Semantic Labelling and Learning for Parity Game Solving in LTL Synthesis*

Jan Křetínský[0000−0002−8122−2881], Alexander Manta[0000−0003−0591−6985], and Tobias Meggendorfer[0000−0002−1712−2165]

Technical University of Munich

Abstract. We propose "semantic labelling" as a novel ingredient for solving games in the context of LTL synthesis. It exploits recent advances in the automata-based approach, yielding more information for each state of the generated parity game than the game graph can capture. We utilize this extra information to improve standard approaches as follows. (i) Compared to strategy improvement (SI) with random initial strategy, a more informed initialization often yields a winning strategy directly without any computation. (ii) This initialization makes SI also yield smaller solutions. (iii) While Q-learning on the game graph turns out not too efficient, Q-learning with the semantic information becomes competitive to SI. Since already the simplest heuristics achieve significant improvements the experimental results demonstrate the utility of semantic labelling. This extra information opens the door to more advanced learning approaches both for initialization and improvement of strategies.

1 Introduction

Reactive synthesis is a classical problem to find a strategy that given a stream of inputs gradually produces a stream of outputs so that a given specification over the inputs and outputs is satisfied. In LTL synthesis the specification is given as a formula of *linear temporal logic (LTL)*. The classical solution technique is the *automata-theoretic approach* [30] that transforms the specification into an automaton. The partitioning of atomic propositions into inputs and outputs then yields a game over this automaton. Subsequently, the game is solved and the winning strategy in the game induces a winning strategy for the original problem. The standard type of automaton to be used in this context is the *deterministic parity automaton (DPA)* since (i) determinism ensures we obtain a well-defined game and (ii) the parity condition yields a *parity game*, which can be solved reasonably cheaply in practice [32,25,8] with good tool support [9,29].

While solving large games still takes significant resources, the bottleneck of this procedure is already the construction of the game. Indeed, after transforming the LTL formula into a non-deterministic automaton [30] this automaton is

* This research was funded in part by the Czech Science Foundation grant No. P202/12/G061, and the German Research Foundation (DFG) projects KR 4890/1-1 "Verified Model Checkers" and KR 4890/2-1 "Statistical Unbounded Verification".

determinized using *Safra's construction* [24] or its improvements [22,26]. This is notoriously known to be practically inefficient [17] despite some tool support [15]. As a result, other approaches for the synthesis problem have been suggested that avoid Safra's determinization, see the related work below. However, recent translators, e.g. [4,16], yield significantly smaller automata to the extent that the traditional parity-game approach, e.g. [20,19], becomes competitive [10,1]. Apart from the smaller sizes of automata, one of their decisive advancements is the ability to generate the automaton on the fly and terminate as soon as a winning strategy is found, possibly way earlier than the whole automaton is constructed.

Yet these approaches suffer from several inefficiencies. In order to tackle them let us observe their roots. Firstly, despite the relative efficiency, solving the parity game can still take significant time. Either the whole game is solved, e.g. using Zielonka's algorithm [32] as in [20], or growing on-the-fly explored parts are repetitively solved, e.g. using strategy improvement as in [19]. In both cases, large parts of the state space are processed and the overall effort is still significant since the strategy improvement is executed many times during the process. Secondly, in the case with on-the-fly exploration, it is not clear in which direction the game should be explored. In the graph game, the available extra information is only the priorities and computing their attractors is a global computation defeating the purpose of on-the-fly exploration.

In this paper, we suggest a framework for a theoretically fundamental improvement of solving parity games that arise from LTL synthesis and we instantiate it with the first simple heuristics. The experimental results confirm the potential of this approach. The main idea is to exploit, to our best knowledge for the first time, the *semantics* of the vertices of the game.

Where does the semantics come from? Since the original specification is translated to an automaton, its states have a strong correspondence to the monitored property. However, Safra's determinization and the subsequent latest appearance record for obtaining a DPA leaves us with permutation over Safra's trees over sets of LTL formulae, whose semantics is extremely hard to decipher. In contrast, the new approach of [5,16] yields a description of each state in terms of a single formula to be satisfied and a list of formulae describing progress of satisfying each sub-goal. This clearer structure allows us to exploit the meaning of available successors and to choose the most promising one in the sense of satisfying the goal of each player. This addresses issue of exploration guidance. The other issue of updating the whole or whole explored part of the state space can be addressed using reinforcement learning [28]. Since the degree how promising a vertex is can be quantified, we can use it as a reward, together with the priorities, in Q-learning. This way we update only the most promising parts of the state space.

Our contribution is the following:

- We introduce a semantics-based framework for heuristics for parity games in LTL synthesis and instantiate it as follows.
- We utilize the semantic labelling of vertices to get a better initial strategy for the parity game, often yielding (i) an optimal solution directly and (ii) a smaller one.

– We show how reinforcement learning can be applied to parity games and accelerated by the semantic labelling.
– We demonstrate the potential of this approach experimentally on formulae from the SYNTCOMP competition [10] as well as random formulae, opening a door for learning-based approaches to automata-based LTL synthesis.

Related Work

Firstly, one can reduce the synthesis problem to emptiness of nondeterministic Büchi tree automata [18]; it has been implemented with considerable success in [11]. The second approach is to use heuristic to improve Safra's determinization procedure [14,15]. The third approach is to consider fragments of LTL. For instance, the generalized reactivity(1) fragment of LTL (called GR(1)) was introduced in [23] and a cubic time symbolic representation of an equivalent automaton was presented. The approach has been implemented in the ANZU tool [12]. Another approach, prominent in competitions like SYNTCOMP [10], is bounded synthesis [27], as implemented by, e.g., BoSy [7] and PARTY [13]. The tool Acacia+ [2] uses symbolic incremental algorithms based on antichains.

Besides, there are learning approaches that utilize some information on the state space. However, this is typically not the information on the property currently to be satisfied, e.g. [21] uses automata learning, but only for safety properties and not focusing on the property itself. Further, [3] takes the property into account, but only as its respective automaton. It tries to decrease the distance to the accepting vertex, which can be very different from making the property easier to satisfy. Moreover, it is not designed for games, although the alternating distance might address this drawback. More importantly, it is not suited for partial models as we need to construct the whole automaton, which is the bottleneck for complex properties.

2 Preliminaries

In this section, we give some basic background knowledge and establish fundamental notation. Due to space constraints, we touch only briefly on several topics and encourage the reader to refer to the mentioned literature.

Basic Notation. We use $\mathbb{N}$ to denote the set of non-negative integers. Given a propositional formula ϕ over a set of propositions AP, we use $\mathrm{sat}(\phi) = \{v \in 2^{\mathsf{AP}} \mid v \models \phi\}$ to denote the set of all satisfying assignments. The constants `tt` and `ff` denote *true* and *false*, respectively.

2.1 Synthesis & Games

The synthesis problem in its general form asks whether a system can be controlled in a way such that it satisfies a given specification under any (possible) environment. Moreover, one often is interested in obtaining a witness to this query, i.e.

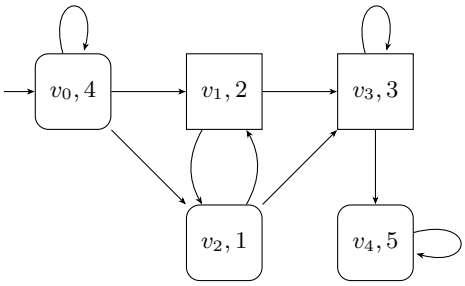


Fig. 1: An example (parity) game. Rounded rectangles belong to the system player and normal rectangles to the environment player. The vertices are additionally labelled with their priorities. For readability, we omit the requirement of alternation.

some *controller* or *strategy* which specifies the system's actions. For example, one might ask whether a robot can be steered over difficult terrain such that it arrives at a particular target location.

Graph games are a standard formalism used in synthesis. A game, denoted by $\mathcal{G} = ((V, E), v_0, P, \textsc{Win})$, consists of a digraph (V, E), a *starting vertex* $v_0 \in V$, a *player mapping* P, and a *winning condition* Win, described later. Each vertex belongs to one of the two players 0 (called *system*) and 1 (called *environment*), specified by the mapping $P : V \to \{0, 1\}$. In other words, the set of vertices is partitioned into player 0's vertices V_0 and player 1's vertices V_1; $V = V_0 \dot\cup V_1$. See Fig. 1 for an example of such a game. For ease of notation, we write $vE := \{(v, u) \in E \mid u \in V\}$ to denote all outgoing edges of some vertex v and define $E_i := \{(u, v) \in E \mid u \in V_i\}$ the set of all edges "controlled" by player i.

To play the game, a token is placed in the initial vertex v_0. Then, the player owning the token's current vertex moves the token along an outgoing edge of the current vertex. This is repeated infinitely, giving rise to an infinite sequence of vertices containing the token $\rho = v_0 v_1 v_2 \cdots \in V^\omega$, called a *play*. The set of all possible plays is denoted by $\mathfrak{P}$.

For simplicity, we assume in the following that all games are *alternating*, i.e. the successors of a vertex belong to a different player than the vertex itself.

Winning conditions are a mapping from plays to the winning player $\textsc{Win} : \mathfrak{P} \to \{0, 1\}$. Numerous kinds of winning conditions have been studied. In this work, we consider the following three:

Safety is defined by a set of vertices T to be avoided. The system player loses iff one of the vertices in the given set is visited.

Co-Safety (or *reachability*) is, as the name suggests, dual to safety. Here, the system player wins iff one of the given vertices is visited at least once. Observe that this exactly corresponds to a safety objective for the environment player.

Parity is based on a *priority assignment* for each vertex $\mathfrak{p} : V \to \mathbb{N}$. The system player wins iff the *maximum* of all infinitely often occurring priorities is *odd*.[1] Formally, we define $\inf(\rho) = \{p \mid \forall j.\exists k \geq j.\mathfrak{p}(\rho_k) = p\}$ and system wins a play ρ iff $\max \inf(\rho)$ is odd. We refer to odd priorities as *good* (for the system player) and to even priorities as *bad* (for the system player).

Note that both safety and co-safety are special cases of the parity condition, with a straightforward linear time transformation.

Strategies are mappings $\sigma_i : V_i \to E$, assigning to each of the player's vertices an edge along which the token will be moved.[2] Observe that once both players fix a strategy, the game is fully determined and a unique run is induced. This means that given a game with a particular winning condition and a strategy for each player, we can decide which of the players wins the game using these strategies. A strategy of a player is called *winning* if the player wins for *any* strategy of the opponent. Thus, we can rephrase the synthesis question to "Is there a winning strategy for the system player?".

For example, consider again the game depicted in Fig. 1. Fixing the strategies $\sigma_0 = \{v_0 \mapsto (v_0, v_2), v_2 \mapsto (v_2, v_3), v_4 \mapsto (v_4, v_4)\}$ and $\sigma_1 = \{v_1 \mapsto (v_1, v_2), v_3 \mapsto (v_3, v_3)\}$ induces the play $v_0 v_2 v_3 v_3 \cdots$. The set of infinitely often seen priorities equals $\{3\}$, hence the system player wins with these strategies. Moreover, the strategy σ_0 is winning, since the play always ends up in either v_3 or v_4.

Strategy Improvement (or *strategy iteration*) is the most prominent way of solving parity games, i.e. answering the above question. In recent times, it received significant attention due to both theoretical and practical advances. We explain the approach only very briefly, since its details are not important for this work.

In essence, strategy improvement works as follows. First, arbitrary initial strategies are picked for both players. Then, the algorithm checks whether one of the current strategies is winning. If yes, this strategy is returned. Otherwise, the algorithm tries to improve one of the strategies by changing its choices in some vertices. If an improvement is not possible, there exists no winning strategy for the respective player. Otherwise, the process is repeated with the new strategy.

It is known that this algorithm converges to the correct result in finite time for any initial strategy. This gives us a straightforward way of optimization, namely the choice of the initial strategy. Intuitively, a heuristic which often comes up with a "good" strategy may improve the runtime significantly over arbitrary or random choice, since then only a few improvement steps are necessary.

Throughout this work, we refer to a reference implementation of SI, denoted `SI`, e.g., when running SI with a particular initial strategy. In our implementation, we used the algorithm of [31], but other variants could be substituted.

[1] Instead of the maximum, one could also decide based on the minimum; similarly instead of "odd", "even" sometimes is considered winning for the system.

[2] Strategies may be significantly more complex, e.g., by using memory. Since "positional" strategies are sufficient for all properties we consider, we intentionally omit the general definition in the interest of space.

2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is a standard logics used in verification and synthesis to specify the desired behaviour of a system. The logic is given by the syntax

$$\phi ::= \mathtt{ff} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi,$$

where $a \in \mathsf{AP}$ is an *atomic proposition*, inducing the *alphabet* $\Sigma = 2^{\mathsf{AP}}$. LTL formulae are interpreted over infinite sequences $w \in \Sigma^\omega$ called ω-words. Intuitively, a word $w = w_0 w_1 \cdots \in \Sigma^\omega$ satisfies the *next* $\mathbf{X}\,\phi$ iff ϕ is satisfied in the next step. The *until* operator $\phi\,\mathbf{U}\,\psi$ is satisfied iff ϕ holds until ψ is satisfied. Apart from the mentioned operators, we also consider *finally* $\mathbf{F}\,\phi := \mathtt{tt}\,\mathbf{U}\,\phi$ and *globally* $\mathbf{G}\,\phi := \neg\,\mathbf{F}\,\neg\phi$, which require that ϕ holds at least once or always, respectively.

Given an LTL formula ϕ, the set of its *sub-formulae* is denoted by $\mathrm{sub}(\phi)$. The *top-level temporal operators* $\mathrm{top}(\phi)$ are all temporal operators not nested inside other temporal operators. For example, the formula $\phi = \mathbf{G}((\mathbf{F}\,a) \wedge b) \wedge \mathbf{F}\,b$ has sub-formulae $\mathrm{sub}(\phi) = \{a, b, \mathbf{F}\,a, (\mathbf{F}\,a) \wedge b, \mathbf{G}((\mathbf{F}\,a) \wedge b), \mathbf{F}\,b, \phi\}$ and top-level operators $\mathrm{top}(\phi) = \{\mathbf{G}((\mathbf{F}\,a) \wedge b), \mathbf{F}\,b\}$.

LTL Synthesis is an instance of the general synthesis problem. Here, the specification to be satisfied by the system is given in form of an LTL formula. Due to recent advances [16,19], the *automata-based approach* [30] to LTL synthesis received significant attention. Essentially, the given LTL formula is translated into an ω-automaton, which in turn is transformed into a parity game. By solving the resulting game, we obtain a solution to the original synthesis question.

Technically, the game is obtained by "splitting". To this end, the set of atomic propositions is split into system- and environment-controlled propositions. Then, the players' actions correspond to choosing which of their propositions to enable. Once both players chose their propositions' values, the automaton moves to the next vertex according to the chosen valuation. See, e.g., [19], for more detail.

Semantic translations from LTL to automata are the key ingredient to our new approach. These translations not only produce a parity game, but also provide a semantic labelling of the game's vertices. In particular, using the approach introduced in [5] and implemented in [16], we obtain for each vertex a list of LTL formulae, roughly corresponding to (sub-)goals which still have to be (possibly repetitively) fulfilled. Due to space constraints, we describe the ideas of these constructions only briefly in Section 3.1. This labelling is not easily derived from the structure of the game graph and provides additional information not accessible to conventional, general-purpose parity game solvers. The primary goal of this paper is to show that this additional information can be exploited for a significant increase in performance.

2.3 Q-Learning

Q-Learning is a well known, simple yet versatile *reinforcement learning* technique [28]. It usually is applied in machine learning to find performant strategies for

(stochastic) systems. The technique roughly works as follows. Each edge (u, v) has a *Q-value* $Q(u, v)$, indicating the *quality* assigned to the respective edge. This value is initialized according to some heuristic and then repeatedly updated through *learning episodes*. Each episode consists of sampling a path through the system, following the maximal Q-value. In order to encourage exploration, randomization is added to this choice. While sampling, the learning agent receives rewards based on his choices. The Q-value of the respective edge is then updated with the obtained reward and the Q-value of it's successor.[3] To smoothen the learning process, the propagated value is weighted by a *learning rate* α. Together, the update essentially is computed by $Q(v, u) \leftarrow (1-\alpha) \cdot Q(v, u) + \alpha \cdot (\mathcal{R}(v, u) + Q(u))$ where v and u are the current and next vertex, respectively, $\mathcal{R}(v, u)$ is the obtained reward, and $Q(u) = \max_{(u,u') \in E} Q(u, u')$ is the Q-value of the successor vertex u.

3 Our Contributions

In this section, we explain the central ideas of our contributions. First, we highlight the peculiarities of the mentioned labelling function. Then, we introduce the concept of *trueness*, which we directly use to augment strategy improvement. Finally, we explain our adaptation of Q-learning to parity games and how we derive a semantic reward signal from the labelling, using trueness.

3.1 Input Details

We assume that we are given a parity game, where each vertex is labelled by a structured list of LTL formulae. The labelling corresponds to the remaining goals to be achieved by the system player (or violated by the environment player). More precisely, the labelling consists of one *master formula* and potentially several *monitors*. The master formula tracks the "overall progress" and all finitely achievable parts of the formula. In particular, the master formula `tt` corresponds to the formula being satisfied by the prefix, analogously `ff` corresponds to a falsified formula. The system player automatically wins if a vertex labelled with `tt` is reached and, similarly, loses on a `ff` vertex. In the special case of reachability or safety specifications, the labelling actually only consists of the master formula.

For more complex specifications, the labelling also exhibits a more intricate structure. Intuitively, there is one monitor for each sub-formula which needs to be satisfied infinitely often (liveness conditions) or may only be violated finitely often (safety conditions). Each monitor tracks a list of formulae which have to be fulfilled in order to satisfy its overall goal. The monitors are ordered according to an appearance-record style construction. "Failing" monitors emit a bad priority and are moved at the beginning of the list, succeeding monitors instead emit a good priority. Both priorities are based on the respective monitor's position in the list. Intuitively, for a fixed word ω, all monitors which only fail finitely often

[3] The exact details of this update vary between different instantiations of Q-learning. For example, a discount factor may be included.

eventually are ranked higher than all the monitors which fail infinitely often. Thus, if such a non-failing monitor emits a good priority, it overrules all of the failing monitors' bad priorities.

Consider, for example, the formula $\mathbf{F}\,\mathbf{G}\,a$, meaning "eventually, a appears every step". Here, there is no ultimate `tt` or `ff` vertex, since this formula cannot be satisfied or violated by any finite prefix. The construction gives us $\mathbf{F}\,\mathbf{G}\,a$ as master formula and $\mathbf{G}\,a$ as a monitored goal. Whenever we see a $\neg a$, this monitor would fail, emit a bad priority, and move to the front of the list. Dually, for every a, it emits a good priority.

The details of this construction are described in [6]. It is implemented in the tool `Rabinizer` [16] which we use for our constructions. An online demo thereof is located at `https://owl.model.in.tum.de/try/`. A simplified example can be found in Fig. 3 later on.

3.2 (Co-)Safety games

Recall that for these games, the labelling we obtain is a single LTL formula per vertex. The system player wants to reach the `tt` vertex and avoid the `ff` vertex. Consequently, the system player naturally is interested in taking "trueness-maximizing" edges, analogously the environment player wants to move away from `tt`. This simple observation directly leads us to the concept of *trueness*.

Trueness of an LTL formula $t : \mathrm{LTL} \to [0, 1]$ intuitively denotes how "close" a given formula is to being satisfied. We compute this value by treating the formula as purely propositional, i.e. each temporal operator is considered to be a fresh variable. Formally, given an LTL formula ϕ, we interpret it as a propositional formula over $\Sigma = 2^{\mathsf{AP} \cup \mathrm{top}(\phi)}$. For this formula, we then determine and scale the ratio of satisfying assignments, i.e. $t(\phi) := |\mathrm{sat}(\phi)|/|\Sigma|$. This value can be computed efficiently by representing the formula as *binary decision diagram (BDD)*, as implemented in `Rabinizer`. Even for formulae with several hundred syntax elements the trueness is computed virtually instantaneously.

At first, this notion may seem rather unintuitive, since the temporal aspect of a formula is not necessarily reflected by the trueness value. For example, the formula $\phi = \mathbf{G}\,a \wedge \mathbf{G}\,\neg a$ has a trueness value of $t(\phi) = \frac{1}{4}$, but actually is unsatisfiable. Nevertheless, trueness proves to be a surprisingly good initialization heuristic, which we explain in the following and further demonstrate in our evaluation.

One particular reason for its performance in our application is due to the way the labelling is constructed. In particular, temporal operators are "unfolded" as an essential step of the construction. For example, the formula $\mathbf{G}\,a$ is unfolded to $a \wedge \mathbf{G}\,a$ while $\mathbf{F}\,a$ yields $a \vee \mathbf{F}\,a$ and $a\,\mathbf{U}\,b$ gives $b \vee (a \wedge (a\,\mathbf{U}\,b))$. The unfolded variants are semantically equivalent to the original formula, but provide us with a one-step propositional "approximation" of the temporal operators. The above ϕ then is unfolded to $\phi \equiv (a \wedge \mathbf{G}\,a) \wedge (\neg a \wedge \mathbf{G}\,\neg a) \equiv \mathtt{ff}$.

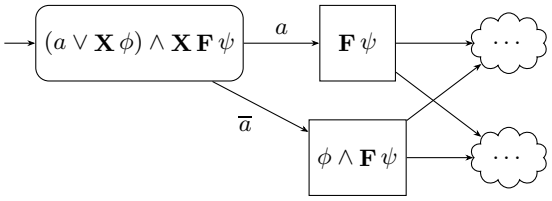


Fig. 2: An example showing the application of trueness initialization. For readability, we only show the vertex labels. ϕ and ψ are some non-trivial LTL formulae.

Initializing strategies based on trueness can be achieved as follows. Fix some game $\mathcal{G}$ with a (co-)safety objective and labelling function $L : V \to \text{LTL}$. The strategies σ_0 and σ_1 are called *trueness-optimal* if they satisfy

$$\sigma_0(v_0) \in \operatorname*{arg\,max}_{(v_0,u)\in E} t(L(u)) \qquad\qquad \sigma_1(v_1) \in \operatorname*{arg\,min}_{(v_1,u)\in E} t(L(u))$$

for all vertices $v_0 \in V_0$ and $v_1 \in V_1$, respectively. Observe that, since we assumed the game to be alternating, all u vertices in the above equations belong to the respective opponent.

This immediately yields our first semantic algorithm $\texttt{SI}_{sem}$, which runs SI initialized with trueness optimal strategies.

For a small example, consider the simplified part of a game depicted in Fig. 2. Here, the system player can choose whether or not to play a in the initial vertex. The successors' trueness value, namely $t(\mathbf{F}\,\psi) = \frac{1}{2}$ and $t(\phi \wedge \mathbf{F}\,\psi) \leq \frac{1}{4}$ (for a non-trivial ϕ), suggest the natural choice of a, leading to $\mathbf{F}\,\psi$. Intuitively, $\mathbf{F}\,\psi$ is "easier" to satisfy than $\phi \wedge \mathbf{F}\,\psi$. Observe that without the labelling and its trueness value this choice would not be as obvious, since the impact of this decision may only become visible much later in the game. Quite surprisingly, this initialization solves a *majority* of randomly generated games *instantly* without the need for any further improvements step, as shown in the experimental evaluation.

3.3 Parity games

To apply our ideas to parity games, there are several hurdles to overcome. Recall that the labelling we obtain for these games has a non-trivial structure, compared to the singleton labelling in the special cases. Hence, we cannot use the trueness value directly. Rather, we need a more intricate way of deriving meaning from the labelling. Because of its simplicity, we decided to use Q-learning. Recall that Q-learning usually works with a single agent, interested in maximizing the obtained reward. In our case, we instead have two antagonistic agents and we need to adapt the Q-learning framework to this setting. Furthermore, there are some technical peculiarities when we want to incorporate priorities and the labelling.

Remark 1. We again stress that our main goal is to show the usefulness of the so far unexploited semantic labelling. In particular, the exact approach to extracting

rewards or even the fact that we use Q-learning is of secondary importance for our research goal. Using more advanced techniques, e.g., extract reward from the formula using neural networks, is left for future work.

Basic concepts For all variants, our Q-values lie between -1, corresponding to a presumably guaranteed loss for the system player, and $+1$, analogously corresponding to a win. As usual in Q-learning, we repeatedly sample paths and update vertex values. But, since the two players are antagonistic, we don't always pick a maximizing action while sampling. Instead, we choose a successor with maximal Q-value in system vertices and a minimal one in environment vertices. Once we encounter a vertex for a second time and consequently would enter a loop, we stop the sampling.

In the following we present three variants of Q-learning. The first, agnostic variant obtains rewards only based on whether an episode is winning or losing. This approach is widely applicable, since basically no domain knowledge is necessary. Not surprisingly, it also is not too efficient. We then present a first adaption, which additionally incorporates priorities. This variant is tailored towards parity games, but does not employ any semantic information provided by the labelling. Our experiments show that it outperforms the first variant, but only by a slim margin. Finally, we present our semantic approach, which on top also considers the labelling, employing the trueness function in several ways. In our experiments, this significantly outperforms the first two ideas and, on some datasets, even beats strategy improvement by a large factor. Note that each variant also incorporates the reward signals of the previous approaches.

Rewards based on winning paths are binary, yielding $+1$ for winning and -1 for losing. In our case, this means that when we stop sampling after entering a loop, we determine whether the loop is winning or losing and propagate the value accordingly. We initialize the Q-values with 0, since there is no a-priori information available. The resulting Q-learning variant is denoted by $\mathtt{QL}_{win}$.

Priority rewards are the first step to a more intricate reward signal. Here, the agent additionally obtains intermediate rewards based on the priority of the edge. Recall that these priorities are natural numbers, and the system player is interested in large, odd priorities. Dually, large even priorities should be avoided.

There are two difficulties associated with this idea. Since we assumed our Q-values to lie between -1 and $+1$, we need to rescale the priorities into this domain. Furthermore, we need to rescale them such that larger priorities significantly "overrule" smaller ones, reflecting the nature of the parity objective. For example, obtaining ten 5 priorities in a row is irrelevant if afterwards one 6 is encountered.

We approach this problem by rescaling the priorities as follows. Let p_i be the priorities occurring in the given game, sorted in ascending order. We first compute the absolute frequency of each priority as $f(p_i) = |\{s \mid \mathfrak{p}(s) = p_i\}|$, i.e. the number

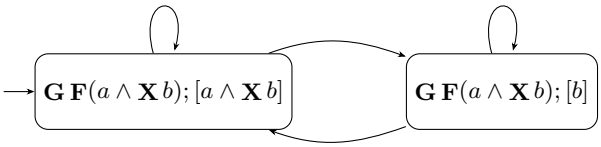


Fig. 3: A simplified example for a monitor labelling as produced by `Rabinizer`. The first formula represents the master formula, while the second one corresponds to the only monitor's formula.

of vertices in the whole game labelled with priority p_i. Then, we define the scaled priorities $\overline{p}_i$ by $\overline{p}_0 = p_0$, $\overline{p}_i = 2 \cdot f(p_{i-1}) \cdot \overline{p}_{i-1} + 1$. Observe that $\overline{p}_i$ is larger than the sum of all $\overline{p}_j$ with $j < i$ occurring in the game. Finally, we re-normalize $\overline{p}_i$ into the $[-1, +1]$ domain by $r_i := (-1)^{p_i+1}\overline{p}_i(1 + \sum_j \overline{p}_j \cdot f(p_j))^{-1}$ Even priorities are mapped to the negative domain, as the system player wants to avoid them. As before, we initialize the Q-values with 0.

We denote this Q-learning variant by $\mathtt{QL}_{pri}$. Note that this approach is applicable to general parity games. In our case, it indirectly uses the labelling, since the priorities of the game are directly derived from the labelling and correspond to progress in the monitors.

Semantic rewards are our idea for exploiting the information provided by the vertex labelling, denoted by $\mathtt{QL}_{sem}$. Firstly, we describe how we assign the initial Q-values. Recall that we cannot apply the trueness value to the whole labelling, since in general it comprises several different LTL formulae corresponding to different goals. Nevertheless, we can still easily exploit the master formula to obtain a sensible value. In particular, we initialize the Q-value of each action based on the trueness of the master formula in the successor vertex. This directly generalizes the approach of the special case of (co-)safety, where the labelling consists only of the master formula.

Now, we introduce our ideas for deriving the reward signal from the labelling. Recall that apart from the master formula we have several monitor formulae, corresponding to goals which we have to fulfil repeatedly. We present a motivating example in Fig. 3. There, we show a (simplified) labelling produced for the formula $\psi = \mathbf{G}\,\mathbf{F}(a \wedge \mathbf{X}\,b)$. In order to satisfy ψ, we repeatedly need to play a and then in the next step b. The master formula is the same in both vertices, hence we need to analyse the monitors instead. Intuitively, playing a seems to be the more natural choice in the initial vertex, since b is easier to satisfy than $a \wedge \mathbf{X}\,b$. Hence, our main idea is to also apply trueness analysis to the monitor labels.

Consequently, the system player should be interested not only in the ad-hoc reward obtained by following good priorities, but also in "progressing" other monitors. To this end, we also give a reward proportional to the change in trueness for all monitors which are ranked higher than the current priority as follows.

Let $e = (u, v)$ be an edge in some game $\mathcal{G}$ with priority assignment $\mathfrak{p}$ and labelling L. By taking this edge, each monitor updates its monitored formulae. In

particular, some monitor may fail or succeed due to this transition. Let m_e be the index of the highest-ranked of such failing or succeeding monitors in the labelling of vertex u. If there is no such monitor, let $m_e = -1$. Then, all monitors ranked higher than m_e made some progress, but did not fail or succeed. Nevertheless, the system player is interested in succeeding at least one of those monitors and let none of them fail. Hence, we incentivise the system player to take actions which additionally bring monitors closer to succeeding or at least do not worsen their state. As basis for this approximation, we again use the trueness value. Note that we still use the winning and priority based rewards, hence the "progress" of the monitor at m_e is already incorporated.

Each monitor is a list of formulae which all have to be fulfilled repeatedly. Since we want all of these goals to be satisfied, we first take the minimum trueness value for all formulae tracked by a particular monitor. Furthermore, since the progress of all monitors ranked lower than m_e is irrelevant due to the success or fail event of m_e, we ignore them. Together, we are left with one value for each monitor ranked higher than m_e. To aggregate these values, we pick the maximal one, letting the learner focus on improving a single monitor instead of slowly progressing all of them. The overall "progress" reward thus is given by

$$\mathsf{progress} = \max_{m > m_e} \left(\min_{\phi \in L(v)_m} t(\phi) - \min_{\phi \in L(u)_m} t(\phi) \right),$$

where $L(v)_m$ are the formulae tracked by monitor m in vertex v.

Note that this is only one of many possible choices. Since we are only interested in showing the general applicability of our idea, we simply picked the best heuristic out of several hand-crafted definitions of progress.

4 Experimental Evaluation

In this section, we evaluate the presented techniques to show their potential. We show how initializing strategy improvement using semantic information leads to significant improvements of the algorithm. We evaluate our Q-learning variants on several data sets, both real-world and randomly generated. We compare it to strategy improvement, showing our semantic variant to be competitive on several models. Further data, left out due to space constraints, can be found in Appendix A.

4.1 Setup

The experiments have been carried out on consumer-grade hardware, a laptop with a 2x2,9 GHz Intel Core i5 and 8GB RAM. We investigate several algorithms and models, which we briefly explain in the following.

Algorithms In our evaluation, we investigate the following algorithms:
- `SI`: A reference SI implementation with random initial strategy.

– $\texttt{SI}_{sem}$: SI with semantic initialization.
– $\texttt{QL}_{win}$: Q-learning with only win/loss as reward signal.
– $\texttt{QL}_{pri}$: Q-learning with priority-based rewards.
– $\texttt{QL}_{sem}$: Q-learning with semantic rewards.

While running the Q-learning variants, we repeatedly check whether the current strategy is winning in the starting vertex in order to determine when to stop the learning. We do not use any information gained by this check during the learning process itself.

Metrics First, we count the *number of evaluation steps* until convergence for each algorithm. Since our implementation is only a prototype, we consider time to be less relevant, and use this metric instead to approximate the time needed by an efficient implementation of each variant. For Q-learning, this equates to the number of vertices visited in all learning episodes; for SI we count the number of iterations times the size of the game, to allow for a fair comparison, giving a slight advantage to SI to be on the safe side. See the below remark for further details.

Second, we investigate the *size of the solution*, i.e. the number of vertices reachable under the identified winning strategy. The size of a solution is a good estimate for its quality. For example, a smaller solution means that its implementation requires less memory, since decisions need to be stored for fewer states. We give the solution size as a fraction of the overall size of the respective game.

As both methods involve randomization – Q-learning during sampling, SI in its initialization – we ran our methods five times on each model. We chose a timeout of 60 seconds for each run to allow for a reasonably fast evaluation. Since timeouts are difficult to properly incorporate into averages, we chose to ignore the few timeouts that occurred, usually less than 5%. See Appendix A for details.

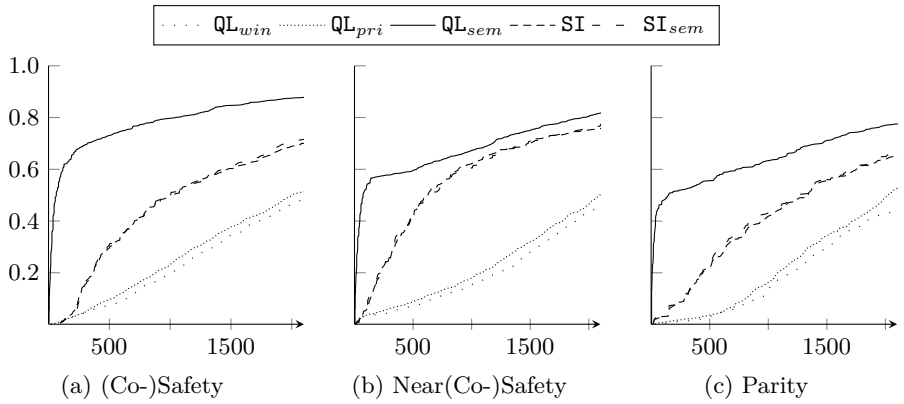
Remark 2. Q-learning and SI evaluate the strategy in vastly different ways. Q-learning picks the currently best action whenever it visits a particular vertex, potentially switching back and forth between two similar actions. In contrast, strategy improvement repeatedly evaluates the current strategy on the whole game, simultaneously changing choices in all vertices which allow for improvement. Intuitively, Q-learning evaluates fewer, important vertices more often, while the evaluations of SI are spread over the whole game. The evaluation of a strategy in SI is costly, since the whole game is considered, while Q-learning simply compares and updates the current Q-values along a single path.

Models We investigate both randomly generated and real-world games.

The random formulae are generated using Spot's [4] `randltl`. We selected three classes of random formulae:
– (Co-)Safety: Pure safety or co-safety formulae
– Near(Co-)Safety: Formulae which mostly consist of either safety or co-safety elements, but contain a few sub-formulae of the other type.
– Parity: Fully random formulae.

Table 1: Percentage of games solved in the starting vertex by the initial strategy and the size of the final solutions. "Near" refers to the Near(Co-)Safety dataset. To obtain more deterministic results, we used additional semantic information obtained from the monitors for tie-breaking in $\texttt{SI}_{sem}$, where applicable.

	Immediately solved games			Solution size		
	(Co-)Safety	Near	Parity	(Co-)Safety	Near	Parity
`SI`	32%	11%	10%	7%	13%	8%
$\texttt{SI}_{sem}$	65%	67%	56%	7%	13%	9%



(a) (Co-)Safety (b) Near(Co-)Safety (c) Parity

Fig. 4: Detailed analysis of all considered methods on randomly generated games. We show the percentage of games solved within the given number of steps.

We investigate the special case of "Near(Co-)Safety" formulae separately, since a lot of real-world specifications often comprise mostly safety and only a few other conditions. This dataset is supposed to imitate this asymmetry.

In order to generate these formulae, we parametrize `randltl` with priorities on the syntax elements. For the "Parity" dataset, we used the default priorities. For the other two, we used the priorities listed in Appendix A. Furthermore, to obtain a reasonable test-set, we filter the generated formulae as follows. First, we remove all formulae where the translation to a parity game using `Rabinizer` takes more than 5 GB of memory or more than 30 seconds, as this would lead to disproportionately large games. Then, we also remove games which have more than 10,000 nodes. We generated 100 such formulae per class.

We also use several real-world formulae from the SYNTCOMP 2017 competition [10]. The specifications are given in the *TLSF* format, which `Rabinizer` can translate to LTL and then to parity games. Again, we filter out games with more than 10,000 nodes, leading to a total of 195 models.

4.2 Results

In Table 1, we present our analysis of the trueness initialization on random formulae. In particular, we compare SI and $\texttt{SI}_{sem}$ with respect to how many games are solved by the initial strategy and the average solution size after the algorithm has converged. In order to evaluate the initialization in $\texttt{SI}_{sem}$, we additionally break ties using the monitor labelling, taking the edge with the largest progress among those optimal w.r.t. the trueness of the master formula. Our presented semantic initialization heuristic immediately identifies a winning strategy for more than half of the cases, while a randomly chosen strategy is only winning with roughly 10% probability. In particular, even in the more complex case of parity games, the trueness of the master formula proves to be a very good initialization heuristic. The solution sizes do not differ significantly on these models, since the solutions usually are rather simple for such randomly generated formulas. We highlight that for random initialization, there only is a negligible difference between the "Near(Co-)Safety" and "Parity" dataset, while our semantic approach works significantly better on the former set.

Remark 3. The performance of our initialization heuristic suggests interesting applications. For example, one could use this new initialization heuristic to explore and solve extremely large games. Observe that the game can be generated on the fly. Hence, when we initially follow trueness-optimal edges, we may immediately identify a solution while only constructing a small fraction of the game.

Fig. 4 shows an evaluation of all our methods on the randomly generated formulae in terms of evaluation steps. The use of priorities consistently improves the performance of QL, but only by a small amount. On the other hand, the use of semantic information vastly improves the performance of QL, outperforming even strategy improvement quite significantly. Moreover, the difference between SI and $\texttt{SI}_{sem}$ is negligible. This is due to strategy improvement running until *global* convergence, hence the algorithm spends effort in some unsolved regions of the game, even if the current strategy is already deciding optimally in most vertices.

Inspired by these results on random games, we applied our algorithms to real-world problems. The results are summarized in Table 2. In our experiments, we considered a large part of the SYNTCOMP set. We additionally hand-picked some classes of formulae to discuss several observations about the semantic rewards.

The naive $\texttt{QL}_{win}$ method is severely underperforming compared to other methods, as expected. Moreover, $\texttt{QL}_{sem}$ often outperforms both other QL variants.

The solution identified by the Q-learning methods often is larger than the one found by $\texttt{SI}_{sem}$. We conjecture that this is due to Q-learning's bias towards exploration – we did not incentivise the learner to yield small solutions. The solution size practically is constant between the different QL methods, suggesting that these larger solutions are due to Q-learning itself. Nevertheless, the solution size is comparable to the one of SI. Moreover, we highlight that $\texttt{SI}_{sem}$'s solutions are significantly smaller than the one identified by SI, although the number of steps

Table 2: Summary of our evaluation on the real-world data set of SYNTCOMP and several sets of randomly generated formulae. We show the geometric average of the number of evaluation steps on the respective data set. "Unrealizable" are all models in the SYNTCOMP data set which are not realizable. Out of the QL variants, we only include the average solution size for $\texttt{QL}_{sem}$, since the solution sizes of all our QL methods are essentially equal.

	Avg. Eval. Steps					Avg. Solution Size		
Class (#models)	$\texttt{QL}_{win}$	$\texttt{QL}_{pri}$	$\texttt{QL}_{sem}$	SI	$\texttt{SI}_{sem}$	$\texttt{QL}_{sem}$	SI	$\texttt{SI}_{sem}$
amba (13)	11540	9765	1271	1119	1089	78%	73%	46%
lily (23)	2179	2052	168	639	580	21%	21%	27%
ltl2dpa (22)	4909	3490	3944	561	552	44%	36%	18%
Unrealizable (53)	1141	1223	101	951	762	3%	4%	5%
Overall (206)	3142	2664	1004	631	531	22%	26%	14%
(Co-)Safety	2177	1993	103	1094	1060	7%	7%	7%
Near(Co-)Safety	2243	1922	151	682	673	12%	13%	12%
Parity	2869	2141	174	1294	1157	7%	8%	9%

until convergence is essentially equal. This suggests that our trueness initialization indeed identifies good initial strategies for such real-world games.

Another interesting observation is that our semantic approaches perform significantly better on unrealizable formulae, although these two cases theoretically are dual to each other. We strongly conjecture that this is due to a bias in the data set. Usually, unrealizable formulae are obtained by injecting small, local faults into an otherwise realizable formula. These local faults are easy to find for our trueness / Q-learning approach. This conjecture is strongly supported by the extremely small solutions found by all approaches.

The lily class seems to be of a similar structure, exhibiting fast convergence rates and small solutions. On the ltl2dpa class, our semantic approach performs rather poorly compared to the priority based variant. This class comprises unusually intricate temporal patterns. We conjecture that a more fine-tuned reward signal may improve performance especially on such models.

5 Conclusion

We have presented the first step towards exploiting semantic labelling in LTL synthesis via the concept of trueness and the subsequent Q-learning. By interpreting the labelling provided by semantic translations, the Q-learning agent can plan ahead instead of only seeing the next vertex. Our first experimental evaluation already shows the potential of this idea.

Future work includes several points of optimization. Firstly, we want to provide a performant implementation of the on-the-fly exploration. Once this is done, an in-depth performance comparison to state-of-the-art tools like Strix is desirable.

Furthermore, we can employ the semantic information to guide the exploration within these on-the-fly tools, as discussed in Remark 3.

Another interesting point is a more sophisticated definition of trueness. Recall that our concept of trueness does not consider temporal aspects of the formula, yet it underpins all of our labelling-based approaches. A different heuristic could yield significant improvements here. Furthermore, one could use more complex learning methods instead of Q-learning. These methods may among other things be able to re-use experience gained while solving a single game.

Finally, we plan on combining our learning methods with strategy iteration. For example, the Q-learning agent can derive a good, but potentially not optimal strategy, and strategy iteration then solves the game with a few adjustments.

References

1. The reactive synthesis competition: SYNTCOMP 2018 results. http://www.syntcomp.org/syntcomp-2018-results/, 2018.
2. Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In *CAV*, 2012.
3. Xu Chu Ding, Mircea Lazar, and Calin Belta. LTL receding horizon control for finite deterministic systems. *Automatica*, 2014.
4. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In *ATVA*, 2016.
5. Javier Esparza and Jan Kretínský. From LTL to deterministic automata: A safraless compositional approach. In *CAV*, 2014.
6. Javier Esparza, Jan Kretínský, Jean-François Raskin, and Salomon Sickert. From LTL and limit-deterministic büchi automata to deterministic parity automata. In *TACAS*, 2017.
7. Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In *CAV*, 2017.
8. John Fearnley. Efficient parallel strategy improvement for parity games. In *CAV*, 2017.
9. Oliver Friedmann and Martin Lange. Solving parity games in practice. In *ATVA*, 2009.
10. Swen Jacobs et al. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In *SYNT@CAV*, 2017.
11. Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *FMCAD*, 2006.
12. Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *CAV*, 2007.
13. Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. PARTY parameterized synthesis of token rings. In *CAV*, 2013.
14. Joachim Klein and Christel Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theor. Comput. Sci.*, 2006.
15. Joachim Klein and Christel Baier. On-the-fly stuttering in the construction of deterministic ω-automata. In *CIAA*, 2007.
16. Jan Kretínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler. Rabinizer 4: From LTL to your favourite deterministic automaton. In *CAV*, 2018.

17. Orna Kupferman. Recent challenges and ideas in temporal synthesis. In *SOFSEM*, 2012.
18. Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, 2005.
19. Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit reactive synthesis strikes back! In *CAV*, 2018.
20. Thibaud Michaud and Maximilien Colange. Reactive synthesis from LTL specification with Spot. In *Proceedings of the 7th Workshop on Synthesis, SYNT@CAV 2018*, 2018.
21. Daniel Neider and Ufuk Topcu. An automaton learning approach to solving safety games over infinite graphs. In *TACAS*, 2016.
22. Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *LICS*, 2006.
23. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, 2006.
24. Shmuel Safra. On the complexity of ω-automata. In *FOCS*, 1988.
25. Sven Schewe. Solving parity games in big steps. In *FSTTCS*, 2007.
26. Sven Schewe. Tighter bounds for the determinisation of Büchi automata. In *FOSSACS*, 2009.
27. Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *ATVA*, 2007.
28. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2018.
29. Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In *TACAS*, 2018.
30. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, 1986.
31. Jens Vöge and Marcin Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *CAV*, 2000.
32. Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 1998.

A Further data

Table 3: Priorities used to generate the random formulae. All unmentioned priorities are set to 0.

Class	$\wedge$	$\vee$	**G**	**F**	**X**	**U**
Safety	7	7	10	0	5	0
Co-Safety	7	7	0	10	5	0
Pseudo-Safety	7	7	10	1	5	1
Pseudo-Co-Safety	7	7	1	10	5	1

Table 4: Overview of all timeouts for each considered algorithm and input class.

Class	$\texttt{QL}_{win}$	$\texttt{QL}_{pri}$	$\texttt{QL}_{sem}$	`SI`	$\texttt{SI}_{sem}$
`amba`	8%	0%	0%	0%	0%
`lily`	0%	0%	0%	0%	0%
`ltl2dpa`	11%	8%	7%	1%	2%
Unrealizable	4%	3%	2%	12%	25%
Overall	11%	7%	9%	4%	10%
(Co-)Safety	0%	0%	0%	0%	0%
Near(Co-)Safety	1%	1%	1%	0%	0%
Parity	1%	4%	3%	1%	1%

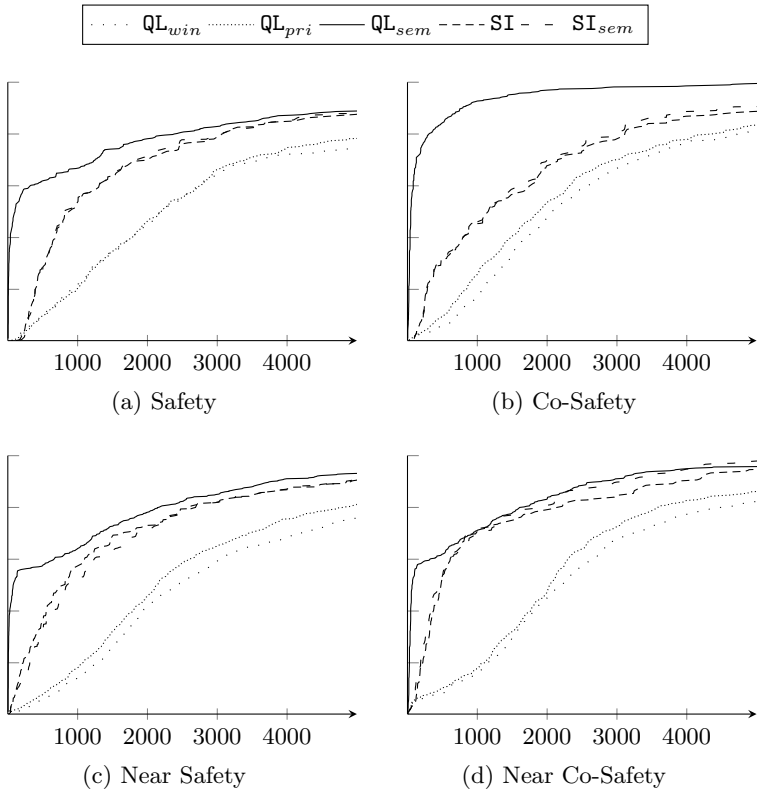
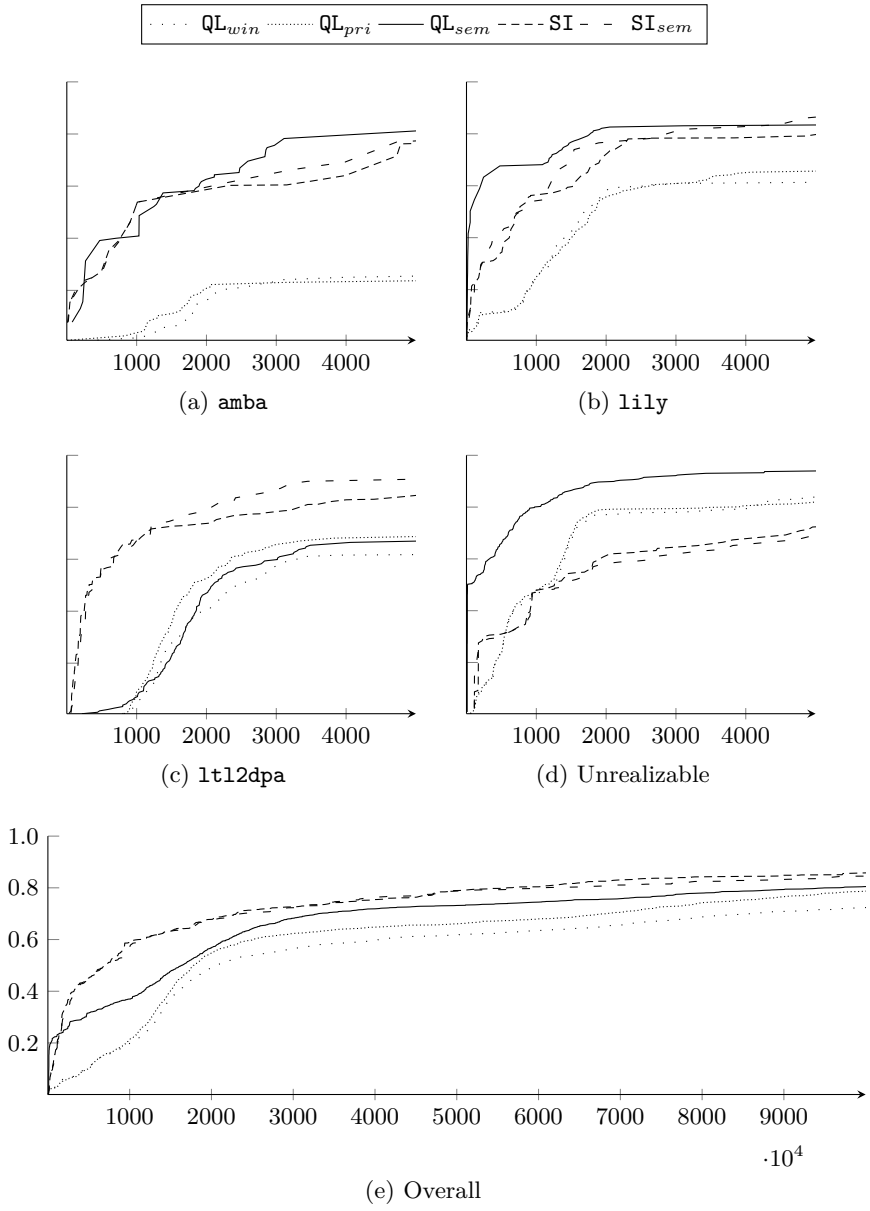


Fig. 5: Data for the "(Co-)Safety" and "Near(Co-)Safety" sub-classes with the same notation as in Fig. 4.



(a) `amba` (b) `lily`

(c) `ltl2dpa` (d) Unrealizable

(e) Overall

Fig. 6: Data for all investigated SYNTCOMP classes with the same notation as in Fig. 4.