International Journal on Software Tools for Technology Transfer (2023) 25:249–266
https://doi.org/10.1007/s10009-023-00716-z

Algebraically explainable controllers: decision trees and support vector machines join forces

Florian Jüngermann[1] · Jan Křetínský[1,2] · Maximilian Weininger[1]

Accepted: 6 July 2023 / Published online: 10 August 2023
© The Author(s) 2023

Abstract
Recently, decision trees (DT) have been used as an explainable representation of controllers (a.k.a. strategies, policies, schedulers). Although they are often very efficient and produce small and understandable controllers for discrete systems, complex continuous dynamics still pose a challenge. In particular, when the relationships between variables take more complex forms, such as polynomials, they cannot be obtained using the available DT learning procedures. In contrast, support vector machines provide a more powerful representation, capable of discovering many such relationships, but not in an explainable form. Therefore, we suggest to combine the two frameworks to obtain an understandable representation over richer, domain-relevant algebraic predicates. We demonstrate and evaluate the proposed method experimentally on established benchmarks.

Keywords Controller representation · Explainability · Synthesis · Decision tree

1 Introduction

Safe and efficient controllers for cyber-physical systems are difficult to obtain manually, in particular in presence of both the discrete type of behaviour and continuous aspects such as complex dynamics in space and/or time. To this end, various model checking tools offer also an automatic *controller synthesis* option, for instance `UPPAAL Stratego` [14], `PRISM` [25], `SCOTS` [35], or `STORM` [16]. In their most basic form, they use discretization to represent the continuous input space with a finite set of states. For each of those states, the synthesized controller describes which actions are allowed. So, the controller can be expressed explicitly as a lookup table, often with millions of rows.

There are two main issues with this representation. First, storing such a large table can require several hundred megabytes of storage. However, the devices on which the controller should run are often embedded chips with very limited storage capacity. This makes it infeasible to store the entire lookup table on the device. Second, the sheer size makes it impossible to understand the behavior of the controller. Safety guarantees of the controller rely on the assumption that the formal model was correct and behaves as expected. To validate this and certify the quality, understanding the controller is crucial. For example, a non-permissive controller for an emergency braking system might try to immediately stop the car. This fulfils the safety requirement, as no crash can occur; however, it is not useful in a real application. These flaws in the model can be detected if we can represent the safe controller in a succinct and explainable way.

✉ J. Křetínský
jan.kretinsky@tum.de

F. Jüngermann
florian.jungermann@tum.de

M. Weininger
maximilian.weininger@tum.de

1 Technical University of Munich, Munich, Germany

2 Masaryk University, Brno, Czech Republic

Running example To demonstrate our approach to these issues, we take a closer look at the adaptive cruise control model (in short `cruise`) from [26], which models a simplified emergency braking system for a car. Synthesizing a safe controller with `UPPAAL Stratego` gives us a controller, but in the form of a huge lookup table with more than six million lines. However, previous work [1] has shown that there is a way to formulate the safe behavior with a handful of sentences or equations. The goal of this paper is to find such a succinct and explainable representation automatically, utilizing techniques from machine learning.

Controller representation with decision trees Recently significant progress has been made [3–6, 10] with

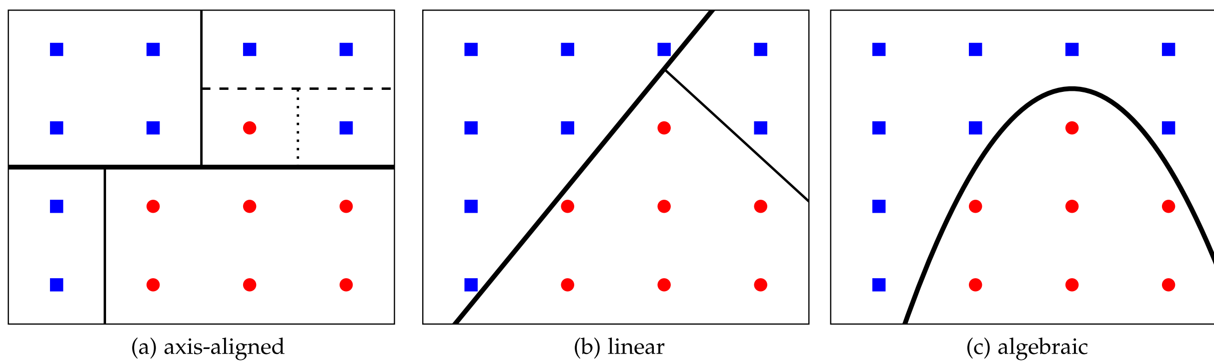


Fig. 1 Example showing how different types of predicates can separate a dataset

representing controllers using decision trees (DTs, a.k.a. nested if-then-else code). Decision trees, e.g. [29], are simple in structure, making them easy to understand, but still expressive enough to represent complex controllers. The open-source tool `dtControl` [6] takes advantage of this and offers an automated way to generate succinct decision trees. It can read controllers from many commonly used model checkers and implements various heuristics to minimize the size of the decision tree.

Traditionally, in a DT, the predicate in the decision node has the form $v_i \leq c$ with some variable v_i and some constant $c \in \mathbb{R}$. Such a split divides up the feature space with a hyperplane orthogonal to the feature axis v_i thus giving it the name *axis-aligned* split [29]. These splits are easy to understand and find efficiently, forming the basis of efficient decision-tree learning.

However, axis-aligned predicates are incapable of capturing more complex relationships, as seen in Fig. 1a. In this example, 5 predicates are needed to separate the red from the blue labels. For a real-world dataset with thousands of data points, this behavior can be even more extreme. For this reason, `dtControl` also supports linear predicates, as proposed by [30]. These splits are still hyperplanes, but now can have arbitrary orientations (see Fig. 1b). This makes the predicates harder to find, more difficult to comprehend as more variables are involved, but can ultimately give significantly smaller DTs.

Extending the notion of using more complex decision predicates, the newest version of `dtControl` allows the use of *algebraic predicates* [6, 39] (see Fig. 1c). A domain expert can provide arbitrary closed-form mathematical expressions that are then used in the decision tree construction. It is even possible to leave some constants unspecified, for which `dtControl` then finds suitable values.

Limitations While `dtControl` has already greatly reduced the size of the controllers in many benchmarks, there is still room for improvement. Most of the implemented heuristics for continuous systems rely on clever ways to determine a non-deterministic controller. This means that we start with a (possibly most) permissive controller, i.e., a controller that can permit several safe actions per state. Then, we dynamically select one action for every state, making the choice in each state deterministic. This allows to represent the resulting strategy more succinctly. However, in some instances, such as the `cruise` model, we want to keep the permissiveness of the controller, for example, to give a human driver the maximum amount of freedom.

To accurately represent the most-permissive controller for the `cruise` model without any determinization heuristics, `dtControl` needs several hundred decision nodes and the resulting DT is hardly explainable. Given the right domain knowledge, significantly smaller decision trees can be found by using algebraic predicates [1, 39]. However, so far, the supplied domain knowledge had to be tailored to the problem by hand.

Our contributions We address these limitations by providing a new, automated way of generating algebraic predicates.

- We use support vector machines (SVM) to learn the predicates from the data. To make this feasible, we utilize a variant of the kernel trick that is informed by our domain knowledge. This results in more accurate but less understandable predicates.
- We introduce several techniques to improve the explainability of the predicates.
- We experimentally evaluate this new approach of combining DT and SVM learning, which we implemented as an extension of `dtControl`. In particular, we receive an explainable DT with only 5 decision nodes for the `cruise` example.

Related work This work extends the open-source tool `dtControl` that was first presented in [5], covered in detail in [20], and since significantly extended [6]. Adapting techniques from machine learning and formal verification,

we combine the insights we receive from the controller data with the domain knowledge to construct smaller and more explainable DTs.

There have been several approaches using non-linear predicates in DTs. [19] explicitly constructs new features by combining existing ones (for example, take their product or ratio), while [8] explicitly uses SVMs inside the decision nodes. Both focus on the DT's ability to generalize rather than exactly represent a controller, thus potentially losing safety guarantees. Moreover, they do not consider explainability. Specifically, they do not explicitly reconstruct algebraic predicates from the SVM. In contrast, our approach not only generates SVM related to the domain knowledge and reconstructs the algebraic predicates, but it also focuses on their transformation into simpler and more explainable ones.

In previous versions of `dtControl` [6, 39], curve fitting [2] was used to find undetermined coefficients in algebraic splitting predicates. This approach is based on regression analysis and uses least square fitting [27, 28]. In our work, however, we use the predicates to separate the data rather than fitting it. For a more detailed comparison, see Sect. 4.1.

Typically, *binary decision diagrams* (BDDs) [12] are used to represent controllers in a compressed way [35, 40]. As BDDs can only represent a binary function $\{0,1\}^n \rightarrow \{0,1\}$, this approach requires us to encode the list of state-action pairs of the controller in binary variables. As a result, the BDDs are hardly explainable. Additionally, the size of the BDD heavily depends on the variable ordering. Finding an optimal ordering is NP-complete [9] and currently known heuristics struggle with high-dimensional inputs.

Algebraic decision diagrams [7] extend BDDs to support representation of a function $\{0,1\}^n \rightarrow S$ with $S \subset \mathbb{N}$. They have been used to represent controllers in, e.g., [37]. However, they suffer from the same issues we discussed for BDDs.

2 Preliminaries

The paper is concerned with the representation of controllers, which we thus now define. While the specific type of dynamics of the system is irrelevant, we assume the states of the system are given by values of state variables:

Definition 1 (Controller)
For a model $\mathcal{M}$ with states $\mathcal{S}$ and actions $\mathcal{A}$, a controller $C : \mathcal{S} \rightarrow 2^{\mathcal{A}}$ selects for every state $s \in \mathcal{S}$ a set of (so-called *safe*) actions $C(s) \subseteq \mathcal{A}$. Moreover, we assume the set of states is $\mathcal{S} \subseteq \prod_{i=1}^{M} \mathrm{Dom}(v_i)$, where, for $1 \leq i \leq M \in \mathbb{N}$, v_i is a state variable with domain $\mathrm{Dom}(v_i)$ and $\prod$ denotes the cartesian product.

Note that this definition allows for *permissive* controllers that can provide multiple possible actions for a state. Further, we call a controller *deterministic* when $|C(s)| = 1$ for all $s \in \mathcal{S}$, meaning it chooses exactly one possible action in every state.[1]

2.1 Representing controllers by decision trees

Definition 2 (Decision Tree)
A decision tree (DT) T is defined as follows:

- T is a rooted full binary tree, meaning every node either is an *inner node* and has exactly two children, or is a *leaf node* and has no children.
- Every inner node v is associated with a decision predicates α_v. A decision predicate (or just predicate) is a boolean function $S \rightarrow \{0,1\}$ over the input S.
- Every leaf ℓ is associated with an output label $a_\ell \in A$.

For learning a DT, numerous methods exists such as `CART` [11], `ID3` [33], and `C4.5` [34]. In principle, they all evaluate different *predicates* (see Sect. 2.3) by calculating some impurity measure (see Sect. 2.2) and then greedily pick the most promising one before splitting the dataset on that predicate and recursively continuing with the two children.

A DT represents a function as follows: every input vector $\mathbf{x} \in S$ is evaluated by starting at the root of T and traversing the tree until we reach a leaf node ℓ. Then, the label of the leaf a_ℓ is our prediction for the input $\mathbf{x}$. When traversing the tree, at each inner node v we decide at which child to continue by evaluating the decision predicate α_v with $\mathbf{x}$. If the predicate evaluates to true, we pick the left child, otherwise the right one.

When we represent a controller with a DT, our input data is the set of states and an output labels describe a subset of safe actions. Figure 2 shows a decision-tree representation of a deterministic and a permissive controller of a battery-powered temperature control system.

2.2 Impurity measures

To evaluate how promising a predicate α is, `dtControl` implements different impurity measures. The most classic impurity measure is entropy, originating from information theory. It measures how much uncertainty is left in a dataset. If the dataset is dominated by one specific label, the entropy is low, whereas a heterogeneous dataset has a higher entropy. For a dataset X with N data points and the label set B, let

[1] Also note that according to our definition, the controller's decision is solely based on its current state, not its past states. In practice, this limitation can often be circumvented by encoding additional information about the past into the state. For example, the decision of whether to water the plants may depend on the precipitation of the last three days. Then we can model our current state as a tuple (p_1, p_2, p_3) where p_i describes the precipitation i days ago.

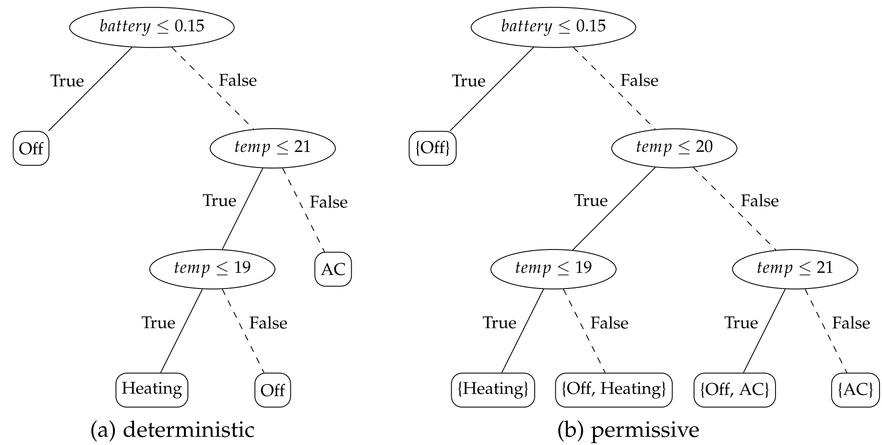


Fig. 2 An example of how a decision tree can represent a controller. (a) shows a determinized controller, (b) a permissive one with multiple safe actions at some states

$n(X, y)$ be the number of data points in X with the label $y \in B$. Then the entropy $H(X)$ is defined as:

$$H(X) = -\sum_{y \in B} \frac{n(X,y)}{N} \log_2 \left(\frac{n(X,y)}{N} \right). \tag{1}$$

To evaluate a predicate α, we calculate the remaining entropy after the split. If α is a binary split and partitions X into $X = X_l \uplus X_r$, we define:

$$H(\alpha, X) = \frac{|X_l|}{|X|} H(X_l) + \frac{|X_r|}{|X|} H(X_r). \tag{2}$$

2.3 Predicates in decision trees

In each decision node of our tree, a predicate function $A \rightarrow \{0, 1\}$ is used to decide at which child we continue. We distinguish three categories of predicates according to their complexity:

Axis-aligned predicates are the simplest and by far the most commonly used type of predicates. They have the form $v_i \le c$ for a constant $c \in \mathbb{R}$. Geometrically speaking, the function $v_i = c$ describes a hyperplane orthogonal to the v_i axis, intersecting at $v_i = c$. That is why they are called *axis-aligned* predicates [29].

To find the best axis-aligned predicate, we make the simple observation that for a feature v_i with k different values, there are only $k - 1$ different relevant values for c. Thus, we can simply evaluate all possible predicates for all features v_i and select the most promising one.

Linear predicates [30] or sometimes called *oblique predicates* have the form $\sum_i a_i v_i \le c$ with $a_i, c \in \mathbb{R}$. This linear combination of different features describes a hyperplane with arbitrary orientation. Hence they are more expressive, but it also makes it harder to find optimal predicates. Algorithms used to find suitable predicates include the OC1 algorithm [30], logistic regression [18, Chap. 4.4], and support vector machines (SVM) [18, Chap. 12] – a machine-learning technique using a hyperplane to split a dataset into two partitions. A hyperplane can be formally defined by the orthogonal vector that goes through the origin $\mathbf{w}$ and the distance from the origin b. Then it can be used as a classifier in the following way:

$$\begin{aligned} c : \mathbb{R}^M &\rightarrow \{-1, 0, 1\} \\ \mathbf{x} &\mapsto \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x} - b), \end{aligned} \tag{3}$$

where sgn is the sign function.

Algebraic predicates as outlined in [1] and implemented in [6, 39], are even more powerful predicates, which can reduce the size of the DT and improve explainability. Algebraic predicates allow any closed-form expressions and hence they are the most expressive. For the same reason, automatically generating good algebraic splitting predicates is difficult and typically requires human guidance.

Figure 1 shows how the three types of predicates work on a toy dataset. As expected, the more expressive the predicate, the fewer predicates are needed to build a perfect classifier.

3 Running example: cruise control

3.1 High-level description

In the cruise control model (in short `cruise`) of [26] (available for download on the website https://people.cs.aau.dk/~marius/stratego/cruise.html), we want to control a car so that it does not crash into another vehicle in front. As a secondary objective, the car should drive as fast as possible, thereby minimizing the distance between both cars.

The model is illustrated in Fig. 3. We only consider two vehicles, our vehicle called *ego* and the next vehicle in front

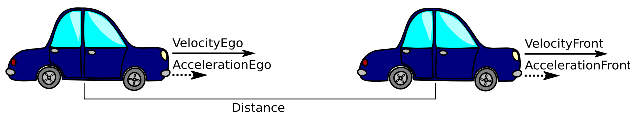


Fig. 3 An illustration of the `cruise` model. Originally appeared in [26]

of us called *front*. We drive on a single lane without cars entering or leaving, therefore this constellation does not change. The state of the system is modeled by the velocities v_e, v_f of the cars and their relative distance d_r; the safety criteria $d_r \geq d_{safe}$ should hold in each state. In the model, both cars choose a constant acceleration a_e, a_f for the duration of one time step t_1. Then, the new state (d_r', v_e', v_f') is given by:

$$v_e' = v_e + a_e t_1, \tag{4}$$

$$v_f' = v_f + a_f t_1, \tag{5}$$

$$d_r' = d_r + (v_f - v_e)t_1 + \frac{1}{2}(a_f - a_e)t_1^2. \tag{6}$$

The model restricts the domains of the accelerations to $a_e, a_f \in \{-2, 0, 2\}$ describing the three actions deceleration, neutral, and acceleration. Similarly, the cars have a bounded minimum and maximum velocity v_{min}, v_{max} and the distance sensor has a limited reach of d_{max}. Depending on the values of these parameters, the size of the generated controller changes considerably. We provide some technical details on the choice of these domains in the next subsection.

When synthesizing a controller for this case study with the tool `UPPAAL Stratego`, we receive a huge lookup table that maps every state of the model (i.e., every tuple (v_e, v_f, d_r)) to the set of safe actions (i.e., a subset of $\{-2, 0, 2\}$). The next subsection discusses the state of the art methods for representing this controller more concisely.

3.2 Insufficiency of the current controller representations

To make the example more concrete, we consider the dataset `cruise_250` (see Sect. 3.4 for technical details). This version of the case study discretizes the state space into 320,523 states. Synthesizing a controller using `UPPAAL Stratego` results in a file with over 400 MiB and comprising 961,569 state-action pairs.[2] `UPPAAL Stratego` only offers to represent the controller as a verbose lookup table explicitly describing every state-action pair. Commonly, controllers are stored as binary decision diagram [12]. However, as discussed in the Related Work section, these are hardly explainable since their decisions are on the bit-level of the state-variables. Moreover, a binary decision diagram for our concrete example still uses over 1,800 nodes.

The tool `dtControl` can read the controller file produced by `UPPAAL Stratego` and process the lookup table into a DT. Different settings of `dtControl` lead to different results: axis-aligned splits yield a DT with 869 nodes and a combination of axis-aligned and linear splits one with 369 nodes, both of which is still too large to be understandable. Using the determinization heuristics discussed in [5], we find a decision tree with only 3 nodes. This controller is perfectly explainable: It simply lets the car decelerate until it reaches minimal velocity. Of course, this behavior satisfies the safety criteria, but it is not helpful in the real world, since it ignores the secondary objective of minimizing the relative distance and thereby actually driving somewhere. So we see that `dtControl` can produce explainable controller representations, but that the determinization can lead to unfavorable results.

To fulfil the secondary objective of minimizing the relative distance, we have two options: We can pre-determinize the controller by always picking the largest safe acceleration, or we keep the maximal permissiveness. In the latter case, the controller acts as an emergency braking system by letting the human driver choose any action as long as it is a safe one. However, recall that without using determinization heuristics, the DT that `dtControl` produces has 369 nodes, so it is too large to be explainable.

3.3 Handwritten strategy

For our running example, we know that there is a small DT representing the most-permissive controller, since it was handcrafted in [1]. It only uses 11 nodes and every predicate is understandable, as it has a clear physical explanation. Before investigating how to generate such a small DT automatically, we illustrate what the necessary predicates look like, as this will guide us in developing our automation process.

In the worst case, the front vehicle starts decelerating in the next time step and continues to do so until it reaches its minimal velocity. For our car, we have to decide what action to take for the next time step t_1: accelerate, stay neutral, or decelerate. To see if it is safe to accelerate, we calculate the relative distance after accelerating for one time step t_1 and then decelerating until the ego vehicle reaches the minimal velocity.

In Fig. 4, we have plotted the velocity-time diagram describing the kinematics of both cars in case the ego vehicle accelerates in the next time step. The front vehicle (red) instantly decelerates with the rate a_{min} and then continues with minimal velocity. The ego vehicle (blue) starts with a higher velocity, accelerates for one step, and then decelerates with the same rate. The distance traveled is the time integral of

[2] Recall that the controller is permissive, i.e., it can allow multiple safe actions for a state. This is why the number of safe state-action pairs is larger than the number of states.

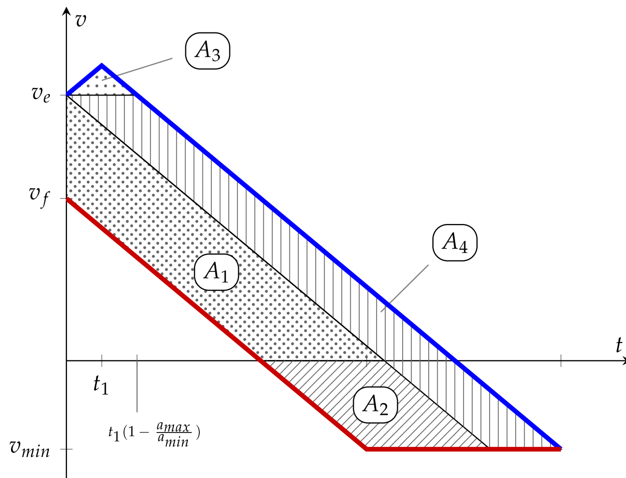


Fig. 4 A velocity-time graph showing the ego vehicle accelerating for one time step. The area between the blue and the red curves describes the change in distance between the two cars

the velocity, so the area between the curves describes the relative distance change. We can partition the area into four sections and calculate the respective areas:

$$A_1 = -\frac{v_e^2}{2a_{min}} - \left(-\frac{v_f^2}{2a_{min}}\right),$$

$$A_2 = v_{min}\frac{v_e - v_f}{a_{min}},$$

$$A_3 = a_{max}t_1^2\left(1 - \frac{a_{max}}{a_{min}}\right),$$

$$A_4 = (v_e - v_{min})\,t_1\left(1 - \frac{a_{max}}{a_{min}}\right).$$

With these values, we can write the predicate deciding whether it is safe to accelerate in the next time step as a quadratic polynomial of our state variables v_e, v_f, d_r.

$$\frac{1}{2a_{min}}v_e^2 - \frac{1}{2a_{min}}v_f^2 - \left(\frac{v_{min}}{a_{min}} + t_1(1 - \frac{a_{max}}{a_{min}})\right) v_e + \frac{v_{min}}{a_{min}}v_f - \left(1 - \frac{a_{max}}{a_{min}}\right) t_1(t_1 a_{max} - v_{min}) + d_r \;\geq\; d_{safe}. \tag{7}$$

3.4 Technical details

The `cruise` model we use in this paper differs slightly from the one provided on the website https://people.cs.aau.dk/~marius/stratego/cruise.html, since the way the case study was modeled warranted some unwanted behaviors. For completeness and reproducibility, we describe our changes and their motivation. This section can be skipped without affecting understanding of the rest of the paper.

Table 1 The parameters used for generating the controllers of the cruise model and the resulting sizes, measured by the number of states and the number of state-action pairs

Name	Parameters			Controller Size	
	v_{min}	v_{max}	d_{max}	#states	#s-a pairs
`cruise_prev`	−10	20	200	295,615	886,845
`cruise_250`	−6	20	250	320,523	961,569
`cruise_300`	−10	20	300	500,920	1,502,760

Domain sizes The `cruise` model is parameterized by the maximum and minimum velocities v_{max} and v_{min} and the maximum distance d_{max}. The version used in [5] and [6], specified these parameters as $v_{max} = 20$, $v_{min} = -10$, and $d_{max} = 200$.

The case study is modeled such that after the front vehicle disappears from our sensor distance, a new car with a random velocity can appear. This, together with domain sizes chosen in previous work, leads to the following unwanted behavior: When the distance between the front and the ego vehicle is at around 150, both cars can drive at full speed. However, when the relative distance approaches 200, the ego vehicle needs to slow down. The reason is that after the front car disappears from sensor distance, the model allows that a new car with arbitrary velocity appears. As a concrete example, this can lead to the following sequence of events:

1. the front vehicle drives with $v_f = 20$ and $d_r = 190$,
2. the front vehicle disappears into the far-away state as $d_r > 200$,
3. a "new" car appears at the end of the sensor range $d_r = 200$. Independent of the velocity the front vehicle had before, the new car can have any velocity, for example $v_f = -10$.

This way, the front vehicle effectively changes its velocity from $v_f = 20$ to $v_f = -10$ in just a couple of time steps. Even the full sensor distance of 200 is not enough for the ego vehicle to react and avoid a crash in this scenario. We fix this flaw by increasing the minimal velocity and the maximum sensor distance so that the ego vehicle has enough time to break if a new car suddenly appears.

Table 1 contains an overview of the parameters used and the resulting size of the controller, measured by the number of states and the number of state-action pairs. The generated controllers are included in [23].

Model modifications Additionally to changing the domain sizes, we made the following small adjustment as in previous work [5, 6]. Since the accelerations are −2,0 or 2 and the time step is 1, all velocities occurring in the model are even. However, when a car appears from the far-away state, it can also have an odd number velocity. To keep the

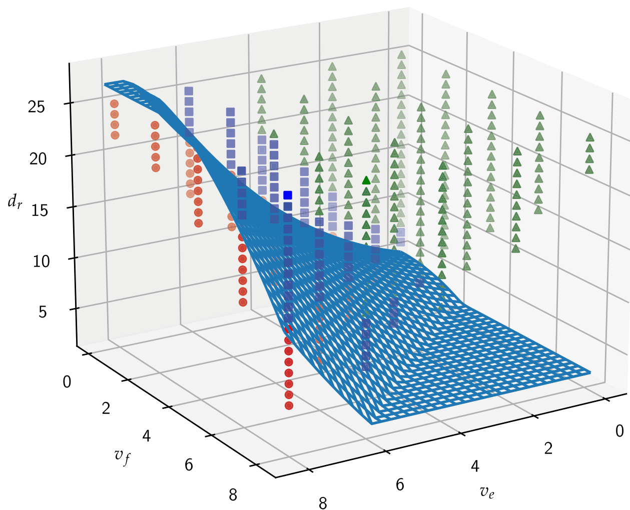


Fig. 5 Visualization of a part of the controller data from the `cruise` case study, together with a handcrafted predicate perfectly separating the data. The coordinate axes describe our three state variables v_e, v_f, d_r and the color of the data points indicates the set of allowed actions: red for $\{-2\}$, blue for $\{-2, 0\}$ and green for $\{-2, 0, 2\}$. The handcrafted strategy perfectly separates the red and blue labels

even velocities, we changed the source code in lines 242ff. to:

```
242  i:int[minVelocityFront/2, maxVelocityFront/2]
243  2*i &lt;= velocityEgo
244  velocityFront = i*2,
245  distance = maxSensorDistance,
246  rVelocityFront = 2 * i * 1.0,
247  rDistance = 1.0*maxSensorDistance
```

The modified model file can be found in [23].

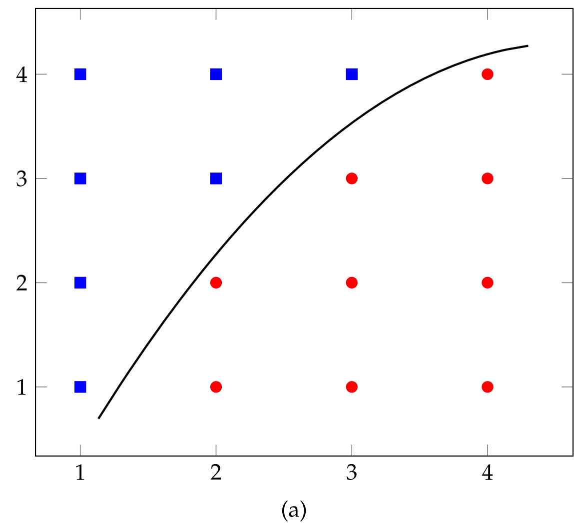
4 Predicates from controller data

Coming up with handcrafted algebraic predicates is a difficult and error-prone process. Indeed, in [4] the authors provided a handcrafted DT with 25 nodes, which was then optimized to one with 11 nodes in the bachelor thesis [1]. Our goal is a process to automatically come up with good algebraic predicates only by looking at the controller.

In Fig. 5, we have visualized a part of the controller data from the `cruise` model together with a handcrafted splitting predicate. This splitting function can be constructed by solely looking at the data and fitting a function to it. We use SVM to perform this task. For completeness, we recall the previous solution of curve fitting in Sect. 4.1 before explaining how SVM can be used and why it is more suited for the task at hand in Sect. 4.2.

4.1 Problems with curve fitting

The recent extensions of `dtControl` [6, 39] enable us to use curve fitting [2] for finding unspecified coefficients. We



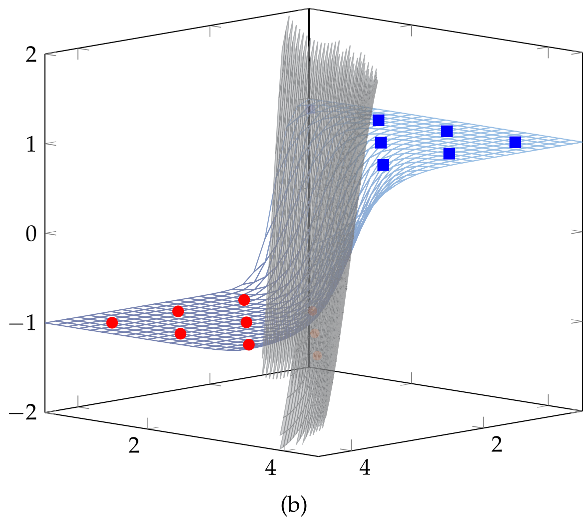


Fig. 6 In the current curve-fitting implementation, a two-dimensional dataset (a) is mapped to a three-dimensional space where the $z \in \{-1, 1\}$ is determined by the label. The old approach then fits a function to the new dataset. Our approach instead tries to separate the data, as the gray surface does in (b)

know from Equation (7) that the handpicked strategy is a quadratic polynomial, so we can try to use a general quadratic polynomial $c_1 v_e^2 + c_2 v_e v_f + c_3 v_f^2 + \cdots$ and determine the coefficients with curve fitting. Unfortunately, this approach fails to find the correct predicate. To understand why, we need to investigate how the curve fitting is implemented.

For now, we always consider a *one versus the rest* split. This means, we pick a label y that we want to separate from the rest and set:

$$y'_i = \begin{cases} +1 & \text{if } y_i = y \\ -1 & \text{else} \end{cases} \quad \text{for all } i.$$

Consider the two-dimensional data from Fig. 6a. What the current version of curve fitting does is the following. First, we

map our two-dimensional data $x_i \in \mathbb{R}^2$ with label $y_i' \in \{-1, 1\}$ to the three-dimensional space where y_i' is used as the third coordinate. Then we use regression analysis to fit a function to the data with least-squares-fitting [27, 28] (see Fig. 6b). In this work, we propose to use a classification approach rather than a regression approach. So, in Fig. 6b, we are interested in the gray function *separating* the data points instead of fitting them. This way, we put most emphasis on the sample points close to the split, rather than weighting every sample equally. Coming back to the two-dimensional space (Fig. 6a), we want to find a function that smoothly separates the labels, ideally maximizing the distance to any specific sample. This is where support vector machines (SVMs) come into play.

4.2 Using support vector machines

Support vector machines do exactly what we want here: find a function that separates the data and maximizes the margins. The main idea of this work is that we can reconstruct the algebraic decision function from the internal coefficients of the SVM. This is not feasible for tools like neural networks [18, Chap. 11], but we will see how and under what conditions it is possible for SVMs in the next sections.

The tool `dtControl` already supports finding linear splitting predicates with SVMs. However, for the `cruise` example, a linear predicate is not enough to perfectly split the data. So we are tempted to use a polynomial kernel to increase the expressiveness of our SVM. However, the runtime of common training algorithms for SVMs is at least quadratic in the number of samples. And in fact, the algorithms implemented in the open-source tool `scikit-learn` [31] do not terminate within an hour for the `cruise` example with a few hundred thousand sample points. Thus, using a general SVM is infeasible, but using only the linear SVM already provided by `dtControl` is insufficient.

We can circumvent the issue by taking advantage of our specific use case. Usually, SVMs are used with high-dimensional datasets like images [15] or language models [32], where the number of features has the same order of magnitude as the number of samples. For the purpose of controller synthesis, the number of state variables is usually small because the number of states grows exponentially with the number of state variables. So, while the kernel trick is useful for high-dimensional data, we can renounce the kernel trick in our case and explicitly construct the higher-dimensional space. A similar idea is also described in [13].

For example, recall that the `cruise` model has the state variables v_e, v_f and d_r, and a controller provides a set of safe actions for every tuple of these variables. We can change this linear state space to be quadratic by adding more terms as follows:

$$(v_e, v_f, d_r, v_e v_f, v_e d_r, v_f d_r, v_e^2, v_f^2, d_r^2)^T . \tag{8}$$

Note that all the new state variables can be directly calculated from the old ones. Thus, we modify the controller by adding (essentially redundant) new state-variables to every line of the lookup table. However, these new state variables allow a linear SVM to find quadratic splitting functions. Thus, we can take advantage of the higher dimensional features while still using the fast linear SVM algorithms, making the computation feasible. This clearly outweighs the moderate increase in dimensions.

Note that in this case, we know, based on our domain knowledge, that polynomials of degree more than two are not necessary. Still, choosing variables for the new state space requires way less manual effort than designing predicates by hand. Moreover, such domain knowledge is not required in general. When evaluating how the approach generalizes in Sect. 6.2, we obtain good results by always using quadratic polynomials, independent of the case study.

To summarize:

- SVM is more suited than curve fitting to find splitting predicates, as it is concerned to find a separating hyperplane, not fit a function to the data.
- On the one hand, the predicates resulting from straightforward application of linear SVM algorithms cannot separate the data. On the other hand, the general SVM algorithms are not performant enough. Thus, we explicitly increase the dimensions by considering the quadratic feature space and then use the more performant linear SVM algorithms.

Problems with higher dimensions At the moment, we only support mapping to the quadratic space, which means our predicates are quadratic polynomials. For higher degree polynomials, our experience is that the gained expressiveness does not justify the significantly increased complexity of the predicates. For example, a cubic predicate with 5 variables already has 55 terms. Even with the methods we will discuss in Sects. 5.1 and 5.2, this predicate will not fulfil our goal of being explainable. Mapping to a space with features like e^x or $\sin(x)$ poses the challenge that we can only fit the coefficient, but not scale the function in x-direction like e^{cx} or $\sin(cx)$, and is therefore left for future work.

Reconstructing the algebraic decision function Assuming that our SVM finds a separating hyperplane that we want to use as a splitting predicate in our DT, how do we reconstruct the algebraic representation? The SVM algorithm finds a hyperplane $(\mathbf{w}^*, b^*)$ with $\mathbf{w}^* \cdot \mathbf{x} - b^* = 0$ where $\mathbf{x}$ corresponds to a transformed set of state variables in the form of Equation (8). This means the w_i are the coefficients of the quadratic polynomial of our state variables.

When implementing it in practice, there is a small intermediate step we want to mention for completeness. For the quadratic optimization algorithm to work properly, the input

data needs to be normalized to have a mean of 0 and a standard deviation of 1. This standardization of course has to be taken into account when exporting the coefficients.

Problems with predicate size The quadratic polynomial predicates we receive by applying linear SVM with the quadratic feature space for the `cruise` example consist of up to 25 terms[3] and may, for example, look like this (rounded to 6 decimal places):

$$\begin{aligned}
&-1.004058e_{choose}d_r + 0.000121d_r^2 + 4.011296e_{choose}v_e \\
&-0.002316d_r v_e + 0.51353v_e^2 + 8.5 \cdot 10^{-5}e_{choose}a_e \\
&-0.000276d_r a_e + 0.002239v_e a_e - 6.4 \cdot 10^{-5}a_e^2 \\
&-3.007296e_{choose}v_f + 0.001317d_r v_f - 0.012358v_e v_f \\
&-0.001334a_e v_f - 0.499783v_f^2 - 0.000224e_{choose}a_f \\
&+0.000261d_r a_f - 0.002111v_e a_f - 6.4 \cdot 10^{-5}a_e a_f \\
&+0.001224v_f a_f + 3.1 \cdot 10^{-5}a_f^2 - 1.004058d_r + 4.011296v_e \\
&+8.5 \cdot 10^{-5}a_e - 3.007296v_f - 0.000224a_f + 23.107387 \le 0
\end{aligned}$$

So the price we pay for a performant algorithm that finds good splits is that the result is hardly explainable. We address this in the next section by providing ways to embellish the predicate and reduce its size.

5 Improving the explainability of the predicates

In the previous section, we described how we can generate predicates from controller data using SVM. In this section, we improve the explainability of these predicates and the resulting DT. For this, we first recall the procedure of generating a DT from a controller, since our improvements touch multiple parts of this process.

The input is a controller, i.e., a function mapping states to sets of allowed actions. These controllers are generated by solving a model (e.g., `cruise`) with a controller synthesis tool (e.g.,`UPPAAL Stratego`).

The DT learning algorithm (see also Sect. 2.1) has two important parameters: the set of predicates it considers and the impurity measure, which is used to evaluate how good predicates are. Using SVM as described in the previous section, we can generate new predicates[4] that are very useful,

[3] Note that this is because there are unnecessary variables a_e, a_f, f_{choose} and e_{choose}, see Sect. 5.1 for more details. Theoretically, predicates could even use up to $9 \cdot 4 = 36$ predicates in this example.

[4] Note that we generate multiple predicates, because we use the 'one-versus-rest' approach, i.e., we train an SVM for trying to split of every single label.

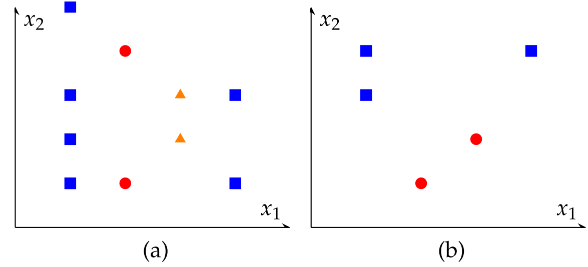


Fig. 7 Examples with redundant features. In (a) x_2 is not needed. In (b) x_1 and x_2 individually look redundant, but only one may be removed. Also, removing x_1 is preferred over removing x_2

but also very long. In Sects. 5.1 and 5.2, we shorten these predicates by removing unimportant terms and rounding the coefficients to nice numbers. Then we describe a new impurity measure named *min-label entropy* in Sect. 5.3. Intuitively, it makes the DT learning algorithm prefer splits that completely separate one label, i.e., one set of allowed actions, and thus improves the explainability of the resulting DT. Finally, we introduce the *predicate priority*, which allows us to further fine tune which predicates are selected. Note that the latter two improvements are independent of using SVM for predicate generation; still, they go well together with it.

5.1 Feature importance

As we discussed in Sect. 3, the state of the `cruise` model is defined by v_e,v_f, and d_r. However, the model checker `UPPAAL Stratego` also exposes four additional state variables. These comprise the current acceleration values a_e,a_f that do not impact the acceleration the cars choose at the next time step, and the variables f_{choose} and e_{choose} that are an artifact from the internal model and have constant values for all relevant states.

To recognize such unimportant variables, we introduce a basic version of a feature importance measure. Consider the two-dimensional dataset shown in Fig. 7a with features x_1 and x_2. To classify a data point, feature x_2 is not needed. We verify this by removing feature x_2 and grouping the data points with the same x_1 value. We can now measure how many 'collisions' occur. If zero collisions occur, the feature is not needed. Otherwise, we can give a rough estimate of the importance of that feature by calculating the ratio of data points where a collision happened.

Note that for a dataset like Fig. 7b, this approach would judge both features as irrelevant. Individually, this is correct, but we can only remove one of them without causing collisions. This is why we calculate the feature importance incrementally. When we find an irrelevant feature, we remove it directly before calculating the importance of the next feature. As a result, the outcome may depend on the order of features we choose. For example, in Fig. 7b, removing x_1

would result in a linearly separable dataset, while removing x_2 would not. In general, there might even be a case where we can either remove a single feature x_i or all three features $x_{i+1}, x_{i+2}, x_{i+3}$. However, we have not observed such behavior so far, so we leave this issue for future work.

5.2 Rounding coefficients

With the feature importance, we remove variables that are clearly useless and reduce the number of terms in the `cruise` predicates from 25 to 9 (namely, exactly the features given in Equation (8)). Removing unimportant features in the previous example simplifies it like this:

$$
\begin{aligned}
&-0.000463d_r^2 + 0.008656d_r v_e - 0.549255v_e^2 - 0.005078d_r v_f \\
&+ 0.046916v_e v_f + 0.496888v_f^2 + 2.043519d_r - 10.25286v_e \\
&+ 6.138132v_f - 39.685041 \leq 0
\end{aligned}
$$

Still, we generate predicates that contain unnecessary terms. For example, we know from our handcrafted predicate (see Equation (7)) that we do not need a d_r^2 term for the `cruise` predicates. However, in the predicate, the respective coefficient has a small positive value. To understand why this is the case, recall that the only objective the SVM has is to maximize the margin between the data points. For that, a small coefficient for d_r^2 seems to be beneficial. If we loosen the maximum margin objective, we can generate a predicate with equivalent accuracy but a simpler algebraic expression. Again, as we are not interested in the classifier's ability to generalize – as long as the accuracy for our controller data stays the same, we do not care about how large the margins are.

So, to embellish our predicate, we proceed in three steps:

1. Setting coefficients to zero.
2. Scaling the entire predicate.
3. Rounding coefficient to integers or nice numbers.

Rounding to zero If we can set a coefficient to zero, the predicate becomes significantly shorter and easier to understand. So this is our primary goal. A natural approach is to try setting a coefficient with a small absolute value to zero and checking if the classification for all samples stays the same. While this suffices for some coefficients, sometimes we need to change the remaining ones to counterbalance the change. Therefore, we temporarily remove the feature and try to re-train the SVM. If we are successful, we permanently remove the feature for this split and try the next feature. Similar to the feature importance approach (Sect. 5.1), the result may again depend on the order of coefficients we are trying to remove. Here, we use the heuristic of trying to remove the coefficient with the smallest absolute value first.

Compared to the feature importance approach, three key differences make this approach more powerful:

- We only consider the subset of the entire dataset available in the current subtree.
- We only focus on separating one specific label (we only have the two labels $+1$ and -1).
- We directly consider the features in the higher-dimensional space such as d_r^2.

Scaling the predicate An additional step to improve readability is to scale the generated predicate. In principle, a predicate $\alpha : 0.5x + 0.1y \leq 0.3$ is equivalent to a scaled predicate $10\alpha : 5x + y \leq 3$ but the second one might be easier to read. The SVM uses an internal scaling constraint but for us this is not relevant. We can again lift this constraint and scale all coefficients as well as the intercept value b arbitrarily. One could think of various heuristic of how to scale the predicate. We decided to use a simple one: We search for the coefficient with the value closest to 1 and scale the predicate so that it becomes exactly 1. This way we have at least one term with a simple coefficient.

General rounding As the last step, we generalize the "rounding to zero" approach and use it on the coefficient we could not set to zero. This way, instead of having a predicate like $8.165839d_r^2 - 2.935846v_r \leq 0$, we can use a better looking one like $8d_r^2 - 3v_r \leq 0$. For that, we try the approach from above with increasing relative precision. For example, for the coefficient of d_r^2, we first try the value 10, then 8, then 8.2, and so on, until we find a value that does not change the classification for any sample. Note that we do not re-train the SVM in this step, but simply change the coefficient and check if the classification stays the same.

With these techniques, we can finally generate pretty predicates. For example, one the predicate we find for the `cruise_250` dataset exactly corresponds to the handcrafted polynomial from Equation (7) after substituting all constants. The only difference is the constant offset.

$$
-0.25v_e^2 + 0.25v_f^2 - 5v_e + 3v_f + d_r + 19.5 \leq 0 \qquad (9)
$$

Numerical errors The rounding procedures introduce the possibility of numerical floating point precision errors. When testing our classifier, we use the internal coefficients of the SVM. The coefficients we output are different though, as we need to undo the normalization we applied. We must ensure that possible precision errors from these transformations do not change the classification. In the original predicate generated by the SVM, the classifier maximizes the margin between the label sets, so we can be quite confident that small precision errors will not change the classification.[5] When trying out rounded coefficients, however, we lose this property. A rounded coefficient might classify everything correctly,

[5] Note that this only holds for cases where we find a perfect split.

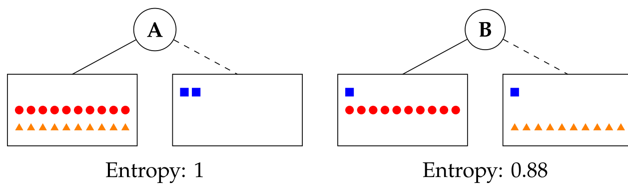


Fig. 8 Two different splits with their respective entropy values. While split B has a better entropy value and is preferred in machine learning, we want to use split A first when building a perfect classifier

but the slightly different transformed coefficient might lead to other results.

As a heuristic against these problems, we over-approximate a change when we try a rounded coefficient. For example, if our current coefficient is 2.953 and we want to try the rounded value 3, we over-approximate the change and try 3.00001 instead. If that works, we can be more confident that the value 3 will not lead to those problems.

Additionally, there can also be numerical errors if the SVM uses very large coefficients (e.g., 10^{18}). We slightly adapted the regularization process to get good performance while also avoiding such large coefficients.

When the tree construction is finished, `dtControl` verifies that every sample is classified correctly or outputs an error rate. In this step, we use the transformed coefficients of the polynomial we output so we can be sure that the DT is as accurate as the tool tells the user.

5.3 Min-label entropy

Now that we have pretty predicates, we shift our focus to the selection of predicates for the next two sections. For the `cruise` dataset, we can construct a decision tree with 37 nodes, only 10% of the size when using linear predicates. Moreover, we have seen that the approach generates the exact predicates we derived by hand in Sect. 3.3. Still, the decision tree is not as compact as the 11-node tree from [1], as we do not directly use those predicates. To understand why this is the case, let us take a look at Fig. 8. We see that split A perfectly separates the blue label from the rest, while B separates the red and orange labels but distributes the blue one among both children. Considering only a single split, we would prefer split B because the dataset is well separated except for the small number of blue samples. The entropy impurity measure comes to the same conclusion and assigns split B a better entropy score.

However, when building a perfect classifier for representing the most-permissive controller, we have a different perspective than in machine learning. At some point, we need to separate the blue labels from the rest. If we do not separate them now and select split B, we have to add additional splits on *both sides* of the split B. If we rather start with split A, we can select split B as the next split in the left child and thus receive a smaller DT.

This effect is especially prevalent if the number of samples per label differs significantly. In the `cruise` example, we observe exactly that: the label “all actions are allowed” has 20 times more data points than any of the other labels. Hence, we introduce a new impurity measure that we call *min-label entropy*:

Definition 3
For a dataset $X = X_l \uplus X_r$ with the label set B, let $n(X, y)$ describe the number of data points in X with label $y \in B$. For a predicate α that splits the dataset into X_l and X_r, we define the *min-label entropy* H^* as:

$$K(p) := -p \log_2(p),$$

$$H^*(X, y) := K\left(\frac{n(X, y)}{|X|}\right),$$

$$H^*(\alpha, X) := \min_{y \in B} \left\{ \frac{|X_l|}{|X|} H^*(X_l, y) + \frac{|X_r|}{|X|} H^*(X_r, y) \right\}. \quad (10)$$

Intuitively, the *min-label entropy* measure estimates for every label y, how difficult it will be to separate the label y in both partions after this split. Then it returns the value of the best label. The strategy we want to provoke with this impurity measure is to first fully separate one label and then continue with the next one. Specifically, if we can completely separate one label like in the example in Fig. 8, the impurity for this split is 0 and we definitely select such a spilt.

5.4 Predicate priority

With the min-label entropy, we reduce the DT size of the `cruise` example to 25. As a last optimization heuristic, we also adjust the priorities of the predicates. When deciding between an axis-aligned and a polynomial predicate, both of which have similar impurity values, we want to choose the axis-aligned one as it is considerably simpler to understand. For that reason, `dtControl` has implemented a priority function for predicate generators. For example, when we give the polynomial predicates the priority 0.5 and the axis-aligned ones the priority 1, we only choose a polynomial predicate if it is at least twice as good in terms of the impurity measure. In fact, we want to choose an even lower value as a priority for another reason. In the `cruise` example, we know that we can find a polynomial that distinguishes cases where we can accelerate from those where we cannot. However, in our handpicked strategy, we have not considered the edge cases when we are already driving at minimal or maximal velocity. If we do not exclude those, the data will not be perfectly separable, meaning we will find a polynomial split that almost classifies everything correctly, but misses a few data points.

While this is not a huge problem, it turns out that it is more effective to first exclude the edge cases with axis-aligned predicates and then perfectly split the data with a complex predicate later. We can achieve it with a low priority value ≤ 0.2 for the polynomial splits in combination with our min-label entropy. This way, we will only choose the complicated splits if they are at least 5 times better. Note that the impurity is 0 if we can perfectly separate one label. So in this case, we are infinitely better than any non-perfect solution.

6 Experimental evaluation

In this section, we will evaluate how well generating quadratic polynomials with SVMs performs in practice. While developing the various techniques and heuristics, we mainly focused on the `cruise` dataset. Thus, we first analyze the results for this dataset in Sect. 6.1 and afterwards investigate how well the approach generalizes to other case studies. We split the latter evaluation into Sect. 6.2 about the performance of the whole approach when compared to the state-of-the-art and Sect. 6.3 specifically investigating the improvements of min-label entropy and predicate priority. We conclude our evaluation with a comment on explainability in Sect. 6.4.

Artifacts All resources, such as generated domain knowledge predicates, model files, and synthesized controllers used in this paper, are available to download at [23]. The repository also contains scripts to reproduce the benchmark tables presented in this paper.

Minimum tree size To better understand the quality of our results, we also compare them to the theoretical minimum-sized DTs. We can give a lower bound on the number of nodes the DT must contain if we want to represent the entire controller without determinizing (i.e., the most-permissive controller) as follows: At every state s, a subset of actions $C(s) \subseteq \mathcal{A}$ is allowed. We define $U := \{C(s) \mid s \in \mathcal{S}\}$ as the set of all possible allowed action subsets that occur in our controller in at least one state. To completely represent the controller, we need at least one distinct leaf in our DT for every distinct element $u \in U$. If we disregard the non-binary splits for categorical variables introduced in [6], we always build a full binary DT. As a full binary tree with n leaves has $n-1$ inner nodes, the lower bound for the total number of nodes of our DT is $2|U|-1$.

If the decision predicates are sufficiently complex, we can always achieve this bound. However, in practice, this is often not even desirable. For example, we see that our DT for the cruise example in Fig. 9a has no minimum size as it contains two leaves with the actions $a \in \{-2, 0\}$. Still, to keep an explainable DT, we would not want to merge those leaves, as one describes that the car cannot accelerate because it has already reached its maximum velocity, while in the other case accelerating would be technically possible but would lead to unsafe behavior.

6.1 Cruise control

Using all the strategies discussed in Sect. 4, we achieve great results for the `cruise` model. For the `cruise_250` dataset, we find a succinct DT with only 11 nodes (see Fig. 9a). This is exactly the number of nodes [1] found with the handcrafted strategy. In fact, we precisely found the handcrafted ‘must break’ and ‘can accelerate’ predicate from the handcrafted strategy, with only a small difference in the constant offset.

For the slightly larger `cruise_300` dataset, we generate a very similar but slightly larger DT with 13 nodes (Fig. 9b). The quadratic predicates change in line with the change of the constant v_{min} (see Sect. 3.4 for a description of how the dataset was changed), and one complex splitting predicate is exchanged for two simpler predicates.

In both cases, the generated DTs are almost 80 times smaller than the ones we obtain with axis-aligned predicates, and 30 times smaller than the ones with linear predicates.

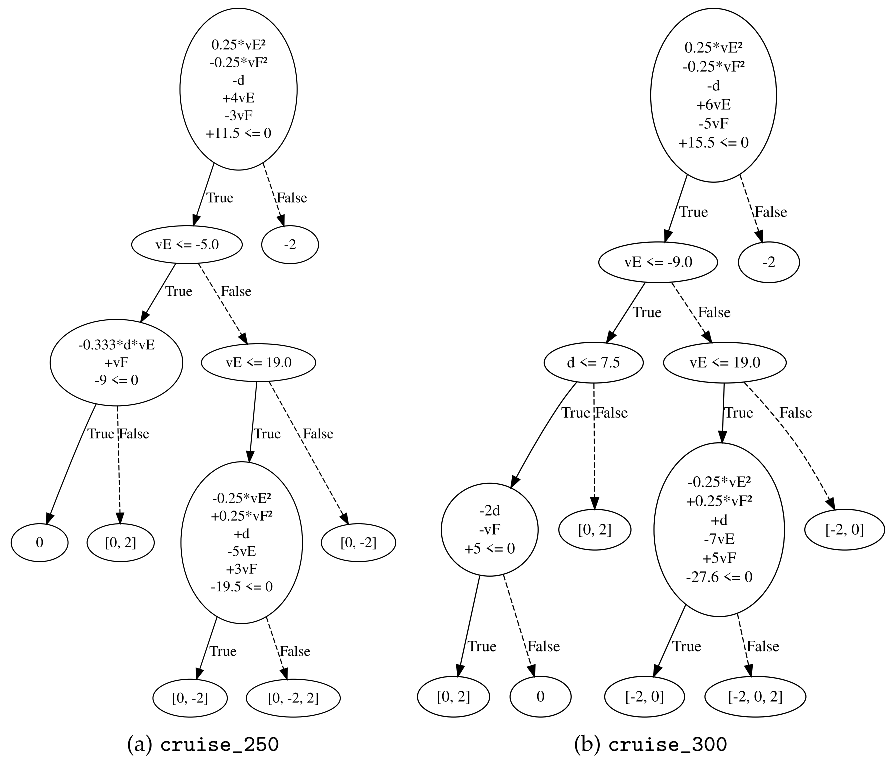
6.2 Generalizing to other benchmarks

To see how our approach and the individual heuristics generalize, we evaluate them on the case studies of cyber-physical systems from [5] as well as on the case studies from the quantitative verification benchmark set [17] that were used in [6]. We avoid using any determinization heuristics so that we generate the most-permissive controllers. We ran all experiments on a server with the operating system Ubuntu 20.04, a 2.2 GHz Intel(R) Xeon(R) CPU E5-2630 v4, and 250 GB RAM. Table 2 contains a selection of the results, with the case studies of cyber-physical systems at the top and quantitative verification at the bottom. In every row, we compare the number of nodes in the generated DT for

- the axis-aligned splitting strategy (Ax.Al.);
- the smallest DT we could generate with axis-aligned and linear predicates[6] (Linear);
- axis-aligned predicates and the quadratic polynomials generated by support vector machines with a priority value of 0.1 (Poly);
- and with the default priority value of 1.0 (PolyPrio1).

In every cell, the top number describes the result using the entropy impurity measure and the bottom number refers to the result using min-label entropy. TO indicates that we were

[6] We calculate this as the minimum over the three splitting strategies logistic regression, linear support vector machines, and the OC1 heuristic.



(a) `cruise_250`

(b) `cruise_300`

Fig. 9 The decision trees for the `cruise` example generated by our data-driven approach

unable to generate a DT within three hours. For comparison, we list the number of states of the controller as well as the theoretical minimum size of the DT.

For completeness, we provide the full experimental results in the Appendix, including all 28 case studies, a comparison with BDDs, and results for different linear strategies.

Scatter plot To complement the table, Fig. 10 visualizes the results in a logarithmic scatter plot. As a reference, we take the smallest tree we could generate with linear predicates and the entropy impurity. Then we compare it to the size of the tree with axis-aligned predicates and our quadratic polynomials. For example, the two blue points near the location $(370, 10)$ are the two cruise datasets. The x-coordinate is the size of the tree with linear predicates and the y-coordinate shows the size of the polynomial or axis-aligned results.

Analysis Our new approach gives smaller DTs for almost all case studies, with the exception of `helicopter` and `cdrive.10`, where the linear solution is smaller by 6%, and `traffic_30m`, where we run into a timeout (see the full table of results in the Appendix). In Table 3 we show the cumulated statistics. Most notably, the number of cases where we find a tree of minimum size has increased from 2 to 10 out of 28.

6.3 Min-label entropy and predicate priority

We applied two significant changes to arrive at the small DTs in the `cruise` example: the min-label entropy impurity measure and the modified predicate priority value. We now analyze how useful they are for the other case studies.

In Fig. 11, we again make use of a logarithmic scatter plot to visualize the data from our tables. As a baseline, we take the size of the DT, generated by our proposed approach using the entropy impurity measure and the default priority 1.0. We compare it to the size when using the proposed min-label entropy (blue) and when using the reduced priority value 0.1 (red).

Min-label entropy The min-label entropy reduces the tree size in 14 out of 17 cases (82%) where we are not already at the minimum size and do not run into a timeout. Interestingly, this behavior is different when using the min-label entropy with axis-aligned splits or linear splits. There, the min-label entropy can only improve the result in 30 out of 106 cases (28%).

Also, we observe 5 cases where our approach only times out when using the min-label entropy, but not when using the standard entropy. A reason for this might be that the min-label entropy encourages the formation of DTs formed like a line. For all case studies where we generate minimum-sized trees like the `10rooms` case study, every leaf has a unique label. With the min-label entropy impurity, every splitting

Table 2 The number of nodes of the generated decision trees using axis-aligned splits, linear splits, and the proposed quadratic polynomial splits with priority 0.1 and 1.0. Each row displays the result using the entropy impurity measure at the top and using min-label entropy at the bottom. TO means time out after 3 hours. As a comparison, we show the number of states of the underlying controller and the theoretical minimum size a decision tree needs to have. The full table is in the Appendix

Case Study	Comparision		Previous		Quadratic	
	States	MinSize	Ax.Al.	Linear	Poly	PolyPrio1
cartpole [22]	271	**169**	253	183	243	189
			263	187	**169**	**169**
10rooms [21]	26,244	**49**	17,297	147	61	61
			17,297	107	**49**	**49**
helicopter [22]	280,539	475	6,339	**3,769**	5,035	3,787
			9,649	4,637	TO	TO
cruise_250 [26]	320,523	9	869	369	353	37
			1,067	363	**11**	25
dcdc [35]	593,089	5	271	139	**129**	199
			265	173	147	273
truck_trailer [24]	1,386,211	1,839	**338,283**	TO	TO	TO
			366,411	TO	TO	TO
aircraft [36]	2,135,056	31	915,877	916,685	725,011	**602,335**
			1,015,903	1,013,949	688,577	630,631
pacman.5	232	**37**	53	49	47	**37**
			81	59	**37**	**37**
philosophers-mdp.3	344	59	391	333	315	251
			403	367	251	**223**
ij.10	1,013	19	1,291	753	897	209
			1,405	735	**141**	177
elevators.a-11-9	14,742	129	16,341	9,865	9,779	2,859
			17,809	9,955	2,023	**1,919**
exploding-blocksworld.5	76,741	149	16,913	2,687	4,511	**829**
			20,273	2,845	TO	TO
wlan_dl.0.80.deadline	189,641	175	3,369	701	693	667
			3,675	2,841	**523**	TO
pnueli-zuck.5	303,427	173	171,371	156,165	114,979	**83,219**
			263,955	221,645	95,879	83,951

predicate separates out one of those labels. Thus, the tree looks like a line. As a consequence, the runtime for finding predicates does not decrease as fast while constructing the tree. When we construct a perfectly balanced tree, the size of the dataset left at the subtree at depth d is only a small fraction (2^{-d}) of the original size. In the case of a line, however, the dataset size only decreases slowly.

Low priority heuristic While the low priority value helps in the cruise example in combination with the min-label entropy, the only other cases where this heuristic brings an improvement are the `dcdc` and `eajs.2.100.5.ExpUtil` case studies (see the full table of results in the Appendix). We conclude that our motivating idea of first separating the "outliers" and then using the more sophisticated splits later does not generalize well. Apparently, it is beneficial to just take the best available split right away in complex models.

6.4 Explainability

We have seen that we can significantly reduce the number of DT nodes with our proposed approach. But how explainable are the trees we generated?

Of course, reducing the number of decision nodes already helps to create an explainable DT. Still, we have to consider that the complexity of the individual splitting predicate increases, thereby potentially reducing explainability. As an example, we consider the `10rooms` case study. Here, we find

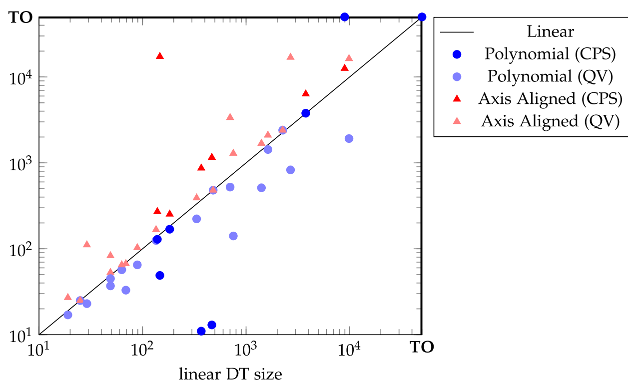


Fig. 10 Performance comparison of different predicate types. Based on the decision tree size using linear predicates, we compare how many nodes the decision trees with axis-aligned splits and quadratic polynomials have. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV)

a DT with 49 nodes, which is the minimum size for a most-permissive DT. Unfortunately, the DT is not particularly explainable, as some predicates comprise up to 35 terms, even after trying to round coefficients to zero. The reason is the large number of 10 state variables. A quadratic polynomial with ten variables can already have 65 terms.

Regardless of the complexity of individual predicates, for some case studies, the minimum DT size is already too large to be easily understandable by a human. Any most-permissive DT for the case studies `helicopter` and `truck_trailer` will have more than 400 and 1,800 nodes, respectively. So, in these cases, we might need to investigate determinized controllers, as discussed in [5, 6].

7 Conclusion

In this work, we have investigated an approach to generate expressive algebraic splitting predicates for decision trees. We proposed learning quadratic polynomials with support vector machines directly from the controller data. Additionally, we introduced a new impurity measure called *min-label entropy* that focuses on separating one specific label first. We integrated both ideas into the decision-tree learning algorithm and implemented it as an extension of the open-source tool `dtControl`. We were able to generate significantly smaller DTs in cases where the determinization heuristics could not be applied. For the cruise model, we generated a tree with the same size as the one created with help of a human expert, and in 10 out of 28 case studies we even found a DT of minimum size.

On the one hand, we showed that more expressive quadratic polynomials can help to generate succinct trees for permissive controllers. On the other hand, a key aspect still to be improved upon is the explainability. Of course, succinct DTs are already easier to understand by nature, but more complex predicates again reduce explainability. Ideally, we would want to automatically generate a justification explaining the coefficients of each complex predicate.

Table 3 Cumulated statistics over all 28 benchmarks. We compare the best linear strategy with entropy impurity with the best of our heuristics

	Linear	Quadratic
Timeout	1	2
Minimal DT	**2** (7%)	**10** (35%)
DT is smaller or equal		**25** (89%)
DT has less than half the size		**8** (29%)

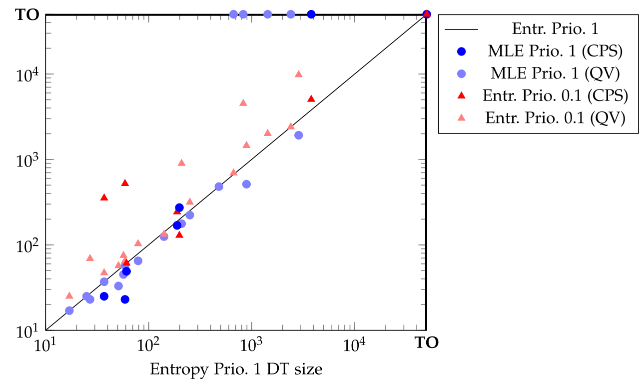


Fig. 11 Performance comparison of the min-label entropy (MLE) and the low priority heuristic. Based on the decision tree size using quadratic polynomials as predicates with entropy and priority 1.0, we compare how the heuristics change the tree size. Every sample corresponds to a case study of cyber-physical systems (CPS) or originates from the quantitative verification benchmark set (QV)

Appendix: Full experimental results

In the main body, we condensed the results of our experimental evaluation in order to improve readability. Here we list the benchmark results for all case studies and approaches. The structure is the same as described in Sect. 6.2 with two slight differences. Firstly, we also give the size of the binary decision diagram (BDD) representation. Secondly, we explicitly list the tree sizes we get with the linear support vector machine, the logistic regression, and the OC1 heuristic splitting strategies. In Table 2 in the main body, we only show the minimum across those.

The BDD sizes for the cyber-physical system cases are the minimum number of nodes from 10 tries. For the quantitative verification case studies, we show the BDD sizes from [6], which correspond to the minimum across 20 tries.

The benchmarks are split into three tables. Table 4 contains the cyber-physical system case studies, Table 5 and Table 6 contain case studies from the quantitative verification benchmark set [17].

Table 4 Benchmark results for the cyber-physical system case studies

Case Study	Comparision				Linear			Quadratic	
	States	BDD	MinSize	Ax.Al.	LinSVM	LogReg	OC1	Poly	PolyPrio1
cartpole [22]	271	312	**169**	253	247	199	183	243	189
				263	263	187	261	**169**	**169**
10rooms [21]	26,244	168	**49**	17,297	157	147	4,515	61	61
				17,297	121	107	7,455	**49**	**49**
helicopter [22]	280,539	**1,348**	475	6,339	5,787	3,769	TO	5,035	3,787
				9,649	9,763	4,637	TO	TO	TO
cruise_250 [26]	320,523	1,820	9	869	721	557	369	353	37
				1,067	817	657	363	**11**	25
cruise_300 [26]	500,920	2,229	9	1,157	991	691	467	521	59
				1,343	1,035	881	507	**13**	23
dcdc [35]	593,089	575	5	271	279	139	179	**129**	199
				265	265	173	179	147	273
truck_trailer [24]	1,386,211	**36,169**	1,839	338,283	TO	TO	TO	TO	TO
				366,411	TO	TO	TO	TO	TO
aircraft [36]	2,135,056	**177,332**	31	915,877	916,685	TO	TO	725,011	602,335
				1,015,903	1,013,949	TO	TO	688,577	630,631
traffic_30m [38]	16,639,662	TO	23	12,573	9,631	8,953	TO	TO	TO
				20,895	9,211	**7,099**	TO	TO	TO

Table 5 Benchmark results for case studies from the quantitative verification benchmark set (part 1)

Case Study	Comparision			Single Feature		Linear			Quadratic	
	States	BDD	MinSize	Ax.Al.	Categ.	LinSVM	LogReg	OC1	Poly	PolyPrio1
triangle-tireworld.9	48	51	**17**	27	28	25	23	19	25	**17**
				31	31	21	25	23	**17**	**17**
pacman.5	232	330	**37**	53	43	51	49	55	47	**37**
				81	81	71	59	81	**37**	**37**
rectangle-tireworld.11	241	495	481	481	**373**[1]	481	481	481	481	481
				481	481	481	481	481	481	481
philosophers-mdp.3	344	295	59	391	181	381	377	333	315	251
				403	307	375	367	393	251	**223**
firewire_abst.3.rounds	610	295	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
				25	**25**	**25**	**25**	**25**	**25**	**25**
rabin.3	704	303	**23**	111	187	51	43	29	69	27
				175	137	31	29	45	**23**	**23**
ij.10	1,013	436	19	1,291	1,291	907	753	771	897	209
				1,405	1,405	893	735	1,131	**141**	177
zeroconf.1000.4.true.correct_max	1,068	535	**45**	83	83	67	63	49	75	57
				79	79	47	**45**	71	**45**	**45**
blocksworld.5	1,124	3,985	367	1,687	1,308	1,515	1,407	1,583	1,451	891
				1,771	1,649	1,535	1,405	1,719	521	**513**

[1]This is better than the minimum size as it uses non-binary splits

Table 6 Benchmark results for case studies from the quantitative verification benchmark set (part 2)

Case Study	Comparision			Single Feature		Linear			Quadratic	
	States	BDD	MinSize	Ax.Al.	Categ.	LinSVM	LogReg	OC1	Poly	PolyPrio1
cdrive.10	1,921	5,134	1,903	2,401	3,122	2,401	2,401	**2,257**	2,401	2,401
				2,089	2,037	TO	TO	TO	TO	TO
consensus.2.disagree	2,064	138	25	67	67	75	69	69	57	51
				105	105	105	93	95	35	**33**
beb.3-4.LineSeized	4,275	913	**57**	65	70	65	65	63	65	59
				85	76	**57**	**57**	89	**57**	**57**
csma.2-4.some_before	7,472	1,059	**65**	103	103	107	105	89	103	79
				185	185	85	**65**	177	**65**	**65**
eajs.2.100.5.ExpUtil	12,627	1,315	65	167	160	173	161	135	133	141
				167	167	167	157	133	133	**125**
elevators.a-11-9	14,742	6,750	129	16,341	16,413	11,243	9,865	13,619	9,779	2,859
				17,809	17,495	11,505	9,955	16,423	2,023	**1,919**
exploding-blocksworld.5	76,741	3,447	149	16,913	8,138	4,503	2,687	5,993	4,511	**829**
				20,273	8,571	5,307	2,845	6,893	TO	TO
echoring.MaxOffline1	104,892	43,165	801	2,101	2,251	1,629	1,625	1,627	2,005	**1,431**
				5,031	TO	TO	TO	TO	TO	TO
wlan_dl.0.80.deadline	189,641	1,541	175	3,369	3,369	2,821	2,563	701	693	667
				3,675	3,675	2,841	2,621	1,049	**523**	TO
pnueli-zuck.5	303,427	**50,128**	173	171,371	171,371	156,165	150,341	125,421	114,979	83,219
				263,955	263,955	221,645	214,801	221,645	95,879	83,951

Funding Open Access funding enabled and organized by Projekt DEAL. This research was funded in part by the German Research Foundation (DFG) projects 383882557 *Statistical Unbounded Verification (SUV)* and 427755713 *Group-By Objectives in Probabilistic Verification (GOPro)*. This paper extends the tool dtControl [6] and through the synergy of algebraic, formal-methods and machine-learning approaches it increases the explainability of controllers, positioning itself into the STTT theme area *Explanation Paradigms Leveraging Algebraic Intuition (ExPLAIn)*.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Akmese, S.M.: Generating richer predicates for decision trees. Bachelor's thesis, Technical University of Munich (2019)
2. Arlinghaus, S.: Practical Handbook of Curve Fitting. Taylor & Francis, London (1994)
3. Ashok, P., Brázdil, T., Chatterjee, K., Křetínský, J., Lampert, C.H., Toman, V.: Strategy representation by decision trees with linear classifiers. In: Parker, D., Wolf, V. (eds.) Quantitative Evaluation of Systems, pp. 109–128. Springer, Cham (2019)
4. Ashok, P., Kretínský, J., Guldstrand Larsen, K., Le Coënt, A., Taankvist, J.H., Weininger, M.: SOS: safe, optimal and small strategies for hybrid Markov decision processes. In: Parker, D., Wolf, V. (eds.) Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Proceedings, Glasgow, UK, September 10-12, 2019, Lecture Notes in Computer Science, vol. 11785, pp. 147–164. Springer Berlin (2019)
5. Ashok, P., Jackermeier, M., Jagtap, P., Křetínský, J., Weininger, M., Dtcontrol, M.Z.: Decision tree learning algorithms for controller representation. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, HSCC'20. Association for Computing Machinery, New York (2020)
6. Ashok, P., Jackermeier, M., Kretínský, J., Weinhuber, C., Weininger, M., Yadav, M.: dtcontrol 2.0: explainable strategy representation via decision tree learning steered by experts. In: TACAS (2). Lecture Notes in Computer Science, vol. 12652, pp. 326–345. Springer, Berlin (2021)
7. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Form. Methods Syst. Des. **10**(2/3), 171–206 (1997)
8. Bennett, K.P., Blue, J.A.: A support vector machine approach to decision trees. In: 1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98CH36227), vol. 3, pp. 2396–2401 (1998)
9. Bollig, B., Wegener, I.: Improving the variable ordering of obdds is np-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
10. Brázdil, T., Chatterjee, K., Chmelik, M., Fellner, A., Kretínský, J.: Counterexample explanation by learning small strategies in Markov decision processes. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Confer-

ence, CAV 2015,, Proceedings, Part I, San Francisco, CA, USA, July 18-24, 2015, Lecture Notes in Computer Science, vol. 9206, pp. 158–177. Springer Berlin (2015)
11. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth, Belmont (1984)
12. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
13. Chang, Y.-W., Hsieh, C.-J., Chang, K.-W., Ringgaard, M., Lin, C.-J.: Training and testing low-degree polynomial data mappings via linear SVM. J. Mach. Learn. Res. **11**, 1471–1490 (2010)
14. David, A., Gjøl Jensen, P., Guldstrand Larsen, K., Mikucionis, M., Haahr, J.: Taankvist. Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, Proceedings, London, UK, April 11-18, 2015, Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer Berlin (2015)
15. DeCoste, D., Schölkopf, B.: Training invariant support vector machines. Mach. Learn. **46**(1–3), 161–190 (2002)
16. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Proceedings, Part II, Heidelberg, Germany, July 24-28, 2017, Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer Berlin (2017)
17. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings, Part I, Prague, Czech Republic, April 6-11, 2019, Lecture Notes in Computer Science, vol. 11427, pp. 344–350. Springer, Berlin (2019)
18. Hastie, T., Tibshirani, R., Friedman, J.H.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd edn. Springer Series in Statistics. Springer, Berlin (2009)
19. Ittner, A., Schlosser, M.: Non-linear decision trees - NDT. In: Saitta, L. (ed.) Machine Learning, Proceedings of the Thirteenth International Conference (ICML'96), Bari, Italy, July 3–6, 1996, pp. 252–257. Morgan Kaufmann, San Mateo (1996)
20. Jackermeier, M.: dtcontrol: Decision tree learning for explainable controller representation. Bachelor's thesis, Technical University of Munich (2020)
21. Jagtap, P., Zamani, M.: QUEST: a tool for state-space quantization-free synthesis of symbolic controllers. In: Bertrand, N., Bortolussi, L. (eds.) Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Proceedings, Berlin, Germany, September 5-7, 2017, Lecture Notes in Computer Science, vol. 10503, pp. 309–313. Springer, Berlin (2017)
22. Jagtap, P., Abdi, F., Rungger, M., Zamani, M., Caccamo, M.: Software fault tolerance for cyber-physical systems via full system restart. ACM Trans. Cyber Phys. Syst. **4**(4), 47:1–47:20 (2020)
23. Jüngermann, F.: Learning Algebraic Predicates for Explainable Controllers: Artifacts (2021). https://doi.org/10.5281/zenodo.4746131
24. Khaled, M., Zamani, M.: pfaces: an acceleration ecosystem for symbolic control. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019, pp. 252–257. ACM, New York (2019)
25. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings, Snowbird, UT, USA, July 14-20, 2011, Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer, Berlin (2011)
26. Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Safe and optimal adaptive cruise control. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Proceedings, Oldenburg, Germany, September 8-9, 2015, Lecture Notes in Computer Science, vol. 9360, pp. 260–277. Springer Berlin (2015)
27. Levenberg, K.: A method for the solution of certain non-linear problems in least squares. Q. Appl. Math. **2**(2), 164–168 (1944)
28. Marquardt, D.W.: An algorithm for least-squares estimation of nonlinear parameters. J. Soc. Ind. Appl. Math. **11**(2), 431–441 (1963)
29. Mitchell, T.M.: Machine Learning. McGraw-hill, New York (1997)
30. Murthy, S.K., Kasif, S., Salzberg, S.: A system for induction of oblique decision trees. J. Artif. Intell. Res. **2**, 1–32 (1994)
31. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
32. Pradhan, S.S., Ward, W.H., Hacioglu, K., Martin, J.H., Jurafsky, D.: Shallow semantic parsing using support vector machines. In: Hirschberg, J., Dumais, S.T., Marcu, D., Roukos, S. (eds.) Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2004, Boston, Massachusetts, USA, May 2-7, 2004, pp. 233–240. The Association for Computational Linguistics (2004)
33. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
34. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Mateo (1993)
35. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: Abate, A., Fainekos, G.E. (eds.) Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016, pp. 99–104. ACM, New York (2016)
36. Rungger, M., Weber, A., Reissig, G.: State space grids for low complexity abstractions. In: 54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015, pp. 6139–6146. IEEE Press, New York (2015)
37. St-Aubin, R., Hoey, J., Boutilier, C.: APRICODD: approximate policy construction using decision diagrams. In: Leen, T.K., Dietterich, T.G., Tresp, V. (eds.) Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA, pp. 1089–1095. MIT Press Cambridge (2000)
38. Swikir, A., Zamani, M.: Compositional synthesis of symbolic models for networks of switched systems. IEEE Control Syst. Lett. **3**(4), 1056–1061 (2019)
39. Weinhuber, C.: Learning domain-specific predicates in decision trees for explainable controller representation. Bachelor's thesis, Technical University of Munich (2020)
40. Zapreev, I.S., Verdier, C., Mazo, M. Jr.: Optimal symbolic controllers determinization for BDD storage. In: ADHS, IFAC-PapersOnLine, vol. 51-16, pp. 1–6. Elsevier, Amsterdam (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.