, O\rangle$, where *B' = B* $\cup \top \cup \{examples \leftarrow \wedge_{e \in E} e\}$, *IC* includes the constraint $\{\perp \leftarrow not examples\}$ and the observation is empty. The constraints guarantee that all answer sets that are solutions to the task cover all the positive and none of the negative examples. The unique identifier of each abducible is used to translate an abductive solution Δ back into its equivalent inductive hypothesis *H* [38, 12].

Modern ASP solvers such as *clingo*, which we use in our implementation, allow the user to specify preferences over the computed answer sets, which constitute parameters for the ILP system [45]. For each abducible a_i, the user can specify a weight w_i. An optimisation statement is of the form *#optimise[*a_i = w_i, ..., a_n = w_n*]*, where *optimise* is either *maximise* or *minimise*. The weights of all the abducibles in each answer set are summed; a minimisation statement considers answer sets with the lowest total weights to be optimal, while the opposite is true for a maximisation statement. ASPAL assigns the length of the rule represented by each abducible as the abducible's weight, and employs a minimisation statement to favour shorter hypotheses. *clingo* also supports aggregates of the form *min*$\{a_1$, ..., $a_n\}$*max*. The user can specify *min* $\geq$ 0 and *max* to define the minimum and maximum number of abducibles that can appear in any answer set [44].

2.2.3 Theory revision

While early applications of ILP focused on learning hypotheses from scratch, the field was expanded to include methods for refining existing theories in various ways [46]. Of the types of theory refinement, theory revision involves modifying a given theory to change its consequences such that, together with a background knowledge, the revised theory entails a set of positive examples and not a set of negative examples [46, 38, 11]. The theory is usually revised to cover all of the positive examples and none of the negative ones, but, as with traditional ILP tasks, provisions can be made to allow for noise. The task can be formalised as the tuple $\langle B, \mathcal{R}, R, E\rangle$, where *B* is the background theory that is fixed, $\mathcal{R}$ is the rule space, *R* $\subseteq \mathcal{R}$ is the revisable theory, and *E* is the set of examples. A revised theory *R'* is a hypothesis for this task if and only if *R'* $\subseteq \mathcal{R}$ and *B* $\cup$ *R'* $\models$ *E* [11].

Early methods for theory revision usually iteratively adjusted the theory using various operators to generalise and specialise over the hypothesis lattice, centring around the deletion and addition of clauses and literals within clauses [46, 47]. At each iteration a locally greedy search is conducted for the best operator to apply, and the iterations terminate when the required example coverage is achieved [11].

An important consideration when applying these revision operators is the concept of the **minimality** of the revision, in that it is usually preferred that the revised theory is as similar as possible to the original theory [11]. Minimality has been defined in various ways, but it is usually understood in terms of the number of revision operators applied, though different types of operator can also be weighted differently [46, 11, 47].

Theory revision through nonmonotonic ILP

[11] proposes that theory revision can be achieved through nonmonotonic ILP, whereby semantically correct inductive hypotheses are computed that prescribe which set of revisions should be applied to change the existing theory syntactically; the revision system is implemented as RASPAL [38]. These changes consist of the addition or deletion of whole rules, or of the body literals of existing rules. Doing so overcomes the issue of iteration-based techniques that may miss non-atomic or otherwise more complex revisions that are not locally optimal, while being complete and guaranteed to be consistent with the examples. This approach uses mode declarations to define the rule space $\mathcal{R}$, so the theory revision task is formalised instead as the tuple $\langle B, M, R, E\rangle$ where *R,R'* $\subseteq \mathcal{R}_M$. The mode declarations define which literals can form the heads of new rules, and which can be included in the bodies of new or revised rules [11].

This method consists of a pre-processing stage, a learning stage and a transformation stage. While the revision operation of adding a new rules corresponds to the ILP task of learning a clause, in order to include the other revision operators in the hypothesis, the mode declarations are extended in the pre-processing stage with *modeh(extension(#rule_id, +vars))* and *modeh(delete(#rule_id, #body_ id))*. The argument *#rule_id* refers to an existing clause in the revisable theory in which a body literal is to be added or removed, and *vars* identifies the list of variables in that clause which are involved in the revision operation [38]. Also in the pre-processing stage, further rules are added for every normal clause $h_i \leftarrow b_{i,1}, ..., b_{i,n}$ in the revisable theory *R*. We provide here the rules added as in [38], which implements the learning stage in terms of ASPAL, though [11] specifies that any nonmonotonic ILP system can be employed and different versions of these rules can be added:

- $h_i \leftarrow$ *try(i, 1, vars(*$b_{i,1}$*)),...,try(i,n,vars(*$b_{i,n}$*)),extension(i,vars(*r_i*))*;
- *try(i,j,vars(*$b_{i,j}$*))* $\leftarrow b_{i,j}$*, not delete(i,j)*, for each *try(i,j,vars(*$b_{i,j}$*))*;
- *try(i,j,vars(*$b_{i,j}$*))* $\leftarrow$ *delete(i,j)*, for each *try(i,j,vars(*$b_{i,j}$*))*;
- $\bot \leftarrow$ *delete(i,j),* $\{$*extension(i,vars(*r_i*))*$\}$*0,* for each *delete(i,j)*.

The indices *i* and *n* respectively identify each clause in the revisable theory and the conditions in each clause; *vars(*r_i*)* is the list of variables in r_i; and *vars(*$b_{i,j}$*)* is the list of variables in $b_{i,j}$. Each *try* clause checks whether the condition $b_{i,j}$ should be kept in the revised version of the *i*th clause of the revisable theory, otherwise the relevant *delete(i,j)* is learnt for the inductive hypothesis. Following the ASP aggregate encoding, {*extension(i,vars(r_i))*}*0* indicates that there are no instances of *extension(i,vars(r_i))*, such that the constraints on each *delete(i,j)* mean that a deletion of a condition can only be learnt if the clause in which the condition appears remains in the revised theory [38].

For the augmented revisable theory $\tilde{R}$ and the extended mode declarations $\tilde{M}$, the theory revision task is formalised as the nonmonotonic ILP task $\langle B \cup \tilde{R}, \tilde{M}, E \rangle$. The inductive hypothesis *H* is the set of operations that must be applied to the original revisable theory *R* in the postprocessing stage to reach the revised theory *R'*. This set can include delete facts, new clauses compatible with the mode declarations, or clauses with an *extension* literal as the head. *R'* is semantically equivalent to *R* $\cup$ *H* and is therefore consistent with *E* [11]. The revised theory is derived from the revisable theory and the inductive hypothesis as follows [38]:

- For each pair $r_i \leftarrow b_1$,...,b_n from the revisable theory and *extension(r_i, vars(r_i))* $\leftarrow b_{n+1}$,...,b_m from the inductive hypothesis, the revised theory includes the clause $r_i \leftarrow b_1$,...,b_n,b_{n+1},...,b_m.

- For every *delete(i,j)* in the hypothesis, the condition $b_{i,j}$ is deleted from the clause r_i that has been retained in the revised theory.

- Any clause in the hypothesis that does not have *delete* or *extension* as its head literal, and was not in the revisable theory, is added to the revised theory.

- Any clause in the hypothesis that does not have *delete* or *extension* as its head literal and does not have an associated *extension* in the hypothesis is not included in the revised theory.

The notion of minimality provided by a given ILP system also ensures the minimality of the theory revision; in the case of ASPAL, this can be provided by a minimisation statement concerning the abducible revision operations [11].

2.2.4 Learning with hypothesis constraints

In the background discussion thus far, we have mentioned that some hypotheses within the space of possible solutions to a given learning task may be preferred over others. These preferences have so far been expressed as the language bias that defines the literals that can appear in the heads or bodies of clauses; the complete coverage of the examples by the hypotheses or the allowance for some noise; the common, but not universal, preference for more compressed hypotheses; and, in the ASPAL implementation of an ILP task, the ability to specify maximum and minimum numbers of rules to be abduced. [12] proposes a method called constraint-driven bias, which allows a user to specify further domain-dependent preferences over the computed hypotheses, such as the structure of the rules and which literals should appear in which patterns. These preferences are expressed as denials over the search space, such that computed hypotheses must also respect these constraints.

[12] proposes that literals compatible with the mode declarations be referenced with labels. The label of such a literal consists of its predicate name and constant arguments, and, in the case of a negated literal, the predicate name is prefixed with "*not_*". [12] gives the example of the body mode declaration *modeb(not p(+int,#int,-int))* and the compatible literal *not p(X, 2, Z)*; this will be given the label *not_p(2)*. Depending on the user's needs, the labels can be ground in the top theory or left unground (in this example, *not_p(X)*) to be instantiated with respect to the computed hypotheses. Variables of which the type is head label can be denoted L_h, while those of body label type can be denoted L_b.

Implementation in ASPAL

[12] proposes several templates for possible domain-dependent constraints and implements them by extending ASPAL, both because constraints already form an essential consideration of any ALP task [42, 48], and because the meta-level encoding of the hypothesis space that is conducted in the TAL approach lends itself to the deployment of meta-constraints over hypothesis structures. The encoding of the constraints requires the following meta-predicates, where the learning system automatically generates for each clause and condition in $\mathcal{R}_M$ the unique clause identifier *R* and body literal position *I*:

- *is_rule(R,L_h)* denotes that there is a clause with identifier *R* and head literal labelled L_h; and,
- *in_rule(R,L_h,L_b,I)* denotes that the clause with identifier *R* and head literal labelled L_h has a body literal labelled L_b at position *I* in the clause.

For a rule $r = h \leftarrow b_1,...,b_n$ with identifier R_{id} in the hypothesis *H*, [12] specifies that the associated meta-level information is encoded as the set of ground literals $\{in\_rule(R_{id},L_h, L_{b_1},1), ..., in\_rule(R_{id},L_h, L_{b_n},n), is\_rule(R_{id},L_h)\}$. For an ASPAL task extended with constraint-driven bias, the top theory $\top_M$ is constructed from $\top \cup \mathcal{M}$, where $\top$ is the conventional top theory of the ASPAL task, and $\mathcal{M}$ is generated as follows. For each rule $h \leftarrow b_1, ..., b_n$, $rule(id(h \leftarrow b_1, ..., b_n), \overline{C})$ in $\top$, $\mathcal{M}$ includes the following rule with an aggregate head to map each rule in $\mathcal{R}_M$ to its meta-level information:

$$n + 1\{is\_rule(i(\overline{C}),l_h), in\_rule(i(\overline{C}),l_h(\overline{C}_h),l_{b_1}(\overline{C}_{b_1}),1), ..., \\ in\_rule(i(\overline{C}),l_h(\overline{C}_h),l_{b_n}(\overline{C}_{b_n}),\mathrm{n})\}\ n + 1 \leftarrow rule(id(h \leftarrow b_1, ..., b_n), \overline{C})$$

If $rule(id(h \leftarrow b_1, ..., b_n), \overline{C})$ is abduced, the mapping forces the meta-level information for the rule *r* to be inferred. The meta-constraints over this meta-level information therefore mean that only rules satisfying the constraints can be abduced and form part of the inductive hypothesis.

As such, [12] implements a learning task $\langle B, M, E, IC \rangle$, where *IC* is the constraint-driven bias, as the abductive task $\langle \tilde{B}, A^{\top}, \tilde{I}, O \rangle$, where the observation *O* is empty, the background theory $\tilde{B} = B \cup \top_M \cup \{examples \leftarrow \wedge_{e \in E} e\}$, $A^{\top}$ is the same set of abducibles as in the conventional ASPAL task, and $\tilde{I} = \{\bot \leftarrow not\ examples\} \cup IC^t$, where IC^t is the ASP encoding of the domain-dependent constraint-driven bias *IC*. The inductive hypotheses are derived by applying the reverse translation to the rule encodings of the abductive solution [12].

2.3 Reinforcement learning

Rather than being explicitly instructed how to achieve their objectives, RL agents are given an indication of how well they are fulfilling or progressing towards their goals in the form of a reward function. Agents learn the optimal policy through trial and error over a multitude of training episodes, in which they experiment with actions and find out the usefulness of performing a given action in a given state by discovering the reward they receive. In some cases, RL agents are designed to evolve their policy at runtime, by balancing continued exploration of alternative actions with the exploitation of the existing policy.

2.3.1 Environment

A RL agent interacts with a dynamic environment that is at least partially observable. The agent must be able to observe the current state of the environment in the form of, for example, sensor readings or symbolic representations, and take one of a number of problem-specific actions that affect its environment [49]. The agent then receives a new observation about the subsequent state of the environment in response to its action, and possibly also a reward.

The agent's environment might be physical, digital or of some other nature, such as symbolic. We highlight here the distinction between the environment of a controller and that of a RL agent. While these two types of agent can indeed function in the same type of environment, in our problem they are separate. Our RL agent acts within a symbolic environment, where its objective is to help find assumptions that reflect the controller's physical or digital environment.

2.3.2 Reward function

The reward function determines the reward presented to the RL agent for reaching a new state by taking a particular action in a given state. The agent must seek to maximise its cumulative reward over the entire episode, while also taking into account a discount factor, that may weight rewards received at all timesteps equally, or may place greater importance on near-term rewards over those received further in the future. With a full discount, the agent is only concerned about its immediate reward.

Reward functions are defined on a case-by-case basis, to incentivise the agent to learn to accomplish the specific task at hand; we detail our implementation in 4.4.4. Depending on the problem, the agent might only receive a reward for reaching certain desirable states such as the terminal state. Equally, the agent may be presented with a negative reward, also called a penalty, for reaching undesirable states, or it may receive a penalty for every action that does not take it to the goal state, to encourage the agent to reach the goal as quickly as possible.

2.3.3 Policies and value functions

The policy of a RL agent is a mapping from each state to the action to be taken in that state; an optimal policy indicates what action should be taken in a given state so as to maximise the cumulative rewards received by the agent during the remainder of the episode. The policy is both determined by, and itself determines, the so-called *state values* in some algorithms, or the *state-action pair values* in approaches such as Q-learning, a model-free RL method that we employ. The Q-value of a given state-action pair is the sum of the rewards that will be received by the agent if it follows a policy beginning from taking that action in that state [49].

In approaches dealing with state values, an optimal policy means that in each state, the action is chosen that is most likely to bring the agent to the state with the highest value of the subsequent possible states; in Q-learning, the action with the highest Q-value for that state is chosen. The learning process involves the agent experimenting with different actions to explore the average rewards received after taking given actions in given states, which allows the agent to update the values, and thereby also its policy. Early approaches used tabular methods for storing and updating these values, but states and actions are usually very numerous if not also continuous, so it is rarely possible to explore the search space exhaustively, nor to store such a large table. This means that the value function, which maps a state or state-action pair to its value, must be approximated. Recent developments have seen deep neural networks employed to this end, creating the field of deep RL [49, 50].

A deep Q-network is a neural network used to approximate the Q-value function. The state observations are given as inputs, and the outputs are the Q-values for all possible actions. The action with the highest value according to the network's output is chosen to be enacted in each state. The loss between the predicted Q-value and the Q-value observed through the agent's experimentation is backpropagated through the network to update its parameters. The network should eventually converge to define an optimal policy, as long as it does not get trapped in a local minimum [51].

Chapter 3

Framework

3.1 Introduction

We seek to update violated assumptions in such a way as to permit as similar behaviours as possible to the original ones while also reflecting the environment, with the objective of enabling the controller to fulfil its guarantees with as little interruption as possible. Crucially, the revised assumptions should also find an optimal balance between weakness and realisability. If the assumptions are too weak, the specification may not be realisable and so the guarantees will not be achieved until a series of strengthening steps is carried out, which may both be costly and move the specification too far from the designer's original intent. On the other hand, insufficiently weak assumptions mean that a narrower range of environment behaviours respect the assumptions, which are consequently more likely to be violated again. If our framework needs to be executed in response to a runtime violation, a benchmark for success would be that it quickly revises the assumptions in such as way that the framework needs to be executed as little as possible afterwards, because the improved assumptions are subject to few subsequent violations.

Our framework consists of several phases, which together work to ensure that the assumptions are modified so as to give a solution that is acceptable (realisable and reflective of the environment's behaviour), and as close as possible to optimal (weak and similar to the original set). Our most significant contribution is the RL phase, which we propose is suited to accomplishing the trade-offs due to its ability to learn a domain-dependent policy without the need for an explicit model of its operating environment [52]. The sections below discuss the various considerations behind the key phases of our approach, and Chapter 4 describes which elements of our general framework we have implemented for our proof of concept, and how we have done so. Figures 3.1 and 3.2 respectively illustrate the main interactions between core components of our framework, and the system's usage at runtime.

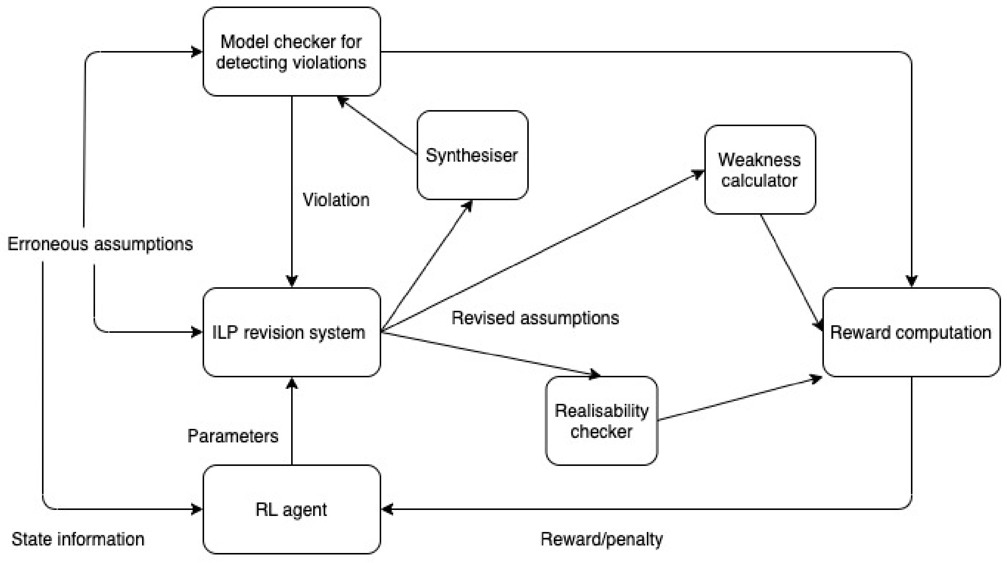

Figure 3.1: Interactions between the core components of the framework

3.1.1 Toy example

We introduce here a toy example that we use later in the chapter for illustrating considerations relating to revisions. We have also used this example for testing our implementation, but we introduce in Chapter 5 a different, more realistic specification for experiments on our proof of concept. The toy example consists of two doors, each of which can be hit or have its doorbell pressed. The true environment behaviour dictates that these events are respectively accompanied by a knocking sound or a ringing sound. The controller's guarantees are that if a door gives a knocking or ringing sound it should be answered. We can formalise this as follows:

Input variables $\mathcal{X}$ = {onebellpressed, twobellpressed, onebellrings, twobellrings, onedoorhit, twodoorhit, oneknocksound, twoknocksound}

Output variables $\mathcal{Y}$ = {onedooranswered, twodooranswered}

System invariants φ^{S}_{inv} =
{**G** (onebellrings $\rightarrow$ onedooranswered),
G (oneknocksound $\rightarrow$ onedooranswered),
G (twobellrings $\rightarrow$ twodooranswered),
G (twoknocksound $\rightarrow$ twodooranswered)}

Initial system conditions φ^{S}_{init} = {($\neg$onedooranswered & $\neg$twodooranswered)}

The designer has correctly specified the initial environment conditions and one environment invariant for each door, but has made a mistake with the other invariant for each door. The correct assumptions are as follows:

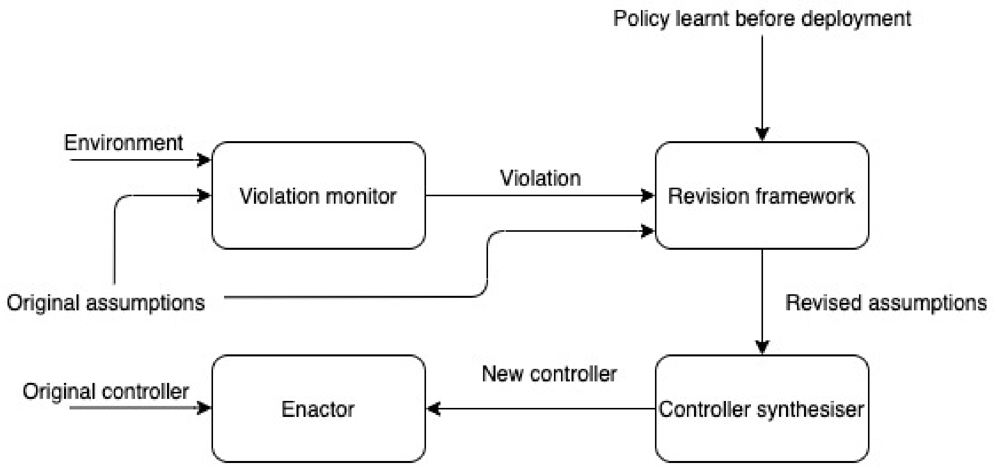

Figure 3.2: The usage of our framework at runtime

Initial environment conditions $\varphi^{\mathcal{E}}_{init}$ = {(¬onebellrings & ¬onedoorhit & ¬onebellpressed & ¬oneknocksound & ¬twobellrings & ¬twodoorhit & ¬twobellpressed & ¬twoknocksound)}

Environment invariants $\varphi^{\mathcal{E}}_{inv}$ =
{**G** (onebellpressed → onebellrings)
G (twodoorhit → twoknocksound)}

The erroneous assumptions that need correcting at runtime with respect to the actual environment behaviour are:

$\varphi^{\mathcal{E}}_{inv}$ =
{**G** (onedoorhit → onebellrings)
G (twobellpressed → twoknocksound)}

3.2 Updating specifications by theory revision

[?]
We believe that viewing the update of assumptions as a theory revision task is a particularly suitable approach to our problem, and specifically a theory revision task conducted through nonmonotonic ILP. We highlight below the key reasons for this being the case. It is also of note that earlier works have approached similar problems as theory revisions tasks. These include the modification of the rules that constitute normative frameworks to bring them in line with violating event traces [11, 53], and the modification of guarantees with respect to counterexample or execution traces [30, 12]. As described in 4, we base the implementation of our assumption revision engine on a similar engine built for revising guarantees in [30].

3.2.1 Similarity and coverage

Facing a violation of one or more assumptions, one response could be to learn from scratch new assumptions consistent with the observed environment behaviour: [8] proposes a comparable method as described in Chapter 7. However, we believe it is preferable instead to correct the existing assumptions, which is the promise offered by theory revision systems [11].

A freshly discovered set of assumptions may not be similar to the original set, where similarity refers to both the syntax of the formulae, and the range of behaviours that they permit. Given that we probably do not have issue with assumptions that are not violated by the observation trace, our desire for behaviour coverage described in 2.1.5 dictates that we would prefer these to remain constant. The various notions of the minimality of revisions which can be incorporated into theory revision systems helps to ensure that the syntax of assumptions is not changed more than is necessary, while the RL component of the framework can provide support for ensuring behavioural coverage, as described later. Moreover, theory revision systems allow us to choose whether to add some or all the assumptions to the revisable theory, or only the violated assumptions, with the remainder left in the fixed background theory.

3.2.2 Realisability

We presume that the original set of assumptions was both minimal and as weak as possible while achieving realisability, meaning that all of the assumptions were necessary for realisability. If one of the assumptions is erroneous, it is feasible that the remainder of the specification remains close enough to realisability that only a small number of modifications are required to reach a new realisable specification. This may allow us to maintain the assumptions at a similar weakness and number as they were in the original specification. Conversely, learning a new weak and minimal specification from scratch, possibly involving refinement steps to reach realisability, may be more time-consuming.

3.2.3 Consistency and completeness

Using a theory revision system means that the modified assumptions are guaranteed by construction to be consistent with the violating traces provided as examples. This means that the assumptions are certain to accurately reflect this observed behaviour (if not all of the environment's possible behaviours). Such a system also ensures that the modified assumptions remain semantically and syntactically coherent, which is essential for the specification to be able to give rise to a synthesisable controller.

Nonmonotonic ILP systems are also complete. This ensures that a revision is conducted whenever one is possible, and therefore additionally that the optimal revision (regardless of how we define optimality) is found [11].

3.2.4 Complex changes

To find a realisable and minimal revised specification, unviolated assumptions may need to be updated alongside the violated ones. A nonmonotonic ILP system, which can execute complex semantic changes over the theory, might be better suited to this task than traditional operator-based theory revision systems, which use logical entailment to guide monotonic changes to the consequences of the theory [11].

3.2.5 Flexibility

Theory revision systems, and particularly those employing nonmonotonic ILP, can be adapted and augmented in several ways that are desirable for our problem. For example:

- We can require the revised theory to cover all or only a certain proportion of the provided examples by allowing specifiable levels of noise. This is useful for our proposed method for ensuring coverage described in 3.3.3.
- Most theory revision systems seek some notion of the minimality of the changes, which is useful for ensuring the syntactic similarity of the updated assumptions with respect to the original specification. Additionally, some engines, including nonmonotonic ILP-based systems, can be adapted to find solutions that are not categorically the most minimal, but involve as few changes as possible while respecting some other optimality criteria. For instance, we can restrict the search space for hypotheses using meta-constraints, which is essential for ensuring the semantic integrity of assumptions in our implementation of the revision (see Chapter 4). Equally we can guide the search space using the aggregates detailed in 2.2.2; setting the maxima and minima of these aggregates are some of the actions of our RL agent.
- As already mentioned, theory revision systems allow the user to define which rules should be included in the revisable theory and which should be kept fixed.

3.2.6 Further thoughts

For the reasons outlined above, a theory revision system forms the core phase of our framework for updating specifications. The background knowledge B includes assumptions we do not wish to update and other fixed information; the mode declarations M define the syntax of the revised assumptions; the revisable theory R contains the assumptions to be updated; and the examples E consist of the violating execution trace, as well as possibly other traces to aid with coverage and realisability, as we explain later. A revision task for guarantees would be similar, but with the revisable theory containing the incorrect goals, as well as other guarantees that may need updating.

The main consideration of an elementary theory revision system is ensuring that the required proportion of examples is covered by a revised theory that is reached by minimal revisions; there may be a number of possible solutions that satisfy these criteria. Such concerns, however, form only some of the considerations relevant for our revised set of assumptions, which need to be optimal also in terms of their weakness and whether they produce a revisable specification.

In addition to this phase, therefore, we propose the addition of other methods not just to eliminate some of the possible solutions of the theory revision task, but actually to guide the revision engine to areas of the hypothesis space that may not be reached by minimal revisions but are nevertheless preferable given our multiple quality criteria. We explain the implementation-specific meta-constraints we use for ensuring the semantic correctness of revised assumptions in Chapter 4, and introduce the general considerations relating to RL element of our framework below.

3.3 RL

3.3.1 Introduction

RL approaches allow us to specify what we want the system to achieve without having to explicitly instruct the system how to achieve it. This means that we can build systems that are able to accomplish tasks that we may not know how to complete ourselves, and also which can accomplish these tasks in different ways depending on the context.

Both of these advantages are beneficial in an approach to our problem. We hope that somewhere in the space of possible revised assumptions exist those that contribute to a realisable specification, while also being as weak and similar to the original assumptions as possible, both syntactically and in terms of behaviour coverage. Parameters can be used to guide the ILP system's search for these solutions, but since our requirements are complex, and may require trade-offs between countervailing preferences, it is not necessarily clear what these parameters should be. One explicit way to achieve the trade-off between weakness and realisability could be to remove the violated assumption and then conduct a counterstrategy-guided refinement process to find a weak and minimal set of assumptions that ensure realisability (see 2.1.3), but doing so might be costly at runtime and not reflect the specification designer's general intent. While this method might still be required if our approach is not able to find a realisable specification, we hope to be able to circumvent it by having the RL system learn which parameter settings correctly restrict the hypothesis space.

There are likely to be more than one set of assumptions that satisfy our criteria. The second advantage of RL mentioned above, the fact that it can learn a domain-dependent policy, means that we can train a system that is able to find the appropriate sets of assumptions that are most likely to reflect a given environment's behaviour. By way of illustration, we note that invariants usually describe a relation between two or more variables, in the form $\mathbf{G}(\mathrm{a} \wedge \mathrm{b} \rightarrow \mathrm{c})$. For a specification containing several single-state invariants, in one environment the assumptions may be violated because the relations assumed at design-time are incorrect, so we would want the RL system to learn, for example, to find a realisable specification with different consequents in the assumptions, as in $\mathbf{G}(a \wedge b \rightarrow d)$. In another environment, the assumed relationships may be correct, but the environment may evolve more slowly that expected, such that it makes better sense for the RL agent to narrow the hypothesis space down to assumptions that are transition invariants that otherwise contain the same variable relationships, as in $\mathbf{G}(a \wedge b \rightarrow \mathbf{X}c)$. At runtime, we want to be able to bring erroneous assumptions in line with environment behaviour in as few attempts as possible. Having a system that has learnt at training-time a context-specific policy for updating assumptions should help us to do this.

3.3.2 Guiding the search

We have designed our framework to try to restrict the search space directly to realisable revised specifications. We provide some justification here for not instead following approaches focused on removing violated assumptions or finding an appropriate weakening of the assumptions that could then be strengthened into a realisable specification.

Removing only the violated assumption

We have already mentioned that the approach of removing a violated assumption and then refining the specification as necessary may be costly at runtime. A further issue with this technique is that the assumption that is removed may not have been the final assumption added in the refinement process. Removing an assumption that was an intermediate refinement would mean that an unknown number of strengthening steps would be required to reach a new realisable specification. This may also mean that the other assumptions added in the original refinement process after the one that has now been removed, may been redundant, as explained in 2.1.4, thus unnecessarily constraining the environment.

Removing a whole refinement branch

A conceivable way around this problem could be to remove as well as the violated assumption, all assumptions added in subsequent refinement steps along that branch. In real-world applications, assumptions are unlikely to be labelled with the order in which they were added to the specification. We might then hope to be able to train a RL system to recognise which assumptions form a branch that must be pruned when the base assumption of that branch is violated. We might even hope that the system can prune the branch to a point from which only a smaller number of refinement steps are required to reach a new specification.

To assess the feasibility of such an approach, we analysed the refinement tree produced by the interpolation-based refinement procedure proposed in [13] (see 2.1.3).

We were not able to intuit any features of assumptions to indicate which assumptions were added in subsequent refinement steps, and the authors of [13] confirmed to us that there do not appear to be features of an unrealisable set of assumptions to indicate the length of any refinement branches stemming from this point.

We could also remove the redundant assumptions by a method such as the one proposed in [2]. By any approach, though, removing a significant quantity of now-redundant assumptions might make the new specification too different from the designer's original intent.

Jumping directly to a new realisable specification

We instead propose a RL system that is able to restrict the search space to allow for jumping directly across from the original specification to another realisable one, rather than weakening upwards along the refinement branch. The system may be able to do this on a first attempt, but will otherwise try to do this in as few attempts as possible. This allows us to avoid strengthening steps where possible and thereby reduce the number of realisability checks, giving a speed-up in repairing an erroneous specification at runtime.

The agent may even be able to learn to enact the equivalent of a refinement step if there are no other realisable specifications with the current number of assumptions, by adding one or more new assumptions consistent with the traces. However, we need to test this hypothesis with more complex specifications in future work.

3.3.3 Core features and actions

We discuss here the state features and revision operations that are relevant for a system seeking to bring one set of assumptions directly to a similar realisable specification. We have identified these by analysing once again the refinement trees produced for [13], to discern similarities between neighbouring leaves, in the hope that if one leaf is violated the system will be able to jump to a nearby leaf that would not have been affected by the violation. While not all refinement techniques form a tree in this way, and the assumptions in our problem may have been specified by a designer rather than added through refinement steps, this analysis nonetheless provided an insight into the types of revision that might be required to adjust the original set of assumptions into another realisable specification that is not too different from the designer's original intent.

While all RL agents learn what action to take in light of the features of the current state, the nature of our objective is such that the relationship between features and actions is semantically even tighter than usual. This is because we have observed that nearby leaves are similar in the patterns and length of their assumptions, meaning that we can infer from features of the violated set assumptions the features that might be present in a neighbouring leaf, and take actions accordingly to revise the assumptions from one to the other. Equally, we propose that a RL agent might be able to learn enough about the internal structure of the sets of assumptions of a given group of realisable specifications to be able to restrict the space of revisions such that it jumps directly from one set to another, even when the difference between them is more significant (though we intend to test this hypothesis further with larger specifications in future work). In this sense too, the RL system's ability to learn a domain-dependent policy is important. The RL agent not only learns about the structure of the sets of assumptions of a group of related realisable specifications, it also learns which of these are most likely to be violated by a specific environment. Of the various unviolated sets, we additionally design the agent to choose the weakest set that is most similar to the original set of assumptions.

A limitation of our tree analysis is that it is much easier to see similarities between leaves that share a common immediate ancestor, i.e. where the specifications only differ in the assumption added in the final refinement, as these final assumptions are usually correlated in the ways we describe below. This means that we are again focusing on violations of the last added assumption rather than those added in intermediate refinements, which we noted was a shortcoming of other approaches. Nevertheless we believe that the reward function of our agent, which we describe in 3.3.4 and which incentivises the agent to find realisable sets of assumptions, will be able to encourage the agent to learn about the structure of assumptions even in leaves that do not have a common immediate ancestor with the original set. The analysis below of immediately neighbouring leaves gives an insight into the reasoning behind the features and actions which we believe are also relevant for our agent to be able to discover a policy for jumping between more distant realisable sets.

Revision operations

In our framework, for the majority of the identified operations the RL agent does not directly decide which one to apply. Instead these operations are undertaken by the ILP revision system, and the RL agent's domain-dependent policy sets parameters for the number of each different type of operation that the ILP system is able to use.

Given an assumption of the form $\mathbf{G}(a \land b \land c \rightarrow z)$, it is clear that any operation that strengthens the assumption would give an assumption that is also violated, meaning we **cannot** do any of the following (each with an illustrative example of the result of applying the operation):

- Remove any of the variables from the antecedent: $\mathbf{G}(a \land b \rightarrow z)$.
- Add a disjunct to the antecedent: $\mathbf{G}(a \land b \land (c \lor d) \rightarrow z)$.
- Add a conjunct to the consequent: $\mathbf{G}(a \land b \land c \rightarrow (z \land y))$.

We are therefore left with the following possible options for revisions; the ones we allow in our proof of concept are reviewed in 4.4.2:

1. Replace one or more of the variables in the antecedent: $\mathbf{G}(a \land b \land d \rightarrow z)$.

2. Add a conjunct to the antecedent, which equates to weakening the assumption: $\mathbf{G}(\mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c} \wedge \mathrm{d} \rightarrow \mathrm{z})$.
3. Replace one or more of the variables in the consequent: $\mathbf{G}(\mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c} \rightarrow \mathrm{y})$.
4. Add a disjunct to the consequent, which again equates to weakening the assumption: $\mathbf{G}(\mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c} \rightarrow (\mathrm{z} \vee \mathrm{y}))$.
5. Turn the single-state invariant into a transition invariant, or vice versa: $\mathbf{G}(\mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c} \rightarrow \mathbf{X}\mathrm{z})$.

We reproduce here the erroneous assumption that needs correcting for Door 1 in the toy example introduced at the beginning of this chapter:

$\mathbf{G}$ (onedoorhit $\rightarrow$ onebellrings)

A trace might be observed that violates this erroneous assumption with the following truth values at timestep S_1:

{onedoorhit = TRUE
oneknocksound = TRUE
onebellpressed = FALSE
onebellrings = FALSE
twodoorhit = TRUE
twoknocksound = TRUE
twobellpressed = FALSE
twobellrings = FALSE
onedooranswered = TRUE
twodooranswered = TRUE}

The revision engine, seeking to apply a minimal number of revision operations, might update the erroneous assumption by applying operation 2, requiring only the addition of a body condition. This might give rise to the following updated assumption:

$\mathbf{G}$ (onedoorhit & $\neg$oneknocksound $\rightarrow$ onebellrings)

While this is a logical revision for the ILP system to make, we can see intuitively that this is an incorrect update given the environment's true behaviour. The benefit of our framework is that the RL agent should learn a domain-dependent policy for applying revisions that reflect the ground-truth behaviour. In this case, the correct revision is not the most minimal one as far as the ILP system is concerned; it is instead revision 3, which requires *onebellrings* to be deleted from the consequent, and *oneknocksound* to be added in its place, to give the correct assumption:

$\mathbf{G}$ (onedoorhit $\rightarrow$ oneknocksound)

The RL should therefore learn to set the ILP system's parameters such that in this situation a consequent variable must be deleted and another added; the policy should set the maximum number of additions to antecedents to be zero.

Relevant features

We discuss here the features that are relevant for the RL agent to decide which revision operations to allow through its parameter selection.

Any operation that weakens the assumptions could make the specification unrealisable, so we might only want to take actions 2 or 4 if the current assumptions are not considered too weak, making the **weakness of the original set of assumptions** an important feature.

The replacement operations 1 and 3 are only possible if there are variables not already appearing in the formula which can be used for the substitution. An indicator for whether a replacement is possible is therefore the **proportion of the total observable variables that appear** in the violated assumption. A similar effect can be achieved by counting the **number of variables that appear in the antecedent and consequent** of the assumption; the more variables already appearing, the less likely it is that there remain unused variables for a substitution.

The added benefit of the length counting method is that it can also indicate which of the weakening actions 2 and 4 might be preferred. During our analysis of the refinement trees we observed that the final assumptions of neighbouring leaves are often in some way counterpoints to each other, such that if one has a longer antecedent and shorter consequent, the other may exhibit the reverse. As such, the longer the antecedent of a violated assumption, the more likely it is that a disjunct should be added to the consequent; if an assumption has a shorter antecedent but already several disjuncts in the consequent, we are more likely to add a conjunct to the antecedent.

If both antecedent and consequent are long, this might indicate that a weakening action is not possible, just as this information might indicate that a replacement action is not possible. This might point to action 5, the addition or removal of a next **X** operator. This requires the RL agent to know whether such an operator is already in place, so we propose that the presence or **number of X operators** also be a state feature for the agent; fewer **X** operators make it more likely that one should be added. If none of the above actions are possible for reaching a realisable specification, this may indicate that a new assumption should be added.

As already mentioned, when updating a violated assumption that was added in an intermediate refinement step, we may also need to revise the assumptions added later in the refinement process. It is therefore useful to have as a feature the **total number of assumptions**, as an indicator for how many overall changes need to be made.

To summarise, an implementation of our framework is likely to include many if not all of the following features for the RL agent; the ones we use in our proof of concept are detailed in 4.4.3:

- the weakness of the original set of assumptions;
- the proportion of the total observable variables that appear in the (violated) assumptions;
- the number of variables that appear in the antecedent and consequent of the (violated) assumptions;
- the number of transition invariants among the (violated) assumptions; and,
- the total number of (violated) assumptions.

Number of examples to cover

There are several other possible features and actions that an implementation of our proposed RL agent might be desired to include, in addition to setting parameters for the number of the revision operations discussed above which the ILP system can use. Our proof of concept detailed in Chapter 4 only implements the parameter selection actions.

One way for the RL system to favour revisions with greater behavioural coverage (as defined in 2.1.5) is through the reward function, which we discuss further in 3.3.4. The pursuit of coverage can also be implemented as an action as follows. We have assumed that all of our invariants are of the form $\mathbf{G}$ ($\bigwedge a_i \rightarrow$ b) or $\mathbf{G}$ ($\bigwedge a_i \rightarrow \mathbf{X}$b). Of the various possible combinations of truth values of antecedent and consequent for a formula of this form, the formulae is only violated if the antecedent is satisfied and the consequent is not - if the antecedent is not satisfied, it does not matter whether or not the consequent is true. This means that for any invariant subject to a violating execution trace, we can generate at least three witness traces that testify to the assumption being satisfied. We can generate further such traces for each of the correct, unviolated assumptions that may need to be updated.

In addition to the violation trace that we provide as an example to the theory revision system, we could provide witness traces to ensure that the revised assumptions still allow the unviolating behaviour permitted by the original assumptions. Forcing the coverage of all the witness traces might mean no hypothesis exists; since coverage is a preference rather than a requirement, it would be better to be able to vary the number of traces to satisfy through an action by the RL agent, so as to force as much coverage as possible without preventing realisability.

One way of implementing such an action could be the choice of how many witness examples to provide (with the revision system insisting on the coverage of all provided examples). However, this would mean that the selection of which examples to give is random, and this arbitrariness could have negative effects: it might be possible to find a solution only covering two of the three witness traces, but setting the number of traces to provide as two might mean that the non-coverable trace still gets given. Nevertheless, theory revision systems also permit noise in the examples by allowing the coverage of only a certain number of the provided examples to be required. Setting the number of provided traces to be covered would be a more sensible action for the RL agent, as it would delegate the identification of which examples can be covered to the revision engine.

This might mean, though, that the revision engine ends up not covering the violation trace, which would undermine the point of our system. Nonmonontic ILP systems that employ ASP, such as ASPAL, can make use of the weights and optimisation statements described in 2.2.2 to overcome this issue. If all examples are considered equally important, the weight penalty for not covering each of them can be the same. However, we could also set the penalty for not covering the violation trace as infinite, meaning it must be covered, while the penalty for not covering the witness traces is some smaller positive integer.

Selection of variables to be used in revisions

A violating trace contains truth values for all of the system's observable variables at each timestep. Some of these variables relate to specific components of the environment and system, and only make sense semantically when grouped together. In our toy example, we would only want to update the assumptions pertaining to Door 1 with variables describing Door 1; since the functioning of each door is not related, it does not make sense to have assumptions describing Door 1 using observations of Door 2. We give here again Door 1's erroneous assumption, and the variable truth values at violating timestep S_1:

$\mathbf{G}$ (onedoorhit $\rightarrow$ onebellrings)

S_1 =

{onedoorhit = TRUE
oneknocksound = TRUE
onebellpressed = FALSE
onebellrings = FALSE
twodoorhit = TRUE
twoknocksound = TRUE
twobellpressed = FALSE
twobellrings = FALSE
onedooranswered = TRUE

twodooranswered = TRUE}

Even if the RL agent has correctly learnt to set the parameters to ensure that the consequent is modified, the revision system may attempt to update the assumption with any of the variable valuations it observes in the trace, even if this is not correct in the real world. For example, it could update the assumption to $\mathbf{G}$ (onedoorhit $\rightarrow$ twoknocksound) or $\mathbf{G}$ (onedoorhit $\rightarrow$ twodoorhit).

To make sure that the correct update is achieved, we could extend our agent's actions to allow the provision of only a certain subset of $\mathcal{V}$ to be used for the update. The agent may decide that for certain entities, only the variables describing that entity may be used to revise the entity's violated assumptions. If the agent only provides the subset {onedoorhit, oneknocksound, onebellpressed, onebellrings}, the revision engine is far more likely to reach the correct assumption $\mathbf{G}$ (onedoorhit $\rightarrow$ oneknocksound). The problem of incorrect updates can also be reduced by the provision of witness traces, described above, and subsequent violation traces, as discussed in 3.3.4.

3.3.4 Reward function

Even after the meta-constraints have restricted the solutions that the revision system can return, there are likely to be numerous possible sets of updated assumptions, of which some are more desirable than others. The RL agent's reward function is what assesses the quality of computed revision, and this feedback is used by the agent to learn about the effects of its actions, so that at runtime it will employ the actions most likely to achieve an optimal revision. The reward function must therefore not just take into account the various quality criteria of a revised specification, but also be able to make a sensible trade-off between our preferences when necessary. Several of our requirements and preferences for the updated assumptions are subject to a predominance hierarchy which must additionally be reflected in the reward function. We discuss the considerations affecting the preferences in the order they appear in this hierarchy, beginning with the most important. We give the finer details of the reward function that we implemented for our proof of concept in 4.4.4.

1. Realisability

The overall aim of our assumption updating system is to have the controller's guarantees fulfilled in as many circumstances as possible. The two issues that prevent the guarantees being fulfilled are if the assumptions are not satisfied or if the specification is unrealisable. The revision system brings the assumptions in line with the environment's behaviour so that they are more likely to remain satisfied by the environment, but the goals will still go unsatisfied unless the revised specification is realisable, making this the key quality criterion for the various possible revisions. If the specification is not realisable, none of our other preferences are relevant.

Updating assumptions in light of violating behaviour may weaken them, rendering the specification unrealisable. We therefore want to favour revisions that leave the specification realisable, or, if this is not possible, that will not require too many expensive strengthening steps to return to realisability. An ideal implementation of the reward function might therefore take into account not just whether the new specification is realisable, but also, in the cases where it is not realisable, how many strengthening steps are required, with more steps earning a smaller reward.

Conversations with the authors of [13] indicated that there are not currently known to be features of unrealisable specifications that identify how many strengthening steps are required before realisability is reached. One way to discover the number of steps is therefore actually to carry out the strengthening process and count the steps. We believe that doing so is undesirable for two reasons.

- After each step, the refined specification must be checked for realisability, which can be time consuming; in some application domains, strengthening to a realisable specification can take several hours [13]. Conducting this procedure on every unrealisable specification produced by the revision engine for which we want this information as part of the reward might make the RL agent's training process intractable, given the large number of training episodes.

- Even if we did wish to allow the training process to run for the amount of time required to compute the number of realisability steps, it is unlikely that this element of the reward function would materially affect the learnt policy. Since the number of strengthening steps that will be required is not known to be determined by any features of the set of assumptions, it is unclear that the RL agent would be able to learn to take actions that give rise to sets of assumptions that, if unrealisable, require as few strengthening steps as possible.

A possible way around the time-consuming process of counting the actual number of strengthening steps might be the use of a heuristic for 'closeness to realisability'. We explored a potential such heuristic for use as part of the reward function. For a given unrealisable specification, [17] proposes the computation of a environmental counterstrategy that prevents the specification from being fulfilled. A countertrace, that is, a single input trace demonstrating unrealisability, may be derived from the counterstrategy, and [17] then displays the counterstrategy or countertrace as a graph. We discussed with the authors of [13] the possibility of using the entropy of this graph as a heuristic for closeness to realisability. A greater entropy would indicate more possible behaviours by which the environment can prevent the specification from being realised, and therefore more strengthening steps required to restrict these behaviours before realisability is achieved. The entropy could be calculated from the maximum eigenvalue of the adjacency matrix of the graph.

Nevertheless, research has not yet been carried out to ensure that the counterstrategy graph's entropy is indeed an accurate and admissible heuristic for the number of strengthening steps required for realisability, and to the extent of our knowledge there are not pre-existing algorithms for obtaining the maximum eigenvalue of the counterstrategy graph's adjacency matrix. In the absence of evidence for the benefit of including this procedure as part of the reward function, we decided not to implement and test this component due to the time constraints of our project, but recognise that doing so may be a valuable piece of future work.

Due to these challenges in identifying a specification's 'distance from realisability', we have opted to include in our current version of the framework only a component that gives a penalty if a realisable specification cannot be found after a certain number of attempts; we do not attempt to make unrealisable specifications close to realisability. We can increase the likelihood that a realisable specification is indeed found within the specified number of attempts as follows. Certain realisability checkers, such as RATSY [16], produce a counterstrategy demonstrating why a given specification is unrealisable. Whereas counterstrategy-guided refinement techniques use such information to direct the addition of an assumption in a strengthening step (see 2.1.3), we propose instead to provide this counterstrategy as a negative example for the theory revision system for its next attempt. On this subsequent attempt, the possible sets of revised assumptions are restricted further to those not satisfied by the countertrace, meaning that the problematic behaviour is excluded.

2. Correctness of assumptions

Our reward function thus far has incentivised the revision system to find a realisable set of assumptions that respect the original violating trace. Given that the system has only seen one environment behaviour trace, the changes enacted may not have brought the assumptions in line with all of the environment's possible behaviours, meaning that they remains erroneous. We give here again Door 1's erroneous assumption, and the variable truth values at violating timestep S_1:

$\mathbf{G}$ (onedoorhit $\rightarrow$ onebellrings)

$S_1 =$

{onedoorhit = TRUE
oneknocksound = TRUE
onebellpressed = FALSE
onebellrings = FALSE
twodoorhit = TRUE
twoknocksound = TRUE
twobellpressed = FALSE
twobellrings = FALSE
onedooranswered = TRUE
twodooranswered = TRUE}

If we have not implemented the variable subset selection action suggested in 3.3.3, the assumption might still be updated to, for example, $\mathbf{G}$ (onedoorhit $\rightarrow$ twoknocksound) or **G** (onedoorhit $\rightarrow$ twodoorhit). Even with the implementation of the variable selection action, assumptions can be updated in other ways that give realisable specifications but do not reflect the environment's true behaviour, and might therefore be violated again.

To encourage the RL agent to learn the true behaviour of the environment, it is necessary to give via the reward function a penalty if the realisable specifications produced by the revision engine continue to be violated in the simulated training executions. This way, at runtime, our framework will be more likely to revise the assumptions to reflect the environment's behaviour in one attempt, rather than the framework needing to be launched persistently in response to continued violations. As with the penalty discussed in the previous subsection for repeated unrealisable outputs, we can increase the likelihood that the framework will produce realisable assumptions that are not violated within the specified number of attempts by not just testing for a violation with subsequent traces, but adding the new violating trace at each attempt as an additional example for the revision system. Doing so means that the revised assumption will satisfy more of the behaviours exhibited by the environment, thus reflecting it more accurately.

3. Weakness and coverage

Realisability and correctness with respect to the environment are the required quality criteria of all revised specifications. Of the probably numerous possible solutions, the preference that is likely most importance is the weakness of the updated assumptions. Weaker assumptions will be satisfied by a greater range of environment behaviour, so the guarantees will be fulfilled in more circumstances. The reward function should therefore contain a component that gives a greater reward for weaker solutions.

A reward function may also be desired to include a component that favours solutions with greater coverage of behaviours permitted by the new and original assumptions. As introduced in 2.1.5, coverage can be measured by the number of witness traces satisfying the old assumptions which also satisfy the updated set.

Weakness and coverage may be contradictory qualities, and there is not an obvious trade-off to be found between them. While we presume that most users would favour weakness (permissability of more behaviours) over coverage (similarity to existing behaviours), there may be cases in which a user wants as much as possible of the original design of the assumptions to be retained. To make the trade-off between these criteria, which are preferences rather than requirements, less arbitrary, the respective values of rewards for these criteria can be defined differently for alternative implementations of our framework.

4. Number of attempts

Since the reward function incentivises the agent to find the weakest possible realisable specification, it may make several attempts in which the specification is too weak and therefore not realisable, before finding the correct balance. At runtime, we want the framework to find this balance in either one attempt or as few tries as possible. Moreover, we have made it easier for the agent to find the correct assumptions during training by adding additional violating traces as examples. At runtime it may also be necessary to continue gathering execution traces so as to correctly model the environment, but it is preferable for the correct assumptions to be found immediately. To speed up the execution of our framework at runtime and reduce the number of times it needs to be launched, we must use the reward function to incentivise the RL agent to find the optimal revision in as few attempts as possible.

3.4 Review

In this chapter we have surveyed the general considerations and variety of components that an instantiation of our framework might be hoped to include. The core framework consists of a revision engine, for which the hypothesis space is constricted by meta-constrains. An RL agent additionally sets parameters for the revision engine to guide the search towards to the solutions we consider acceptable and optimal. It learns the policy for doing this through its reward function, which quantifies the quality of revised assumptions and how quickly they were produced. The agent's actions and reward calculation can be augmented in a variety of ways to help ensure that the best revised specifications are found.

Chapter 4

Implementation

4.1 Introduction

We instantiate our proposed framework to update assumptions written in the Spectra specification language [54], using the tools highlighted in this chapter for the theory revision system, RL implementation and realisability checks. Our proof of concept includes the majority of the elements that we have indicated an instantiation of our framework might be hoped to have; though we have not implemented some of the supplementary actions proposed for the RL agent, nor the check for subsequent violations or coverage. We intend to incorporate and test these elements in future work.

4.2 Theory revision

We have modified the theory revision system built for adapting requirements models in [30] to update assumptions, and also with permission borrowed their parsers and translators for reading between different specification languages and data representations. Their revision system employs RASPAL [11, 38], the revision engine implemented in terms of ASPAL [11, 38], using clingo as the ASP solver [45]. There are several reasons that we have made this choice for our implementation:

- Adapting requirements with respect to execution traces is very similar to modifying assumptions, so building upon the existing code avoids unnecessary replication of labour. Having the existing revision system as the base layer of our framework also means that we can test the use of RL for revising guarantees in future work.

- RASPAL brings the benefits of nonmonotonic ILP systems highlighted in **??**, such as completeness and consistency. A drawback of RASPAL is that with larger programs, the computation can take a large amount of time (over ten minutes, if not significantly more). As described in 4.4.4, we consequently have had to introduce a timeout for occasions when the revision computation takes more than a certain number of minutes, so that training does not become intractable. This means that some potentially viable solutions may be lost. A new, faster revision system is due to be released coinciding with the end of this project, which may help to overcome the training problem and speed up the execution of our framework at runtime. We intend to test our implementation with this new engine as part of future work.

- RASPAL and clingo provide support for two other key elements of our implementation, namely meta-constraints (2.2.4) and min/max aggregates (2.2.2). clingo is an efficient solver whose scalability is comparable to SAT solvers; the computation of answer sets is very fast for smaller programs, but can become intractable as the search space grows [30].

4.2.1 Formalisation

Rules

Similarly to earlier works involving learning with LTL specifications [30, 12, 55], the syntactic formulation of the assumptions and guarantees is represented through normal logic programs and an Event Calculus formalism [56]. Each assumption is represented by three rules, with head literals named *current_holds*, *target_holds* and *theta_holds*. The bodies of these rules respectively indicate which variables appear in the antecedent of the assumption; which variables appear in the consequent of the assumption; and whether the assumption is a single-state or transition invariant. A variable *a* that is true at timepoint *T1* in trace *S* is represented by a fluent (a condition that can change over time) thus: *holds_at(a,T1,S)*. If it is false at that timepoint, it is represented instead by the fluent *not_holds_at(a,T1,S)*. As such an assumption labelled *firstassumption* with formula $\mathbf{G}(\text{doorhit} \rightarrow \text{bellrings})$ is represented by the following three rules:

```
current_holds(firstassumption,T1,S):-
  holds_at(doorhit,T1,S),
  timepoint(T1),
  trace(S).

target_holds(firstassumption,T2,S):-
  holds_at(bellrings,T2,S),
  timepoint(T2),
  trace(S).

theta_holds(firstassumption,T1,T2,S):-
  timepoint(T1),
  timepoint(T2),
  trace(S),
  eq(T1,T2).
```

If the fluent *holds_at(doorhit,T1,S)* is true at a certain timepoint in a trace, then the antecedent of *sillyassumption* is satisfied at that timepoint, represented by the head literal *current_holds(sillyassumption,T1,S)* being true. *target_holds(sillyassumption,T2,S)* captures similar information for the consequent. The literal *eq(T1,T2)* in the rule with head *theta_holds(sillyassumption,T1,T2,S)* indicates that *sillyassumption* is a single-state invariant. If this rule instead contained the literal *next(T2,T1,S)*, it would mean that the assumption is a transition invariant, i.e. $\mathbf{G}(\text{doorhit} \rightarrow \mathbf{X}\text{bellrings})$.

This representation means that the *holds_at* or *not_holds_at* literals that appear in the body of a rule with head *target_holds* represent a conjunction of variables in the consequent of the assumption. Using RASPAL's *extension* and *delete* mode declarations (see below) we can either add or remove literals in these rules, that is, we can only add conjuncts to the consequent and not disjuncts. As explained in 3.3.3, however, we never wish to add conjuncts to the consequent. We therefore prevent our revision engine from doing this by techniques explained later; we also assume a pre-processing step that transforms all assumptions into a form with only one variable in the consequent. The weakness of this approach is that, given the formalisation, our present implementation can only handle transition invariants with one **next** variable. It also means that we cannot use the number of variables in the consequent as a feature, nor the addition of disjuncts to consequents as an action, as suggested in 3.3.3.

Background theory

The background theory includes the following:

- A set of domain-independent axioms for determining whether assumptions are satisfied by a trace, and facts indicating which timepoints are equal or sequential;
- Facts indicating which variables are input or output variables, and which can be used in a revision (see 3.3.3);
- The guarantees and initial assumption represented by Event Calculus rules as above.

In our implementation, the revisable theory consists of all the assumptions rather than only the violated ones, as we assume that the violation detection component only indicates that a violation occurred, rather than which invariants in particular were violated. Additionally, even unviolated assumptions may need to be updated to reach a realisable leaf. If this were not the case, rules representing the unviolated assumptions could instead be added to the background theory.

Mode declarations

The mode declarations are as follows; RASPAL extends them with *extension* and *delete* mode head declarations (see 2.2.3):

```
modeh(current_holds(assumption, +timepoint, +trace)).
modeh(target_holds(assumption, +timepoint, +trace)).
modeh(theta_holds(assumption, +timepoint, +timepoint, +trace)).
modeb(holds_at(usable_atom, +timepoint, +trace)).
modeb(not_holds_at(usable_atom, +timepoint, +trace)).
modeb(eq(+timepoint, +timepoint)).
modeb(next(+timepoint, +timepoint, +trace)).
```

Examples

Traces provided as examples are translated into fluents indicating which variables are true at each timepoint. For example, if at timepoint 1 in a trace called *firsttrace* we have

S_1 =

{doorhit = TRUE
knocksound = TRUE
bellpressed = FALSE
bellrings = FALSE
dooranswered = TRUE}

this would be represented as:

```
holds_at(doorhit, 1, firsttrace).
holds_at(knocksound, 1, firsttrace).
not_holds_at(bellpressed, 1, firsttrace).
not_holds_at(bellrings, 1, firsttrace).
holds_at(dooranswered, 1, firsttrace).
```

4.3 Meta-constraints

Meta-constraints are added to this base layer of the revision engine. We detail here the constraints required to ensure the semantic correctness of revised assumptions, all of which have been decomposed into the three separate Event Calculus rules. The same denials with different, relevant labels are added to ensure that any new assumptions added to the revisable theory, like the revised existing assumptions, obey these constraints. In the following, *w_in_rule(R, H, B)* is defined by the rule

```
w_in_rule(R, H, B) :- in_rule(R, H, B, I), b_index(I).
```

where *b_index(I)* indicates the position of the literal in the clause.

The first three constraints are borrowed from the implementation of adaptations to requirements models in [30].

```
:- 0 {w_in_rule(ID, H, B)} 0, is_rule(ID, H).
```

This forces all rules to contain at least one literal. Without it, the revision engine simply removes the consequent of violated assumptions without trying to replace it, as encouraged by the in-built minimality of the revision approach, implemented as a clingo minimisation statement.

```
:- assumption(A), atom(F),
w_in_rule(ID, current(A), holds_at(F)),
w_in_rule(ID, current(A), not_holds_at(F)).

:- assumption(A), atom(F),
w_in_rule(ID, target(A), holds_at(F)),
w_in_rule(ID, target(A), not_holds_at(F)).
```

These two constraints indicate that a variable cannot be both true and false in the antecedent or consequent, e.g. we cannot have $\mathbf{G}(a \wedge \neg a \rightarrow b)$.

We have added the following meta-constraints ourselves for ensuring the correct semantics of revised assumptions.

```
:- assumption(A), atom(F),
w_in_rule(ID, current(A), holds_at(F)),
w_in_rule(ID2, target(A), holds_at(F)), w_in_rule(ID3, theta(A), eq).

:- assumption(A), atom(F),
w_in_rule(ID, current(A), not_holds_at(F)),
w_in_rule(ID2, target(A), not_holds_at(F)), w_in_rule(ID3, theta(A), eq).
```

These two constraints avoid vacuous assumptions by ensuring that no variable can be true (or false) in both the antecedent and consequent of a single-state invariant, e.g. we cannot have $\mathbf{G}(a \rightarrow a)$.

```
:- assumption(A),
w_in_rule(ID, current(A), next).

:- assumption(A),
w_in_rule(ID, current(A), eq).

:- assumption(A),
w_in_rule(ID, target(A), next).

:- assumption(A),
w_in_rule(ID, target(A), eq).
```

The above four assumptions indicate that antecedents and consequents cannot contain the literals that indicate whether the assumption is a single-state or transition invariant.

```
:- assumption(A),
w_in_rule(ID, theta(A), holds_at(_)).
```

```
:- assumption(A),
w_in_rule(ID, theta(A), not_holds_at(_)).
```

Similarly, these two constraints ensure that the *theta* rules can never contain the literals meant to appear in the antecedents and consequents.

```
:- assumption(A),
w_in_rule(_, theta(A), next),
w_in_rule(_, theta(A), eq).
```

This ensures that an invariant can only ever either be either single-state or transition.

```
:- assumption(A),
w_in_rule(_, target(A), C1),
w_in_rule(_, target(A), C2), C1!=C2.
```

We use this constraint to ensure that only one variable can ever appear in the consequent of revised assumptions, as per 3.3.3.

```
:- assumption(A), output_literal(Body),
w_in_rule(ID, current(A), Body),
w_in_rule(ID, theta(A), eq).
```

The realisability checkers RATSY and Spectra, the latter being the tool we use for our realisable checks, assume that the environment makes its play before the controller in the context of the GR(1) games introduced in 2.1.3. They therefore do not accept output literals appearing in the antecedents of assumptions unless they are transition invariants; the constraint above ensures that this is true of our revised assumptions.

```
:- assumption(A), output_literal(Body),
w_in_rule(ID, target(A), Body).
```

This constraint ensures that output variables never appear in the consequents of assumptions, given both the reasoning for the previous constraint and the definition of environment invariants given in 2.1.2: $\mathbf{G}B(\mathcal{V} \cup \mathbf{X}\mathcal{X})$.

For each assumption with name *thisassumption*, we also automatically generate the following six constraints:

```
:- is_rule(ID, current(thisassumption)), is_rule(ID2, current(thisassumption)), ID != ID2.
:- is_rule(ID, target(thisassumption)), is_rule(ID2, target(thisassumption)), ID != ID2.
:- is_rule(ID, theta(thisassumption)), is_rule(ID2, theta(thisassumption)), ID != ID2.

:- 0 {is_rule(ID, current(thisassumption))} 0.
:- 0 {is_rule(ID, theta(thisassumption))} 0.
:- 0 {is_rule(ID, target(thisassumption))} 0.
```

The first three ensure that no rules are 'extended' into the revisable theory more than once, and that new assumptions always use fresh names. The latter three constraints force the inclusion in the revised theory of the three rules required by the semantics of each assumption. A possible downside of doing this is that we can never remove an erroneous assumption entirely and must always try to correct it. However, since the objective of our framework is to find another realisable leaf (see 3.3.2), rather than to move up a refinement branch by removing a violated assumption, we consider this not to be a concern.

4.4 RL agent

4.4.1 Overview

Our agent is used to direct the search for optimal revisions by restricting the space of possible hypotheses further than the meta-constraints already have done. As detailed below, the search space is restricted through setting several parameters for the RASPAL layer, meaning that each of the agent's actions is one possible combination of values for the parameters. Consequently each action is taken as a single attempt at finding an acceptable revision, the quality of which is evaluated by the reward function, before another attempt is made if necessary - if the learnt policy is optimal, only one action (i.e. one attempt) will need to be applied during an execution of the RL system. Since the agent is seeking the best action to apply for a given violated set of assumptions, information regarding which contributes to the 'state' of the agent's 'environment', Q-learning is an appropriate RL algorithm to employ. Moreover, given the very large number of possible actions (combinations of parameters) and states (the weakness component of which is continuous), we have opted to use a deep q-network to approximate the Q-value function.

We have implemented our system using TF-Agents [57], a deep RL library based on TensorFlow, a widely used machine learning library [58]. The TF-Agents library provides a ready-to-use deep q-network, offers tutorials to help with the set-up of the basic framework, and is considered well supported [59]. However, the library also has some limitations that we discuss in 4.4.2, and which mean that, when scaling up the framework in future work, we are likely to use a different library or build the system from scratch.

4.4.2 Actions of agent

We have implemented the core actions discussed in 3.3.3, and not the supplementary actions that we propose could be used for augmenting the framework. There are three sets of actions, each implemented differently.

Number of deletions

As described in 2.2.3, RASPAL computes the operations that must be applied to the revisable theory to reach the revised set of assumptions. It does so by encoding each possible operation rule as a fact $rule(id(h_i \leftarrow \overline{b_i}), \overline{C})$ that can be abduced. clingo allows us to place each abducible a_i in an aggregate *min*$\{a_1, ..., a_n\}$*max*. We create a separate aggregate for deletions applied to each of current, theta and target rules, and have six parameters that the RL agent can set, each of which indicate the minimum and maximum permitted numbers of each type of deletion. Deletions of body literals of current (resp. target) rules indicate that variables in the antecedents (resp. consequents) of assumptions must be replaced; deletions of body literals of theta rules indicate that the assumption should be turned from a single-state to transition invariant, or vice versa.

Number of body literal additions

When generating its top theory, RASPAL iteratively adds to each rule as many body literals as are permitted by the mode declarations. We have extended the existing code from [30] to allow separate limits on the number of body literals that can be added to existing current, theta and target rules, and also the number of conditions that can appear in the three rules representing any new assumptions that are included in the revised theory. We set hard limits of one body literal for new and existing theta and target rules. This is because, as already mentioned, we only want one variable appearing in consequents, and each assumption can only be either a single-state or a transition invariant. The number of variables appearing in antecedents are not subject to such limits, so the maximum numbers of conditions appearing in new current rules and in extensions of existing current rules are two parameters that can be set by the RL agent. This gives the agent control over how many variables to add or substitute in antecedents.

Number of new assumptions

The ninth parameter settable by the agent in our implementation is the number of new assumptions that must be added to the revised specification. Names and the necessary constraints are generated for the chosen number of assumptions, which are then abduced by RASPAL according to the mode declarations. The length of these new rules is guided by the parameter in the previous subsection.

Application of actions

To summarise the above, we have an array of nine parameters that can be set by the agent:

1. Minimum number of deletions of variables appearing in antecedents;
2. Maximum number of deletions of variables appearing in antecedents;
3. Minimum number of transformations of single-state invariants to transition invariants, or vice versa;
4. Maximum number of transformations of single-state invariants to transition invariants, or vice versa;
5. Minimum number of deletions of variables appearing in consequents;
6. Maximum number of deletions of variables appearing in consequents;
7. Maximum number of new variables to be added to the antecedents of existing assumptions;
8. Maximum number of variables to be included in the antecedents of new assumptions; and,
9. The required number of new assumptions.

Ideally, at each timestep, the agent would directly choose a value within a certain range for each parameter in the array. A limitation of the TF-Agents library for our purposes is that it cannot handle multi-dimensional actions of this type; actions for the deep q-network can only be encoded as a single discrete value, which can be interpreted in different ways depending on the environment. While we intend to seek an alternative solution to this issue in future work, we have implemented the following workaround for this proof of concept.

Our agent is allowed to choose as its action any integer between 0 and 19,682. We convert this number into base 3, and, if necessary, add preceding zeroes to ensure that the number always consists of nine numerals. Transforming the action in this way allows us to represent values in the range 000000000 to 222222222. From each numeral, we extract a setting for each of the nine parameters.

The major downside of this technique is that we can only set each parameter in the range 0 to 2. For larger specifications than our toy examples and relatively small case study, we are likely to want a significantly larger range to allow the update of more assumptions at a time. If applying our framework to the revision of guarantees, we are even more likely to want to apply numerous modifications at a time to ensure the correctness of the goal model. However, TF-Agents' deep q-networks, which map state observations to Q-values for each possible action, have limited dimensions, meaning the actions cannot exceed a certain value [57]. To have parameters up to even 333333333, when converted from base 4 to base 10, would require 262,144 actions; 19,682 suffices for our small test cases.

4.4.3 State observations

Unlike its actions, TF-Agents does allow an array of observations, which in our case consist of the following:

1. The weakness of the existing set of assumptions, represented as a value between 0 and 1. A weakness of 0 means there is only one trace that satisfies the formulae, i.e. the environment is very restricted; a weakness of 1 means the formulae are equivalent to *true*, i.e. the assumptions are very weak. The value is taken from the entropy of an automaton representing the assumptions; with permission we use code built for the implementation of [14].
2. The number of assumptions in the original set.
3. The average number of variables in the antecedents of assumptions.
4. The lowest number of variables in the antecedents of assumptions.
5. The highest number of variables in the antecedents of assumptions.
6. The number of transition invariants in the set.

The observations above are all collected automatically in our implementation.

7. The number of violated assumptions.
8. The average number of variables in the antecedents of violated assumptions.
9. The lowest number of variables in the antecedents of violated assumptions.
10. The highest number of variables in the antecedents of violated assumptions.
11. The number of violated transition invariants.

In the above we count only the number of variables in antecedents, since in our representation we only ever have one variable in the consequent of assumptions. Observations 7-11 are hard-coded for the examples in our proof of concept, though we intend to automate their collection in future work. After a violation is initially detected by model checking all the assumptions and the execution trace, we could identify which specific assumptions are violated by model checking each one individually.

For the smaller examples with which we test our proof of concept, we suppose that observations 7-11 are more useful for the agent than 2-6, as usually only the violated assumptions need to be updated. Larger specifications may require the modification even of unviolated assumptions, as discussed in 3.3.2 and 3.3.3, heightening the need for observations 2-6. Revisions of goal models will similarly require information about all of the guarantees.

We also include the following "meta-observations", which do not relate to the nature of the revisable theory. They are instead used to quantify the RL agent's attempts to find an acceptable solution, as explained in 4.4.4:

12. A count of the number of sequential failures of the revision system to find any solution.

13. A count of the number of sequential attempts that resulted in the revised specification being unrealisable.

14. A count of the number of sequential attempts that resulted in the revised specification being realisable, but violated by another execution trace, i.e. still not reflective of the environment's behaviour.

15. A count of the total number of attempts that were unsuccessful for any reason.

4.4.4 Episode structure, action evaluation and reward function

Ideally, our agent would always find in a single attempt a realisable specification that reflects the environment, and which we additionally hope has optimal weakness and coverage. In reality, this may take multiple attempts, each of which constitutes one step in the agent's episode of activity, but we must set limits so that the number of attempts is finite and the episode terminates. We detail here our algorithm for ensuring this, and for assigning the reward given at each step (i.e. the reward function). We have not set a discount for the weighting of the agent's future rewards, meaning it seeks to maximise its cumulative rewards over the entire episode rather than just the immediate reward for each action.

Timeout

The agent selects an action value that we process into the array of parameters as explained in 4.4.2. The parameters are passed to the revision engine which attempts to find a solution within those parameters. We found during training that for certain combinations RASPAL's computation could not terminate within a reasonable time. To allow training to continue, we instigated a timeout and give the agent a large penalty (-1000) if the timeout is reached and end the episode immediately. By ending the episode the agent learns that no future rewards can ever be achieved by taking this action, so we encourage the agent to avoid the actions that cause training to halt. Setting a time limit of course means that some possible solutions may not be found, but we set the time limit high to try to avoid this. In any case, if the subprocess cannot terminate, we never find these possible solutions anyway nor any others, so we consider the time limit an acceptable workaround, given that we are able to find alternative specifications.

No output

For certain parameter combinations RASPAL cannot compute a revised theory, as there are no answer sets satisfying those parameter settings. While this constitutes a failed attempt, it is not critical, as the agent must be allowed to explore to find an optimal solution, and the fruitless combination may have been 'wrong' by only a small number of parameter values. We therefore do not give a penalty for every such failure, as with an undiscounted future reward, the agent may try to end the episode quickly rather than risk accumulating penalties for every failed attempt. Conversely, we want to incentivise the agent to continue trying to find a solution.

Notwithstanding this, the agent must learn that parameters that return no solution are undesirable, so after ten such attempts we provide a penalty of -100 and end the episode. The penalty here is smaller than that for a timeout, so the agent learns that it is preferable to give a best effort by making ten attempts that give no solution, than seeking to end the episode immediately through a timeout. For training purposes, those ten failed attempts are informative about the effects of those ten actions; the agent learns much less by conducting a single action that does not terminate. We implement the ten-attempt limit through state feature number 12 from 4.4.3, where this value begins the episode at 0 and is incremented by 1 for each void output; if the value reaches 10, the penalty is given and the episode ended. If the revision engine at any point does manage to find a solution, this value is reset to zero. We keep track of the total number of reattempts by incrementing also feature 15; this value does not get reset during the episode.

The benefit of including the number of empty solutions as a feature, as with the other "meta-features" 13 and 14 detailed below, is that it provides the agent with an indication of how many more attempts it can make before the episode is terminated with a penalty, rather than with a reward for an acceptable solution (see below). This allows the learner to experiment with new parameters that might, for example, give a weaker realisable specification than the ones it is already able to produce. If it is unsuccessful and observes that it is nearing its final permitted attempt, it can resort to an action that it knows will return a realisable specification, albeit a stronger one than might be optimal. This way, we can aim for optimality whenever possible; if this hopeful attempt results in failure, we can still receive a suboptimal, but nevertheless acceptable, solution.

Unrealisable solution

For any solution that is returned, we test its realisability using the Spectra realisability checker [54]. As above, we do not immediately penalise unrealisable specifications, but instead give a penalty of -10 if such solutions are produced five times, implemented by incrementing the values of features 13 and 15, and end the episode. Feature 13 is reset to 0 if a realisable specification is found. Again, the penalty is smaller than that given at previous stages, so that the agent learns that it is preferable to produce five unrealisable specifications than ten empty solutions or a timeout. We want to incentivise the agent to continue trying to make progress towards an optimal solution while it seeks to maximise its future cumulative reward. We allow only five such attempts rather than ten, as the revision engine returns empty solutions more quickly than populated ones, so at runtime we can afford a greater number of the former type of failure. Nevertheless, these limits are somewhat arbitrary - a certain user may believe that a realisable solution always exists and it is preferable for the agent always to keep seeking it at runtime rather than giving up. In this case, the limit may be considerably higher, or there may be no limit at all.

In future work, we intend to experiment also with the checker RATSY, using which we can extract a countertrace that demonstrates unrealisability [16]. As described in 3.3.4, we could add each countertrace as a negative example for RASPAL, so that on the subsequent attempt, if there is one, the revision system is more likely to produce a realisable solution.

Erroneous solution

Our implementation gives room for testing whether a realisable revised specification is in fact correct with respect to the environment, by checking for further violations. Some of the components required for performing this were not ready to include for our proof of concept, but incorporating this step will be an important element of future work.

A new controller is synthesised from the realisable updated specification, likely using the Spectra synthesis tool, which is relatively fast [54]. Spectra's creators have built simulators for several of the specifications they provide [60]; if we test our framework on one of these specifications, we can stop the execution of the new controller if the new assumptions are violated again by the environment's behaviour. We can then extract the new violation trace from the simulator and provide it as an additional example for the RASPAL revision system.

Similarly to the realisability check above, we propose to give a (in this case smaller) penalty of -1 and end the episode if a correct set of updated assumptions cannot be found within 5 attempts, measured by incrementing features 14 and 15. The reason for the decreasing penalty is the same as for the previous checks.

Optimality

Any realisable set of revised assumptions that is not violated by another execution trace is an acceptable solution, in response to which we end the episode; ideally every episode would end in this way after one step. We must also give the agent a reward to indicate that this is the desired outcome, but we vary the size of the reward depending on the quality of the solution. We have implemented only the weakness measurement but not the coverage computation, with which we will experiment in future work.

The reward is assigned as follows. As with the original weakness calculation undertaken for the state observation (see 4.4.3), the weakness value is between 0 and 1, with 1 indicating the greatest weakness. We only calculate the weakness of acceptable specifications, as this value is otherwise irrelevant; conducting unnecessary calculations would needlessly increase computation time. From the weakness value, we subtract the value of state feature 15, namely the total number of attempts that were unsuccessful for any reason. This means that the reward is inversely proportional to the number of attempts that were required to reach an acceptable solution, incentivising the agent to find this as quickly as possible; the weakness value encourages the the solution not just to be returned quickly, but also to be optimal.

We so far have *reward = weaknessValue - totalFailures*. To this value we add a constant that is equal to (or greater than) the maximum number of unsuccessful attempts. With our limits, this is 36 with no check for subsequent violations, or 144 with this check. We ensure by this constant that an acceptable solution, even if it required the maximum number of tries, is preferable to ending the episode with an unacceptable solution. Without this constant, the agent would learn that it is preferable to terminate the episode through some kind of failure than to keep seeking a solution for as long as permitted, since the penalty for ending in failure might be less than the subtracted value for the number of attempts.

Chapter 5

Experimental Results

5.1 Case study

We have tested our framework using a specification similar to one offered by the creators of Spectra, which describes the activities of an automated forklift [61]. The forklift can either be far away or nearby, and at each timepoint, it chooses one action out of staying in position, changing position, lifting an object or dropping an object. We have converted the original specification's enumeration variables to Boolean form and named the assumptions and goals. To maintain the integrity of the original enumeration variables, we have added domain-dependent constraints to ensure that only one of the *command* variables can be true at any one time. The 'correct' version of our specification, with no erroneous assumptions, is as follows, given in the Spectra specification language; Spectra uses a *next* operator to indicate the **X** symbol we have been using previously:

```
module ForkLift

env boolean fardistance ;
sys boolean liftcommand ;
sys boolean dropcommand ;
sys boolean gocommand ;
sys boolean idlecommand ;

assumption – idle_far
G (idlecommand=true & fardistance=true → next( fardistance=true ) );

assumption – idle_close
G (idlecommand=true & fardistance=false → next( fardistance=false ));

assumption – go_close
G (gocommand=true → next( fardistance=false ) );
```

assumption – close_nolift
G (fardistance=false & liftcommand=false → next(fardistance=false));

assumption – close_withlift
G (fardistance=false & liftcommand=true → next(fardistance=true));

guarantee – s_init
liftcommand=true;

guarantee – s_11
G (fardistance=true → next(dropcommand=false));

guarantee – s_12
G (fardistance=false → next(gocommand=false));

From this we have created four test cases, each by making one of the assumptions erroneous in a different way with respect to the true environment behaviour. In all cases, the specification remains realisable, as it would be for a specification written by a designer with erroneous assumptions.

Case 1:
assumption – go_close
G (gocommand=true → **fardistance=false**);

Case 2:
assumption – idle_far
G (**dropcommand**=true & fardistance=true → next(fardistance=true));

Case 3:
assumption – idle_far
G (idlecommand=true & fardistance=true → next(fardistance=**false**));

Case 4:
assumption – close_nolift
G (fardistance=**true** & liftcommand=false → next(fardistance=false));

For each case, we have written an execution trace that:

- satisfies all of the guarantees and correct assumptions;

- violates only the erroneous assumption; and,
- demonstrates the true behaviour that should be reflected in the updated assumption.

We give the trace for Case 2 by way of illustration:

```
→ State: 1.1 ←
fardistance = TRUE
liftcommand = TRUE
dropcommand = FALSE
gocommand = FALSE
idlecommand = FALSE
→ State: 1.2 ←
fardistance = TRUE
liftcommand = FALSE
dropcommand = FALSE
gocommand = FALSE
idlecommand = TRUE
→ State: 1.3 ←
fardistance = TRUE
liftcommand = TRUE
dropcommand = FALSE
gocommand = TRUE
idlecommand = FALSE
→ State: 1.4 ←
fardistance = FALSE
liftcommand = TRUE
dropcommand = FALSE
gocommand = FALSE
idlecommand = FALSE
→ State: 1.5 ←
fardistance = TRUE
liftcommand = FALSE
dropcommand = TRUE
gocommand = FALSE
idlecommand = FALSE
→ State: 1.6 ←
fardistance = FALSE
liftcommand = FALSE
dropcommand = FALSE
gocommand = FALSE
idlecommand = TRUE
End
```

In all the experiments, steps in which no solution was found by the revision engine took one or two seconds. Steps that did produce a revision usually took up to a minute, though in some cases lasted up to our timeout for RASPAL of two minutes.

5.2 Experiment 1

We want to test whether the agent is able to converge to a policy that generalises to unseen violations, since at runtime the assumptions that turn out to be erroneous may be different from the ones that were incorrect during training. The agent should be able to learn to perform similar corrective actions in similar, though different, situations.

Test cases 2 and 4 are similar, in that they both involve an incorrect literal in an antecedent containing two literals; they therefore give rise to identical state features aside from, possibly, the weakness of the assumptions. Our first experiment therefore used one of these cases for training, and the other for evaluation; we ran this same experiment twice. Four evaluation episodes were run for every 100 training steps, and the average reward on the evaluation episodes calculated. On each occasion, the experiment was left running for approximately 10 hours.

Figures 5.1 and 5.2 plot the average reward on the evaluation case against the number of training steps. We have plotted the timeout penalty as -120 rather than -1000 so that more of the curve can be seen as the agent learns to find the optimal realisable solution in fewer attempts. As it converges on such a policy, the average reward increases by only 1 or 2, after having jumped up from the penalties. As explained in 4.4.4, a reward of over 36 indicates that the solution was found on the first attempt.

5.2.1 Run 1

As illustrated in Figure 5.1, on the first run of the experiment the agent converged on an ‘optimal’ policy after about 3,400 training steps, such that it was able to find a realisable specification with weakness value 0.7169914246 after only one attempt on the evaluation case. The policy revised assumption *silly_idle_far* to

G (**dropcommand**=true & fardistance=true → next(fardistance=**false**));

It changed the truth value of the consequent rather than replacing *dropcommand* in the antecedent with *idlecommand*, as we would desire given the ground truth assumption in the original specification. While this revision is consistent with the trace and gives a realisable specification, the assumption remains erroneous. This demonstrates the importance of implementing the further violation checks in our agent in future work, to ensure the updated assumption actually reflects the environment’s behaviour.

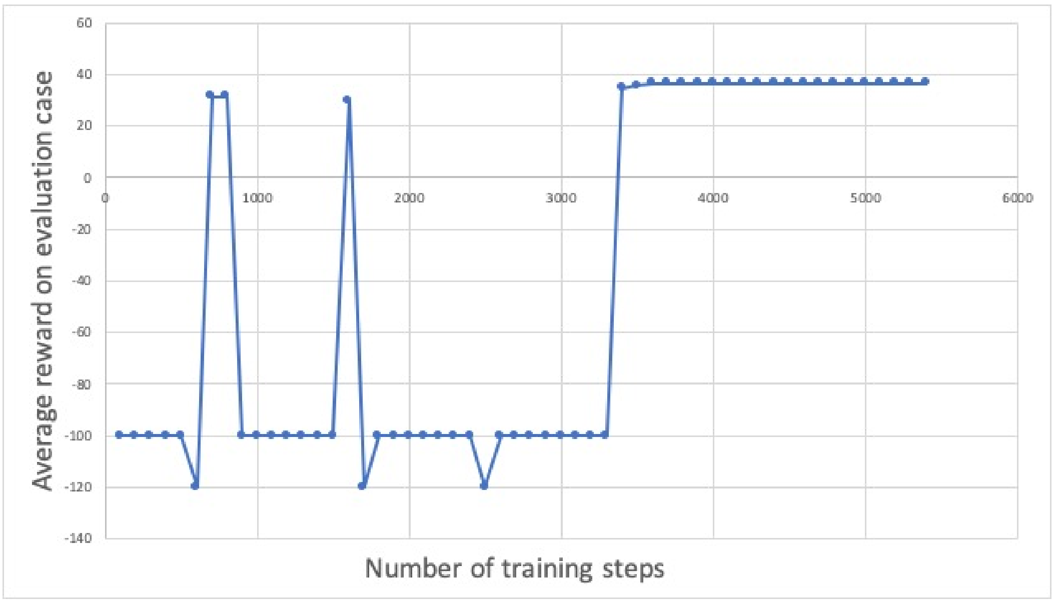

Figure 5.1: Experiment 1, run 1

5.2.2 Run 2

Figure 5.2 shows that on the second run of the experiment, the learner converged on a timeout policy for the evaluation environment. While this appears on the surface to be problematic, the same policy was producing a realisable specification after two attempts in the training environment, and we believe that this would also have been true of the evaluation environment if the timeout had been set slightly higher (with further training, the learner would also probably have converged on a policy that found the acceptable solution in one attempt rather than two). Equally, if the timeout had been set lower, the learner would likely have converged on an alternative policy in the training environment, such as the one seen in the first run of this experiment. Either way, this demonstrates the benefit to be derived from a faster revision engine than RASPAL, such as the one under development currently, for ensuring that acceptable solutions are not lost.

The action that caused the timeout in the training environment, when tested separately without the timeout, found the correct revision to the erroneous assumption. However, it also incorrectly removed the *liftcommand=true* variable from the antecedent of the assumption *close_withlift*. Again, this illustrates the need for the subsequent violation checking phase to ensure that realisable solutions are correct for environment behaviours other than that seen in the initial violation trace.

5.3 Experiment 2

We also want to test whether the agent is able to broaden its range of actions so that it can correct erroneous assumptions that have similar state features but which are violated in different ways. Case 3 has the same features as Cases 2 and 4, apart

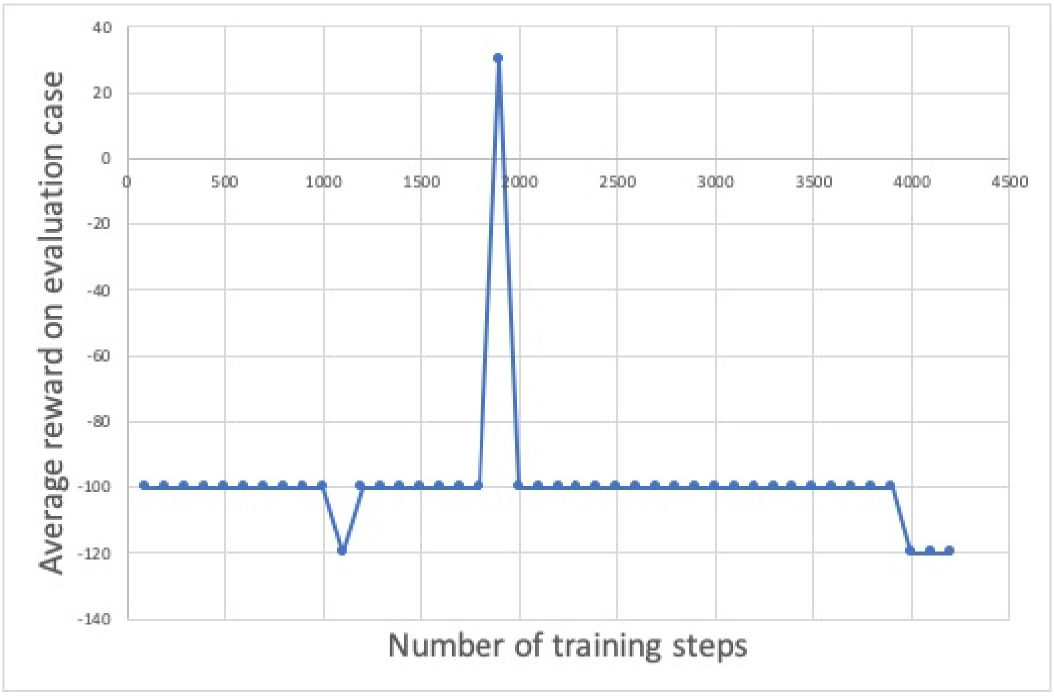

Figure 5.2: Experiment 1, run 2

from possibly the weakness, but the consequent needs updating rather than one of the variables in the antecedent. We left Case 2 as the evaluation case, but alternate between training on Case 3 and Case 4 for each episode.

Figures 5.3 and 5.4 show that the agent was able to converge on a policy for finding an optimal solution on the first attempt within only 500 steps on the first run, but took slightly longer on the second run. In both cases, convergence occurred significantly faster than in Experiment 1, meaning that training only took up to three hours. However, as in Run 1 of Experiment 1, the policies incorrectly modified the consequent rather than the antecedent, meaning that we have not been able to show the agent responding differently to different violations of assumptions that share similar features. However, we do not consider this a failing of our agent: implementing the further violation checks in future work should remedy the issue of incorrect updates, which should force the policy to be more diverse as necessary.

5.4 Experiment 3

Our third experiment tests whether our system might be scalable. The setup is as for Experiment 2, but training now alternates between Cases 1, 3 and 4. The erroneous assumption in Case 1 lacks the *next* operator that it should contain to be correct, meaning that the state features observed for this case are different, as is the necessary revision to be executed. By adding this case to the others on which the RL agent is trained, we are testing that it is able to handle a greater range of state observation values, and that it can vary its actions for different observations.

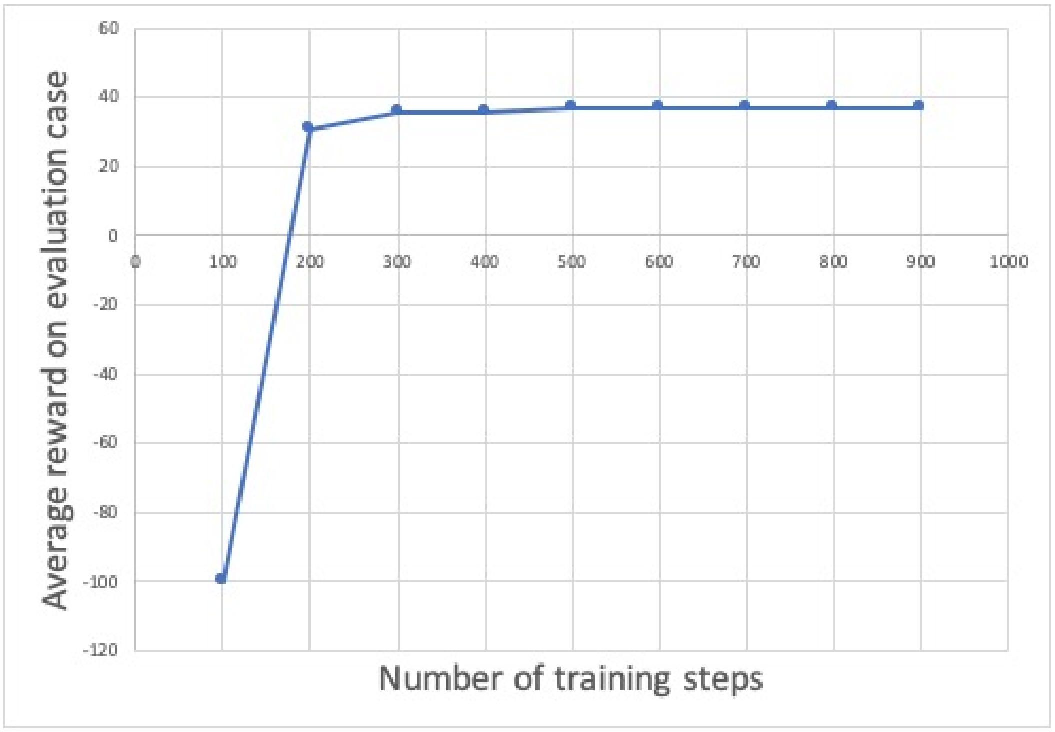

Figure 5.3: Experiment 2, run 1

Figures 5.5 and 5.6 show that on both runs, the agent quickly converged on a policy for finding a solution in one or two attempts, with training taking less than two hours. On the second run, as with many of the previous experiments, the revision updated the consequent rather than the *dropcommand* variable in the antecedent as we would have desired. On the first run, the policy additionally deleted the *fardistance* variable from the antecedent, to give

G (dropcommand=true → next(fardistance=false));

which is further from what we consider to be the correct assumption. The policy also added an unnecessary new assumption, which was identical to the existing assumption *go_close*.

5.5 Evaluation

The experiments show that in all cases, aside from one affected by the designated timeout, the learner converged on a policy for finding an acceptable revision within only one or two attempts. The RL system is able to train on multiple sets of erroneous assumptions with related violation traces, which is promising for the scalability of our system, since we want our agent to have a policy for dealing appropriately with a range of different violations to different sets of assumptions. In fact, the experiments appear to indicate that convergence occurred faster with training on more

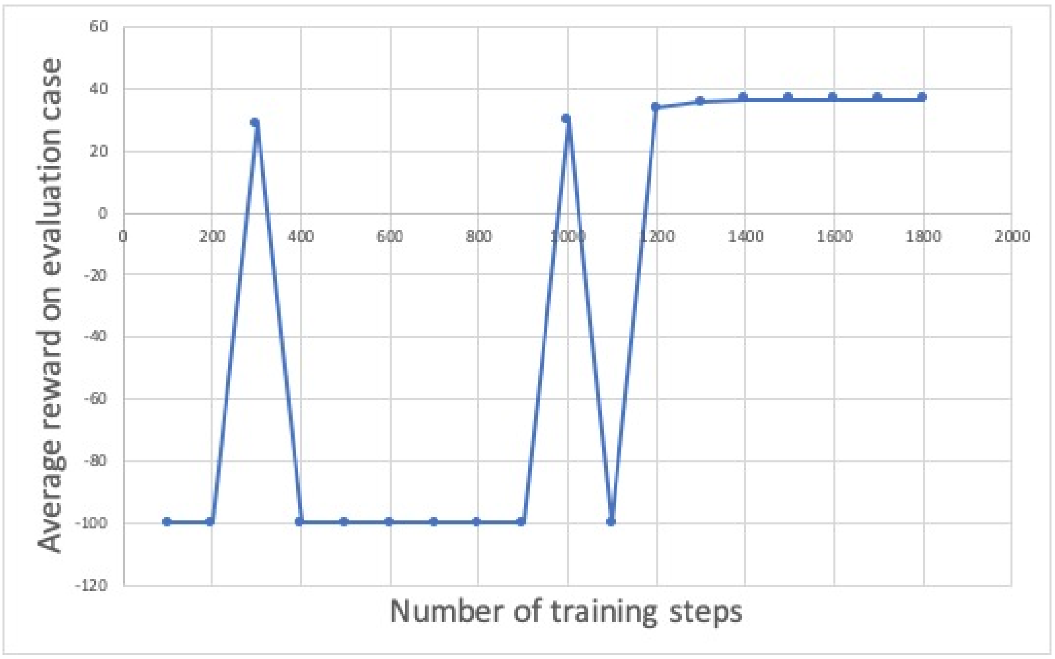

Figure 5.4: Experiment 2, run 2

environments. This finding seems counterintuitive so must be subjected to further experimentation in future work. We suggest that it may be related to having a greater range of feature observations, since each different set of starting assumptions shows slightly different feature values to the agent.

In many cases, the revision enacted by the agent was not the one we consider to be correct with respect to our knowledge of the environment's behaviour, though it was nevertheless a logical update for the revision engine to have made. As noted throughout, this highlights the importance of checking the update for further violations to make sure the revised assumptions correctly reflect the environment. Equally, providing longer example traces to the revision engine, which would contain more information about the environment's actual behaviour, would help make sure the updates are sensible.

We note that we have not observed any penalties of -10 when testing our implementation. This penalty was to be applied when an unrealisable specification was produced five time, causing the episode to terminate. Instead, either a realisable specification was produced within a certain number of attempts, or penalties were given for a timeout or no output by RASPAL. The lack of -10 penalty might be related to the size and complexity of the specifications on which we have tested the framework; with more complex specifications, we might be more likely to see revisions being produced that do not give rise to a realisable specification.

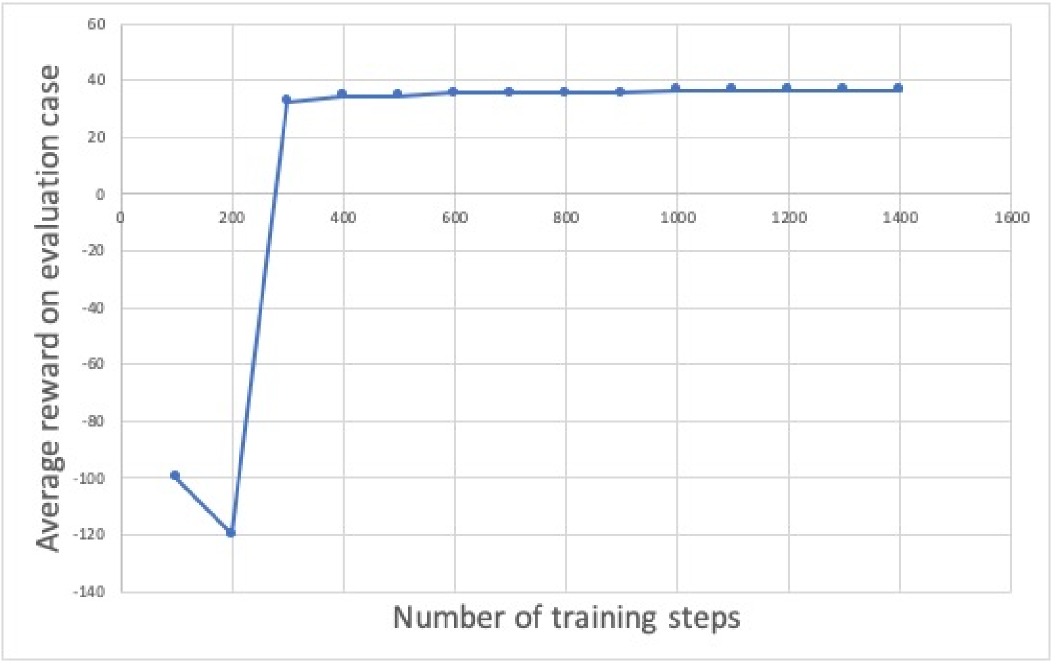

Figure 5.5: Experiment 3, run 1

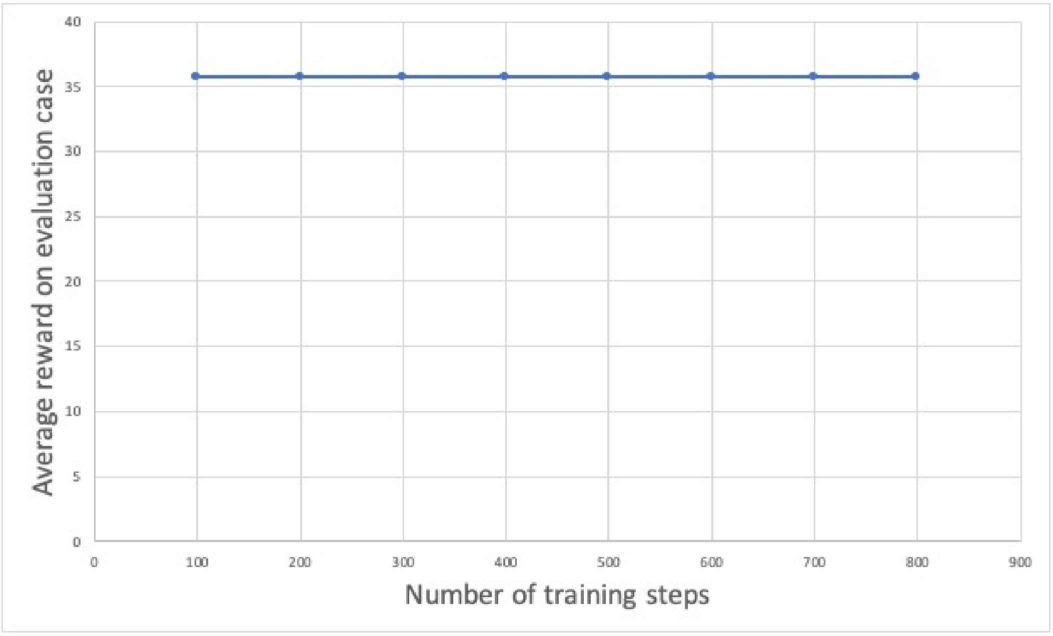

Figure 5.6: Experiment 3, run 2

Chapter 6

Discussion

6.1 Limitations

We review here the various limitations of our framework and implementation which we have highlighted in the chapters above, and point to possible solutions to these issues.

6.1.1 Different performance on training and evaluation cases

It may sometimes occur that the policy learnt during training fails on the evaluation case, which is a situation we want to avoid when applying our framework to real-world runtime situations. For example, the second run of Experiment 1 showed that a policy that appeared to be optimal on the training case led to a timeout on the evaluation case. The likelihood of this occurring can firstly be reduced by training on more cases, so the agent learns about the effects of its actions in a broader range of situations, and improves its policy in response.

We might also hope to remedy this by more closely integrating the evaluation episodes with the training. In our current setup, there is no feedback between the average reward received on the training environment and the policy learnt by the agent, allowing it to converge on a policy during training that fails at runtime. In the situation we saw in the second run of Experiment 1, an alternative implementation might have allowed the timeout to be decreased reactively, making the policy learnt during training no longer optimal. The agent would then have to seek a new policy, that would hopefully mean a solution could be found on the evaluation case. Similarly, we might vary the amount of exploration conducted by the agent, to ensure its learnt policy is not only locally optimal. Just as it would be better to train on more cases, evaluating on more examples would similarly help to avoid the learnt policy inadvertently failing at runtime.

6.1.2 Scalability of system

As explained in 4.4.2, the TF-Agents library that we have used for our implementation of the RL agent does not accept multi-dimensional actions, and the deep Q-network can only hand uni-dimensional actions of a limited size. This means that the range of the parameters that we ask the agent to set for the revision system is restricted. More complex specifications might require more assumptions to be updated at a time, or greater numbers of conditions to be added and deleted. For our framework to be able to handle this, in future work we intend to experiment with different RL libraries and with implementing the agent from scratch to improve scalability.

6.1.3 Representation of assumptions

In 4.2.1 we explained that, given our representation of assumptions as three normal clauses, and in order to prevent the revision system from strengthening assumptions by adding conjuncts to the consequent, we have had to prevent consequents from containing more than one variable. However, in this representation, the variables that a transition invariant indicates are to be true at the next timestep must also appear in the consequent. This means that our system is limited to dealing with transition invariants with only one **next** variable, whereas real-world specifications often contain assumptions with more such variables.

A solution to this obstacle might require a different formalisation of the assumptions, where a preprocessing step could remove the **next** operator and label the variables that appeared after it. For example, $\mathbf{G}\ (\mathrm{a} \rightarrow \mathbf{X}(\mathrm{b} \vee \mathrm{c}))$ might instead be represented as $\mathbf{G}\ (\mathrm{a} \wedge \neg\, \mathrm{bx} \rightarrow \mathrm{cx})$. This would also necessitate a broader set of mode declarations, to allow the revision system to use each variable with either its usual or its 'next' meaning.

6.1.4 Failure of framework to find a solution

When experimenting with our proof of concept on small example specifications and traces, the learnt policy was always able to find acceptable or optimal solutions within a small number of attempts. It is unclear whether this will be the case with more complex specifications, for which it might be more difficult to find solutions; on the other hand, larger specifications usually consist of large numbers of short assumptions that might still be relatively easy to fix. The objective of training is to allow the agent to explore the search space at design time, so that at runtime only a small number of attempts are required to find a solution. If at runtime, in an unseen environment, the agent is still not able to find a solution within the specified number of attempts, we might consider incrementing the limit, if the user would rather the agent continue trying to find an appropriate revision, rather that trying to update the specification by hand.

We consider a greater obstacle to be the necessity of a timeout for the revision engine's computation. Again, during training the agent should be able to find a policy that sets the revision engine's parameters so that the search space is restricted enough that solutions can be returned before the timeout is reached. Nevertheless, our experiments have shown that a policy that returns a solution within the timeout on a training example might not be able to do so on the runtime environment. We might therefore consider increasing the timeout at runtime, presuming that the user would rather the agent continue trying to compute a solution, which might be less expensive than the controller's task being abandoned and the specification repaired manually. We also hope to reduce this problem in future work by using new, faster revision systems than RASPAL. Adding only the violated assumptions, or a subset of the total assumptions, to the revisable theory might similarly help with scalability, since larger programs lead to longer computation times.

There may remain situations in which our framework is incapable of correctly updating the assumptions, regardless of the permitted time or number of attempts. This should not necessarily be the case, given that if a solution exists, the agent should be able to alter the parameters until the solution is eventually found. As we have already mentioned, we can increase the probability of this result by adding further violation traces as positive examples for the revision system, and traces demonstrating unrealisability as negative examples. Nevertheless, if our framework does not appear to be having any success, a user might have to resort to existing techniques for the controller gracefully failing the task, and the specification might have to be rewritten by hand and counterstrategy-guided refinement methods for reaching realisability.

6.2 Alternative implementation of reward function

A possible alternative to our implementation of the RL agent's reward function might give a penalty for every unsuccessful attempt, even those which do not terminate the episode; the penalty could increase with each subsequent attempt, with the aim of encourage the agent to find an acceptable solution more quickly. However, if the agent receives a penalty for every step of the episode, it is likely to seek to terminate the episode in failure sooner rather than risk further penalties, as already mentioned. We could get around this by applying a full discount to future rewards, so that the agent would only seek to maximise its immediate, rather than cumulative return.

We believe this may lead to undesirable effects, as the agent is discouraged from exploring for more optimal solutions. After a certain amount of training, the agent may have learnt that a particular action produces an acceptable solution for which it receives a reward, but this solution does not exhibit particularly good weakness or coverage. Nevertheless at each timestep the agent, seeking to maximise its immediate reward, would prefer to receive its suboptimal reward than a penalty for an unsuccessful attempt, especially if the penalty increases with each step. On the other hand, our implementation encourages the agent to search for higher quality specifications when possible, but allows an acceptable suboptimal solution to be returned otherwise.

6.3 Applicability to other problems

Our framework could be used with little modification for revising requirements models. This can be necessary when the goals are found to be incorrect, or the assumptions on which they rely change [12, 30]. We discuss here some of the considerations that might be necessary when adapting our framework for goal revision.

6.3.1 Goal models and revision

The field of requirements engineering refines the high-level goals of a specification into sub-goals that, if their conjunction is satisfied, mean also that their parent goal is satisfied. In this way, goals and their refinements (sub-goals) form directed graphs wherein refinements are mutually dependent for the satisfaction of their parent [12, 62]. This graph must be:

- **complete** - the sub-goals and the environment assumptions must satisfy the parent goals;
- **consistent** - the sub-goals and the environment assumptions must be consistent; and,
- **minimal** - the set of sub-goals necessary for satisfying a parent goal must be minimal [30].

If the goals are found to be incorrect or the environment assumptions change, the goals must be revised. Here the aim is not to ensure realisability and weakness, as with assumptions, but instead completeness, consistency and minimality. The locality of changes, and coverage in the sense of behavioural similarity, are also preferred [12, 30].

[12] borrows a simplified example from [63] of an automated aeroplane landing system with the high-level goal $\mathbf{G}(\mathit{MovingOnRunway} \rightarrow \mathbf{X}\mathit{ReverseThrustEnabled})$. This is refined into the left sub-goal $\mathbf{G}(\mathit{MovingOnRunway} \rightarrow \mathit{WheelsTurning})$ and the right sub-goal $\mathbf{G}(\mathit{WheelsTurning} \rightarrow \mathbf{X}\mathit{ReverseThrustEnabled})$. Figure 6.1 illustrates this model. At runtime it may transpire that the runway is slippery, so the plane is able to move without the wheels turning, but we still want the reverse thrust to be activated. We therefore need to update the child goals such that the parent remains satisfied.

6.3.2 Existing approaches for revising goals

[30] has demonstrated the use of RASPAL for revising requirements. A nonmonotonic ILP system is particularly appropriate for this task, as it allows more complex semantic changes to be executed than might be possible with revision engines that use only atomic operators [11]. This is helpful given the need to maintain correct goal models: changes to one goal might necessitate complex modifications also to

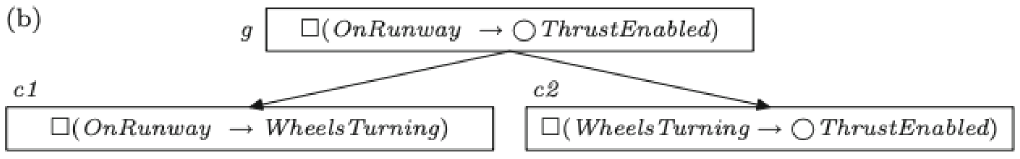
(b)
g
□(OnRunway → ◯ ThrustEnabled)
c1
□(OnRunway → WheelsTurning)
c2
□(WheelsTurning → ◯ ThrustEnabled)

Figure 6.1: Simple aeroplane goal model [12]

its siblings, children or parents [12, 30].

[12, 30] have shown the use of meta-constraints for preserving the semantics of the goal model. A revision system that is only guided by the minimality of the revision might replace the antecedent of the right child with a propositional atom that is observed in the violating execution trace such as *WheelsPulseOn*, giving $\mathbf{G}(\textit{WheelsPulseOn} \rightarrow \mathbf{X}\textit{ReverseThrustEnabled})$. However, it may neglect to revise the other child goal in such a way as to preserve the correctness of the goal model; meta-constraints allow us to ensure that the hypothesis space is restricted to only such acceptable revisions. A meta-constraint that forces any variable that appears in the antecedent of the right sub-goal to appear also in the consequent of the left sub-goal would result in a semantically correct revision of the left child: $\mathbf{G}(\textit{MovingOnRunway} \rightarrow \textit{WheelsPulseOn})$.

6.3.3 Using RL for revising guarantees

Adding our RL method to the existing approaches for revising goals is likely to be beneficial for guiding the search space towards acceptable and optimal solutions. A reward function for this agent would be designed to ensure the necessary and preferred qualities of a revised goal model which we mentioned in 6.3.1.

We suggest here a number of additional supplemetary features and actions for the RL agent that may be useful in the context of guarantee revision, but are less relevant for updating assumptions. When revising assumptions in complex specifications, it is conceivable that several may need to be revised at a time, possibly including unviolated ones, to reach a realisable specification and remain minimal in the number of assumptions. On the other hand, whenever a guarantee is modified, it is highly likely that other guarantees will also need changing to preserve the correctness of the goal model. Whereas it may be unclear which unviolated assumptions need updating when a violated one is modified, it is likely to be more obvious which guarantees in the goal model are related, as parent, child and sibling guarantees will by their nature share some variables.

For assumptions, therefore, especially in larger specifications, we are likely to want to include all the assumptions in the revisable theory, unless some are specifically not desired to be changed. However, an action for a RL agent that is updating guarantees could be the choice of which guarantees to include in the realisable theory. This would necessitate it receiving as observations which variables appear in the incorrect guarantees, so it can include in the revisable theory other guarantees containing these variables. More guarantees in the revisable theory might also mean the agent acts to ensure more witness traces are covered as examples. The agent could also select, as an additional action, the types of meta-constraint that might be required for ensuring the correctness of the goal model, based on which guarantees are going to be revised.

Chapter 7

Related work

Previous works have recognised that assumptions made about the environment at design time may be violated at runtime, and numerous methods have been proposed for automatically dealing with this uncertainty about the environment's behaviour. These methods generally do not involve the production of new assumptions or re-synthesis, which until now has usually been triggered by a human-in-the-loop.

One approach that does attempt to modify erroneous assumptions at runtime is [8], extended in [64]. This work updates the assumption by incrementally adding disjuncts representing the observed, violating behaviour to the existing assumption. While tailoring the assumption update in this way does ensure coverage 2.1.5, it may also lead to overly specific and complex assumptions. Assumptions that are weakened in this way are also likely to result in the specification becoming unrealisable, which the authors of [8] seek to remedy at runtime by asking the user to provide additional liveness assumptions. Our contribution is a framework that may avoid these issues.

7.1 Coping with environment uncertainty

Other techniques for dealing with environmental uncertainty can be roughly split into building in resilience at design time; responding to violations at runtime without re-synthesis; and learning more accurate environment models.

7.1.1 Built-in resilience

[7] presents a method for synthesising controllers that are robust to possibly inaccurate models of the environment. The controllers are designed to be correct not just for the assumed environment model, but also for a family of models that include it, thereby giving room for divergence. However, the controller is as such robust only to certain violations, so there may still be unforeseen situations in which it is required to update assumptions and conduct re-synthesis. The same can be said of the work in [65], which proposes the formation of a robust winning strategy from the combination of separate strategies, each of which is tolerant to certain unexpected events.

Likewise, [8] adds so-called fall-back transitions during synthesis such that the robot may be able to preserve its safety requirements when assumptions are violated and proceed towards its goals if and when the violation ends. When this is not possible, the assumption is updated by the weakening method described above.

In a similar vein, the framework in [9] enables the synthesis of controllers that can cope with intermittent violations of assumptions, as long as the violations are separated by periods in which the controller can recover. Again, this is not sufficient for cases in which the violations are not limited and temporary, so the need remains for our method for updating assumptions that are more enduringly erroneous. Likewise, the algorithm in [66] addresses the synthesis of controllers from safety specifications, but does not tackle situations in which the assumptions are found to be erroneous.

7.1.2 Runtime responses without re-synthesis

[1] highlights the importance of describing environmental assumptions explicitly, as erroneous implicit assumptions can mean that the guarantees are not satisfied, and the specification is therefore not realisable. [10] builds on this observation by offering a method for monitoring assumptions at runtime, to ensure that the guarantees are not left unsatisfied due to silent assumption violations. Specifications of the assumptions are used to synthesise monitors that use Stream Runtime Verification, a form of runtime verification that can handle continuous data inputs rather than only Boolean observations. The reactions to detected violations consist of logging for offline analysis, or otherwise remediatory planning by the controller to gracefully fail the mission or to attempt to satisfy the guarantees in spite of the violations.

[67] extends other studies in runtime verification and assurance, such as [68], by offering a programming framework that allows for two controller modes. The advanced controller mode gives high performance for a robot operating under nominal conditions, while the lower performance safe controller mode is used to keep the robot operating safely when monitors detect that an assumption violation is possible. While beneficial in some circumstances, this approach does not address the need for updating erroneous assumptions when required. [69] presents a parallel methodology for self-adaptive systems, wherein a tiered framework of operational strategies, each based on different assumptions, allows the system to transition between models when the assumptions of higher tiers are broken.

7.1.3 Updating environment models at runtime

Another avenue of research seeking to address the problem of inaccurate or incomplete understandings of environment behaviour is based around non-RL methods for updating environment models at runtime. While the proposed methods are not used to modify the assumptions based on the updated models, it is feasible that this extension could be applied. However, the benefit of our RL approach is that assumptions can be modified in accordance with the learned policy without the need for an explicit model.

The work in [70] uses stochastic gradient descent to estimate observation probabilities for rules from execution traces at runtime; the new rules are then added to the existing model. [71] similarly uses probabilistic rule learning to update models. These algorithms are implemented for self-adaptive systems, which make use of an environment model that may need updating. [72] likewise proposes a procedure for self-adaptive systems that builds controllers that allow the system either to proceed towards its goals, or else to learn more about the environment. As this approach is aimed at self-adaptive systems, its major weaknesses in our setting are two-fold. The environment must be able both to report a current state ID and to be reset if the goals are found to be unsatisfiable with the current behaviour, in order to allow the system to make use of its new knowledge on another attempt. Neither of these two requirements are applicable to our problem, where an agent operating in a continuous environment needs to be able to handle violations in real time.

[73] defines a fragment of LTL to describe a mission for a robot incrementally to learn a model of the environment. While the algorithm is complete, the task is executed in infinite time, which is again not appropriate for our purposes of reacting to violations as they arise.

7.2 Current uses of RL in similar problems

Our use of RL to determine parameters for ILP systems and to modify assumptions is a novel contribution. Nevertheless, RL has been used for similar problems, which we survey below.

7.2.1 RL in self-adaptive systems

Autonomic/self-adaptive systems respond to changes in the environment by selecting an appropriate action for reconfiguring themselves in some way. Due to the complexity of the systems, the space of possible actions is often very large and the optimal adaptation may be domain-dependent, suggesting a parallel with our problem. Moreover, as with our problem, the best adaptation can depend on numerous quality criteria to be maximised and traded off against one another, such as service times and quality.

RL algorithms have been implemented to find an optimal adaptation policy for self-adaptive systems such as news web applications [74], virtual machines [75] and other software systems [76, 77] subject to varying workloads; robots with changing priorities [78, 79]; as well as for evolving the possible adaptation actions themselves [80]. [81] proposes that the space of adaptation actions be structured as a feature model in order to speed up the convergence of learning at runtime, while [82] proposes a model-based approach for RL in self-adaptive systems for the same reason.

However, in our problem the training takes place at design time, as is also the case for the self-adaptive system RL agent in [74], meaning that the speed of convergence is less of a concern. Equally, our model-free RL approach is able to guide the revision engine to acceptable solutions while avoiding the need for an explicit model of the environment's behaviour.

7.2.2 Other related uses of RL

RL algorithms have also been used in settings that are arguably more closely related to our problem than are self-adaptive systems, but in ways that are less reflective of our proposed approach.

[83] shares the objective of the works in 7.1.1 for seeking to synthesise controllers with a degree of robustness to environmental uncertainty, but uses RL in its method. RL is used to learn control policies for Markov Decision Processes with unknown stochastic behaviour, such that the policy maximises the probability of satisfying the given LTL specification. However, it does not enable responses to assumption violations at runtime.

[84] uses RL to accelerate falsification of cyber-physical systems through a reward function that seeks to minimise a robustness value of properties, in a comparable way to our own use of preferences in our reward function. However, here the properties are in STL rather than LTL. Correspondingly, in [85] RL is employed to speed up the search for violating inputs while conducting fuzz testing for networking devices. However, we intend to use RL in the response to violations rather than in their generation.

[86] demonstrates the application of RL to symbolic reasoning, but it is used here to learn heuristics for solving quantified Boolean formulae. We instead use RL to restrict the search space for our symbolic reasoning component.

Chapter 8

Conclusion

8.1 Achievements

We have designed a RL agent that can learn a context-specific policy for setting parameters for a nonmonotonic ILP theory revision system. Doing so restricts the system's search for revised sets of assumptions to those that produce a realisable specification that correctly reflects the environment's behaviour. The framework can also be used to favour updates that give weak assumptions that permit similar environment behaviours to the designer's original intent.

We have instantiated the key elements of our framework as a proof of concept. Our implementation demonstrated the ability of the RL agent to converge, within an acceptable amount of training time, on policies that could update different violated assumptions appropriately in a single attempt at runtime. This contribution is significant for both the formal methods and ILP communities.

8.2 Future work

There are numerous possible avenues for continuing our research, which we intend to pursue in future work. These relate to addressing the limitations of our current implementation; implementing the supplementary elements of our proposed framework, and subjecting it to further testing and evaluation.

8.2.1 Addressing limitations

The two most significant limitations of our current implementation relate to the scalability of our system. Firstly, the RASPAL revision engine can be slow when tackling larger programs, and the timeout we have introduced to mitigate this can mean that suitable solutions are not found. We intend to instantiate our framework on a soon-to-be-released ILP revision system that may provide quicker updates.

The TF-Agents library we have used to build our agent limits the number and type of actions with which we can equip our agent. We will explore other implementations of the RL system to allow a greater range of parameter values to be offered to the ILP system, allowing it to revise larger and more complex specifications.

8.2.2 Augmenting the implementation

There are a number of components of our proposed framework that we have not yet implemented, as well as other ways in which we hope to extend it. Mostly notably, our experiments demonstrated the necessity of further violation checks for ensuring that the revised assumptions correctly reflect the true environment behaviour. Spectra's tool for generating traces automatically will significantly aid this process. Equally, we wish to include the addition of counterstategies as negative examples to facilitate our system's search for realisable specifications.

While RASPAL's favouring of minimal revisions helps to an extent with ensuring syntactic similarity with the original assumptions, we have not yet implemented our proposals for seeking behavioural coverage either by the RL agent's actions or reward function. We might also augment our current implementation by extending our parsers to handle more of the Spectra specification language.

8.2.3 Further evaluation

We intend to conduct further experiments on our system to evaluate both the quality of its revisions and its scalability. We wish, for example, to test whether the revised assumptions are generally weaker than those produced by other approaches, such as the counterstrategy-guided refinement techniques. Moreover, it will be important to compare the quality and speed of revisions produced by our framework and a revision system without the help of an RL agent, to demonstrate the benefit of our proposal.

We will also experiment with larger and more complex case studies, a greater variety of training examples, to gain an insight into the applicability of our framework for real-world use. We also wish to test it on other problems, such as adaptations of requirements.

Bibliography

[1] Nicolás D'ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing Nonanomalous Event-Based Controllers for Liveness Goals. *Association for Computing Machinery Transactions on Software Engineering and Methodology (TOSEM)*, 22(1), March 2013. pages 1, 3, 67

[2] Davide G. Cavezza, Dalal Alrajeh, and András György. Minimal Assumptions Refinement for GR(1) Specifications. *CoRR*, abs/1910.05558, 2019. pages 1, 3, 5, 7, 10, 26

[3] A. Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. *Automata Languages and Programming*, 372:179–190, January 1989. pages 1

[4] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) Designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 364–380, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. pages 1, 3, 5, 7

[5] Shahar Maoz and Jan Oliver Ringert. On the software engineering challenges of applying reactive synthesis to robotics. In *Proceedings of the 1st International Workshop on Robotics Software Engineering*, RoSE '18, page 17–22, New York, NY, USA, 2018. Association for Computing Machinery. pages 1

[6] Thein Than Tun, Amel Bennaceur, and Bashar Nuseibeh. OASIS: Weakening User Obligations for Security-critical Systems. *2020 IEEE 28th International Requirements Engineering Conference*, pages 113–124. pages 1

[7] Ufuk Topcu, Necmiye Ozay, Jun Liu, and Richard M. Murray. On Synthesizing Robust Discrete Controllers under Modeling Uncertainty. In *Proceedings of the 15th Association for Computing Machinery International Conference on Hybrid Systems: Computation and Control*, HSCC '12, page 85–94, New York, NY, USA, 2012. Association for Computing Machinery. pages 1, 66

[8] Kai Weng Wong, Rüdiger Ehlers, and Hadas Kress-Gazit. Correct High-level Robot Behavior in Environments with Unexpected Events. In *Robotics: Science and Systems*, 2014. pages 1, 22, 66, 67

[9] Rüdiger Ehlers and Ufuk Topcu. Resilience to Intermittent Assumption Violations in Reactive Synthesis. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, page 203–212, New York, NY, USA, 2014. Association for Computing Machinery. pages 1, 67

[10] Sebastian Zudaire, Felipe Gorostiaga, Cesar Sanchez, Gerardo Schneider, and Sebastian Uchitel. Assumption Monitoring Using Runtime Verification for UAV Temporal Task Plan Executions. [Under review]. pages 1, 67

[11] Domenico Corapi. *Nonmonotonic Inductive Logic Programming as Abductive Search*. PhD thesis, Imperial College London, https://doi.org/10.25560/9814, 2011. pages 2, 11, 12, 13, 14, 15, 21, 22, 23, 37, 63

[12] Duangtida Athakravi, Dalal Alrajeh, Krysia Broda, Alessandra Russo, and Ken Satoh. Inductive Learning Using Constraint-Driven Bias. In Jesse Davis and Jan Ramon, editors, *Inductive Logic Programming*, pages 16–32, Cham, 2015. Springer International Publishing. pages 2, 11, 13, 15, 16, 17, 21, 38, 63, 64

[13] Davide G. Cavezza and Dalal Alrajeh. Interpolation-Based GR(1) Assumptions Refinement. *Tools and Algorithms for the Construction and Analysis of Systems*, 10205:281–297, 2017. pages 3, 5, 7, 8, 25, 26, 33

[14] Davide Giacomo Cavezza, Dalal Alrajeh, and András György. A Weakness Measure for GR(1) Formulae. *Lecture Notes in Computer Science*, page 110–128, 2018. pages 3, 9, 10, 46

[15] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining Assumptions for Synthesis. In *Proceedings of the 9th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, MEMOCODE '11, page 43–50, USA, 2011. IEEE Computer Society. pages 3, 8

[16] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY – a New Requirements Analysis Tool with Synthesis. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, page 425–429, Berlin, Heidelberg, 2010. Springer-Verlag. pages 3, 34, 49

[17] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. pages 152 – 159, December 2009. pages 3, 8, 33

[18] Imperial College. Guide to MSc Individual Projects. `https://www.doc.ic.ac.uk/lab/msc-projects/ProjectsGuide.html#projas`. pages 4

[19] Sebastián Zudaire, Leandro Nahabedian, and Sebastián Uchitel. Assured UAV Mission Adaptation using Dynamic Update of GR(1) Discrete Event Controllers. *SEAMS '20: Proceedings of the 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, [Under review], 2020. pages 4

[20] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society. pages 5

[21] Giuseppe De Giacomo and Moshe Y. Vardi. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, page 854–860. AAAI Press, 2013. pages 6

[22] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. *Journal of Computer and System Sciences*, 78:911–938, 2012. pages 7

[23] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–484, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. pages 8

[24] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. Symbolic Repairs for GR(1) Specifications. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1016–1026. IEEE Press, 2019. pages 8

[25] R. Alur, S. Moarref, and U. Topcu. Counter-Strategy Guided Refinement of GR(1) Temporal Logic Specifications. In *2013 Formal Methods in Computer-Aided Design*, pages 26–33, 2013. pages 8, 9

[26] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 501–516. Springer, 2015. pages 8

[27] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment Assumptions for Synthesis. *Lecture Notes in Computer Science*, page 147–161, 2008. pages 9

[28] S. A. Seshia. Combining Induction, Deduction, and Structure for Verification and Synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015. pages 9

[29] Ludwig Staiger. On the Hausdorff measure of regular ω-languages in Cantor space. *Discrete Mathematics Theoretical Computer Science*, 17:357–368, May 2015. pages 10

[30] Dalal Alrajeh, Antoine Cailliau, and Axel van Lamsweerde. Adapting Requirements Models to Varying Environments. 42nd International Conference on Software Engineering, 2020. pages 10, 21, 37, 38, 40, 44, 63, 64

[31] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.*, pages 493–498, 2004. pages 10

[32] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems, pages 327–341, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. pages 10

[33] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *The 4th IEEE International Symposium on High-Assurance Systems Engineering*, HASE '99, page 239–248, USA, 1999. IEEE Computer Society. pages 10

[34] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg, 1984. pages 11

[35] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991. pages 11

[36] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994. pages 11

[37] Chiaki Sakama. Nonmonotomic Inductive Logic Programming. In Thomas Eiter, Wolfgang Faber, and Miros law Truszczyński, editors, *Logic Programming and Nonmotonic Reasoning*, pages 62–80, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. pages 11

[38] Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. Learning Through Hypothesis Refinement Using Answer Set Programming. In Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors, *Inductive Logic Programming*, pages 31–46, Berlin, Heidelberg, September 2014. Springer Berlin Heidelberg. pages 11, 12, 13, 14, 15, 37

[39] Timothy Kimber. *Learning Definite and Normal Logic Programs by Induction on Failure*. PhD thesis, Imperial College London, https://doi.org/10.25560/9961, 2011. pages 11

[40] Chiaki Sakama and Katsumi Inoue. Brave induction: A logical framework for learning from incomplete information. *Machine Learning*, 76:3–35, July 2009. pages 12

[41] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive Logic Programming as Abductive Search. In Manuel Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54–63, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. pages 12

[42] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992. pages 12, 16

[43] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1594–1597. AAAI Press, 2008. pages 12

[44] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive Logic Programming in Answer Set Programming. In Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *Inductive Logic Programming*, pages 91–97, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. pages 12, 13

[45] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011. pages 13, 37

[46] S. Wrobel. First Order Theory Refinement. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 14–33. IOS Press, 1996. pages 13, 14

[47] James Wogulis and Michael J. Pazzani. A Methodology for Evaluating Theory Revision Systems: Results with Audrey II. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 1128–1134. Morgan Kaufmann, 1993. pages 13, 14

[48] Antonis Kakas, A. Michael, and Costas Mourlas. Aclp: Abductive constraint logic programming. *The Journal of Logic Programming*, 44:129–177, July 2000. pages 16

[49] Mance E Harmon and Stephanie S Harmon. Reinforcement learning: A tutorial. Technical report, Wright Lab Wright-Patterson AFB OH, 1997. pages 17, 18

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, February 2015. pages 18

[51] Ankit Choudhary. A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python. `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`. pages 18

[52] David O. Aihe and A. Gonzalez. Context Driven Reinforcement Learning. In *Swedish American Workshop on Modeling and Simulation*, 2003. pages 19

[53] Domenico Corapi, Alessandra Russo, Marina De Vos, Julian Padget, and Ken Satoh. Normative design using inductive learning. *Theory and Practice of Logic Programming - TPLP*, 11, July 2011. pages 21

[54] Shahar Maoz and Jan Oliver Ringert. Spectra: A Specification Language for Reactive Systems. *ArXiv*, abs/1904.06668, April 2019. pages 37, 48, 49

[55] Dalal Alrajeh, Jeff Kramer, Axel van Lamsweerde, Alessandra Russo, and Sebastián Uchitel. Generating Obstacle Conditions for Requirements Completeness. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 705–715. IEEE Press, 2012. pages 38

[56] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, January 1985. pages 38

[57] Sergio Guadarrama and Anoop Korattikara and Oscar Ramirez and Pablo Castro and Ethan Holly and Sam Fishman and Ke Wang and Ekaterina Gonina and Neal Wu and Efi Kokiopoulou and Luciano Sbaiz and Jamie Smith and Gábor Bartók and Jesse Berent and Chris Harris and Vincent Vanhoucke and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. [Online; accessed 25-June-2019]. pages 43, 45

[58] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org. pages 43

[59] Thomas Simonini. On Choosing a Deep Reinforcement Learning Library. `https://blog.dataiku.com/on-choosing-a-deep-reinforcement-learning-library`. pages 43

[60] Syntech. Spectra Examples. `http://smlab.cs.tau.ac.il/syntech/examples/index.html`. pages 49

[61] Syntech. basic_past_ForkLiftSpec.spectra. `https://github.com/SpectraSynthesizer/spectra-specs/blob/master/forklift/01.basic_past_ForkLiftSpec.spectra`. pages 51

[62] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley Publishing, 1st edition, 2009. pages 63

[63] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Elaborating Requirements Using Model Checking and Inductive Learning. *IEEE Transactions on Software Engineering*, 39(3):361–383, 2013. pages 63

[64] K. W. Wong, R. Ehlers, and H. Kress-Gazit. Resilient, Provably-Correct, and High-Level Robot Behaviors. *IEEE Transactions on Robotics*, 34(4):936–952, 2018. pages 66

[65] Sumanth Dathathri, Scott C. Livingston, and Richard M. Murray. Enhancing Tolerance to Unexpected Jumps in GR(1) Games. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, page 37–47, New York, NY, USA, 2017. Association for Computing Machinery. pages 66

[66] Daniel Neider and Oliver Markgraf. Learning-Based Synthesis of Safety Controllers. *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 120–128, October 2019. pages 67

[67] Ankush Desai, Shromona Ghosh, Sanjit A. Seshia, Natarajan Shankar, and Ashish Tiwari. SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 138–150, 2019. pages 67

[68] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. Combining Model Checking and Runtime Verification for Safe Robotics. *Runtime Verification*, 10548:172–189, 2017. pages 67

[69] Nicolas D'Ippolito, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. Hope for the Best, Prepare for the Worst: Multi-Tier Control for Adaptive Systems. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 688–699, New York, NY, USA, 2014. Association for Computing Machinery. pages 67

[70] Moeka Tanabe, Kenji Tei, Yoshiaki Fukazawa, and Shinichi Honiden. Learning Environment Model at Runtime for Self-Adaptive Systems. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1198–1204, New York, NY, USA, 2017. Association for Computing Machinery. pages 68

[71] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. Learning revised models for planning in adaptive systems. *Proceedings - International Conference on Software Engineering*, pages 63–71, May 2013. pages 68

[72] M Keegan, V. Braberman, N. D'Ippolito, N. Piterman, and S. Uchitel. Control and Discovery of Reactive System Environments. [Under review]. pages 68

[73] 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*, pages 5177–5182, 2012. pages 68

[74] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari. Adaptive Action Selection in Autonomic Software Using Reinforcement Learning. *4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 175–181, 2008. pages 68, 69

[75] X. Bu, J. Rao, and C. Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Transactions on Parallel and Distributed Systems*, 24(4):681–690, 2013. pages 68

[76] R. M. Bahati and M. A. Bauer. Adapting to Run-Time Changes in Policies Driving Autonomic Management. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 88–93, 2008. pages 68

[77] Yingcheng Sun, Xiaoshu Cai, and Kenneth Loparo. Learning-based Adaptation Framework for Elastic Software Systems. In *SEKE*, July 2019. pages 68

[78] Dongsun Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 76–85, 2009. pages 68

[79] Dinh-Khanh Ho, Karim Ben Chehida, Benoit Miramond, and Michel Auguin. Learning-Based Adaptive Management of QoS and Energy for Mobile Robotic Missions. *International Journal of Semantic Computing*, 13(04):513–539, 2019. pages 68

[80] T. Zhao, W. Zhang, H. Zhao, and Z. Jin. A Reinforcement Learning-Based Framework for the Generation and Evolution of Adaptation Rules. *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 103–112, 2017. pages 68

[81] Andreas Metzger, C. D. Quinton, Zolt'an 'Ad'am Mann, Luciano Baresi, and Klaus Pohl. Feature-Model-Guided Online Learning for Self-Adaptive Systems. *ArXiv*, abs/1907.09158, 2019. pages 68

[82] Han Nguyen Ho and Eunseok Lee. Model-Based Reinforcement Learning Approach for Planning in Self-Adaptive Software System. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, IMCOM '15, New York, NY, USA, 2015. Association for Computing Machinery. pages 68

[83] M. Hasanbeig, Y. Kantaros, A. Abate, D. Kroening, G. J. Pappas, and I. Lee. Reinforcement Learning for Temporal Logic Control Synthesis with Probabilistic Satisfaction Guarantees. *2019 IEEE 58th Conference on Decision and Control (CDC)*, December 2019. pages 69

[84] Takumi Akazaki, Shuang Liu, Yoriyuki Yamagata, Yihai Duan, and Jianye Hao. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. *Lecture Notes in Computer Science*, 10951:456–465, 2018. pages 69

[85] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. Runtime verification of p4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI ML*, NetAI'19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery. pages 69

[86] Gil Lederman, Markus Rabe, Sanjit Seshia, and Edward A. Lee. Learning Heuristics for Quantified Boolean Formulas through Reinforcement Learning. In *International Conference on Learning Representations*, [Under review], 2020. pages 69