DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Learning Support Vector Machines for Predicting Winning Strategies in LTL Synthesis

Sabine Rieder

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Learning Support Vector Machines for Predicting Winning Strategies in LTL Synthesis

Einsatz von Support Vector Machines für das Erzeugen gewinnender Strategien in LTL Synthese

Author:	Sabine Rieder
Supervisor:	Prof. Dr. Jan Křetínský
Advisors:	M.Sc Muqsit Azeem
	M.Sc. Kush Grover
Submission date:	November 8th, 2021

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, November 8th, 2021 Sabine Rieder

Acknowledgments

I want to thank the entire team working on the owl machine learning project, but especially Muqsit, Kush and Jan, for helpful discussions and hints. Thank you for listening to all my ideas and problems and giving me feedback!

In addition, thanks to Christian Backs for answering my questions about the implementation.

Lastly, I would like to thank Saskia and Karin for proofreading the thesis and helping me with the formatting.

Abstract

We apply Support Vector Machines (SVMs) to parity games resulting from Linear temporal logic (LTL) synthesis in order to predict a winning strategy for the games. The thesis builds on recent advances in the field of translations from LTL to Deterministic Parity Automatons (DPAs). These translations allow to store semantic information regarding the LTL formula in the states of the game resulting from the DPA. Namely, we focus on the translation described in [25][6], which uses a Limit Deterministic Büchi Automaton (LDBA) as intermediate step. Each state of a game resulting from this translation consists of a list of monitors keeping track of a subgoal of the original formula. The goal is to fulfill one of the monitors infinitely often.

We follow four main ways to improve the performance of a SVM on such a game: Firstly, we present different ways to combine features computed on the monitors of a state, resulting in a fixed number of features describing the state. Secondly, we introduce new features that utilize the internal structure of a monitor. Thirdly, we separate the learning data into different classes based on the meaning of a state. These classes are then used to train different SVMs on each of them. Lastly, we present a way to detect states for which there is a simple winning strategy. This is based on a computation of a set of system propositions, which would fulfill the formula, if it was played all the time.

The combination of the above approaches allows a prediction of winning strategies for the named translation, which yields similar results to the paper [1], which builds upon a translation from LTL to DPAs using a Deterministic Rabin Automaton (DRA) as intermediate step.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Preliminaries 3
2.1 Basic Definitions . . . 3
2.1.1 Linear Temporal Logic . . . 3
2.1.2 Automata . . . 4
2.2 Translations form LTL to DPA . . . 5
2.2.1 Using LDBAs as intermediate step . . . 5
2.2.2 LTL to DPA using DRA . . . 11
2.3 Synthesis . . . 13
2.4 Support Vector Machines . . . 14
2.5 Underlying work . . . 14
2.5.1 Guided Research of Backs . . . 15
2.5.2 Unpublished Work by Backs . . . 15

3 Approaches 18
3.1 Working with the Monitors . . . 18
3.1.1 Variable Number of Monitors . . . 18
3.1.2 Formulas within a Monitor . . . 20
3.1.3 Change of Feature Values along an Edge . . . 21
3.2 Additional Features . . . 21
3.2.1 Edge Priority . . . 22
3.2.2 Obligation Set and Fail Set . . . 23
3.2.3 Progress in a Monitor . . . 26
3.2.4 Number of Monitors . . . 28
3.2.5 Combination of the Parts of a Monitor . . . 28
3.3 Normalization of Features and Relation between Feature Values of States 29
3.4 Separating the Learning Data . . . 30
3.4.1 Different Classes for Transient and Recurrent States . . . 30
3.4.2 Dividing Based on The Number of Monitors . . . 31
3.4.3 Dividing based on Change in the Master Formula . . . 32
3.4.4 Training Different Support Vector Machines for Safety and Co-Safety formulas . . . 32
3.5 Additional Obligation Set Check . . . 32

4 Experimental Results **36**
4.1 General Information on the Experiments 36
4.1.1 Test Data 36
4.1.2 Metric 36
4.1.3 Training and Evaluation of the SVMs 37
4.2 Evaluation of the Different Functions for Monitors 38
4.3 Change Features 41
4.4 Additional Features 42
4.5 Normalizing the Features 47
4.6 Dividing the Learning Data into Different Classes 49
4.6.1 Transient and Recurrent States 49
4.6.2 Number of Monitors 52
4.6.3 Change in the Master Formula 55
4.6.4 Different Categories of Games 59
4.7 Additional Obligation Set Test 59
4.8 Comparison of the LDBA and the DRA Approach 62

5 Future Work **65**

6 Conclusion **67**

Bibliography **68**

1 Introduction

In the reactive synthesis problem, one is given a specification over the input and output of a system. The goal is to find a strategy producing a stream of outputs for each stream of inputs, such that the specification is met. We consider the problem of *LTL synthesis*, where the specification is given as a formula in Linear temporal logic (LTL). The problem can be solved using the automata-theoretic approach. To do this, the LTL formula is translated into a Deterministic Parity Automaton (DPA). The winning condition of this DPA defines a parity game between the system player and the environment player. The environment player can control the input propositions and the system player the output propositions. To solve the reactive synthesis problem, a winning strategy for the system player is computed. In this thesis, we use a strategy iteration algorithm as described in [28] [27] to accomplish this. If the initial strategy for the strategy iteration algorithm is good, the runtime decreases. Therefore, this thesis aims to find a good initial strategy.

In contrast to Safra's construction [22] for determinization of automata, recent advances in the translation from LTL to DPA [16][6] allow for storing semantical information in the states of the DPA. Namely, each state consists of a *master formula* stating the remaining goal to be fulfilled and keeping track of the finite part of the formula, and possibly several *monitors* measuring the progress of the infinite part of the formula. Each of the monitors contains a list of formulas, which, together, are enough to fulfill the overall goal, if they hold. These information have been used in different ways in order to find a winning strategy. Křetínský et al. defined a heuristic named *trueness* [13], which measures, how close a master formula in a state is to being satisfied. They use this heuristic to build an initial strategy. Furthermore, they implemented Q-Learning on the game to construct winning strategies. A similar approach was followed by Backs in [1]. He trained a Support Vector Machine (SVM) with the goal of predicting, if an edge belongs to a winning strategy. In [1], he suggested different features for the edges of a game, including the trueness of the master formula of the successor of an edge. All of these features are defined on the edge itself or the master formulas of the two nodes adjacent to the edge and none of them considers the monitors. A SVM trained on these features can then be used to find the edge most likely to be part of a winning strategy for each state of the game. From this information, an initial strategy is constructed, which maps each state to the edge chosen by the SVM.

The approach presented in [1] was evaluated using the translation form LTL to DPA described in [5][12][16]. It works well for safety and co-safety formulas. For safety formulas, the goal is to avoid an event. Co-safety formulas, on the other hand, can be seen as reachability of good events. However, on games constructed from other classes of formulas, the suggested approach still achieves better results than trueness, but they are not as good as on the safety and co-safety formulas.

Later on, Backs suggested in unpublished work, for which the code can be found in [2], to consider the monitors, as well, and introduced new features for them. This work is based on the translation from LTL to DPA described in [25] [6], because the monitors resulting from it have more internal structure than the monitors from the previous translation.

In this thesis, we propose different approaches to improve on the unpublished work. In a first step we discuss different methods to receive a fixed number of values for each state and feature based on the monitors, even though the number of monitors for a state varies. Then, we introduce new features. Afterwards, we discuss different classes of states and the results of a SVM trained not on all states, but only on the states of one of these classes. The classes follow the idea, that some states focus on the finite part of the formula, while other states keep track of the infinite goals. In a last step, we present a heuristic to detect states, for which the winning strategy can be computed without sending a request to a SVM.

The outline is as follows: chapter 2 contains the basic definitions and concepts needed in the thesis. Chapter 3 then describes the different approaches made during the thesis. The results are discussed in chapter 4. We end the thesis with a suggestion for future work in chapter 5 and a conclusion in chapter 6.

Related Work

As already mentioned, this work is closely related to [13] and [1]. In both papers, different learning approaches were applied to translations from LTL to DPA, which store semantic information in the states of the game. These information are then used in order to find a winning strategy. Since the thesis builds on the work of [1], it will be described in more detail in section 2.5.

The literature also presents other ways to solve the reactive synthesis problem. In [20], an algorithm exploiting the information for the translation described in [25] [6] is presented. The labels of the nodes are used to construct a forward search algorithm. This way, parts of the automaton do not need to be constructed.

On can also use Safra's determinization [22] to construct a deterministic automaton form a LTL formula. Afterwards, a winning strategy can be extracted from the deterministic automaton. However, the runtime depends on the size of the automaton, so it is crucial to keep it small. Since Safra's construction is known to be inefficient in practice [17], [11] suggests heuristics to improve it.

There are also other approaches, which avoid Safra's construction. In the bounded synthesis problem, only systems with a bounded number of states are considered. Examples can be found in [23], [10] and [4].

Furthermore, the LTL synthesis problem can be reduced to checking emptiness of non-deterministic Büchi tree automata [18].

2 Preliminaries

In this chapter we give basic definitions and briefly describe the different translations from LTL to DPA needed for the thesis. Additionally, we give a more detailed explanation of the reactive synthesis problem. Furthermore, part 2.5 contains a short discussion of previous work done on the topic, which we use as a basis.

2.1 Basic Definitions

2.1.1 Linear Temporal Logic

For this thesis, we consider LTL formulas in negation normal form (NNF), because all formulas occurring in the labels of the states are of this form. Similar to [5] and [25] we define:

Definition 1. *A LTL formula in negation normal form over the set of atomic propositions AP is given by the syntax:*

$$\varphi ::= \mathbf{true} | \mathbf{false} | a | \neg a | \varphi \wedge \varphi | \varphi \vee \varphi | \mathbf{X}\varphi | \mathbf{F}\varphi | \mathbf{G}\varphi | \varphi\mathbf{U}\varphi | \varphi\mathbf{W}\varphi | \varphi\mathbf{R}\varphi | \varphi\mathbf{M}\varphi \tag{2.1}$$

where $a \in AP$.

An ω-word w is an infinite sequence of letters $w[0]w[1]...$. The infinite suffix $w[i][i+1]...$ is denoted by w_i. The semantics is inductively defined as follows:

$$\begin{array}{ll}
w \models \mathbf{tt} & \\
w \not\models \mathbf{false} & \\
w \models a & \text{iff } a \in w[0] \\
w \models \neg a & \text{iff } a \notin w[0] \\
w \models \varphi \wedge \psi & \text{iff } w \models \varphi \text{ and } w \models \psi \\
w \models \varphi \vee \psi & \text{iff } w \models \varphi \text{ or } w \models \psi \\
w \models \mathbf{X}\varphi & \text{iff } w_1 \models \varphi \\
w \models \mathbf{F}\varphi & \text{iff } \exists k. w_k \models \varphi \\
w \models \mathbf{G}\varphi & \text{iff } \forall k. w_k \models \varphi \\
w \models \varphi\mathbf{U}\psi & \text{iff } \exists k. w_k \models \psi \text{ and } \forall 0 \leq j < k. w_j \models \varphi \\
w \models \varphi\mathbf{W}\psi & \text{iff } (\exists k. w_k \models \psi \text{ and } \forall 0 \leq j < k. w_j \models \varphi) \text{ or } (\forall k. w_k \models \varphi) \\
w \models \varphi\mathbf{M}\psi & \text{iff } \exists k. w_k \models \varphi \text{ and } \forall 0 \leq j \leq k. w_j \models \psi \\
w \models \varphi\mathbf{R}\psi & \text{iff } (\exists k. w_k \models \varphi \text{ and } \forall 0 \leq j \leq k. w_j \models \psi) \text{ or } (\forall k. w_k \models \psi)
\end{array}$$

The weak until operator **W**, the release operator **R** and the strong release operator **M** can be transformed into equations using only $\mathbf{F}, \mathbf{G}, \mathbf{U}$ by the following equivalences:

$$\begin{aligned}
\varphi \mathbf{W} \psi &\equiv (\varphi \mathbf{U} \psi) \vee \mathbf{G} \varphi \\
\varphi \mathbf{R} \psi &\equiv \psi \mathbf{W} (\psi \wedge \varphi) \equiv (\psi \mathbf{U} (\psi \wedge \phi)) \vee \mathbf{G} \psi \\
\varphi \mathbf{M} \psi &\equiv \psi \mathbf{U} (\psi \wedge \varphi)
\end{aligned}$$

2.1.2 Automata

For this thesis, we need four different types of atomata: Limit Deterministic Büchi Automaton (LDBA), Deterministic Rabin Automaton (DRA), Generalized Deterministic Rabin Automaton (GDRA) and Deterministic Parity Automaton (DPA). Their definitions can be found in this chapter.

A LDBA is one of the automata that can be used as intermediate step to transform a LTL formula into a DPA. The definition of the LDBA is taken from [25].

Definition 2. *A limit deterministic Büchi automaton is a tuple* $B = (\Sigma, Q, \delta, q_0, \alpha)$ *with alphabet* Σ*, finite set of states* Q*, transition function* $\delta : Q \times \Sigma \to 2^Q$*, initial state* $q_0 \in Q$ *and accepting condition* $\alpha = \{F_1, ..., F_n\}$ *with* $F_i \subseteq T$*, where* $T = \{(s, \nu, t) | s \in Q, t \in \delta(s, \nu)\}$ *is the set of transitions in the automaton. Additionally, there must be a partition* $Q = Q_N \uplus Q_D$ *for the set of states* Q *fulfilling the following two conditions:*
1. $\forall q \in Q_D, \nu \in \Sigma.\ \delta(q, \nu) \subseteq Q_D$ *and* $|\delta(q, \nu)| = 1$
2. $\forall F \in \alpha.\ F \subseteq Q_D \times \Sigma \times Q_D$
A run $r = t_1 t_2 ... \subseteq T^\omega$ *is defined as a sequence of transitions in which for all* $t_i = (s_1, \nu_1, t_1)$*,* $t_{i+1} = (s_2, \nu_2, t_2)$ *it has to hold that* $t1 = s2$*.*
A run r *is called accepting, if, for each* $F \in \alpha$*, there is at least one transition of* F *occurring infinitely often during* r*.*

Under conditions 1 and 2 of Definition 2, a run needs to leave the non-deterministic component Q_N after finitely many steps and change to the deterministic component Q_D in order to be an accepting run.

The DRA and the GDRA are another option for the translation from LTL to DPA. This translation is defined in the papers [5], [12] and [16]. The following definitions for the automata are based on the notation of [8], which is a more detailed version of [5].

Definition 3. *A Deterministic Rabin Automaton (DRA) is a tuple* $B = (\Sigma, Q, \delta, q_0, \alpha)$ *with alphabet* Σ*, finite set of states* Q*, transition function* $\delta : Q \times \Sigma \to Q$*, initial state* $q_0 \in Q$ *and accepting condition* $\alpha = \{(F_1, I_1), ..., (F_n, I_n)\}$ *with* $F_i, I_i \subseteq T$*, where the set of transitions* T *is defined as for the LDBA.*
A run $r = t_1 t_2 ... \subseteq T^\omega$ *is again defined as a sequence of transitions, where the start and end point of two consecutive transitions are equal. A run* r *on a DRA is accepting, if there is a pair* $(F_i, I_i) \in \alpha$ *s.t. all transitions in* F_i *are visited finitely often and at least one transition of* I_i *is visited infinitely often.*

A Generalized Deterministic Rabin Automaton (GDRA) is defined as the DRA , but with a different accepting condition:

$$\alpha = \{(F_1, \{I_{11}, ..., I_{1m_1}\}), ..., (F_n, \{I_{n1}, ..., I_{nm_n}\})\}$$

with $F_i, I_{ij} \subseteq Q \times \Sigma \times Q$.
A run on a GDRA is called accepting, if there is a pair $(F_i, \{I_{i1}, ..., I_{im_i}\})$ *s.t. each transition of* F_i *is visited finitely often and for each* I_{ij} *there is a transition visited infinitely often.*

The most important automaton for this thesis is the DPA. It is later on used to define a game between the system player and the environment player, so that a strategy for the system player can be computed, which fulfills the LTL specification.

Definition 4. *A Deterministic Parity Automaton (DPA) is a tuple* $B = (\Sigma, Q, \delta, q_0, \alpha)$ *with alphabet* Σ*, finite set of states* Q*, transition function* $\delta : Q \times \Sigma \to 2^Q$ *and initial state* $q_0 \in Q$*. The acceptance condition* $\alpha : T \to \{1, ..., c\}$ *maps a transition to a natural number called color.*

To set $inf(r) = \{\alpha(t) | t \text{ occurs infinitely often in } r\}$ *contains all colors, which appear infinitely often in a run* r*.*
In this thesis, a run r *is accepting, if the minimum of* $inf(r)$ *is odd.*

Notice, that it is also possible to have different acceptance conditions for a DPA, e.g. the minimum of $inf(r)$ needs to be even. However, we follow the implementation [14] of the translation described in [25][6].

2.2 Translations form LTL to DPA

For this thesis, two translations from a LTL formula to a DPA are relevant. The translation on which we build our results converts the LTL formula into a LDBA [25] and then transforms the LDBA into a DPA [6]. This translation has the advantage of keeping information explicitly stored in the monitors of each state.

The second translation was used by Backs in [1] and builds on a DRA as intermediate step. We only present a short overview of it, in order to briefly compare the results obtained by the two approaches.

2.2.1 Using LDBAs as intermediate step

This section gives an overview of the main construction used during the thesis. Details can be found in [25] and [6]. Firstly, step the LTL formula is translated into a LDBA, utilizing the so called after formula. This formula defines the conditions under which the remainder of the formula after reading a letter holds true[25]:

Definition 5. *Let φ be a formula and let $\nu \in 2^{AP}$. Then the after formula is defined as follows:*

$$
\begin{aligned}
af(\mathbf{true}, \nu) &= \mathbf{true} \\
af(\mathbf{false}, \nu) &= \mathbf{false} \\
af(a, \nu) &= \begin{cases} \mathbf{true} & \text{if } a \in \nu \\ \mathbf{false} & \text{if } a \notin \nu \end{cases} \\
af(\neg a, \nu) &= \begin{cases} \mathbf{false} & \text{if } a \in \nu \\ \mathbf{true} & \text{if } a \notin \nu \end{cases} \\
af(\varphi \wedge \psi, \nu) &= af(\varphi, \nu) \wedge af(\psi, \nu) \\
af(\varphi \vee \psi, \nu) &= af(\varphi, \nu) \vee af(\psi, \nu) \\
af(\mathbf{X}\varphi, \nu) &= \varphi \\
af(\mathbf{F}\varphi, \nu) &= af(\varphi, \nu) \vee \mathbf{F}\varphi \\
af(\mathbf{G}\varphi, \nu) &= af(\varphi, \nu) \wedge \mathbf{G}\varphi \\
af(\varphi\mathbf{U}\psi, \nu) &= af(\psi, \nu) \vee (af(\varphi, \nu) \wedge \varphi\mathbf{U}\psi)
\end{aligned}
$$

The missing LTL operators $\mathbf{W}, \mathbf{R}, \mathbf{M}$ are not part of paper [25], but can be defined using LTL equivalences.

The automaton returned by the construction in [25] consists of an initial component and an accepting component. In the initial component, each state is labeled by a LTL formula. There is one state for every formula reachable through the after function. The edges between states follow the after function, as well. Intuitively, the states keep track of the formula, that needs to be fulfilled by the rest of the words. In the paper [25], the formula $c \vee \mathbf{XG}(a \vee \mathbf{F}b)$ is used as an example. We extend this formula and use $\psi = c \vee \mathbf{XG}(a \vee \mathbf{F}b) \vee \mathbf{G}d$ as our running example in this section. In the new formula, the part $\mathbf{G}d$ is a safety formula in the monitors, which helps explaining later on in this section. The initial component for the LDBA for the described formula is shown in figure 2.1. The initial state is state 5. From there, an edge with label c leads to a sink with label **true**, which is the after formula of ψ, when c is part of the first letter. The other two edges of state 5 depend on whether $\mathbf{G}d$ is violated.

The accepting component, on the other hand, consists of several subcomponents. Each of these represents a subset of $\mathbf{G}$-subformulas of the original formula φ. For the running example $\psi = c \vee \mathbf{XG}(a \vee \mathbf{F}b) \vee \mathbf{G}d$, the set of all $\mathbf{G}$-subformulas is then $\{\mathbf{G}(a \vee \mathbf{F}b), \mathbf{G}d\}$. This means, there will be a subcomponent for $\mathbf{G}(a \vee \mathbf{F}b)$ and one for $\mathbf{G}d$. Theoretically, there is also one component for the entire set of $\mathbf{G}$-subformulas, but it is optimized away in the implementation, because a word fulfilling both of the formulas is already accepted by each one of the first two subcomponents. In addition, there is also a subcomponent for the empty set. However, this can easily be captured by state 6 of the initial component, so it is optimized away in the implementation, as well.

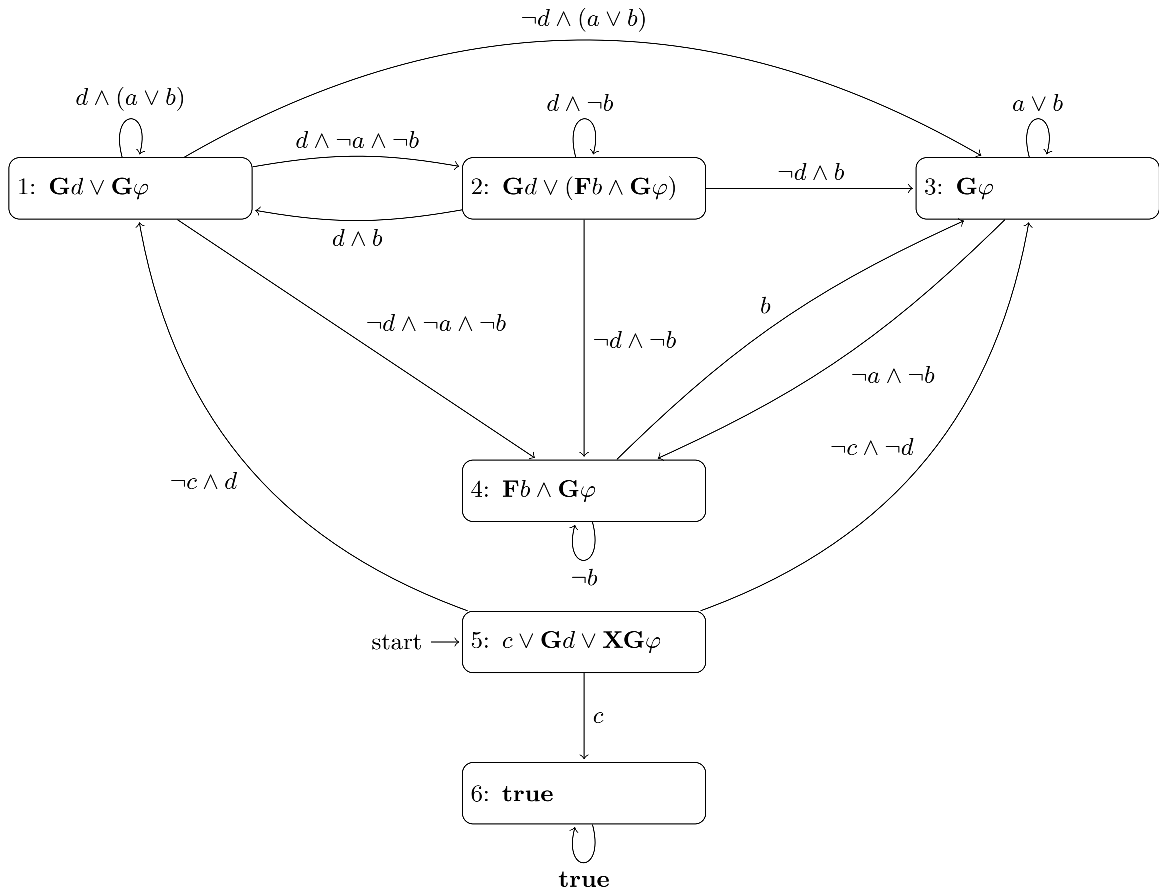

Figure 2.1: This is the initial component of the LDBA for the formula $c \vee \mathbf{XG}(\varphi) \vee \mathbf{G}d$ with $\varphi = a \vee \mathbf{F}b$. The initial state is state 5.

For each component, the subset of **G**-subformulas represented by the component are the assumptions within it. Intuitively, the master formula should to be true if the assumptions hold and these assumptions are checked separately.

Therefor, each of the subcomponents is a product of different deterministic automata. The first automaton measures the progress in the master formula and is similar to the initial component with a slight difference in the labels of the states. For each state label, all occurrences of **G**-formulas, which are part of the assumptions, is replaced by **true**. All occurences of **G**-formulas not contained in the assumptions are replaced by **false**. This automaton will then check the finite part of the formula.

Additionally, there is one automaton for each **G**-formula contained in the assumption. This automaton checks, that the formula indeed holds.

Figure 2.2 shows the accepting components for the formula $\psi = c \vee \mathbf{XG}(a \vee \mathbf{F}b) \vee \mathbf{G}d$ as returned from the implementation. The implementation adds more details to the states of the accepting component than the theoretical approach does. Here, each state consists of a safety formula, a current co-safety formula, stored in the field *current*, a list of co-safety formulas, which need to hold all the time, and a list of f-co-safety formulas, which are co-safety formulas that need to hold eventually. Additionally, there is an index telling the user which co-safety formula is currently checked. The safety formula is a formula representing the safety properties of the state. If this formula turns out to be false, the state will turn into a **false**-sink. In figure 2.2, a safety formula is e.g. $\mathbf{G}d$.

The co-safety goals can be seen as reachability. The implementation always works on one co-safety-goal at a time. The goal is indicated by the field *index* and is stored in the field *current*. Whenever it is satisfied, the index is increased and the next co-safety goal is tested. In order to ensure fulfilling the goals at all times, each step carries out a conjunction of the original goal and the goal stored in the list *next* which represents what needs to be fulfilled at the moment. This can be seen in the edge from state 3 to state 7. Here, the proposition a was not played, so $\mathbf{F}b$ remains to be fulfilled. The co-safety goal is updated and put into a conjunction with its original form. The same would happen for each goal in the field *next*, assuming there would be one. The f-co-safety goals are not updated, because they only need to hold at one step, so there is no need to track a remainder. Whenever the list of co-safety goals is fully traversed, the index turns negative to indicate the focus is now on the f-co-safety formulas and one by one they are moved to the field *current* and tested following the same rules as above.

An edge is part of an accepting set, if all the co-safety goals in next and the set of f-co-safety goals are fulfilled. In this case, the edges from state 7 to state 7, from state 8 to state 7, from state 9 to state 9 and from state 6 of the initial component to state 6 are part of the acceptance condition.

The initial component is connected to the accepting component using ϵ-jumps. A jump leads from a node in the initial component with labeling φ to a state in each subcomponent. The state in the subcomponent corresponds to φ, such that each of the **G**-subformulas is replaced as above and all the automata checking the assumptions are in their respective initial state. For example, state 1 of figure 2.1 has an ϵ edge to the

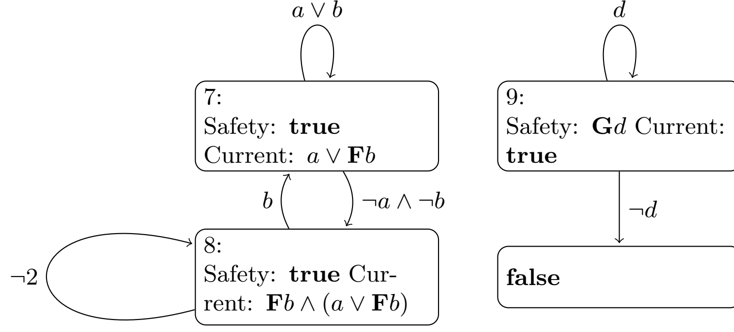

Figure 2.2: The accepting component of the LDBA for the formula $c \vee \mathbf{XG}(\varphi) \vee \mathbf{G}d$ with $\varphi = a \vee \mathbf{F}b$ as returned by the implementation. The index is 0 for all states, because there is only one goal in each accepting component, and the next list consists of the one formula in current for the same reason. The Master-formula is **true** for each state, since the only finite part to be checked is c, and this would already be part of the initial component. There is no f-co-safety goal.

states 7 and 9 of figure 2.2.

Finally, a dpa can be defined from this LDBA. The translation is described in [6]. The main idea is to keep track of all the possible runs a word could have on the automaton. However, there might be infinitely many of them. To cope with this problem, an ordering of the runs is defined. Whenever two runs join, the run with the lower rank in the ordering is kept and the other one is discarded. This is seen as a negative event and an even color corresponding to the position of the run in the ordering is displayed. On the other hand, if a run visits an accepting edge, this is a positive event and an odd color is returned. Again, the priority will correspond to the position of the run in the ordering. In total, the lowest ranked run which visits an accepting edge or merges with another run will be responsible for the overall color of the edge for the DPA.

This is implemented in [14] as follows: Each state of the DPA consists of a master-formula and several monitors. The master formula φ is given by the labeling of the state in the initial component of the LDBA and is basically defined using the after-formula. The monitors are all the states of the accepting component the run could be in. They are given in a list $m_1, ...m_n$. The edge for the letter $\nu \in 2^{AP}$ is then given as follows: The new master formula is defined as $af(\varphi, \nu)$. For each of the monitors, we have a look at the corresponding state in the LDBA. Each monitor is updated using the transition function δ of the LDBA. So, the new list of monitors is $\delta(m_1, \nu), ..., \delta(m_n, \nu)$. If there are duplicates $m'_j, m'_i, j < i$ in this list, m'_i will be removed and an even priority defined as $2Ind(m'_i)$ where $Ind(m'_i) = i$ is the position in the list. The same holds, if the monitor m_i turns into a **false** sink. On the other hand, if a monitor m_i succeeds all its subgoals,

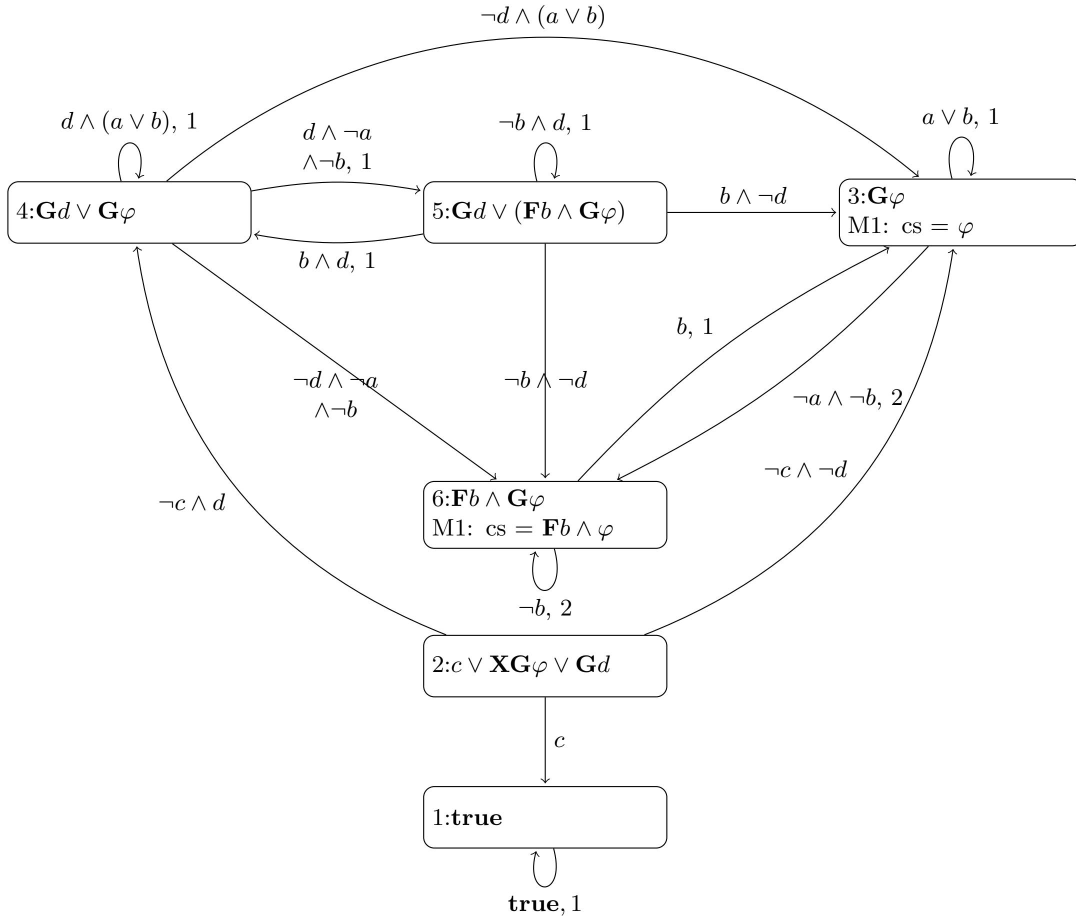

Figure 2.3: The DPA constructed from LDBA for the formula $c \vee \mathbf{XG}(\varphi) \vee \mathbf{G}d$ with $\varphi = a \vee \mathbf{F}b$. The initial state is state 2. The M in the labeling of a state is a monitor with co-safety goal *cs*. Only states 3 and 6 contain monitors.

it will output a good priority, namely $2Ind(m_i)+1$. Every other monitor will not output any priority. The overall color of the edge will then be the minimal returned priority of all the monitors.

Additionally, there are some optimizations. Firstly, if an edge leaves a strongly connected component, it does not have a color, because this edge can only appear once and the color is irrelevant. Secondly, some monitors are optimized away. Figure 2.3 shows the created DPA for the formula $c \vee \mathbf{XG}(a \vee \mathbf{F}b) \vee \mathbf{G}d$. The monitors corresponding to $\mathbf{G}d$ are not present. In the strongly connected component checking for this formula, $\mathbf{G}d$ is the only goal necessary to win the game, so all the monitors can be optimized away.

2.2.2 LTL to DPA using DRA

The translation using a DRA as intermediate automaton consists of three steps. First, the LTL formula is translated into a GDRA as originally described in [5] and explained in more detail in [8]. Then, a DRA is created like in [12]. Lastly, the DRA is translated into a DPA as described in [16]. This section will only contain the main ideas of the translations.

The GDRA is a product of automata. There is one automaton generated using the after formula as described in definition 5. Additionally, there is one monitor automaton for each G-subformula of the original formula. Each of these automata has a different acceptance condition, depending on which other G-formulas are true. This is only needed in case of nested formulas. The details can be found in [5]. The overall acceptance condition checks the conditions of the monitors for each subset of monitors. Additionally , it verifies, that the monitors in the subset are enough to satisfy the master formula. This is not stored explicitly in the states as it is the case for the LDBA construction, but is contained in the acceptance condition. Figure 2.4 shows the DRA for the formula $c \vee \mathbf{XG}(a \vee \mathbf{F}b) \vee \mathbf{G}d$. There is one monitor for each of the G-subformulas. These monitors do not contain any information on connections between them, as it is the case in the LDBA approach. The acceptance condition ensures the fulfillment of the overall formula and the subformulas, e.g. C_1 checks, that the monitor for $\mathbf{G}(a \vee \mathbf{F}b)$ succeeds and it is enough to satisfy the master formula of the state. C_2 checks for the monitor $\mathbf{G}d$. The other conditions check combinations of the above.

Afterwards, the GDRA is transformed into a DRA as described in [12]. the goal is to have a single set of transitions, out of which one transition needs to be visited infinitely often, instead of a conjunction of such sets. For one of these conjunctions, one can create a DRA accepting the same words by adding a new layer for each of the sets, which should be visited infinitely often. If one transition of the set corresponding to the layer is visited, one moves on to the next layer. Afterwards, several of these automata need to be combined, since the acceptance condition is a disjunction of such conjunctions. To do so, the automata for each acceptance condition are multiplied with each other to receive a joint state space. The overall automaton is then created by combining all acceptance conditions in a disjunction.

For the last step, index appearance records are used. The details can be found in [16]. To do so, a permutation of the indices of the accepting conditions of the DRA are stored. Whenever a prohibited set is visited along an edge, its index will be moved to the end in the permutation. If the prohibited set F_i is visited, it will return a bad priority depending on its position in the permutation, meaning $2Ind(i)$, where Ind represents the position in the permutation. If I_i is visited and F_i is not visited, the set will lead to a good priority, namely $2Ind(i) + 1$. At the end, the lowest priority is returned as before.

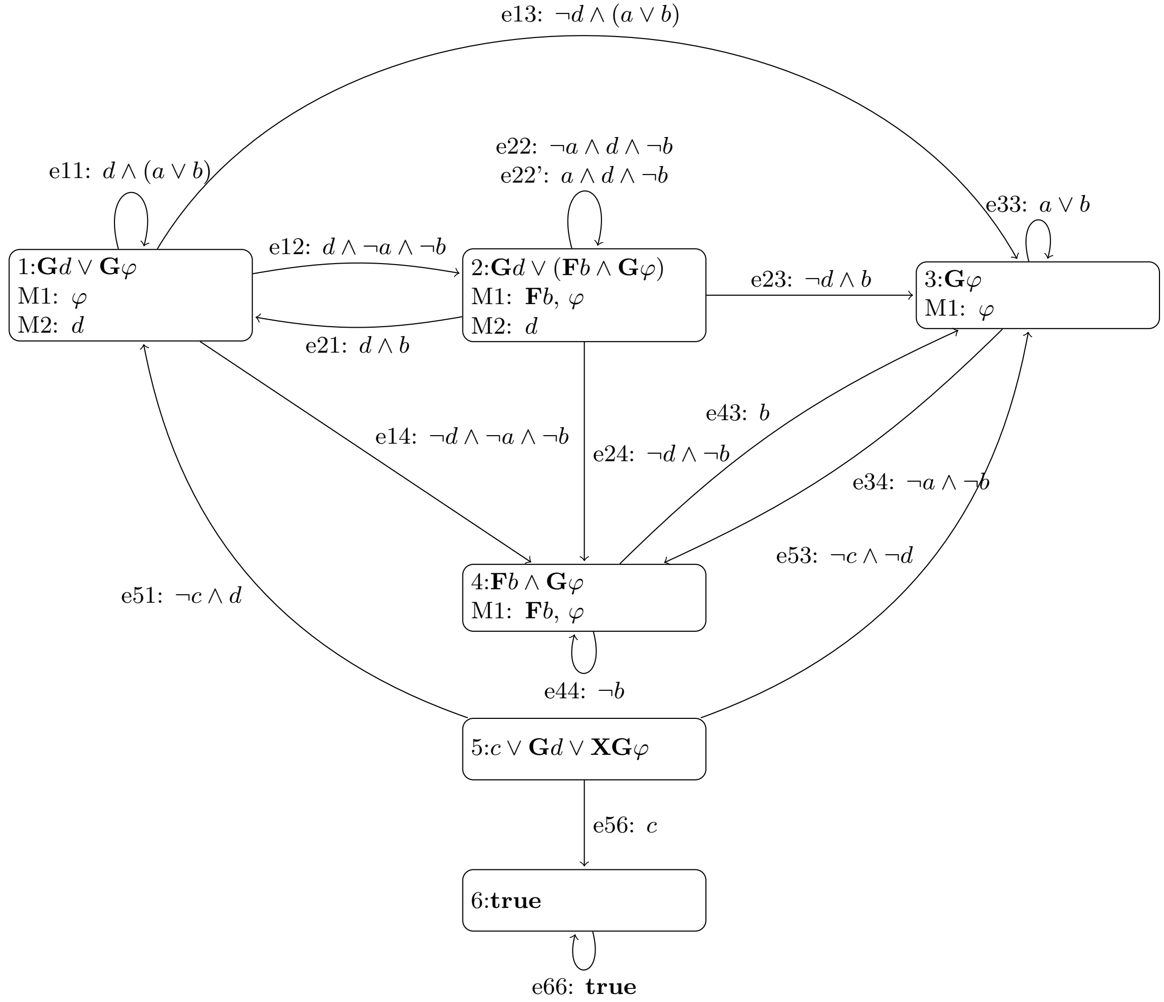

Figure 2.4: The GDRA for the formula $c \vee \mathbf{XG}(\varphi) \vee \mathbf{G}d$ with $\varphi = a \vee \mathbf{F}b$. The initial state is state 5. The acceptance conditions is $\{C_1, ..., C_5\}$ with $C_1 = (\emptyset, \{\{e11, e33, e43, e21\}\})$, $C_2 = (\emptyset, \{\{e11, e12, e21, e22, e22'\}\})$, $C_3 = (\emptyset, \{\{e11, e21\}, \{e11, e12, e21, e22, e22'\}\})$, $C_4 = (\{e22\}, \{\{e22'\}, \{e11, e12, e21, e22'\}\})$, $C_5 = (\emptyset, \{\{e66\}\})\}$.

2.3 Synthesis

The reactive synthesis problem is to find a system that fulfills a given specification while not depending on the environmental behavior [3]. For LTL synthesis, we receive a LTL formula as specification and a partitioning of the propositions into the ones controlled by the environment player and the ones controlled by the system player. We then construct a DPA from the LTL formula the transformations explained above. Afterwards, the DRA is extended in order to create a graph game out of it. This game is then used to find a winning strategy for the system player using the system propositions.

A graph game as defined in [13] is a tuple $G = ((V, E), v_0, P, Win)$, where (V, E) is a directed graph, v_0 is the starting vertex and a function $P : V \rightarrow \{0, 1\}$ mapping each state to a respective player, who ones the state and picks the transition taken for it. In this thesis, player 0 is called the environment player and player 1 is the system player.

For the above transformations, these sets are constructed by splitting each state of the DPA into several states regarding the propositions. For each state of the automaton, we construct one state for the environment player. Afterwards, new system states are added in order to store the environment player's choice. Here, the system player can choose his actions. Then, the overall set of propositions containing the environment player's and the actions chosen by the system player will lead to the environment state corresponding to the state in the original DPA reached by the played set of propositions.

for example, consider state 4 of figure 2.3 with environment propositions $\{d\}$ and system propositions a, b, c. The state 4 would be the environment state. Additionally, there would be two system states, one for the choice of d and one for $\neg d$. In this states, the system player could choose an arbitrary action. Assuming the environment player played $\neg d$ and the system player a, b, the next state would be state 3.

A game on this graph will start at the initial vertex v_0. The player owning this vertex will then choose an outgoing transition and move to the respective end state of the transition. Here, the game will proceed similarly to the first step. Again, the player owning the state will choose a successor state. This will be continued infinitely often. As described in [13], one can now define a play as an infinite sequence of vertices $\rho = v_0 v_1 v_2 \in V^\omega$, where each two vertices v_i, v_{i+1} need to be connected by an edge.

There are different possibilities to define the winning condition on such a play. We use a parity condition. This is based on an assignment of a priority in form of a natural number to each vertex or to each transition. For each play we take into account the infinitely often occurring priorities. If the minimum of these numbers is odd, the system player wins the game. In principle, it is also possible to look at the maximum of all infinitely often occurring priorities or the system player could win if the selected number is odd. E.g. the implementation with a DRA as intermediate step returns a game where the minimal priority occurring infinitely often needs to be even for the system player to win. However, since we work mostly with the construction described in 2.2.1, we use the winning condition of the implementation of this approach throughout the paper.

As priorities, the implementation uses the colors assigned to the edges as described in definition 4.

2.4 Support Vector Machines

A Support Vector Machine (SVM) is a machine learning method for supervised learning, which can be applied to both, classification and regression [24]. This section focuses on classification, as this is needed within the thesis. The idea of a SVM is to separate the data by a hyperplane [26] given by $\{x|wx + b = 0, x \in \mathbb{R}\}$ with $w \in \mathbb{R}^n, b \in \mathbb{R}$ with a maximal margin around it. If the margin is strict, the SVM is called a hard margin SVM. The data to be separated is given by $(x_1, y_1), ...(x_m, y_m)$, where x_i is the input vector and y_i is the class of the sample, namely $y_i \in \{-1, 1\}$. New points x can then be classified using the function $sign(wx + b)$. The problem of computing such a hyperplane is defined as follows [24]:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} w^T w \\ \text{s.t.} \quad & y_i(wx_i + b) \geq 1, i = 1, ...m \end{aligned}$$

The constraint ensures, that each data point lies on the correct side of the margin. The size of the margin can be computed by the perpendicular distance of the two closest points on each side of the hyperplane to the hyperplane [24]: $(\frac{w}{w^T w}(x_1 - x_2)) = \frac{2}{w^T w}$, with $|(wx_1) + b| = 1$ and $|(wx_2) + b| = 1$ and $y_1 = -y_2$. The objective in the above optimization problem maximizes this distance.

If the data is not linearly separable, a soft margin SVM can be used. This is a generalization of the hard margin, where samples are allowed to violate the margin and the hyperplane. It is achieved by adding slack variables ξ_i to the optimization problem (see [24] for more details):

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} w^T w + C \sum_{i=1}^{m} \xi_i \\ \text{s.t.} \quad & y_i((wx_i) + b) \geq 1 - \xi_i, i = 1, ...m \\ & \xi_i \geq 0, i = 1, ...m \end{aligned}$$

The slack variables are part of the objective function and are multiplied with penalty C to ensure, that the data is still separated as clearly as possible.

The *confidence* of a sample x is then defined as its distance to the hyperplane.

2.5 Underlying work

In this section, we will describe previous work on the topic of the thesis, which we use as a basis for our approaches.

2.5.1 Guided Research of Backs

In his guided research [1], Backs used a SVM to predict edges for winning strategies in LTL syntheis. He implemented a SVM and came up with basic features to use, which only depend on the master formula of a state. Because of this, they can be applied to both of the translations explained above. His features consist of:

- The *height* of the syntax tree of the master formula of a state.
- The *trueness* of the master formula of a state. The trueness is defined in [13]. It is an approximation of how close a formula is to being satisfied. First, the LTL formula is transformed to a formula of propositional logic by replacing every occurrence of a temporal operator on the top level of the formula by a fresh variable. The trueness value is then the fraction of satisfying assignments divided by the number of all possible assignments.
- *Disjuncts* is the number of disjunctions on the top level of the master formula.
- *System control* states, if a master formula can approximately be controlled by the system player. It has a value of 1, if this is most likely the case and a value of 0 otherwise. It is computed by traversing the syntax tree recursively. A proposition can be controlled, if it belongs to the system player, a conjunction if all conjuncts can be controlled, a disjunction if one of the disjuncts can be controlled and a temporal operator is assumed to be controllable, if its operand can be controlled.
- *Accepting* is a feature computed on an edge. Assuming the system player wins for an odd color, then this feature will be 1 for all edges with odd color and 0 for all others. It works the same way for an even winning condition.

For the first four features, a difference value is introduced. Assuming an edge e from state s to state t, the difference of feature f is defined as:

$$\Delta f(e) = f(t) - f(s)$$

Backs then used the so called change features and the values of the above features applied to the successor of each edge to train a SVM . The SVM tries to learn for an edge, if it is part of a winning strategy. In order to apply the SVM to predict an edge for a state, the SVM is asked for the confidence value of each edge of the state. The edge with the highest confidence value is returned as initial strategy. Afterwards, a strategy iteration algorithm is executed to find a winning strategy.

2.5.2 Unpublished Work by Backs

Additionally to the guided research in [1], Backs also implemented other ideas in [2]. Since there is no report available for this work, we give a short overview of his ideas.

The main problem in the report [1] is that monitors are not taken into account. Backs implemented some basic ideas applicable to the LDBA translation to change that.

Firstly, he added features similar to the master features described in [1] for each part of the monitor. For example, there are the features *safetyTruenessMonitor, current-TruenessMonitor* and *nextTruenessMonitor*, which compute the Trueness for the safety formula, the current formula and the average of the trueness for all formulas the field *next*, which holds all co-safety formulas, of a single monitor. There are similar features for all master-features defined in [1]. These features are computed for each monitor separately. The problem is, that each state can have a different number of monitors and a SVM can only work with a fixed amount of features. To overcome this, Backs computed for each feature the average over all monitors in a state. This way, he could compute the features for the successor of an edge and compare the different values to make a decision. For example the feature *safetyTrueness* of an edge e leading from s to state t with monitors $monitors(t)$ can be computed as follows:

$$safetyTrueness(e) = \text{avg}_{m \in monitors(t)} trueness(safety(m))$$

where $safety(m)$ is the safety formula of a monitor.

Backs also implemented a way to compute the difference of a monitor feature along an edge similar to the idea for master features. The difference is defined as the difference in the average over all the monitors, e.g. for $\Delta safetyTrueness$:

$$\begin{aligned} \Delta safetyTrueness(e) = \quad & \text{avg}_{m \in monitors(t)} trueness(safety(m)) \\ & - \text{avg}_{m \in monitors(s)} trueness(safety(m)) \end{aligned}$$

We sometimes call this computation the list based approach to the change feature, because the value for the entire list of monitors is computed for each state separately.

In addition to that, a new feature is introduced: *tempOps* is the number of temporal operators in the top level of a formula.

Furthermore, Backs divided the learning data into transient and recurrent states, after he realized, that the SVM performs differently on these two classes. A recurrent state is one, that is visited infinitely often during a game, whereas a transient state stops appearing at some point in time. He trained a SVM only on transient states and one only on recurrent states. Backs then combined them using the following idea: whenever the confidence of the SVM for transient states for some edges of a state is above a certain level, the bes edge is returned. If it is below this level for all edges of a state, the SVM trained on recurrent states is asked for advice. The results of this idea can be seen in table 2.5.2. The SVM trained on recurrent states with only the monitor features performed badly for the safety class. This is not surprising, since both, safety and co-safety classes, do not have any monitors in the games. The result of the SVM only depends on the accepting feature in this case. Additionally, it can be seen, that the combination of SVMs trained on transient and recurrent states is not necessarily better then using just one of them. E.g. for the ltl2dpa category it is better to just use the monitors-recurrent SVM . A reason for this is, that the described approach classifies most states using the SVM for transient states. In this test run, the secondary classifier

category	master-joint	master-transient	monitors-recurrent	transient, recurrent combined	trueness
large-co-safety	0.0014	0.0014	0.0044	0.0010	0.0010
small-co-safety	0.0000	0.0000	0.0000	0.0000	0.0000
small-p-co-safety	0.0063	0.0142	0.0193	0.0142	0.0255
large-safety-56	0.0158	0.0160	0.0853	0.0150	0.0158
small-safety	0.0000	0.0000	0.0617	0.0000	0.0000
small-p-safety	0.0000	0.0000	0.0000	0.0000	0.0467
large	0.0511	0.0539	0.0572	0.0510	0.0778
small	0.0031	0.0031	0.0590	0.0149	0.0625
lilydemo	0.1573	0.1949	0.2006	0.1674	0.2121
ltl2dba	0.1166	0.1166	0.1166	0.1166	0.2753
ltl2dpa	0.1110	0.1110	0.1110	0.1311	0.3383
Average	0.0283	0.0317	0.0505	0.0322	0.0668

Table 2.1: Number of states not mapped to an edge in the winning strategy divided by number of all states. Master-joint is a SVM trained on all states with features only on the master formula, master transient uses the same features, but was trained on transient states only, monitors recurrent uses only the monitor features and was trained on recurrent states.

was only consulted in approximately 30% of the cases, even though most of the states in the games are actually recurrent. Overall, the best SVM was the one trained on only the master features, but all states. Therefor, we compare our results to this SVM, which we often call the *original* one.

3 Approaches

This chapter contains a description of the different ideas we implemented in order to improve the results of the SVM . In section 3.1, we explain different functions that can be used to combine the monitors of a state and we discuss different ways to compute the change of feature values along an edge. The next two sections concern the different features of a state. Section 3.2 contains newly added ones. Since some of the presented features are unbounded, we also tried to normalize them as described in 3.3. In section 3.4 we then present similar approaches to the one in section 2.5.2, where we divide the learning data into different classes and train a SVM for each of these classes. The last section 3.5 contains an additional check before sending a request the SVM in order to avoid unnecessary queries.

3.1 Working with the Monitors

The first idea in order to improve the results of [1] is to include features computed on the monitors of a state and not only rely on the master formula. This is especially interesting for edges alongside which the master formula does not change. As described in section 2.5.2, Backs already suggested to use the average over all values of a feature computed on each monitor separately. In this section we discuss different approaches to improve Backs' idea. The first challenge is to cope with the different number of monitors in each state and the different number of co-safety goals. Furthermore, we define different ways to compute the change value of a monitor feature along an edge.

3.1.1 Variable Number of Monitors

The main problem when defining features on the monitors is that each state has a different number of monitors and a SVM can only work with inputs of fixed size. Backs tried to overcome this by computing the average over all the values of the monitors (see section 2.5.2). However, the average does not take the position of a monitor into account. As described in section 2.2.1, a monitor ranked at a higher position in the list can lead to better priorities, if it succeeds, and it can also outweigh monitors listed at a lower position, where lower position means it is located more towards the end of the list. Using this fact, we follow two main directions. On the one hand, we implement additional functions similar to the average and on the other hand, we also try to only work with the "important" monitors. Both approaches are described in the sections below.

Different Functions Applicable to the Feature Values of the Monitors for One State

In addition to the average, we suggest three functions, namely weighted average, a function using minimum and maximum over the values of a monitor as in [13] and a function returning a list of feature values.

The weighted average follows the idea that a higher ranked monitor is more important for the progress of the game than a lower ranked one. It is defined as follows:

Definition 6. *Let $w \in \mathbb{R}$ be a weight, $m_1, ..., m_n$ the monitors of state s and let $f : \mathbb{M} \rightarrow \mathbb{R}$ be a feature from the monitors to the real numbers, e.g. the trueness feature applied on the safety formula of a monitor.*

$$wavg(s, f, w) = \frac{f(m_n) + wf(m_{n-1}) + 2wf(m_{n-2}) + ... + (n-1)wf(m_1)}{w(\sum_{i=1}^{n-1} i) + 1}$$

The next function is based on the idea that the system player wants to succeed one monitor infinitely often, and was suggested in [13]. We call it progress function or customized function, since for every feature we can use maximum and minimum differently, depending on if a high feature value is actually desired.

Definition 7. *Let $Mon(s)$ be the monitors of state s and let $f : \mathbb{M} \rightarrow \mathbb{R}$ be a feature taking a monitor as input. Then progress is defined as follows:*

$$progress(s, f) = max_{m \in Mon(s)}(f(m))$$

This definition is based on features like trueness, where the goal is to improve the feature value. If the value should be reduced, *min* can be used instead of *max*.

It is also possible to return a list of values instead of a single one. We exploit this fact by computing the feature for each monitor up to a certain threshold position t in the monitor ranking of a state and let the SVM learn for itself how important each monitor is. In order to do so, a default value $d(f)$ is needed for each feature f. Whenever the list of monitors of a state is shorter than the threshold, this default value is returned for the missing positions. We called the function following this idea the monitorList function.

Definition 8. *Let $MonPos(s, i)$ be the monitors of state s at position i in the ranking of s and let $f : \mathbb{M} \rightarrow \mathbb{R}$ be a feature. We define the monitorList function as follows:*

$$monitorList(s, f, t) = \begin{pmatrix} f(MonPos(s, 1)) \\ f(MonPos(s, 2)) \\ ... \\ f(MonPos(s, t)) \end{pmatrix}$$

For simplicity, we did not include the default value in the definition.

All of the above functions can then be applied similarly to the average described in section 2.5.2.

Focusing on Monitors at a High Position

Instead of using all monitors of a state in order to compute a feature with the above functions, one can also try to focus on "important" monitors as suggested in [13]. A monitor is considered important, if it could lead to a priority as least as good as the one on the edge for which we want to make a prediction.

For example, assume the edge has color 5. This color is emitted by the success of the monitor ranked at position 2 in the list. Consequently, the monitors ranked at position 0 and 1 would lead to a color better than the current one, if they would succeed. In total, we take the monitors 0 to 2 into account for computing the features. Every monitor at a position larger than 2 cannot overwrite the current color, so they are considered less important. Notice that this is just a heuristic. In principle, the monitor at position 2 may only succeed once and afterwards the game could be won by letting some monitor at position 3 succeed. If the color of the current edge is 4, monitor 2 fails and cannot overwrite its failure. In this case, we would only consider monitors 0 and 1.

This can be applied to the functions defined in the last chapter. One would replace the function $Mon(s)$ in the definitions by $MonColour(s, c)$, which returns the monitors of state s leading to a color at least as good as c, if they succeed. Additionally, $MonPos(s)$ is replaced by $MonPosColour(s, t, c)$ returning all monitors that could lead to a better color and are ranked at a position before the threshold.

3.1.2 Formulas within a Monitor

Another problem is that each monitor consists of several subformulas, which are described in section 2.2.1. In order to apply the functions of section 3.1.1, we need to produce one value for each monitor. Backs suggested in his unpublished work, which is described in section 2.5.2, to have one feature for the safety formula, one feature for the current field of the monitor and one feature for the list of next co-safety goals. The f-co-safety goals are not taken into account, because they only need to hold at some point in the future, so they are not affected by actions taken, whenever they are not the co-safety goal we are currently trying to succeed.

If we follow this suggestion, we only need to focus on the list of next co-safety goals, since all other fields contain just a single formula. The first approach is to use functions similar to those in the last section.

We decide against using the weighted average within a monitor, since all subgoals are equally important, independent of their position. The average function, on the other hand, is an option and can be used. For the progress function, we decided to follow the ideas of [13]:

Definition 9. *Let m be a monitor and let $f : Formula \to \mathbb{R}$ be a feature. Then the progress function on the list of the next co-safety goals of the monitor is defined as:*

$$progressNext(m, f) = min_{e \in nextCoSafety(m)} f(e)$$

The minimum is chosen, since all formulas within the list need to be fulfilled at some point in time for the monitor to succeed. If one of the co-safety goals is hard to satisfy,

the monitor is likely to not progress, independent of the other co-safety goals. In case a low feature value indicates a positive event, the maximum instead of the minimum of all values should be considered.

The described approach has the drawback that the safety and co-safety functions of the monitors are treated separately. This does not reflect the reality of the game, because an easy co-safety formula like **true** does not lead to success, if the safety formula of the monitor fails and vice versa. In order to solve this problem, we suggest new features, which are described in section 3.2.

3.1.3 Change of Feature Values along an Edge

There are different ways on how to compute the change value of features defined on monitors along an edge. Backs suggested to compute the values for the starting and end point of the edge separately and then subtract them from each other. This does not reflect the change within a single monitor. In principle it is possible that the first monitor of a state improves along an edge and the second one worsens by the same value. This does not have any impact on the suggested change value, but could be helpful due to the different position of the monitors in the list. Because of this, we try to match the monitors with each other and compute the change for each pair before applying one of the functions described in section 3.1.1 to the list of change values.

We propose two main ways to find matching monitors. The first one, called LDBA - successor, computes the successor of a monitor in the underlying LDBA and evaluates the change based on this. The problem here is that the computation is costly. Furthermore, if one monitor fails, the monitors behind it in the list will move forward. This change of the values regarding the position of the monitors is not reflected, when computing matching monitors based on the successor. If a monitor with a good feature value is now ranked further to the beginning of the list, it might be helpful in the future, because not the monitors themselves define the color, but the position of the monitor in the list. In this case it might be more interesting to know, how the list of monitors changed.

The second approach, called position-based, follows the described idea. Here, we compute the change value between two monitors ranked at the same position in the start and end point of the edge.

For both approaches, we need to have a default value, in case one of the monitors has no matching partner. This default value can again be defined for each feature separately.

3.2 Additional Features

We also add new features for the SVM to learn from. In a first step, we improve the *accepting* feature to take into account the value of the color. Additionally, we suggest features called *obligation set* and *fail set*. These aim to measure the control the system player has on the formula.

All of the previously named features compute values separately for each part of the monitor. As already pointed out in the previous section, this might not be the best way

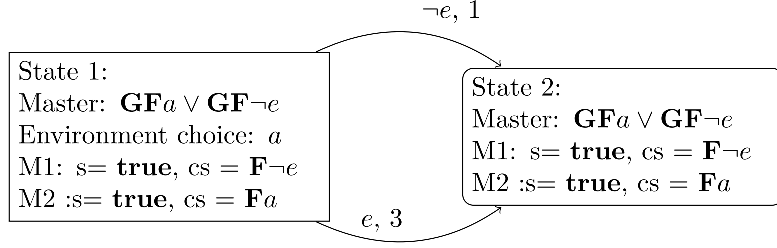

Figure 3.1: A very simple example where the edge priority might influence the choice. The rounded corners indicate the environment state. The pictures only shows a part of a game. The letter s indicates the safety formula of a monitor, cs is the co-safety formula.

for the monitors, as the safety and co-safety formulas depend on each other. Therefore, we also present features focusing on the monitor as a whole or even on the entire state. These are described in section 3.2.3 and 3.2.5.

3.2.1 Edge Priority

In the guided research project [1] Backs added a feature called accepting. This tells the SVM, if the color emitted by an edge is good for the system player, meaning it is even, if the winning condition is even and vice versa. However, there are some states for which the actual value of the color matters, like shown in figure 3.1. Here, both edges are accepting and lead to the same successor state. However, playing $\neg e$ will lead to the better priority for the system player. This is mostly relevant, when there are monitors in between or the environment player could make the last monitor fail.

In order to make these information available to the SVM, we add a new feature representing the edge priority. Winning colors are mapped to values above 0, loosing ones to values below. Furthermore, the colors are mapped according to their value in such a way, that the best color gets the highest score. Since zero is also a possible value for a color, an additional 1 is added to each priority before the mapping. Additionally, there is a case distinction on the winning condition. In the case of a maximal winning condition in the game, all values, which do not belong to the system player will be mapped below zero and the colors of the system player are not changed. For a minimal winning condition, one needs to map small colors to large values in the feature, since the SVM uses only linear functions and cannot learn that a small value above zero is good, but negative values should be as large as possible. To do this, one computes the maximal color within the game and reverses the ordering of the colors. Afterwards one can proceed as in the first case.

For example, let G be a game with minimal odd winning condition and maximal priority of 7. Let e be an edge with priority $prio(e) = 3$. Firstly, the priorities are increased by one. Then the scale of priorities needs to be reversed, because the winning condition is minimal. There are only two smaller priorities than color 3 in the game, so

it will be mapped to a value of 5, which is the third smallest odd number. Since $prio(e)$ is odd, it will not be mapped below zero. In total the feature value is 5.

3.2.2 Obligation Set and Fail Set

Obligation Set

This feature is based on an obligation set similar to the one in [19]. The idea is to find a set of propositions, that would fulfill the formula, if they were played all the time. If such a set is present and depends only on the system player, he can win the game easily.

Definition 10. *Let φ be a formula. We define $os(\varphi)$ as follows:*

$$
\begin{aligned}
&os(\mathbf{true}) = && \{\{\}\}\\
&os(\mathbf{false}) = && \{\}\\
&os(a) = && \{\{a\}\}\\
&os(\neg a) = && \{\{\neg a\}\}\\
&os(\varphi \wedge \psi) = && \{\varphi_1 \cup \psi_1 | \varphi_1 \in os(\varphi) \wedge \psi_1 \in os(\psi)\\
& && \wedge \nexists i \in AP : ((i \in \varphi_1 \wedge \neg i \in \psi_1) \vee (\neg i \in \varphi_1 \wedge i \in \psi_1))\}\\
&os(\varphi \vee \psi) = && os(\varphi) \cup os(\psi)\\
&os(\mathbf{X}\varphi) = && os(\varphi)\\
&os(\mathbf{F}\varphi) = && os(\varphi)\\
&os(\mathbf{G}\varphi) = && os(\varphi)\\
&os(\varphi \mathbf{U} \psi) = && os(\psi)\\
&os(\varphi \mathbf{W} \psi) = && os(\varphi \vee \psi)\\
&os(\varphi \mathbf{R} \psi) = && os(\psi)\\
&os(\varphi \mathbf{M} \psi) = && os(\varphi \wedge \psi)
\end{aligned}
$$

For example, let $\varphi = (\mathbf{F}a \wedge (b\mathbf{U}\neg a)) \vee ((b\mathbf{W}c) \wedge \mathbf{G}a)$. Then we have $os(\mathbf{F}a) = \{a\}$, but $os(b\mathbf{U}\neg a) = \{\neg a\}$. Together, this will result in $(\mathbf{F}a \wedge (b\mathbf{U}\neg a)) = \{\}$, meaning that there is no single set that can satisfy the formula. On the other side $os((b\mathbf{W}c) \wedge \mathbf{G}a) = \{\{b, a\}, \{c, a\}\}$ and together we have $os(\varphi) = \{\{a, c\}, \{a, b\}\}$, so by playing either $\{a, c\}^\omega$ or $\{a, b\}^\omega$ or any superset of the two, the formula will be satisfied.

As one can see, this method does not yield a set for every formula. For example, the formula $(\mathbf{F}a \wedge (b\mathbf{U}\neg a))$ can be easily satisfied by playing $\{\}\{a\}^\omega$, but there is no obligation set for it. However, this feature can still be useful to detect formulas which can be easily fulfilled.

In the next step, the obligation set needs to be turned into a feature for the SVM. It is defined as the maximal ratio of system player propositions within an obligation set.

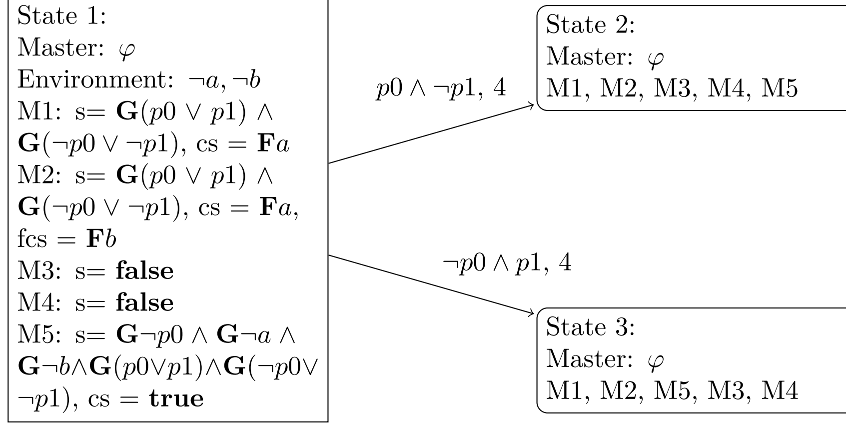

Figure 3.2: This is part of a game generated from $(\mathbf{G}(p0 \leftrightarrow \neg p1) \wedge ((\mathbf{F}(\mathbf{G}(((\mathbf{F}a) \vee (\mathbf{GF}b) \vee (\mathbf{FG}(a \vee b)) \vee (\mathbf{FG}b)))) \leftrightarrow (\mathbf{GF}p0)))$ with environment propositions $\{a, b\}$. The missing edge leads to a false state. The list of monitors in the environment states represents the ordering of the successors of the monitors from state 1. The safety formula of M3 and M4 in state 1 is made false by the choice of the environment player.

Definition 11. *Let φ be a formula the obligation set feature is defined as follows:*

$$osf(\varphi) = max_{s \in os(\varphi)} \begin{cases} 1 & \text{if } s = \emptyset \\ \frac{|\{a | a \in s \wedge a \in systemPropositions\}|}{|s|} & \text{otherwise} \end{cases}$$

If a member of the obligation set is empty, this means that **true** was part of the formula and the formula is always true, no matter of the actions.

We can also use the obligation set to find edges, which improve one of the members of the obligation set. This can be useful, if the master formula does not change along an edge. Then, the obligation set can give a hint on what will satisfy the monitors. The consequence of failing a monitor is removing it from the list. If it originates from a subgoal, which needs to hold eventually, the monitor reappears at the end ot the list. Such an example can be seen in figure 3.2. The transition to state 2 lets monitor 5 fail. Consequently,the monitor is removed, but since it needs to hold eventually is shows up again at the end of the list. Now, the environment player can repeat the same actions as before to reach state 1 again and the circle can be repeated. Since 4 is a losing color for the system player, the environment player wins. State 3 on the other hand allows the system player to progress. This is also reflected by the obligation set for φ. It contains $\{\neg p0, p1, \neg a, \neg b\}$, but no set for $\{p0, \neg p1, \neg a, \neg b\}$, because this does not lead the system player to success for the reason explained. Since the environment player played neither a nor b, the system player can try to play $\neg p0$ and $p1$ to make progress. The edge represented by this choice will have value 1 in the *improveObligationSet* feature, the other edges will have value 0.

Definition 12. *Let s be a system state with environment choice $env(s)$ and master formula φ and let e be an edge with propositions $prop(e)$ starting at s. The improveObligationSet feature is defined as follows:*

$$improveObligationSet(e) = \begin{cases} 1 & \textit{if } \exists o \in os(\varphi).o \subseteq prop(e) \cup env(s) \\ 0 & \textit{otherwise} \end{cases}$$

It is important to notice that this feature is risky for formulas like $(\mathbf{G}\neg d \wedge \mathbf{F}\neg b) \vee \mathbf{XF}(d \wedge b)$ with environment propositions b. If the environment player chooses to play b, the improveObligationSet feature suggests for the system player to play d. This will fail $(\mathbf{G}\neg d \wedge \mathbf{F}\neg b)$ and the environment player can win. Overall, the feature might lead to unexpected results, if the master formula changes along an edge. If it stays the same, it would be possible to switch to a different member of the obligation set later on.

Fail Set

A similar set to the obligation set can be computed to identify failing formulas. We call this a fail set. Here, the goal is to find a set of propositions, that would make a formula false in the next step, if they were played. This can be useful to decide, if the environment player can make a monitor fail in the next step by violating the safety formula. The environment player would most likely follow through with it, because failing a monitor leads to a color in favor of the environment player.

Definition 13. *Let φ be a formula. We define $os(\varphi)$ as follows:*

$$\begin{aligned}
fs(\mathbf{true}) &= && \{\} \\
fs(\mathbf{false}) &= && \{\{\}\} \\
fs(a) &= && \{\{\neg a\}\} \\
fs(\neg a) &= && \{\{a\}\} \\
fs(\varphi \wedge \psi) &= && os(\varphi) \cup os(\psi) \\
fs(\varphi \vee \psi) &= && \{\varphi_1 \cup \psi_1 | \varphi_1 \in os(\varphi) \wedge \psi_1 \in os(\psi) \\
& && \wedge \nexists i \in AP : ((i \in \varphi_1 \wedge \neg i \in \psi_1) \vee (\neg i \in \varphi_1 \wedge i \in \psi_1))\} \\
fs(\mathbf{X}\varphi) &= && \{\} \\
fs(\mathbf{F}\varphi) &= && \{\} \\
fs(\mathbf{G}\varphi) &= && fs(\varphi) \\
fs(\varphi\mathbf{U}\psi) &= && fs(\varphi \wedge \psi) \\
fs(\varphi\mathbf{W}\psi) &= && fs(\varphi \wedge \psi) \\
fs(\varphi\mathbf{R}\psi) &= && fs(\psi) \\
fs(\varphi\mathbf{M}\psi) &= && fs(\psi)
\end{aligned}$$

The fail set feature is computed similar to the obligation set feature described in 11.

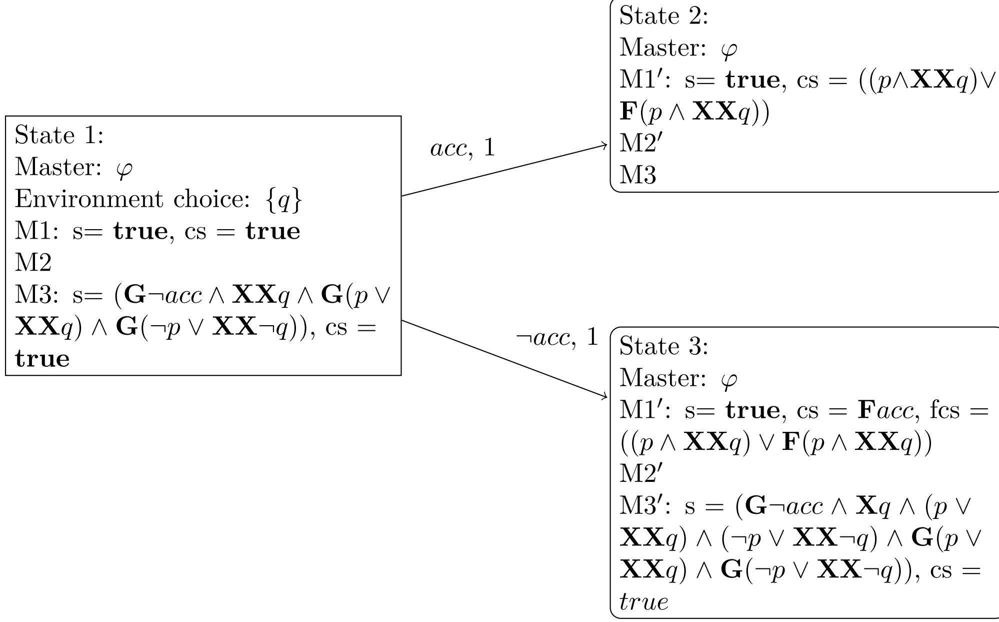

Figure 3.3: This is part of a game generated from $((G(F((p) \rightarrow (X(X(q)))))) \rightarrow (G(F(acc))))$ with environment propositions $\{p, q\}$ and system proposition acc. Monitor $M2'$ is the updated version of monitor $M2$ according to the edge choice and *fcs* is an open co-safety goal that has to hold at some point.

3.2.3 Progress in a Monitor

For the system player to win, at least one monitor needs to be fulfilled infinitely often. For this reason, it is important to the system player to make progress in the monitors. This is not captured when only taking the features of the monitors into account.

For example, in figure 3.3, the first monitor of state 1 is fulfilled by the environment player and will succeed. The system player has now the chance to play acc or $\neg acc$. If he chooses acc, the co-safety formula of the first monitor will be rather complicated, as one can see in state 2, and the last monitor will fail. State 3, on the other hand, has a simpler co-safety formula in monitor 1 and keeps the last monitor. Even though state 3 seems to be the way to go when only using the previous features, state 2 is actually better in regard of a winning strategy, since monitor 1 is progressed. Even though monitor 3 does fail, it will not output a bad priority, since the higher ranked monitor 1 did succeed. In order to teach the SVM to progress monitors whenever possible, we add some features, which will be described below. Since these features are computed on monitors, the functions described in section 3.1.1 can be used again to compute the feature for a state.

However, all of the described features have a similar meaning, so using them in the same SVM might lead to unexpected results due to the dependency between the features.

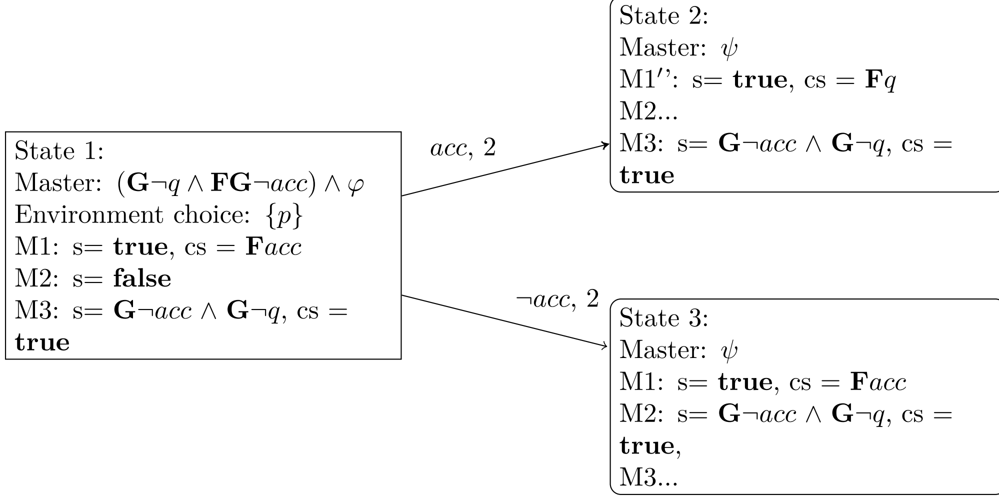

Figure 3.4: This is part of a game generated from $(((\mathbf{G}((p) \rightarrow (\mathbf{F}(q)))) \wedge (\mathbf{G}((\neg(p)) \rightarrow (\mathbf{F}(\neg(q)))))) \leftrightarrow (\mathbf{G}(\mathbf{F}(acc))))$ with environment propositions $\{p, q\}$ and system proposition acc. Monitor $M2$ in state 1 is made false by the environment choice. The monitor 2 of state 2 and 3 of state 3 are not important for the example, so they are abstracted away. States 2 and 3 have the same master formula. Monitor 1 does not fail along any edge. In state 2 it has progressed to check the next co-safety formula.

Co Safety success

The first idea for measuring the progress withing a monitor is to use the percentage of succeeding co-safety formulas along an edge.

Definition 14. *Let m be a monitor of state s with co-safety and f-co-safety formulas $cosafety(m)$ and let $e = (s, \nu, t)$ be an edge. The percentage of succeeding co-safety formulas is then computed as*

$$percentageSuccCoSafety(m, e) = \frac{|\{c | c \in cosafety(m), c \text{ } succeeds \text{ } along \text{ } e\}|}{|cosafety(s)|}$$

This feature can be implemented using the distance between the two indices of s and t. In order to find matching monitors, we use a heuristic based on the position in the list and the safety formula. Starting at the lowest position in both, s and t, one checks if the safety formula of the current monitor in s matches the formula of the current monitor in t. If this is not the case, one can assume that the monitor in s failed and go to the next in the list of s, while not changing in t. This allows to compute matching monitors fast.

A drawback of the feature *percentageSuccCoSafety* is that it treats failing monitors and monitors without any improvement the same way by giving them a value of 0. We

improved on that by giving failing monitors, found as described above, a value of -1. As one can see in figure 3.4, there are special cases, where monitors originate in formulas of the form $FG\varphi$. Failing these monitors has no large impact, since they will reappear in the next step. For this reason, reappearing monitors can get a different value, e.g. -0.5 or even 0.

Binary Progress Feature and Charging Level

The above approach does not take newly appearing monitors into account. To capture that, we add a feature to represent the charge value of a monitor. Here, we view each already fulfilled co-safety goal of a monitor as some load the monitor has. If all of the features are fulfilled, then the monitor is fully charged and it will succeed and output a good priority. This removes the entire charge and the monitor starts checking its goals again.

Definition 15. *Let m be a monitor with co-safety and f-co-safety formulas $cosafety(m)$ and let $checked(m)$ be the already checked co-safety formulas. Then:*

$$chargeLevel(m) = \frac{|checked(m)|}{|cosafety(m)|}$$

The feature *percentageSuccCoSafety* can be seen as the change value for the $chargeLevel$ feature.

Since we do not have information about the progress of each monitor within the $chargeLevel$ feature, we added a binary feature *doesProgress* indicating if a monitor makes any progress along an edge. It checks, whether $percentageSuccCoSafety(m, e) > 0$ for an edge e and monitor m. As stated, the feature is binary, so it does not distinguish between failing monitors and monitors without progress.

3.2.4 Number of Monitors

We also define a feature counting the number of monitors for a state. Our assumption is that a state with many monitors might have a higher chance to fulfill one of them than a state with a lower number of monitors. This is of course just a heuristic. If a state has only one monitor which one only depends on the system player, it is more useful than having many monitors depending on the environment player. However, a large change in this feature along an edge might still be an indicator for a major change in the master formula.

3.2.5 Combination of the Parts of a Monitor

The co-safety and safety formulas of a monitor are not independent of each other, since an easy co-safety formula does not help the system player, if the environment player can violate the safety formula.

For example, let e be an environment proposition and s be a system proposition. The monitor with safety-formula $\mathbf{G}e$ and co-safety formula $\mathbf{GF}s$ is not reliable for the

system player, since the environment player can fail it. However, the co-safety formula only depends on the system player and might lead the SVM to take actions towards satisfying it at the expense of failing another monitor.

To solve this problem, we implement a basic version of a feature combining safety and co-safety formulas:

Definition 16. *Let $f : Formula \to \mathbb{R}$ be a feature taking a LTL formula as input and let $w \in \mathbb{R}$ be a weight. We define the combined monitor feature $comb : \mathbb{M} \to \mathbb{R}$ as follows:*

- *If the safety formula of the monitor is false, a default value indicating the failing of the monitor is returned, e.g. 0.*
- *Otherwise, let $v = min_{c \in cosafety(m)} f(c)$ be the minimum of the feature on all co-safety formulas of the monitor.*
  - *If the safety formula can be turned false by the environment player in the next step, we return v*
  - *Otherwise, return wv*

The function $cosafety(m)$ returns the co-safety and f-co-safety formulas of a monitor m.

As before, taking the maximum value of the feature over all monitors is possible for features, for which a low value indicates a positive event.

The weight ensures that monitors, which are more likely to succeed get a higher value for the feature. If the environment player can fail a monitor in the next step, this monitor might not be reliable, so it receives a lower value. Still, the monitor might be useful. Assume, for example, the monitor m_2 originates from $\mathbf{FG}a$ and there is a monitor m_1 with safety formula **true** and co-safety formula $F\neg a$. If the environment player decides to play $\neg a$, the monitor m_1 will succeed and give a good priority, which cannot be overwritten by the failing of monitor m_2, since m_1 is placed before m_2 in the list. In this case, the environment player might want to choose a and, consequently, the co-safety formulas of m_2 are important for the system player.

The described combination of features can be applied to the successor state of an edge to decide on the quality of its monitors. As inner features for the co-safety formulas, we use *trueness* and the obligation set feature, since they have some semantic meaning. In principle, it is also possible to use other functions.

3.3 Normalization of Features and Relation between Feature Values of States

Another idea for the features is the normalization of the unbounded features. Unbounded features like *height* need to be mapped to a bounded domain before applying the SVM. Backs used the function $x \mapsto arctan(x/10)$ to do so [1]. This function does not take into

account that a high value in the *height* feature of the successor t of an edge might be caused by a high value in the starting point s. If this is the case, the value $height(t)$ might have even decreased compared to $height(s)$, but will still be considered high. A first approach is to divide the value $height(t)$ by $height(s)$. However, this feature will still be unbounded. Using min-max normalization is a different approach to that problem.

Definition 17. *Let $x \in [x_{min}, x_{max}] \subset \mathbb{R}$. The min-max normalization is then defined as:*

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

For unbounded features, this definition cannot be applied directly, since we do not know the maximum or minimum value and it would be inefficient to compute all feature values for every state of the game. This would not allow for an on-the-fly approach later on. Therefore, we consider only the transitions (s, ν, t) starting at a state s. Then we compute the set of values of the unbounded feature f for all states related to s: $V = \{f(t) | \exists \nu.(s, \nu, t) \in T\} \cup \{f(s)\}$. This set can than be used to normalize the feature values within and yields bounded values for the feature.

This approach still allows for a comparison of the feature values for different successors of a state, since the features relating to them are all normalized using the same x_{min} and x_{max}. Additionally, the SVM will not need to distinguish between a transition (s, ν, t) with $height(s) = 50, height(t) = 49$ and a transition (s', ν, t') with $height(s') = 10, height(t') = 9$. In both cases, the height decreases by 1, so the normalization will scale both states to the same values. Notice, that there might be some problems, if one edge has an outlying value, for example if the state s also has a transition to the **false**-sink, which has $height(\mathbf{false}) = 1$. In this case, the normalization does not have any advantages to using *arctan*.

3.4 Separating the Learning Data

In the unpublished work [2], Backs suggested to train two different SVMs, one on transient states and one on recurrent states. The idea is that winning strategies of different classes of states behave differently. In the thesis, we suggest three more subdivisions of the learning data. For the first one, we consider the number of monitors in each state and its successors. The second one improves on that and also considers the change in the master formula. Our last idea is not related to the previous ones, but is based on safety and co-safety formulas. These two classes of formulas do not have monitors in the games created from them and we train SVMs for them separately.

3.4.1 Different Classes for Transient and Recurrent States

In the case of transient and recurrent states, it can be assumes that the transient states focus more on improving the finite part of the formula and recurrent states focus on the infinite part. Thus, the SVM for transient states is trained with the features defined on

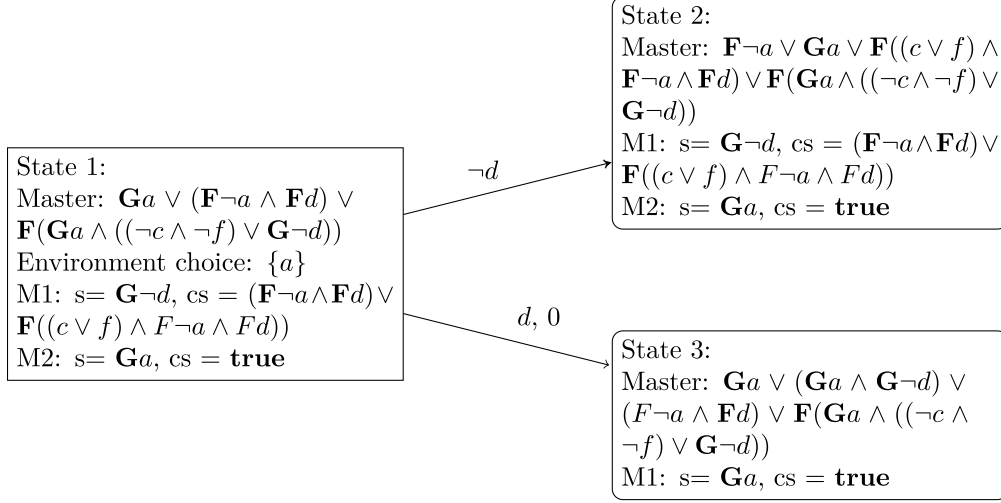

Figure 3.5: This is part of a game generated from $\mathbf{X}\neg\mathbf{G}(\mathbf{F}\neg a \oplus ((c \vee f) \wedge \mathbf{F}d))$ with environment propositions $\{a, c\}$ and system propositions $\{d, f\}$.

the master formula and the SVM for recurrent states is trained on the features for the monitors.

The problem with this approach is that deciding if a state is transient or recurrent is hard, if we do not have a winning strategy and, since we want to compute a winning strategy for a game using the SVMs, we do not have it to decide, which SVM to use. Therefore, we define a a transient state to be a state in a game, that cannot reach itself using all edges of the game. Such a state focuses most likely on the finite part of the formula. However, this still does not allow for an on-the-fly approach later on, since we would need to compute the entire game first.

3.4.2 Dividing Based on The Number of Monitors

Following the idea that some states focus on the master formula, whereas other states depend on the monitors, we map each state to one of the following three classes:

Class 1: This class contains states which do not have monitors and states for which no successor has a monitor.

Class 2: States with monitors belong to this class, if some of the successors of the state also contain monitors and some do not.

Class 3: In this class are only states with monitors, where all successor states have monitors, too.

For states in the first class, one can focus on the master formula. For the second class, the monitors need to be taken into account, as well. However, using only the monitors

is not enough, because some of the successors depend on the master formula. For the last class, we assume that the monitors are more important.

During our experiments, we found some states of the second and third class, for which our assumptions are too restrictive. In class 2, there are states, like state 1 in figure 3.2, which have an edge to the **false**-sink. For all the other successors, the master formula does not change and the choice of the edge only depends on the monitors. Before sending a request to the SVM , the edge to the **false**-state is removed, because it is trivial not to take it. For such a state, the underlying assumptions are not met and the monitors are more important than the master formula of the successors.

For class 3, there are some states, as shown in figure 3.5, for which the master formula changes, in contrast to the assumptions for this class. It might be useful to additionally consider the master formula for those states.

3.4.3 Dividing based on Change in the Master Formula

To solve the above problems, we divide states based on the change in the master formula, with exception of class 1. This class has only small errors in the experiments, so we keep it as it is. Instead of class 2 and class 3 we defined classes 4 and 5. Both contain states with at least one monitor and at least one successor which has again at least one monitor. Additionally, the following conditions hold:

Class 4: States contained in this class have at least one successor with a different master formula.

Class 5: For states in class 5, the master formula stays the same along all transitions starting at the state.

To avoid problems with the **false**-sink, we remove edges to it before classifying a state.

3.4.4 Training Different Support Vector Machines for Safety and Co-Safety formulas

A co-safety formula defines a property, that wants to reach a certain state, safety formulas want to avoid an unpleasant event. The monitors of games derived from co-safety and safety formulas are optimized away, because the above properties can be checked without the help of an additional monitor only by the master formula. Since they have a different structure than the other games and both aim for different goals, we suggest to train one SVM for co-safety formulas and one for safety formulas in addition to the previously described cases.

3.5 Additional Obligation Set Check

The idea of this section is to use the obligation set to detect states, which can be easily won by the system player. If there is a set of system propositions, that would satisfy a formula, if it would be played all the time, the system player has a winning strategy

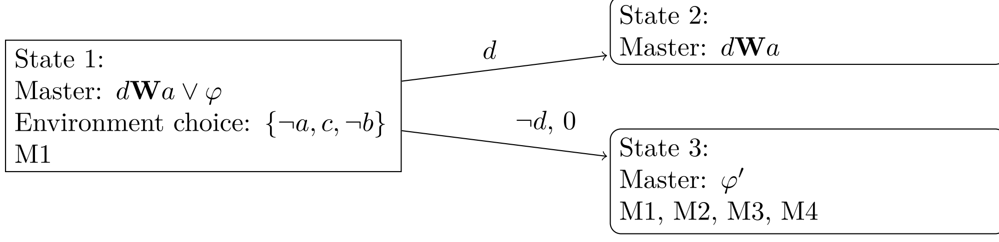

Figure 3.6: This is part of a game generated from $b \leftrightarrow (\mathbf{XX}c\mathbf{U}(d \wedge \mathbf{F}c)))\mathbf{UX}(d\mathbf{W}a)$ with environment propositions $\{a, b, c\}$ and system propositions $\{d\}$. State 1 has one monitor, state 3 has four monitors. The formula φ turns to $d\mathbf{W}a$, if d is played and φ' otherwise. The missing color of the above edge is optimized away by the implementation.

and we do not need to ask an SVM for advice. Backs already implemented a similar check for what he called trivial states [1]. These are states that either have an edge to a **true**-sink or have only one edge which does not lead to a **false**-sink. We improved on that by adding a check for an obligation set before sending a query to the SVM .

This idea can be particularly helpful for situations like the one shown in figure 3.6. State 2 has no monitor, but a rather simple master-formula. The system player can win by always playing d. However, the SVM often aims for state 3 and tries to work with the monitors, since it assumes that all monitors failed in state 2. In this case, the obligation set test improves the result.

We implement the check by first asking for an obligation set controlled by the system player for the master formula of the starting state. If there is one, we play the actions of this set. This is necessary, because of master formulas like $\mathbf{FG}a$ with system proposition a. Both successor states of the edges with letter $\{a\}, \{\neg a\}$ will have an obligation set of the form $\{a\}$, so only checking for the obligation sets of successors is not enough. If the state we currently try to solve does not have such an obligation set, we check for its successors and only ask the SVM for advise, if none of them has such a set.

This approach cannot detect cases where two obligation sets contradict each other. For example a master formula $F_1 = \mathbf{G}a \vee (d \wedge \mathbf{F}\neg a)$ with system proposition d and environment proposition a forces the environment player to fulfill one of the disjuncts, assuming the system player plays d now. The obligation set for such a formula contains the sets $\{a\}, \{\neg a, d\}$. Nevertheless, we cannot compare them directly, since the formula $F_2 = \mathbf{G}a \vee \mathbf{G}(\neg a \wedge d)$ has the same obligation set as $\mathbf{F}_1$, but is not satisfied by playing a at the beginning and $\neg a$ later. In other words, the environment player can fail the formula, even though the obligation set is the same as the obligation set of F_1.

To overcome this, we keep track of a type for each obligation set. The first type is *Globally*, meaning the set needs to be played all the time. *Finally* is used for a set which satisfies the formula, if it is played at an arbitrary time and *Now* fulfills the formula, if

it is played now.

Definition 18. *An extended obligation set is of the form* $\{(os_1, type_1), ..., (os_n, type_n)\}$ *with* $type_i \in \{Finally, Globally, Now\} = Types$ *and* $os_i \in 2^{AP}$*. Let* φ *be a formula. We define* $eos(\varphi)$ *as follows:*

$$
\begin{aligned}
eos(\mathbf{true}) = \quad & \{(\{\}, G)\} \\
eos(\mathbf{false}) = \quad & \{\} \\
eos(a) = \quad & \{(\{a\}, Now)\} \\
eos(\neg a) = \quad & \{(\{\neg a\}, Now)\} \\
eos(\varphi \land \psi) = \quad & \{\varphi_1 \sqcup \psi_1 | \varphi_1 \in eos(\varphi) \land \psi_1 \in eos(\psi) \\
& \land \nexists i \in AP : ((i \in \varphi_1 \land \neg i \in \psi_1) \lor (\neg i \in \varphi_1 \land i \in \psi_1))\} \\
eos(\varphi \lor \psi) = \quad & eos(\varphi) \cup eos(\psi) \\
eos(\mathbf{X}\varphi) = \quad & \{(\psi, Globally) | (\psi, t) \in eos(\varphi)\} \\
eos(\mathbf{F}\varphi) = \quad & \{(\psi, F(t)) | (\psi, t) \in eos(\varphi)\} \\
eos(\mathbf{G}\varphi) = \quad & \{(\psi, Globally) | (\psi, t) \in eos(\varphi)\} \\
eos(\varphi \mathbf{U} \psi) = \quad & eos(\psi) \\
eos(\varphi \mathbf{W} \psi) = \quad & eos(\mathbf{G}\varphi) \cup eos(\psi) \\
eos(\varphi \mathbf{R} \psi) = \quad & eos(\mathbf{G}\psi) \cup eos(\varphi \land \psi) \\
eos(\varphi \mathbf{M} \psi) = \quad & eos(\varphi \land \psi)
\end{aligned}
$$

Where $F : Types \to Types$ *means that the set has to hold finally and is defined as:*

$$
\begin{aligned}
F(Finally) \quad & = Finally \\
F(Globally) \quad & = Globally \\
F(Now) \quad & = Now
\end{aligned}
$$

The operator $\sqcup$ *defines the union of two members of the extended obligation set as:*

$$
\begin{aligned}
(\varphi_1, Now) \sqcup (\varphi_2, Now) \quad & = (\varphi_1 \cup \varphi_2, Now) \\
(\varphi_1, Now) \sqcup (\varphi_2, Globally) \quad & = (\varphi_1 \cup \varphi_2, Globally) \\
(\varphi_1, Now) \sqcup (\varphi_2, Finally) \quad & = (\varphi_1 \cup \varphi_2, Now) \\
(\varphi_1, Finally) \sqcup (\varphi_2, Now) \quad & = (\varphi_1 \cup \varphi_2, Now) \\
(\varphi_1, Finally) \sqcup (\varphi_2, Globally) \quad & = (\varphi_1 \cup \varphi_2, Globally) \\
(\varphi_1, Finally) \sqcup (\varphi_2, Finally) \quad & = (\varphi_1 \cup \varphi_2, Finally) \\
(\varphi_1, Globally) \sqcup (\varphi_2, Now) \quad & = (\varphi_1 \cup \varphi_2, Globally) \\
(\varphi_1, Globally) \sqcup (\varphi_2, Now) \quad & = (\varphi_1 \cup \varphi_2, Globally) \\
(\varphi_1, Globally) \sqcup (\varphi_2, Now) \quad & = (\varphi_1 \cup \varphi_2, Globally)
\end{aligned}
$$

We can then check for contradictions between *Finally* sets and every other set and *Now* sets with other *Now* sets. *Globally* sets cannot be compared with each other, because an environment player with action a could easily win the formula $\mathbf{G}a \vee \mathbf{G}\neg a$ or the formula $a \vee \mathbf{G}\neg a$.

The check for contradictions works as follows: We take all *Finally* sets and all *Now* sets of $eos(\varphi)$ and add them to a list L. We then check for contradictions recursively. If there is only one environment variable v left, we set it to true and remove v from all sets, that contain it. We remove all sets containing $\neg v$ from L. Then we check, if the remainder of L does contains a set only depending on the system player. If so, we do the same thing for the assumption v is false. If both paths have a possible set, we return true, otherwise false, because a set in both cases means that the system player can fulfill one of the obligation sets, no matter what the environment player does.

For example, assume $L = [(\{a, b\}, t_1), (\{a\}, t_2)]$ with environment proposition a. We set a to true and what remains is $L' = [(\{b\}, t), (\{\}, t_2)]$. Both members can be easily fulfilled by the system player, the first by playing b, and the second does not have any restrictions. We now check for setting a to false. Then $L' = []$, which means, there is no set, that could be satisfied. Consequently, we return false.

If there is more than one variable, we call the function recursively. We first set the value of the environment proposition v we want to check to true and construct L' as above. Then we call the function again on L' with the next environment variable as parameter. If it returns true, we set v to false, update L' and check the result. If both paths return true, we return true, else false.

After we went through the list of *Finally* and *Now* sets, we call the same formula on the list of *Finally* sets combined with one *Globally* set at a time. This will check that either the environment player plays always the *Globally* set or one of the finally set will eventually be true.

Adding more than one *Globally* set at the same time is not possible, since the formula $\mathbf{G}a \vee \mathbf{G}\neg a$ will lead to the extended obligation set $\{(\{a\}, Globally), (\{\neg a\}, Globally)\}$. Comparing these two sets would suggest that the system player always has an option to play. However, the word $(\{a\}\{\})^\omega$ will not satisfy the formula.

This method will not detect all contradictions. For example, the formulas $(Fa \wedge G\neg b) \vee (G(a \leftrightarrow b))$ and $GFa \vee GF\neg a$ only contain *Globally* sets in the extended obligation set and these will not be compared with each other. Furthermore, every part of a formula containing $\mathbf{X}$ will be a *Globally* set. This is necessary due to formulas of the form $\mathbf{X}a \wedge \mathbf{XX}b$. Here, the two parts refer to different time steps, so they cannot be compared. In order to solve this, one could try to add more types of sets.

4 Experimental Results

This section describes the experimental results. Firstly, we give an overview of the test data, define the metric we use to evaluate the different SVMs and give details about the training process. Afterwards, we follow the same structure as chapter 3. We start by discussion the different functions for monitors and methods for change values, then continue with the evaluation of additional features and the normalization of features, and finish with tests on the separation of the learning data. After the discussion of improvements on the SVMs, we present the results of applying the additional obligation set check.

4.1 General Information on the Experiments

4.1.1 Test Data

The test and training date used in this section are based on the automata stored in [2]. This way, the results are as recomputable and as comparable as possible. There are still slight differences in each test run, because the strategy iteration algorithm, which computes the training data and a winning strategy from the suggestion of the SVM, depends on the ordering of the edges for a state. This ordering is not deterministic in Java, because it is implemented using a set.

The data was already used in [1] to evaluate the results. It consists of the classes *large-co-safety, small-co-safety, small-p-co-safety, large-safety, small-safety, small-p-safety, large, small, lilydemo, ltl2dba, ltl2dpa.* The first eight classes of formulas were generated randomly and each consist of 100 formulas. The first 75 are used for the training of the SVMs, the last 25 for the tests in this chapter. The classes with *safety* and *co-safety* in the name contain safety and co-safety formulas, respectively. The letter p in the name states, that the class is nearly safety or co-safety.

The last three classes are taken from the SYNTCOPMP dataset. The classes *lilydemo* and *ltl2dpa* consist of 22 formulas, *ltl2dba* contains only 19 formulas. Here, the first half of the data is used for training the SVMs and the second half for testing.

It is interesting to know that in the games generated from formulas from the class *ltl2dba* every state contains at least one monitor. States in games from the *safety* and *co-safety* class do not contain any monitor.

4.1.2 Metric

For the evaluation of different SVMs, we consider only states which have a winning strategy. For each game, we count the number of these states which the SVM maps

to an edge not in the winning strategy and divide it be the number of all states with a winning strategy. We call this the error for winning states of one game. In order to minimize this value, we compute the winning strategy by applying a strategy iteration algorithm to the strategy returned by the SVM.

To get the error for winning states for an entire class of games, we build the average over the error of all games within it.

As a side node on the strategy iteration algorithm, we want to mention, that there are two ways to compute the winning strategy using the implemented algorithm. The strategy iteration algorithm can either only return the best possible edge or all edges which can be added to the initial strategy. In figure 3.1 for example, the best edge is the edge with priority 1, even though the other edge is also acceptable in this simple example. Therefore, using only the best edge results in higher value for the error for winning states than necessary. If the strategy iteration algorithm returns all possible edges for the winning strategy, both edges of figure 3.1 are included. Consequently, we use the second option for the evaluation.

4.1.3 Training and Evaluation of the SVMs

In this section, we discuss decisions regarding the training and evaluation of the SVMs.

For the training of the SVM, we exclude trivial states from the training data. As state is trivial, if it has an edge to a **true**-sink or only one edge not leading to a **false**-sink. The reason is that the SVM is never asked for advice for such states, because they are solved directly.

Additionally, we remove two optimizations suggested in the implementation [14] of the translations from LTL to DPA for training and evaluating the SVMs. The first heuristic tries to produce games as small as possible. If the complement of a formula produces a smaller game, this game is returned and the roles of the system and environment player in this game are reversed. As a consequence, the system player then has to fail the formula in the label of a state. Since the SVM is not aware of that fact and treats all games the same way, the learning is affected, so we remove the optimization.

The second optimization compresses the colors of edges. Consider the state in figure 3.1. There is no color of value 2 possible, because the second monitor cannot fail, since the safety formula is **true**. The color compression would then map color 3 to color 1 and combine both edges. In general, the optimization creates a list of all colors occurring in a strongly connected component. Afterwards, gaps between them are removed. E.g. for the occurring colors $2, 6, 7$, color 2 would be mapped to 0 and color 6 to 2, because it is the first free even color. Color 7 would be mapped to 3. If there is no odd color between two even colors afterwards or no even color between two odd colors, the two colors are combined and the remaining colors are again shifted forward. For the above example, the optimization would return $0, 1$ as possible colors.

This optimization is problematic when computing important monitors based on the color of the edge as described in section 3.1.1. If the optimization is applied in the above example to an edge originally of color 7 the new color of the edge is 1. According to

the definition of important monitors, we would only consider the monitor at position 0, but actually the first four monitors should be taken into account. For that reason, we remove the color compression.

4.2 Evaluation of the Different Functions for Monitors

In this section we compare the different functions described in section 3.1.1 and the influence of using all monitors or only the important once as described in section 3.1.1. All SVMs are trained on all states and contain the features *trueness, height, tempOps, disjuncts, systemControl* applied to the master formula, together with the respective change value of the feature along the edge, as in [1]. We also include the features for the monitors as presented in section 2.5.2, meaning each of the master features *trueness, height, tempOps, disjuncts, systemControl* is applied separately to the fields *current, next, safety* of a monitor. Afterwards, we use the functions to construct one value for each state. For the functions *waverage* and *monitorList*, we used the *average* on the next co-safety goals of a monitor to get a single value for it. The change features for the monitors are generated by first computing the value of each state and then subtracting the value of the successor of the edge from the value of the starting point, as suggested by Backs [2].

We include the SVM trained using only the master features for comparison. As stated in section 2.5.2, this SVM is named *original*.

The tables 4.1, 4.2 and 4.3 show the error for the winning states for each SVMs using the different functions with important monitors and with all monitors. The values of these tables are all generated during the same test run.

As one can see, for the function *average* the SVM using only the important monitors performs slightly better. This matches the assumption, that monitors ranked at a higher position than the color of the edge are not as important for the result. For the function *progress*, the results are more diverse. This is due to the fact, that goal of the function is to find the most promising monitor, and this monitor can in general be ranked at a higher position in the list. However, non of the functions could outperform the use of only the master features. We assume, this is due to poor generalization of the SVM . The results of the training accuracy can be seen in figure 4.4. It did increase for all suggested functions. As already discussed in section 3, the features used here, do not entirely match the meaning of the monitor formulas, since e.g. the position is important and the different formulas of a monitor depend on each other.

In table 4.2 one can see the results for weighted average, when using different weights. We run tests with weights $1, 2, 0.5$, because all of the bounded features have a maximal value of 1, so we use values close to that. The weights 0.5 and 2 performed similarly. Further analysis shows, that the weight 0.5 performs slightly better on states with changes in the master formula and the weight of 2 has a very slight advantage on states which depend more on the monitors. This makes sense, because the weight 2 gives more priority to monitors ranked in the front of the list. Additionally, one can see, that using only the important monitors is of advantage here, as well.

category	original	average-color	average	progress-color	progress
large-co-safety	0.01	0.01	0.01	0.01	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.04	0.06	0.07	0.04	0.01
large-safety	0.06	0.06	0.06	0.06	0.01
small-safety	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.01	0.05	0.05	0.01	0.00
large	0.05	0.10	0.10	0.10	0.06
small	0.03	0.05	0.13	0.07	0.07
lilydemo	0.20	0.11	0.12	0.19	0.25
ltl2dba	0.14	0.18	0.21	0.09	0.09
ltl2dpa	0.13	0.17	0.14	0.14	0.19
Average	0.04	0.06	0.07	0.05	0.05

Table 4.1: Error for winning states for the strategies suggested by the SVMs. The word *color* in the name of an SVM means that only the important monitors where used.

The results for the *monitorList* function can be seen in table 4.3. We use the thresholds of 3 and 5, because 3 is approximately the average number of monitors in the test data for states, which have monitors, and 5 is a slightly higher value to see, if including more monitors than the average turns out to be useful. We can not increase the number further due to computational limits. In this case, using only the important monitors, does not improve the results. We assume, that this is due to the fact that the way we detect this monitors is only a heuristic and the SVM can learn a more detailed use of the monitors, if it is given as a list. Furthermore, there is no large difference between using 5 or 3 monitors, except, if one decides not to cut according to the edge color. Again, the function performs not as good as using only the master features. Nevertheless, the *monitorList* function has the highest training accuracy, as shown in figure 4.4.

For the future experiments, we proceed with weighted average with weight 2, since it did perform best, and *monitorList* with length 5 due to the high training accuracy.

category	original	waverage-w1-color	waverage-w1	waverage-w0.5-color	waverage-w0.5	waverage-w2-color	waverage-w2
large-co-safety	0.01	0.01	0.01	0.00	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.04	0.10	0.11	0.01	0.01	0.01	0.00
large-safety	0.06	0.06	0.08	0.01	0.02	0.01	0.02
small-safety	0.00	0.00	0.01	0.00	0.00	0.00	0.00
small-p-safety	0.01	0.05	0.14	0.03	0.03	0.03	0.03
large	0.05	0.09	0.12	0.04	0.05	0.04	0.04
small	0.03	0.12	0.18	0.01	0.01	0.01	0.01
lilydemo	0.20	0.26	0.22	0.13	0.16	0.14	0.16
ltl2dba	0.14	0.14	0.15	0.09	0.11	0.08	0.18
ltl2dpa	0.13	0.35	0.21	0.11	0.17	0.10	0.14
Average	0.04	0.09	0.10	0.03	0.03	0.03	0.03

Table 4.2: Error for winning states for the strategies suggested by the SVMs. Again, the word *color* in the name of an SVM means that only the important monitors where used. The w in the name indicates the weight.

name	master-branch	monitorList-3-color	monitorList-3	monitorList-5-color	monitorList-5
large-co-safety	0.01	0.01	0.00	0.01	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.04	0.08	0.04	0.10	0.04
large-safety	0.06	0.06	0.11	0.06	0.08
small-safety	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.01	0.05	0.04	0.14	0.04
large	0.05	0.09	0.08	0.13	0.09
small	0.03	0.03	0.05	0.16	0.04
lilydemo	0.20	0.11	0.14	0.12	0.13
ltl2dba	0.14	0.35	0.34	0.39	0.35
ltl2dpa	0.13	0.19	0.17	0.25	0.21
Average	0.04	0.06	0.06	0.10	0.06

Table 4.3: Error for winning states for the strategies suggested by the SVMs. Again, the word *color* in the name of an SVM means that only the important monitors where used. The number in the name indicates the threshold.

master	progress-color	progress	monitor-List-3	monitor-List-3-color	monitor-List-5	monitor-List-5-color	avg
77.05	82.60	81.22	85.33	85.39	88.68	85.86	84.79
avg-color	wavg-1	wavg-1-color	wavg-2	wavg-2-color	wavg-0.5	wavg-0.5-color	
83.83	83.58	82.49	83.52	82.57	83.63	82.45	

Table 4.4: The training accuracy in percent for the different SVMs. The training accuracy is defined as the number of edges of the training data mapped to the class given to the SVM.

4.3 Change Features

This section contains the results for the different ways to compute the change value of a feature defined on monitors. For this, we will not use the stored automata, since they do not contain all information necessary to compute the exact successor of a monitor. The results can be seen in table 4.5. We only include the categories for which games contain states with monitors. On average, there is no large difference for the approaches.

For the *monitorList* function, the change value computed based on the position performed better than the one computed based on the exact successor. We assume, the reason is that the position based approach focuses, similar to the *monitorList* function, on the list instead of single monitors. The computation suggested by Backs preformed as good as the position based approach. However, since we want the SVM to learn a rule for states with monitors, when using features computed on the monitors, we continue with the position based approach, because it performs best on the class *ltl2dba*, where each state has at least one monitor.

For the weighted average, the approach suggested by Backs performs good for most of the classes, except for *small*. It seems, that the weighted average focuses more on the value of entire list than on a single monitor, so using a technique to find matching monitor does not add useful information. Furthermore, the weighted average uses the color. For the monitors above the color, it is not necessary to know exactly, which monitor improves, because it is more important to improve one of the monitors and not worsen the others.

For the following experiments, we use the position based change value for the *monitorList* function and the change over the value of the entire state for the weighted average.

category	monitor-List-5-ldba	monitor-List-5-pos	listmons-5-list	waverage-w2-list-color	waverage-w2-ldba-color	waverage-w2-pos-color
small-p-co-safety	0.02	0.01	0.02	0.02	0.03	0.03
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00
large	0.04	0.05	0.03	0.02	0.02	0.03
small	0.03	0.03	0.03	0.05	0.04	0.03
lilydemo	0.23	0.22	0.19	0.14	0.18	0.24
ltl2dba	0.11	0.10	0.15	0.17	0.18	0.17
ltl2dpa	0.12	0.09	0.08	0.12	0.09	0.11
Average	0.08	0.07	0.07	0.07	0.07	0.08

Table 4.5: Error for winning states for the different ways to compute the change features. The SVMs are the ones performing best in the last section. The *pos* in the name means that the change feature was computed based on the position in the list, *list* means that the feature was computed as suggested in section 2.5.2 and *ldba* is the computation based on the exact successor.

4.4 Additional Features

In this section, we analyze the different additional features described in section 3.2. We start by applying the features only to the master formula. The original SVM in [1] used *height, trueness, disjuncts, system control* and the *accepting* feature. We define the set of new features as *height, trueness, disjuncts, the feature defined on the obligation set, improveObligationSet, the feature defined on the fail set, number of monitors* and the *priority* feature. We exclude the accepting feature, since the SVM often learns unexpected weights, e.g. negative weights for either *accepting* or *priority*, when they are used in combination. This is due to the similar meaning of the features. The same holds for system control and the feature defined on the obligation set.

The results can be seen inf table 4.6. On average, there is no large difference between the two approaches. However, the SVM trained with the new features performs slightly better on the categories *large-safety*, *large* and *ltl2dpa* and was only worse for *lilydemo* and *ltl2dba*. The differences here are caused by states for which aiming for a lower priority of the edge holds no advantage over an edge with higher priority or the two edges suggested by the SVMs are similar and the strategy iteration algorithm decided on one of them.

The results for the different features applied to the monitors using the *monitorList* function can be found in table 4.7. The naming of the SVMs is as follows:

all The new features are applied to the master formula and each part of the monitors. Furthermore, both variants to measure the progress of a monitor, meaning the charge level with the binary progress indicator and the percentage of succeeding

category	master-original	master-new-Features
large-co-safety	0.00	0.00
small-co-safety	0.00	0.00
small-p-co-safety	0.01	0.01
large-safety	0.02	0.01
small-safety	0.00	0.00
small-p-safety	0.00	0.00
large	0.06	0.03
small	0.00	0.00
lilydemo	0.14	0.16
ltl2dba	0.12	0.18
ltl2dpa	0.11	0.06
Average	0.03	0.03

Table 4.6: Comparison of the error for winning states of a SVMs trained on the master formula only using the original or the new features.

co-safety formulas with a recharge value of -0.5, and the number of monitors are used. We also include the combination monitor features from section 3.2.5.

comb-charge This SVM uses only the combination features from section 3.2.5 and use the *chargeLevel* and the binary progress feature to measure the progress of each monitor. In addition, the new features are applied to the master formula.

comb-co-safety-succ-0 This SVM is similar to the above one, except for the progress of a monitor. Here, the percentage of successful co-safety formulas is used instead of *chargeLevel* in combination the binary feature. Failing monitors get a a value of -1 for *percentageSuccCoSafety* and reappearing failing monitors get a value of 0.

comb-co-safety-succ-0.5 Again, this is similar to the SVM directly above it. The difference is that reappearing failing monitors get a value of -0.5 in *percentageSuccCoSafety*.

newF-charge This applies the new features to the master formula and all parts of the monitors. Additionally, the charge level and the binary progress feature measure the progress of each monitor.

newF-co-safety-succ-0 This is similar to *newF-charge*, but the progress is measured using $percentageSuccCoSafety$ and the value for reappearing failing monitors is 0.

newF-co-safety-succ-0.5 Here, the value for reappearing failing monitors is -0.5.

newF-comb The SVM applies the new features to the master formula and all parts of the monitor and also considers the features computed using the combination of formulas of the monitors.

newF The new features are applied on the master formula and all parts of the monitors.

original The feature *accepting* is applied to the edges and the features *height, tempOps, disjuncts, trueness* and *systemControl* are applied to the master formula and all parts of the monitors.

The last two columns of table 4.7 show the results using the original features and the new ones on the master-formula and the monitors. As one can see, the results for the new features outperform the old ones. The reason for the similar average is the number of games in each category and the rounding of the numbers. The first 8 categories contain 25 games, *lilydemo* and *ltl2dpa* only eleven and *ltl2dba* only nine.

Adding the combination features to the new features does not have any advantage. The SVM with the combination features has unexpected weights for some of the features, like the weight for the combination of the fail set with trueness is negative for the monitor in the second position. On the other hand, the training accuracy for this SVM is 90.05%, whereas the SVM using only the new features has a training accuracy of 88.79%, so the bad results might be due to overfitting.

To compare the progress features, we trained two SVMs for each of them, one in combination with the new feature, and one with the combined features. For both cases, the percentage of successful co-safety formulas with a value of 0 for failing monitors which reappear in the next step performed best, but the difference is small. We assume, the reason is, that the *percentageSuccCoSafety* contains more information in itself than the *chargeLevel* and does not require as much learning from the SVM. Furthermore, failing a reappearing monitor is not a problem, so giving them a value of 0 seems reasonable.

In total, there is no clear best approach. For *large*, *small*, *lilydemo* and *small-p-co-safety* approaches using the combination features and the progress features performed best. For the classes *ltl2dpa* one should use the new features in combination with the progress or on their own. The differences in the *ltl2dba* category are caused by states, where keeping the formula of the second monitor does not lead to success. Such a state can be found in figure4.1. Here, the SVM with the combination features aims for the simpler second monitor using the edge *acc*, probably caused by the combination of fail set and trueness, and the SVM with the new features aims for progressing the safety formula by playing $\neg acc$.

The SVMs for the weighted average behave similar to the ones with the *monitorList* function. The results can be seen in figure 4.8. The difference is that giving failing monitors which reappear in the next step a value of -0.5 for the percentage of succeeding co-safety formulas is of general advantage here. The weighted average focuses more on the list of monitors as a whole, than on single monitors, and failing a monitor has an impact on the list, so it should be reflected in the features. However, the difference in the results for the values 0 and -0.5 is small.

In total, one can see, that the SVMs using the weighted average perform again slightly better than the ones with the *monitorList* function.

category	all	comb-charge	comb-co-safety-succ-0	comb-co-safety-succ-0.5	newF-charge	newF-co-safety-succ-0	newF-co-safety-succ-0.5	newF-comb	newF	original
large-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.02
large-safety	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
small-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
large	0.04	0.03	0.02	0.02	0.04	0.04	0.04	0.05	0.04	0.04
small	0.02	0.01	0.00	0.00	0.01	0.00	0.01	0.00	0.00	0.01
lily-demo	0.16	0.11	0.12	0.12	0.15	0.15	0.15	0.18	0.17	0.18
ltl2dba	0.16	0.19	0.18	0.19	0.16	0.16	0.16	0.16	0.16	0.19
ltl2dpa	0.09	0.12	0.12	0.12	0.08	0.07	0.08	0.09	0.07	0.12
Avg	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03

Table 4.7: Comparison of the error for winning states for the *monitorList* function with threshold 5. The change feature is based on the position.

category	all	comb-charge	comb-co-afety-succ-0	comb-co-safety-succ-0.5	newF-charge	newF-co-safety-succ-0	newF-co-safety-succ-0.5	newF-comb	newF	original
large-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
large-safety	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
small-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.03	0.00	0.00	0.00	0.03	0.03	0.03	0.03	0.03	0.03
large	0.05	0.02	0.02	0.03	0.02	0.03	0.03	0.03	0.03	0.02
small	0.01	0.00	0.00	0.00	0.01	0.01	0.01	0.02	0.01	0.02
lily-demo	0.15	0.10	0.09	0.09	0.13	0.08	0.08	0.10	0.10	0.13
ltl2dba	0.10	0.18	0.18	0.18	0.14	0.14	0.14	0.14	0.14	0.15
ltl2dpa	0.13	0.09	0.08	0.06	0.07	0.08	0.07	0.08	0.07	0.08
Avg	0.03	0.02	0.02	0.02	0.03	0.02	0.02	0.03	0.03	0.03

Table 4.8: Error for winning states when using the weighted average function with weight 2 and list based change features.

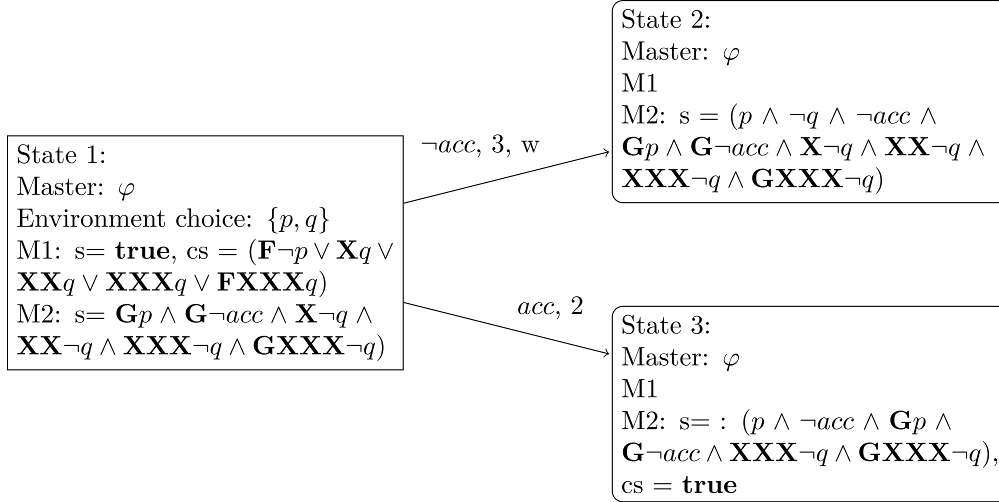

Figure 4.1: This is part of a game generated from $((\mathbf{G}(\mathbf{F}(p \rightarrow (\mathbf{X}(\mathbf{X}(\mathbf{X}q)))))) \leftrightarrow (\mathbf{G}(\mathbf{F}acc)))$ with environment propositions $\{p, q\}$ and system proposition acc. The keyword w denotes the winning edge

4.5 Normalizing the Features

This section contains the results for normalizing the unbounded features. They can be seen on table 4.9. The SVM *wavg-2* uses the features of *comb-co-safety-succ-0.5*, the SVM *monitorList* builds on the features of *comb-co-safety-succ-0* and *master* applies the new features to the master formula.

There is no large difference between the approaches. The overall tendency suggests not to normalize the features. This is surprising, because the weights for the features are similar for the SVMs working with normalized features or the original values of the features. Despite our efforts, we are not able to find a common structure for those states. We assume, there are a number of factors involved. Some of them might be the number of monitors and the trueness of the master formula. The normalized SVMs tend to give those two values a higher weight. Additionally, the number of disjuncts and the height of the master formula seem to be less important.

category	monitorList normalized	monitorList	wavg-2 normalized	wavg-2	master normalized	master
large-co-safety	0.00	0.00	0.00	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.00	0.01	0.00	0.01	0.01	0.01
large-safety-56	0.01	0.01	0.01	0.01	0.01	0.01
small-safety	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00
large	0.03	0.02	0.03	0.03	0.03	0.03
small	0.00	0.00	0.00	0.00	0.00	0.00
lilydemo	0.12	0.12	0.10	0.09	0.18	0.16
ltl2dba	0.18	0.18	0.18	0.18	0.18	0.18
ltl2dpa	0.13	0.12	0.09	0.06	0.06	0.06
Avg	0.03	0.03	0.02	0.02	0.03	0.03

Table 4.9: Error for winning states of an SVM using normalized features compared to a SVM with the original features. All SVMs are trained using the new features. The last two SVMs only use the master formula.

4.6 Dividing the Learning Data into Different Classes

4.6.1 Transient and Recurrent States

This approach follows the idea of Backs in [2]. We separate the transient form the recurrent states and train a SVM on each of them. For the transient part, we expect the master formula to be important, so we train an SVM only using the new features applied to the master formula. For the recurrent states, we have a look at the results of the last chapter, which can be seen for only the recurrent states in table 4.10 . The SVM *wavg-comb-co-safety-succ-0.5* performs best on the recurrent states, so we train it specifically on them.

Table 4.11 shows the results of the test run. The combined SVM follows a different approach than suggested by Backs [2]: For transient states, we ask the transient SVM for advice and for recurrent states the SVM trained on recurrent states, instead of only sending a request to the recurrent SVM, if the results from the SVM for transient states are not clear enough. Since we cannot compute the transient and recurrent states based on a strategy, because we do not have one up front, we follow a slightly different notation; a state in a game is transient if it cannot reach itself. Otherwise it is recurrent. However, this definition still does not allow for an on-the-fly algorithm.

The table also contains SVMs with the same features as the transient and recurrent SVMs, but trained on all states.

As expected, the SVMs trained on recurrent states performs slightly better on states of this class than a similar SVM trained on all states. For transient states, this is not the case, as can be seen for the *large* category. Here, the SVM only trained on the transient states learned a negative value for the edge priority. We assume, this is the case, because most edges leaving a strongly component are not assigned a priority. Since transient states cannot reach themselves, most of the edges starting from them have no priority and, therefore, there are only very view values to learn from. Nevertheless, it seems to be easy to find edges for transient states, since the errors are low.

The results also show that the combination of the two SVMs leads to better results than obtained when using each SVM on its own. Especially, the results for the recurrent states improve. This is often caused by the strategy iteration algorithm. It does not necessarily include a strategy closest to the initial strategy. For this particular case, the algorithm sometimes excludes edges to other winning states from the strategy. An example can be seen in figure 4.2. Please notice that this picture is not based on the stored automata. As one can see, both, state 2 and state 3, have a winning strategy. In the third state, playing $\{f, d\}$ is enough to not let the first monitor fail and succeed the second one, because the second monitor does not contain co-safety goals. However, the strategy computed from the recurrent SVM only returns the edge to state 1, whereas the winning strategy computed from the combination accepts both. Therefore, an improvement of the metric might help.

category	ml-comb-co-safety-succ-0	ml-newF-co-safety-succ-0	ml-newF-comb	ml-newF	ml original	wavg-comb-co-safety-succ-0.5	wavg-newF-co-safety-succ-0.5	wavg-newF-comb	wavg-newF	wavg original
large-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.01	0.02	0.02	0.02	0.02	0.01	0.01	0.01	0.01	0.01
large-safety	0.03	0.02	0.02	0.02	0.04	0.03	0.02	0.02	0.02	0.04
small-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.04	0.04	0.04
large	0.02	0.05	0.05	0.05	0.05	0.04	0.03	0.04	0.04	0.06
small	0.01	0.00	0.00	0.00	0.01	0.00	0.02	0.02	0.01	0.02
lily-demo	0.14	0.18	0.21	0.19	0.21	0.09	0.07	0.09	0.09	0.16
ltl2dba	0.18	0.16	0.16	0.16	0.20	0.18	0.14	0.14	0.14	0.10
ltl2dpa	0.13	0.08	0.10	0.08	0.14	0.07	0.08	0.08	0.08	0.14
Avg	0.03	0.03	0.03	0.03	0.04	0.02	0.03	0.03	0.03	0.04

Table 4.10: Error of the SVMs from the last chapter on only the recurrent states. The word *ml* stands for *monitorList*.

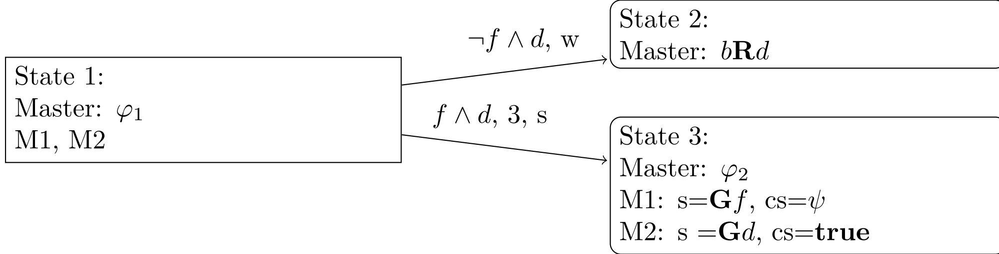

Figure 4.2: A state of the game for the formula $((\mathbf{X}b\mathbf{M}(c \leftrightarrow \mathbf{X}f)) \leftrightarrow (f\mathbf{W}\mathbf{false}))\mathbf{R}(b\mathbf{R}d)$ with environment propositions $\{c, b\}$ and system propositions $\{d, f\}$. The missing edge leads to a **false**-sink and the priority for the edge to state 2 was optimized away. The edge with letter w is part of the winning strategy, s is returned by the recurrent SVM. For simplicity, we do not show the monitors of state 1 explicitly; state 2 does not have monitors.

name	combined	recurrent	recurrent-joint	transient	transient-joint
large-co-safety	0.0005	0.0010	0.0010	0.0010	0.0010
t	0.0000	0.0000	0.0000	0.0000	0.0000
r	0.0011	0.0023	0.0023	0.0023	0.0023
small-co-safety	0.0000	0.0000	0.0000	0.0000	0.0000
t	0.0000	0.0000	0.0000	0.0000	0.0000
r	0.0000	0.0000	0.0000	0.0000	0.0000
small-p-co-safety	0.0069	0.0044	0.0057	0.0045	0.0062
t	0.0078	0.0000	0.0005	0.0078	0.0000
r	0.0073	0.0073	0.0093	0.0033	0.0103
large-safety	0.0107	0.0148	0.0093	0.0068	0.0116
t	0.0027	0.0000	0.0000	0.0027	0.0037
r	0.0268	0.0410	0.0281	0.0141	0.0275
small-safety	0.0000	0.0000	0.0000	0.0000	0.0000
t	0.0000	0.0000	0.0000	0.0000	0.0000
r	0.0000	0.0000	0.0000	0.0000	0.0000
small-p-safety	0.0000	0.0000	0.0000	0.0133	0.0000
t	0.0000	0.0000	0.0000	0.0000	0.0000
r	0.0000	0.0000	0.0000	0.0167	0.0000
large	0.0333	0.0262	0.0291	0.0620	0.0316
t	0.0177	0.0044	0.0076	0.0204	0.0060
r	0.0303	0.0276	0.0360	0.0773	0.0344
small	0.0017	0.0032	0.0007	0.0154	0.0009
t	0.0004	0.0031	0.0006	0.0004	0.0004
r	0.0039	0.0044	0.0011	0.0167	0.0022
lilydemo	0.0687	0.0786	0.0794	0.2361	0.1360
t	0.0175	0.0733	0.0294	0.0175	0.0175
r	0.0762	0.0762	0.0883	0.2787	0.1545
ltl2dba	0.0856	0.0856	0.1777	0.1777	0.1777
t	0.0000	0.0000	0.0000	0.0000	0.0000
r	0.0856	0.0856	0.1777	0.1777	0.1777
ltl2dpa	0.0585	0.0691	0.0607	0.1346	0.0559
t	0.0008	0.0323	0.0008	0.0008	0.0008
r	0.0745	0.0745	0.0658	0.1435	0.0621
Average	0.0164	0.0170	0.0202	0.0394	0.0236
t	0.0043	0.0057	0.0025	0.0046	0.0020
r	0.0191	0.0200	0.0239	0.0451	0.0274

Table 4.11: Comparison of the different SVMs on transient (t), recurrent (r) and all states. The recurrent SVM uses the features of *wavg-comb-co-safety-success-0.5* trained on only recurrent states, the transient applies the new features to the master formula. The word *joint* in the name indicates, that the SVM is trained on all states.

4.6.2 Number of Monitors

In the next step, we separate the states by the number of monitors in them and their successors. We assume that for states with monitors in them and all of their successors these monitors are more important than the master formula. States with no monitors or states, where none of the successors has a monitor, should rely on the master formula.

Again, we use the results form section 4.4, which can be seen in table 4.12 for class 2 states and table 4.13 for class 3 states, to detect, which SVMs to use. In both cases, the SVM *wavg-comb-co-safety-succ-0.5* performs best, so we will train one SVM with these features for each of the classes. The SVM for class 1 only depends on the master formula and uses the new features. For comparison, we include a SVM with the original features applied to the master formula and all parts of the monitors. The function used to compute one value per monitor features is also indicated in the name.

In the test run, we include a combination of the SVMs, which will ask one of the three SVMs for advice, depending on the type of state. The results can be seen in table 4.14. The table excludes categories without states of class 2 and 3 and the results for class 1, because for this class all SVMs lead to similar errors of values below 0.5%.

The combination worked as expected. It mostly keeps the results of the single SVMs for their respective type of state, meaning the result of e.g. class 2 correspond to the results of the SVM trained on this class when used on its own. The slight differences are caused by the ordering in which the edges are given to the SVM and the changes induced from the strategy iteration algorithm.

However, for class 2, the SVMs of class 1 and class 3 perform better on states of this class, than the SVM trained specifically for this states. We assume, that this class has both, states, for which the master formula is crucial, and states, which focus on the monitors. For example, states with one edge to a **false**-sink and all other edges to states with the same master formula belong to this class. In that case, the master formula should not have impact on the result of the SVM, because the edge to the sink will be excluded before sending the request to the SVM. Additionally, class 2 also contains state, where the master formula changes and influences the result, for example when going from a state with master formula $d\mathbf{W}a \vee \varphi$ to $d\mathbf{W}a$, like it can be seen in figure 3.6. In the state 2 of this example, the monitors are optimized away and the edge to take should be determined by the master formula. These two special cases make it hard to learn a general rule for class 2.

For the states of class 3, the SVM trained using features of the master formula in addition to the features of the monitors leads to better results than the SVM trained only on features for the monitors. This contradicts our assumption. Furthermore, the SVM trained for class 1 also leads to good results for states of class 3. Indeed, this class also contains states with changes in the master formula, so it should be considered when choosing an edge.

It is interesting to see that for the overall average the SVM leading to the best results is the SVM trained on states of class 3. This is due to the fact, that we only consider classes with monitors in the table and class 3 contains both, states focusing on the monitors and states for which the master formula has an impact.

category	ml-comb-co-safety-succ-0	ml-newF-co-safety-succ-0	ml-newF-comb	ml-newF	ml-original	wavg-comb-co-safety-succ-0.5	wavg-newF-co-safety-succ-0.5	wavg-newF-comb	wavg-newF	wavg-original
small-p-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
large	0.05	0.09	0.11	0.09	0.09	0.06	0.07	0.06	0.06	0.10
small	0.05	0.02	0.02	0.02	0.09	0.01	0.06	0.05	0.05	0.08
lily-demo	0.11	0.26	0.30	0.30	0.31	0.12	0.15	0.19	0.19	0.28
ltl2dpa	0.12	0.07	0.09	0.07	0.12	0.06	0.07	0.08	0.07	0.13
Avg	0.07	0.08	0.10	0.09	0.11	0.05	0.07	0.07	0.07	0.11

Table 4.12: Error for the SVMs from the last chapter on states of class 2. The word *ml* stands for *monitorList*. We exclude categories not containing states of class 2.

category	ml-comb-co-safety-succ-0	ml-newF-co-safety-succ-0	ml-newF-comb	ml-newF	ml-original	wavg-comb-co-safety-succ-0.5	wavg-newF-co-safety-succ-0.5	wavg-newF-comb	wavg-newF	wavg-original
small-p-co-safety	0.03	0.07	0.07	0.07	0.06	0.02	0.03	0.03	0.03	0.03
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.24	0.24	0.24	0.24
large	0.10	0.05	0.05	0.04	0.11	0.01	0.00	0.03	0.04	0.12
small	0.01	0.09	0.09	0.09	0.11	0.00	0.02	0.04	0.02	0.01
lily-demo	0.18	0.11	0.13	0.10	0.13	0.08	0.04	0.04	0.04	0.05
ltl2dba	0.18	0.16	0.16	0.16	0.19	0.18	0.14	0.14	0.14	0.10
Avg	0.09	0.09	0.09	0.08	0.11	0.05	0.05	0.07	0.06	0.08

Table 4.13: Error for the SVMs from the last chapter on states of class 3. We exclude categories which do not contain states of class 3.

category	combination	combination: class 3 monitor	class 1	class 1 joint	class 2	class 3	class 2, 3, joint	class 3 monitor
small-p-co-safety	0.0039	0.0057	0.0045	0.0062	0.0065	0.0039	0.0057	0.0057
class 2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 3	0.0150	0.0220	0.0174	0.0237	0.0249	0.0150	0.0220	0.0220
small-p-safety	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 3	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
large	0.0273	0.0302	0.0437	0.0324	0.0432	0.0230	0.0291	0.0292
class 2	0.0798	0.0829	0.0789	0.0680	0.0813	0.0629	0.0636	0.1289
class 3	0.0132	0.0970	0.0390	0.0283	0.0676	0.0132	0.0147	0.0971
small	0.0025	0.0115	0.0009	0.0009	0.0103	0.0002	0.0007	0.0099
class 2	0.0515	0.0515	0.0000	0.0000	0.0515	0.0027	0.0053	0.0050
class 3	0.0011	0.1526	0.0086	0.0086	0.0216	0.0011	0.0043	0.1526
lilydemo	0.1833	0.2407	0.1464	0.1507	0.1716	0.0809	0.0840	0.1460
class 2	0.3435	0.3435	0.2237	0.2362	0.3435	0.1183	0.1183	0.1353
class 3	0.0655	0.2652	0.1035	0.0973	0.0351	0.0655	0.0788	0.2652
ltl2dba	0.1777	0.1777	0.0856	0.1777	0.0856	0.1777	0.1777	0.1777
class 3	0.1777	0.1777	0.0856	0.1777	0.0856	0.1777	0.1777	0.1777
ltl2dpa	0.0985	0.0985	0.0609	0.0614	0.0985	0.0895	0.0607	0.0690
class 2	0.0985	0.0985	0.0609	0.0614	0.0985	0.0895	0.0607	0.0690
Average	0.0466	0.0550	0.0359	0.0410	0.0442	0.0345	0.0336	0.0424
class 2	0.1037	0.1046	0.0686	0.0668	0.1042	0.0589	0.0519	0.0769
class 3	0.0451	0.1302	0.0429	0.0561	0.0455	0.0451	0.0489	0.1302

Table 4.14: Error of the different classes. The *class 1* SVM is *newF* trained on only the master states, the *class 2* and *class 3* SVMs are *wavg-comb-co-safety-succ-0.5* and *class 3 monitor* means, the previously mentioned SVM is only trained using features on monitors, not on the master formula. The category *ltl2dpa* does not contain states of class 3 and *ltl2dba* does not contain states of class 2.

4.6.3 Change in the Master Formula

We again look at the rest run from section 4.4 to detect useful SVMs. Table 4.15 and 4.16 show the results for class 4 and 5, respectively. We train the same SVM as before, namely *wavg-comb-co-safety-succ-0.5*. Additionally, we train the SVM *ml-newF-co-safety-success-0* for class 5, where *ml* is the abbreviation for *monitorList*. The SVMs for class 5 do not use features computed on the master formula, since the master formula does not change for all edges of the state and should not influence the result.

The results of this experiment can be seen in table 4.17. Notice, that these runs where executed separately form the other tests in this chapter due to an error in the implementation.

As expected, the SVMs trained on a specific type of states lead overall to better results on these states than the SVM trained on all states. For states of class 5, both SVMs trained on this class return worse results than the corresponding SVM trained on all states for some categories, e.g. *small-p-co-safety* and *large*. The differences are caused by states in which the SVMs for class 5 focus on not failing monitors, while the SVMs trained on all states try to make progress in one monitor, even on the expense of failing another one. This is caused by different weights for the features *priority* and *percentageSuccCoSafety*. The first feature is more important than the second one for the SVMs of class 5 and the ordering is reversed for the SVMs trained of all states.

The SVM *wavg-comb-co-safety-succ-0.5* trained on class 5 states performs best out of all the SVMs in the test run, except for the combined SVMs. This is surprising, since the SVM uses only the monitor features and the feature *edgePriority* and *improveObSet*. For states without monitors, the SVM can only use the last two features to make decisions, which means, that these features are of high importance for the game.

The idea to combine the results of each class by using a different SVM depending on the class of a state improves the results. The SVMs for states of class 1 and 4 behave as expected. However, the results for states of class 5 are are worse by a small factor compared to the corresponding SVM on its own. This mostly originates from the category *small-p-co-safety* and is caused by the strategy iteration algorithm. An abstract example can be seen in figure 4.3. Here, the ordering of the monitors is not important, since both monitors have **true** as a safety formula, which means, they cannot fail. The winning strategy computed from the combination of SVMs only accepts one of the edges, even though both edges lead to the same result. This error value for the combination.

category	ml-comb-co-safety-succ-0	ml-newF-co-safety-succ-0	ml-newF-comb	ml-newF	ml-original	wavg-comb-co-safety-succ-0.5	wavg-newF-co-safety-succ-0.5	wavg-newF-comb	wavg-newF	wavg-original
small-p-co-safety	0.03	0.07	0.06	0.06	0.08	0.02	0.02	0.01	0.02	0.01
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
large	0.05	0.08	0.08	0.08	0.06	0.06	0.05	0.06	0.06	0.09
small	0.01	0.02	0.02	0.02	0.05	0.00	0.02	0.02	0.01	0.03
lily-demo	0.17	0.20	0.23	0.21	0.23	0.09	0.07	0.08	0.08	0.14
ltl2-dba	0.09	0.09	0.09	0.09	0.10	0.09	0.09	0.09	0.09	0.11
ltl2-dpa	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Avg	0.06	0.08	0.08	0.07	0.08	0.04	0.04	0.04	0.04	0.07

Table 4.15: Error for the SVMs from the last chapter on states of class 4. Again, categories without states of class 4 are excluded.

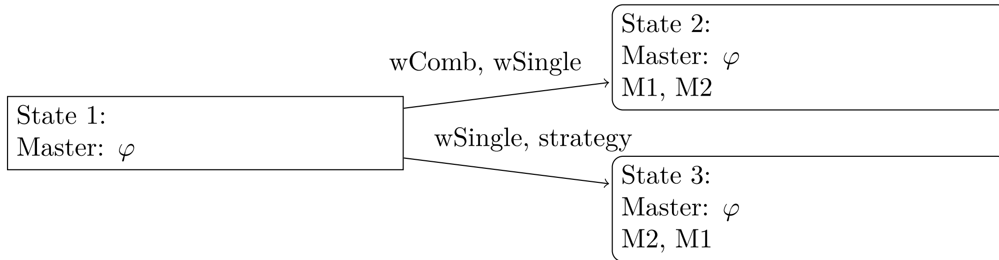

Figure 4.3: Abstraction of a problematic state, as e.g. found in the game for the formula $\mathbf{F}(\mathbf{F}a \wedge \mathbf{F}d \wedge \mathbf{GX}(c \vee \mathbf{FX}e))$. The monitors each have a safety formula of **true**. The edges part of the winning strategy are indicated by *wComb* for the combination and *wSingle* for the SVM on it own. The edge with keyword *strategy* is returned in both initial strategies.

category	ml-comb-co-safety-succ-0	ml-newF-co-safety-succ-0	ml-newF-comb	ml-newF	ml-original	wavg-comb-co-safety-succ-0.5	wavg-newF-co-safety-succ-0.5	wavg-newF-comb	wavg-newF	wavg-original
small-p-co-safety	0.10	0.09	0.09	0.09	0.07	0.06	0.10	0.10	0.10	0.07
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.28	0.28	0.28
large	0.00	0.02	0.05	0.03	0.09	0.00	0.00	0.03	0.03	0.03
small	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.20	0.10	0.00
lily-demo	0.05	0.05	0.08	0.08	0.08	0.06	0.07	0.09	0.09	0.10
ltl2-dba	0.20	0.18	0.18	0.18	0.23	0.19	0.16	0.16	0.16	0.13
ltl2-dpa	0.15	0.09	0.12	0.09	0.16	0.07	0.08	0.08	0.08	0.16
Avg	0.09	0.07	0.09	0.08	0.11	0.06	0.10	0.11	0.10	0.11

Table 4.16: Error for the SVMs from the last chapter on states of class 5. Categories without states of class 5 are not considered.

category	combination, class 5: wavg	combination, class 5: monitorList	wavg class 4	wavg class 5	wavg joint	monitorList c5	monitorList joint
small-p-co-safety	0.0047	0.0044	0.0048	0.0059	0.0057	0.0038	0.0068
class 4	0.0168	0.0168	0.0168	0.0237	0.0244	0.0176	0.0267
class 5	0.0865	0.0673	0.0962	0.0769	0.0577	0.0192	0.0962
small-p-safety	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 4	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 5	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
large	0.0254	0.0254	0.0236	0.0146	0.0291	0.0452	0.0244
class 4	0.0475	0.0475	0.0475	0.0799	0.0580	0.1377	0.0504
class 5	0.0400	0.0400	0.0050	0.0250	0.0000	0.0500	0.0000
small	0.0043	0.0116	0.0043	0.0004	0.0007	0.0558	0.0025
class 4	0.0148	0.0148	0.0148	0.0015	0.0036	0.1357	0.0139
class 5	0.0000	0.1000	0.0000	0.0000	0.0000	0.1000	0.0000
lilydemo	0.0992	0.1073	0.1203	0.1276	0.0983	0.1205	0.1315
class 4	0.1113	0.1113	0.1113	0.1780	0.1119	0.1545	0.1906
class 5	0.0551	0.0699	0.0938	0.0551	0.0551	0.0699	0.0461
ltl2dba	0.0856	0.1170	0.1777	0.0856	0.1777	0.1170	0.1820
class 4	0.0922	0.0922	0.0922	0.0922	0.0922	0.0922	0.0922
class 5	0.1028	0.1342	0.1949	0.1028	0.1949	0.1342	0.1992
ltl2dpa	0.0750	0.0665	0.1025	0.0750	0.0607	0.0665	0.1191
class 4	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
class 5	0.0820	0.0858	0.1117	0.0820	0.0662	0.0858	0.1490
Average	0.0298	0.0334	0.0407	0.0299	0.0350	0.0474	0.0439
class 4	0.0451	0.0451	0.0451	0.0623	0.0463	0.1085	0.0586
class 5	0.0605	0.0784	0.0839	0.0569	0.0633	0.0759	0.0865

Table 4.17: Error of the different SVMs. The SVM for *class 1* uses the new features applied to the the master formula, the *wavg* SVMs are trained on the features of *wavg-comb-co-safety-succ-0.5*, the *monitorList* SVMs are trained like *monitorList-newF-co-safety-succ-0*. The first two SVMs are combinations, where only the last SVM is changed as stated in the name. The SVMs for class 5 do not consider the master formula.

4.6.4 Different Categories of Games

The last separation of the training data concerns the types of games. We train a different SVM for safety and co-safety formulas. The SVMs apply the new features to the master formula. Table 4.18 shows the results. The SVM trained on the safety class did not perform well, not even on its own games. The point of these games is to avoid some bad event. This can not be captured by the existing features, since the only feature included, which aims for not failing the master formula, is the fail set. The other features try to succeed it. The feature based on the fail set has indeed a high influence in the SVM trained on safety formulas.

On the other hand, the SVM for the co-safety formulas leads to good results. It seems, that focusing on achieving something good works well for most games. Additionally, most of the features presented in this thesis and by Backs [1] work in that direction.

name	newF	co-safety-newF	safety-newF
large-co-safety	0.00	0.00	0.00
small-co-safety	0.00	0.00	0.00
small-p-co-safety	0.01	0.00	0.12
large-safety	0.01	0.01	0.02
small-safety	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.13
large	0.03	0.01	0.16
small	0.00	0.01	0.15
lilydemo	0.12	0.11	0.35
ltl2dba	0.18	0.09	0.55
ltl2dpa	0.06	0.06	0.50
Average	0.02	0.02	0.13

Table 4.18: Error of training SVMs based on the type of formula.

4.7 Additional Obligation Set Test

In this section, we present the results obtained when using the additional obligation set check before sending a request to the SVM. For comparison, we use a SVM trained on the features presented in [1] for the master formula, called *original*, one for the new features on the master formula, called *newF*, either trained on all formulas or only the co-safety formulas. In addition, we use *wavg-w2-newF-co-safety-succ-0.5*, called *wavg-newF* in the results of this section, from section 4.4 and *wavg-w2-comb-co-safety-succ-0.5*, called *wavg-comb*, trained on states of class 5 without features on the master formula, because these are the SVMs leading to the best results in the previous sections.

The results can be seen in table 4.19. The additional obligation set check reduces the error for all SVMs. One can also see, that the overall best results are achieved when using the SVMs trained with features for the monitors, especially for the categories *large, small* and *ltl2dpa*. This categories depend on the monitors. However, for the class of

category	original	original check	newF	newF check	newF co-safety	newF co-safety check	wavg-newF	wavg-newF check	wavg-comb	wavg-comb check
large-co-safety	0.0014	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0015	0.0010
small-co-safety	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
small-p-co-safety	0.0048	0.0110	0.0062	0.0110	0.0045	0.0110	0.0073	0.0110	0.0045	0.0110
large-safety	0.0158	0.0040	0.0116	0.0040	0.0142	0.0040	0.0079	0.0040	0.0362	0.0040
small-safety	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0045	0.0000
small-p-safety	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0333	0.0000	0.0133	0.0000
large	0.0513	0.0243	0.0287	0.0131	0.0097	0.0111	0.0261	0.0104	0.0171	0.0135
small	0.0031	0.0003	0.0009	0.0003	0.0069	0.0072	0.0119	0.0076	0.0004	0.0003
lily-demo	0.1565	0.1565	0.1692	0.1631	0.1603	0.1603	0.0906	0.0906	0.1367	0.1327
ltl2dba	0.0782	0.0782	0.1777	0.1777	0.0856	0.0856	0.1423	0.1423	0.0856	0.0856
ltl2dpa	0.1144	0.1144	0.0584	0.0584	0.0605	0.0605	0.0695	0.0695	0.0750	0.0750
Avg	0.0267	0.0231	0.0253	0.0231	0.0197	0.0199	0.0235	0.0188	0.0224	0.0186

Table 4.19: Error for winning states for the SVMs. The *check* in the name of the SVMs indicates, that an additional obligation set check was performed.

safety formulas, the SVMs only trained on the master formula achieve the best results.

Table 4.20 shows the run time for each SVM and trueness. The obligation set check significantly reduces the time. The reason is, that the SVMs are realized as a local server and the program needs to refer to them for advice. The obligation set check can solve some of the states, before sending the request. Nevertheless, *trueness* is still faster by a factor of 100.

category	original	original check	newF	newF check	newF co-safety	newF co-safety check	wavg-newF	wavg-newF check	wavg-comb	wavg-comb check	trueness
large-co-safety	0.14	0.09	0.12	0.08	0.13	0.15	0.12	0.09	0.12	0.51	0.14
small-co-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-co-safety	0.03	0.02	0.03	0.0281	0.02	0.02	0.05	0.03	0.03	0.02	0.02
large-safety	5.38	0.26	5.32	0.24	5.39	0.13	5.83	0.25	5.01	0.17	0.04
small-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
small-p-safety	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
large	13.08	0.11	12.96	0.09	13.25	0.15	13.40	0.06	12.83	0.15	0.07
small	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lily-demo	699.52	98.79	712.29	98.68	685.24	101.68	695.64	101.32	692.99	103.91	1.25
ltl2-dba	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ltl2-dpa	71.85	73.33	69.97	72.41	71.85	73.62	72.51	74.93	71.10	71.98	0.10
Avg	55.42	12.89	56.11	12.80	54.46	13.11	55.28	13.19	54.85	13.16	0.12

Table 4.20: Average of the run time in milliseconds. The *check* in the name of the SVMs indicates, that an additional obligation set check was performed.

4.8 Comparison of the LDBA and the DRA Approach

This section compares the results obtained when using the DRA as intermediate automaton and the features suggested in [1] with the results obtained in the last section. The naming of the SVMs is as before. In addition, there is a SVM *master combination*. This is a combination of different SVMs based on classes 1, 4 and 5 as presented in section 4.6.3. The SVMs for class 4 and 5 are trained on using the features *wavg-comb-co-safety-succ-0.5* applied to states of their respective class. The SVM for class 4 in addition uses the new features applied to the master formula, while the SVM in class 5 only considers the monitors, the priority of an edge an the *improveObligationSet* feature. The row with title *trans/rec combination* shows the results for the combination of one SVM for transient and one for recurrent states, as described in section 4.6.1. Additionally, we include the results for the heuristic *Trueness* applied to a game resulting from the translation with an intermediate LDBA. An additional obligation set check is applied for each of the SVM, except for the rows *original*, *DRA* and *trueness LDBA*.

The number of winning games directly solved by the initial strategy can be found in table 4.21. As one can see, the DRA approach has an advantage for co-safety and safety formulas. However, we want to mention that the LDBA approach also solves all co-safety formulas correctly in some test runs. This depends on the ordering in which the edges are are processed.

For the categories *large* and *lilydemo*, the translation with the intermediate LDBA leads to better results. This is expected, since both classes contain many states with monitors. Surprisingly, the translation with an intermediate DRA can solve more games directly than the presented SVMs.

The errors of the SVMs for the LDBA approach in this category are mostly caused by one of the following cases: In the first case, the SVM aims for a simpler safety formula, like it can be seen in 4.4. The safety-formula of monitor 2 is easier in state 2, however, the winning strategy leads to state 3, because playing *acc* lets the last monitor fail. The second case can be seen in figure 4.5. In this case, the SVM keeps the second monitor from failing, but does not progress the first. Because of the **X** operator in the safety goal, the current implementation is not able to recognize this monitor as a reappearing one. Both of these cases are captured by the color of the edge for the translation with the intermediate DRA. As a consequence, the SVM for this approach can focus on the color instead.

In total, the approach presented in this thesis can solve approximately as many games optimally as the approach suggested in [1]. For the translation using a LDBA as intermediate step, the approaches presented in this thesis solve 12 more games optimally than an SVM using the features of [1] an 25 more games than the heuristic *trueness* applied to the obtained DPA when using the construction with an intermediate LDBA.

category (number of winning games)	original	wavg-newF	wavg-comb	newF	trans/-rec combination	master combination	DRA	trueness LDBA
large-co-safety (13)	12	12	12	12	12	12	13	12
small-co-safety (12)	12	12	12	12	12	12	12	12
small-p-co-safety (12)	12	12	12	12	12	12	12	11
large-safety (13)	9	12	12	12	12	12	13	9
small-safety (12)	12	12	12	12	12	12	12	12
small-p-safety (12)	12	12	12	12	12	12	12	10
large (15)	11	14	13	13	12	14	11	7
small (15)	12	14	15	15	15	15	15	10
lilydemo (9)	5	5	4	5	6	5	4	5
ltl2dba (8)	2	2	3	2	3	3	5	1
ltl2dpa (11)	3	6	6	6	6	6	6	1
Overall (132)	102	113	113	113	114	115	115	89

Table 4.21: Number of games solved correctly by the different approaches. We consider a game to be solved optimally, if there is a winning strategy and the heuristic maps all reachable states to edges in the winning strategy. For all SVMs an additional obligation set check was performed.

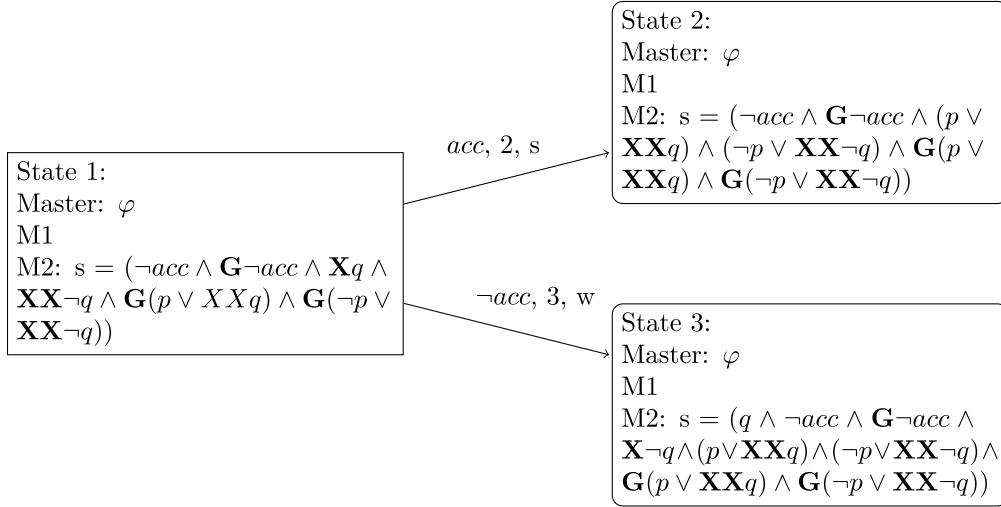

Figure 4.4: This is part of a game generated from $((\mathbf{G}(\mathbf{F}((p) \leftrightarrow (\mathbf{X}(\mathbf{X}(q)))))) \leftrightarrow (\mathbf{G}(\mathbf{F}(acc))))$ with environment propositions $\{p, q\}$ and system propositions $\{acc\}$. The monitor $M1$ is not affected by the choice of the system player. The rounded edges indicate an environment state, again.

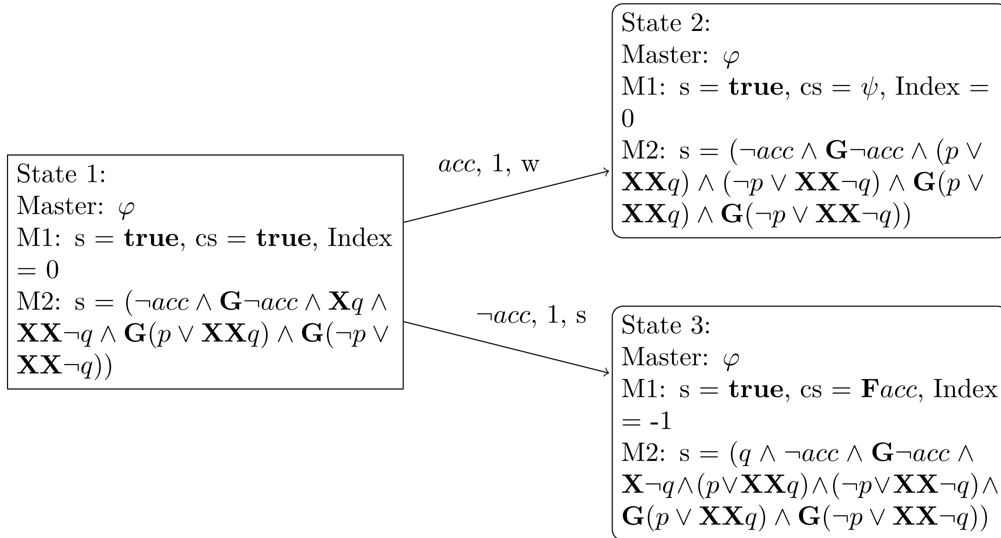

Figure 4.5: This is part of a game generated from $((\mathbf{G}(\mathbf{F}((p) \leftrightarrow (\mathbf{X}(\mathbf{X}(q)))))) \leftrightarrow (\mathbf{G}(\mathbf{F}(acc))))$ with environment propositions $\{p, q\}$ and system propositions $\{acc\}$.

5 Future Work

This chapter contains suggestions for future work. First of all, the problematic states from the last chapter can be considered. Here, it might be possible, to add new features to avoid such cases. Furthermore, one can improve the implementation, such that reappearing states with a **X** operator are noticed as well.

Additional features might also be helpful for safety formulas and the safety formulas within a monitor. In this thesis, the only feature focusing on not failing a formula is the one based on the fail set. These information are also important for the combined features, which appear to be promising.

A different direction is, to combine different SVMs. One could for example use Platt scaling [21] to compare the values of the SVMs. This could be used for states of class 2. Here, SVMs for class 1 and 3 lead to better results than one trained on states of class 2. Using Platt scaling, one could compare the other two SVMs and pick the better suggestion.

Another point is, that the advice of the SVM depends on the ordering of the edges. If several edges have the same confidence value, the first edge is returned. This makes the analysis of the results more complicated. A way around that could either be to order the edges in Java or to make the choice of the SVM unique, e.g. by expanding edges and using predictions on the successor states until a difference is found.

The strategy iteration also depends on the ordering of the states and edges. As already stated, the results obtained in this thesis might not be completely recomputable, because the strategy iteration might return different strategies. Finding an ordering of the edges could help here, as well.

We also want to point out, that the used metric in this thesis does not totally reflect the quality of the initial strategy. We use the percentage of winning states mapped to a wrong edge. However, there might be winning states, which are nearly never visited by a winning strategy starting at the initial node. These states are still considered in our results. One could try to find a different metric to avoid this.

In his unpublished work, Backs suggested a metric, which computes a distance from the initial strategy computed by the SVM to the optimal strategy returned by a strategy iteration algorithm starting at the given initial strategy.

To do so, the metric starts with the initial state and adds it to the list of reachable states. For each state in this list, one compares the initial strategy with the winning strategy. If they agree, the metric moves to the successor state. This is a node controlled

by the environment player due to the structure of the game, so there is no strategy computed on that state and the edge chosen in this state is not predictable. Because of that, all successors of the environment node are added to the queue of reachable states. Then the metric proceeds with the next state in the list.

If the initial strategy and the winning strategy return different edges, the metric counts this as wrong and increases the distance counter by 1. Afterwards, the heuristic, e.g. a SVM, responsible for the initial strategy is asked to choose between all edges of the winning strategy for the state and pick one. Using this edge, the metric proceeds as before.

There are only the above described cases, because a heuristic like the SVM only returns one edge for each state, even though there could be more edges in a winning strategy. If a heuristic is indifferent between two edges, it returns the first one.

When we applied this metric, we realized that the results vary widely, even if one of the pre-computed games is used. The reason is that the implementation uses sets in Java to store the edges. A set does not guarantee that its values will always be returned in the same order. Since the SVM depends on the ordering of the edges, if it is indifferent between two edges, this can influence the results.

However, it might still be worth consideration to improve on the metric.

Furthermore, the strategy iteration algorithm we used, removes cycles from the initial strategy before starting. This can lead to edges of the initial strategy not being part of the computed winning strategy, even though in theory, there is a winning strategy containing them. These edges will then be considered as wrong. One could try to overcome this by computing more than one winning strategy and compare the initial strategy returned from the SVM with all of them in order to find the optimal strategy closest to the initial one.

In consideration of the runtime, the improvement of the obligation set check or the introduction of new heuristics like that to avoid sending a request is also be plausible.

Another approach could be, to use a different translation. We suspect, that for the translation using DRA as intermediate step, it is hard to find features relating to the monitors and the progress of them, because most information is hidden in the priority of an edge and not as accessible as it is in the translation we used. Still, it is a possibility to compute the average of some features over all the monitors. Another translation worth consideration is the symmetric translation described in [9] [7].

6 Conclusion

We presented different approaches to improve the results of the SVM developed in [1] for predicting winning edges in parity games resulting from LTL synthesis. We used the translation from LTL to DPA with an intermediate LDBA.

In comparison tho the SVM suggested in [1], when applied on the translation using a LDBA as intermediate step, our approaches improve the number of games solved by the initial strategy obtained form the SVM and reduce the number of winning states mapped to an edge not in the winning strategy. The combined features for safety and co-safety formulas of the monitor, as well as the progress features and the additional obligation set check proofed to be helpful. The last one reduced the run time significantly. The separation of the training data increased the number of winning states mapped to a winning edge for the class the SVM was trained on. In particular, the separation based on the change in the master formula reduced the error and using the combination of these SVMs we were able to combine the advantages of the individual SVMs.

However, our approaches cannot solve as many games obtained from safety and co-safety formulas as the implementation of [1] applied to the translation with an intermediate DRA . This approach also leads to better initial strategies for the class *ltl2dba*. Our presented SVMs, on the other hand, result in better initial strategies for the class of large games.

Bibliography

[1] C. Backs. Machine learning for prediction of edge performance in LTL synthesis. October 2020.

[2] C. Backs. Unpublished work on edge prediction for LTL synthesis. `https://gitlab.lrz.de/i7/owl-ml`, 2021.

[3] R. Bloem, K. Chatterjee, and B. Jobstmann. *Graph games and reactive synthesis.* In *Handbook of Model Checking.* E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. Springer International Publishing, Cham, 2018, pages 921–962.

[4] R. Ehlers. Unbeast: Symbolic bounded synthesis. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 272–275, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011.

[5] J. Esparza and J. Křetínský. From LTL to deterministic automata: A safraless compositional approach. In *International Conference on Computer Aided Verification*, pages 192–208. Springer, 2014.

[6] J. Esparza, J. Křetínský, J.-F. Raskin, and S. Sickert. From LTL and limit-deterministic büchi automata to deterministic parity automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 426–442. Springer, 2017.

[7] J. Esparza, J. Křetínský, and S. Sickert. A unified translation of linear temporal logic to ω-automata. *J. ACM*, 67(6), October 2020.

[8] J. Esparza, J. Křetínský, and S. Sickert. From LTL to deterministic automata. *Formal Methods in System Design*, 49(3):219–271, 2016.

[9] J. Esparza, J. Křetínský, and S. Sickert. One theorem to rule them all: A unified translation of ltl into ω-automata. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, 384–393, Oxford, United Kingdom. Association for Computing Machinery, 2018.

[10] A. Khalimov, S. Jacobs, and R. Bloem. Party parameterized synthesis of token rings. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 928–933, Berlin, Heidelberg. Springer Berlin Heidelberg, 2013.

[11] J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182–195, 2006. Implementation and Application of Automata.

[12] J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, pages 7–22, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012.

[13] J. Křetínský, A. Manta, and T. Meggendorfer. Semantic labelling and learning for parity game solving in LTL synthesis. In *International Symposium on Automated Technology for Verification and Analysis*, pages 404–422. Springer, 2019.

[14] J. Křetínský, T. Meggendorfer, and S. Sickert. Owl: A library for ω-words, automata, and LTL. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 543–550. Springer, 2018.

[15] J. Křetínský, T. Meggendorfer, S. Sickert, and C. Ziegler. Rabinizer 4: From LTL to your favourite deterministic automaton. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 567–577, Cham. Springer International Publishing, 2018.

[16] J. Křetínský, T. Meggendorfer, C. Waldmann, and M. Weininger. Index appearance record for transforming rabin automata into parity automata. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–460, Berlin, Heidelberg. Springer Berlin Heidelberg, 2017.

[17] O. Kupferman. Recent challenges and ideas in temporal synthesis. In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, pages 88–98, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012.

[18] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 31–44, Berlin, Heidelberg. Springer Berlin Heidelberg, 2006.

[19] J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL satisfiability checking revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning*, pages 91–98, 2013.

[20] M. Luttenberger, P. Meyer, and S. Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica*, 57:3–36, 2020.

[21] J. Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[22] S. Safra. On the complecxity of ω-automata. *FOCS*:319–327, 1988.

[23] S. Schewe and B. Finkbeiner. Bounded synthesis. In *International Symposium on Automated Technology for Verification and Analysis*, pages 474–488. Springer, 2007.

[24] A. Shmilovici. *Data mining and knowledge discovery handbook.* In O. Maimon and L. Rokach, editors. Springer, Boston, MA, 2005. Chapter Support Vector Machines.

[25] S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-deterministic büchi automata for linear temporal logic. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 312–332, Cham. Springer International Publishing, 2016.

[26] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, New York, 1995.

[27] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, pages 202–215, Berlin, Heidelberg. Springer Berlin Heidelberg, 2000.

[28] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.