dtControl 2.0: Explainable Strategy Representation via Decision Tree Learning Steered by Experts [⋆]

Pranav Ashok[1], Mathias Jackermeier[1], Jan Křetínský[1], Christoph Weinhuber[1], Maximilian Weininger[1], and Mayank Yadav[2]

[1] Technical University of Munich, Germany
firstname.lastname@tum.de
[2] Department of Computer Science and Engineering, I.I.T. Delhi,
Hauz Khas, New Delhi - 110016, India
cs1180356@iitd.ac.in

Abstract. Recent advances have shown how decision trees are apt data structures for concisely representing strategies (or controllers) satisfying various objectives. Moreover, they also make the strategy more explainable. The recent tool `dtControl` had provided pipelines with tools supporting strategy synthesis for hybrid systems, such as `SCOTS` and `Uppaal Stratego`. We present `dtControl 2.0`, a new version with several fundamentally novel features. Most importantly, the user can now provide domain knowledge to be exploited in the decision tree learning process and can also interactively steer the process based on the dynamically provided information. To this end, we also provide a graphical user interface. It allows for inspection and re-computation of parts of the result, suggesting as well as receiving advice on predicates, and visual simulation of the decision-making process. Besides, we interface model checkers of probabilistic systems, namely `STORM` and `PRISM` and provide dedicated support for categorical enumeration-type state variables. Consequently, the controllers are more explainable and smaller.

Keywords: Strategy representation · Controller representation · Decision Tree · Explainable Learning · Hybrid systems · Probabilistic Model Checking · Markov Decision Process

1 Introduction

A *controller* (also known as strategy, policy or scheduler) of a system assigns to each state of the system a set of actions that should be taken in order to achieve a certain goal. For example, one may want to satisfy a given specification of a robot's behaviour or exhibit a concurrency bug appearing only in some interleaving. It is desirable that the controllers posses several additional properties, besides

[⋆] We thank Tim Quatman for implementing JSON-export of strategies in `STORM` and Pushpak Jagtap for his support with the `SCOTS` models.

achieving the goal, in order to be usable in practice. Firstly, controllers should be *explainable*. Only then can they be understood, trusted and implemented by the engineers certified by the authorities, or used in the debugging process [10]. Secondly, they should be *small* in size and efficient to run. Only then they can be deployed on embedded devices with limited memory of a few kilobytes, while the automatically synthesized ones are orders of magnitude larger [49]. Thirdly, whenever the primary goal, e.g. functional correctness, is accompanied by a secondary criterion, e.g. energy efficiency, they should be *performant* with respect to this criterion.

Automatic controller synthesis is able to provide controllers for a given goal in various domains, such as probabilistic systems [32, 16], hybrid systems [45, 15, 30, 18] or reactive systems [35]. In some cases, even the performance can be reflected [15]. However, despite recent interest in explainability in connection to AI-based controllers [2] and despite typically small memories of embedded devices, automatic techniques for controller synthesis mostly fall short of producing small explainable results. A typical outcome is a controller in the form of a look-up table, listing the actions for each possible state, or a binary decision diagram (BDD) [13] representation thereof. While the latter reduces the size to some extent, none of the two representations is explainable: the former due to its size, the latter due to the bit-level representation with all high-level structure lost. Instead, learning representations in the form of decision trees (DT) [38] has been recently explored to this end [6, 3]. DTs turn out to be usually smaller than BDD but do not drown to the bit level and are generally well known for their interpretability and explainability due to their simple structure. However, despite showing significant potential, the state-of-the-art tool dtControl [4] uses predicates without natural interpretation, and moreover, the best size reductions are achieved using *determinization*, i.e. making the controller less permissive, which negatively affects performance [6].

Example 1 (Motivating example). Consider the cruise control model of [34], where we want to control the speed of our car so that it never crashes into the car in front while, as a secondary performance objective, keeping the distance between the two cars small.

A safe controller for the this model as returned by `Uppaal Stratego`, is a lookup table of size 418 MB with 300,000 lines. The respective BDD has 1,448 nodes with all information bit-blasted. Using adaptations of standard DT-construction algorithms, as implemented in `dtControl`, we can get a DT with 987 nodes, which is still too large to be explained. Using determinization techniques, the controller can be compressed to 3 nodes! However, then the DT allows only to decelerate until the minimum velocity. This is safe, as we cannot crash into the car in front, but it does not even attempt at getting close to the front car, and thus has a very bad performance.

One can find a strategy with optimal performance, retaining the maximal permissiveness, not determinizing at all, which can be represented by a DT with 11 nodes. A picture of this DT as well as reasoning how to derive the predicates from the kinematic equations is in Appendix A.

However, exactly because the predicates are based on the *domain knowledge*, namely the kinematic equations, they take the form of *algebraic predicates* and not simply linear predicates, which are the only ones in dtControl and commonly in the machine-learning literature on DTs. △

This motivating example shows that using domain knowledge and algebraic predicates, available now in dtControl 2.0, one can get smaller representation than when using existing heuristics. Further, it improves the performance of the DT, and it is easily explainable, as it is based on domain knowledge. In fact, the discussed controller is so explainable that it allowed us to find a bug in the original model. In general, using dtControl 2.0 a domain expert can try to compress the controller, thus gain more insight and validate that it is correct. Another example of this has been reported from the use of dtControl in the manufacturing domain [31].

While automatic synthesis of good predicates from the domain knowledge may seem as distant as automatic synthesis of program invariants or automatic theorem provers, we adopt the philosophy of those domains and offer *semi-automatic techniques.*

Additionally, if not performance but only safety of a controller is relevant, we can still benefit from determinization without drawbacks. To this end, we also provide a new determinization procedure that generalizes the extremely successful MaxFreq technique of [4] and is as good or better on all our examples.

To incorporate the changes just discussed, namely algebraic predicates, semi-automatic approach, and better determinization, we have also reworked the tool and its interfaces. To begin with, the software architecture of dtControl 2.0 is now very modular and allows for easy further modifications, as well as adding support for new synthesis tools. In fact, we have already added parsers for the tools STORM [16] and PRISM [32], and thus we support probabilistic models as well. Since these models also contain categorical (or enumeration-type) variables, e.g. protocol states, we have also added support for categorical predicates. Furthermore, we added a graphical user interface that not only is easier to use than the command-line interface, but also allows to inspect the DT, modify and retrain parts of it, and simulate runs of the model under its control, further increasing the possibilities to explain the DT and validate the controller.

Summing up, the main improvements of dtControl 2.0 over the previous version [4] are the following:

- Support of algebraic predicates and categorical predicates
- Semi-automatic interface and GUI with several interactive modes
- New determinization procedure
- Interfaces for model checkers PRISM and Storm and experimental evidence of improvements on probabilistic models compared to BDD

The paper is structured as follows. After recalling necessary background in Section 2, we give an overview of the improvements over the previous version of the tool from the global perspective in Section 3. We detail on the algorithmic contribution in Sections 4 (predicate domains), 5 (predicate selection) and 6 (determinization). Section 7 provides experimental evaluation and Section 8 concludes.

Related work. DTs have been suggested for representing controllers of and counterexamples in probabilistic systems in [10], however, the authors only discuss approximate representations. The ideas have been extended to other setting, such as reactive synthesis [11] and hybrid systems [6]. More general linear predicates have been considered in leaves of the trees in [3]. `dtControl 2.0` contains the DT induction algorithms from [6, 3]. The differences to the previous version of the tool `dtControl` [4] are summarized above and schematically depicted in Figure 2.

Besides, DTs have been used to represent and learn strategies for safety objectives in [40] and to learn program invariants in [20]. Further, DTs were used for representing the strategies during the model checking process, namely in strategy iteration [9] or in simulation-based algorithms [42]. Representing controllers exactly using a structure similar to DT (mistakenly claimed to be an algebraic decision diagram) was first suggested by [21], however, no automatic construction algorithm was provided.

The idea of non-linear predicates has been explored in [28]. In that work, however, it is not based on domain knowledge, but rather on projecting the state-space to higher dimensions.

BDDs [13] have been commonly used to represent strategies in planning [14], symbolic model checking [32] as well as to represent hybrid system controllers [45, 30]. While BDD [13] operate only on Boolean variables, they have the advantage of being diagrams and not trees. Moreover, they correspond to Boolean functions that can be implemented on hardware easily. [17] proposes an automatic compression technique for numerical controllers using BDDs. Similar to our work, [49] considers the problem of obtaining concise BDD representation of controllers and presents a technique to obtain smaller BDDs via determinization. However, BDDs are difficult to explain due to variables being bit-blasted and their size is very sensitive to the chosen variable ordering. An extension of BDDs, algebraic or multi-terminal decision diagrams (ADD/MTBDD) [7, 19], have been used in reinforcement learning for strategy synthesis [26, 47]. ADDs extend BDDs with the possibility to have multiple values in the terminal nodes, but the predicates still work only on boolean variables, retaining the disadvantages of BDDs.

2 Decision tree learning for controller representation

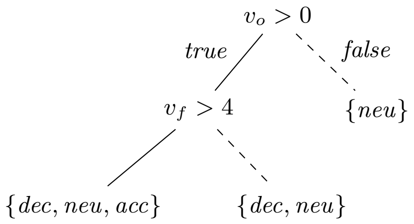
In this section, we briefly describe how controllers can be represented as decision trees as in [4]. We give an exemplified overview of the method, pinpointing the role of our algorithmic contributions.

A (non-deterministic, also called permissive) *controller* is a map $C : S \mapsto 2^A$ from states to non-empty sets of actions. This notion of a controller is fairly general; the only requirement is that it has to be memoryless and non-randomized. These kind of controllers are optimal for many tasks such as expected (discounted) reward, reachability or parity objectives. Moreover, even finite-memory controllers can be written in this form by considering the product of the state space with the finite memory as the domain, for example, like in LTL model checking.

Decision trees (DT), e.g. [38], are trees where every leaf node is labelled with a non-empty set of actions and every inner node is labelled with a *predicate* $\rho : S \mapsto \{\mathit{true}, \mathit{false}\}$.

v_o	v_f	d	actions
0	0	5	$\{\mathit{neu}\}$
2	6	10	$\{\mathit{dec}, \mathit{neu}, \mathit{acc}\}$
2	6	15	$\{\mathit{dec}, \mathit{neu}, \mathit{acc}\}$
4	4	15	$\{\mathit{dec}, \mathit{neu}\}$

(a)


$v_o > 0$
true
false
$v_f > 4$
$\{\mathit{neu}\}$
$\{\mathit{dec}, \mathit{neu}, \mathit{acc}\}$
$\{\mathit{dec}, \mathit{neu}\}$

(b)

Fig. 1: An example controller based on the cruise-control model in the form of a lookup table (left), and the corresponding decision tree (right).

Example 2 (Decision tree representation). As an example, consider the controller given in Figure 1a. It is a subset of the real cruise-control case study from the motivating Example 1. A state is a 3-tuple of the variables v_o, v_f and d, which denote the velocity of our car, the front car and the distance between the cars respectively. In each state, our car may be allowed to perform a subset of the following set of actions: decelerate (*dec*), stay in neutral (*neu*) or accelerate (*acc*). A DT representing this lookup table is depicted in Figure 1b.

Given a state, for example $v_o = v_f = 4, d = 10$, the DT is evaluated as follows: We start at the root and, since it is an inner node, we evaluate its predicate $v_o > 0$. As this is true, we follow the true branch and reach the inner node labelled with the predicate $v_f > 4$. This is false, so we follow the false branch and reach the leaf node labelled $\{\mathit{dec}, \mathit{neu}\}$. Hence, we know that all three possibilities of decelerating, staying neutral and accelerating are allowed by the controller. △

To construct a DT representation of a given controller, the following recursive algorithm may be used. Note that it is heuristic since constructing an optimal binary decision tree is an NP-complete problem [27].

Base case: If all states in the the controller agree on their set of actions B (i.e. for all states s we have $C(s) = B$), return a leaf node with label B.

Recursive case: Otherwise, we split the controller. For this, we select a predicate ρ and construct an inner node with label ρ. Then we partition the controller by evaluating the predicate on the state space, and recursively construct one DT for the sub-controller on states $\{s \in S \mid \rho(s)\}$ where the predicate is true, and one for the sub-controller where it is false. These controllers are the children of the inner node with label ρ and we proceed recursively.

For selecting the predicate, we consider two hyper-parameters: The *domain* of the predicates (see Section 4) and the way to *select* predicates (see Section 5). The selection is typically performed by selecting the predicate with the lowest *impurity*; this is a measure for how homogenous (or "pure") the controller is after the split, in other words the degree to which all the states agree on their actions.

We also consider a third hyper-parameter of the algorithm, namely *determinization* by *safe early stopping* (see Section 6). This modifies the base case as follows: if all states in the controller agree on at least one action a (i.e. for all states s we have $a \in C(s)$), then we return a leaf node with label $\{a\}$. This variant of early stopping ensures that, even though the controller is not represented exactly, still for every state a safe action is allowed.

Hence, if the original controller satisfies some property, e.g. that a safe set of states is never left, the DT construction algorithm ensures that this property is retained. This is because our algorithm represents the strategy exactly (or a safe subset, in case of determinization) and does not generalize as DTs typically do in machine learning. DTs are suitable for both tasks, as both rely on the strength of DTs exploiting underlying structure.

Remark 1. Note that for some types of objectives such as reachability, determinization of permissive strategies might lead to a violation of the original guarantees. For example, consider a strategy that allows both a self-looping and a non-self-looping action at a particular state. If the determinizer decides to restrict to the self-looping action, the reachability property may be violated in the determinized strategy. However, this problem can be addressed when synthesizing the strategy by ensuring that every action makes progress towards the target.

3 Tool

`dtControl 2.0` is an easy-to-use open-source tool for representing memoryless symbolic controllers as more compact and more interpretable DTs, while retaining safety guarantees of the original controllers. Our website `dtcontrol.model.in.tum.de` offers hyperlinks to the easy-to-install `pip` package[3], the documentation and the source code. Additionally, the artifact that has passed the TACAS 21 artifact evaluation is available here [5].

The schema in Figure 2 illustrates the workflow of using `dtControl`, highlighting new features in red. Considering `dtControl` as a black box, it shows that given a controller, it returns a DT representing the controller and also offers the possibility to simulate a run of the system under the control of the DT, visualizing the decisions made. The controller can be input in various formats, including the newly supported strategy representations of the well-known probabilistic

[3] `pip` is a standard package-management system used to install and manage software packages written in Python.

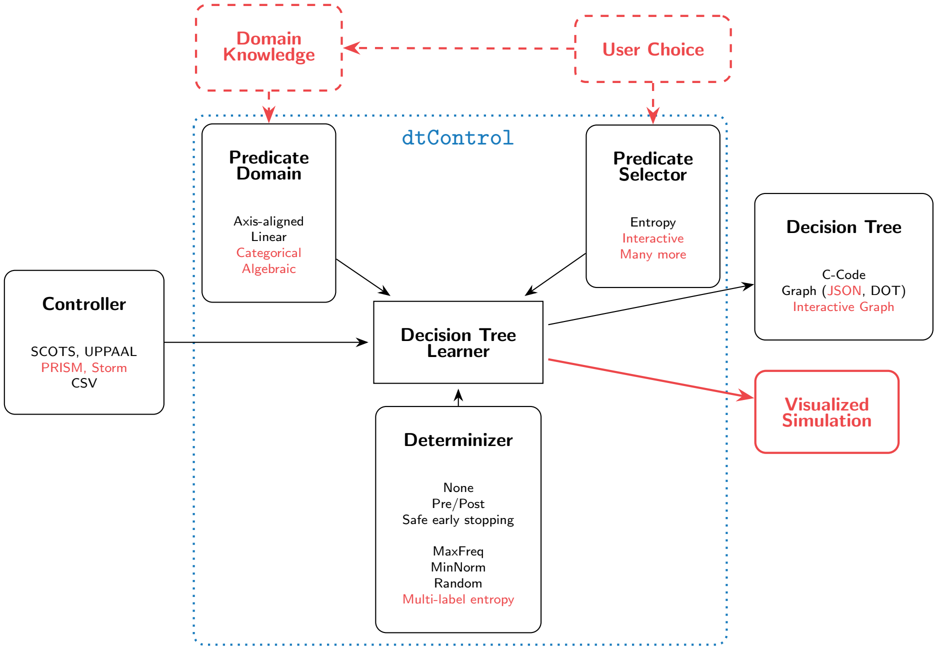
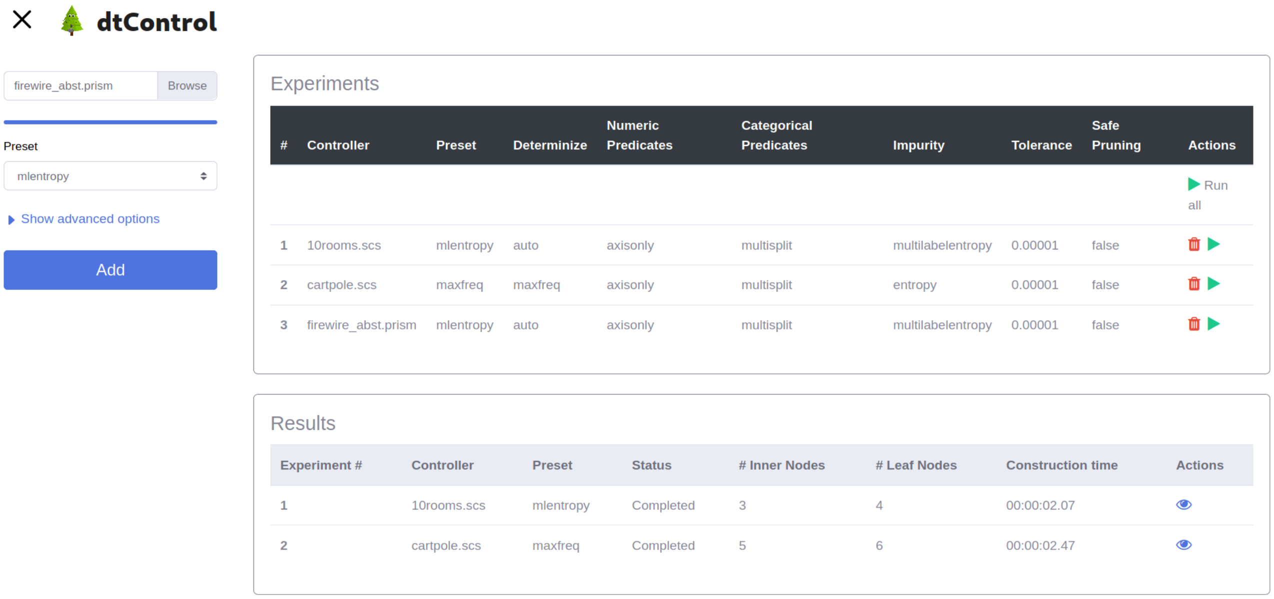


Fig. 2: An overview of the components of `dtControl 2.0`, thereby showing software architecture and workflow. Contributions of this paper are highlighted in red.

model checkers `PRISM` [32] and `STORM` [16]. The DT is output in several machine readable formats, and as C-code that can be directly used for executing the controller on embedded devices. Note that this C-code consists only of nested if-else-statements. The new graphical user interface also offers the possibility to inspect the graph in an interactive web user interface, which even allows to edit the DT. This means that parts of the DT can be retrained with a different set of hyper-parameters and directly replaced. This way, one can for example first train a determinized DT and then retrain important parts of it to be more permissive and hence more performant for a secondary criterion. Figure 3 shows a screenshot of the newly integrated graphical user interface.

Looking at the inner workings of `dtControl`, we see the three important hyper-parameters that were already introduced in Section 2: predicate domain, predicate selector, and determinizer. For each of these, `dtControl` offers various choices, some of which were newly added for version 2.0. Most prominently, the user now has the possibility to directly influence both the predicate domain and the predicate selector, by providing domain knowledge and thus also additional predicates, or by directly using the interactive predicate selection. More details on the predicate domain and how domain knowledge is specified can be found in Section 4. The different ways to select predicates, especially the new interactive mode, are the topic of Section 5. Our new insights into determinization are described in Section 6. To support the user in finding a good set of hyper-



firewire_abst.prism

Preset: mlentropy

Experiments

#	Controller	Preset	Determinize	Numeric Predicates	Categorical Predicates	Impurity	Tolerance	Safe Pruning	Actions
									▶ Run all
1	10rooms.scs	mlentropy	auto	axisonly	multisplit	multilabelentropy	0.00001	false	🗑 ▶
2	cartpole.scs	maxfreq	maxfreq	axisonly	multisplit	entropy	0.00001	false	🗑 ▶
3	firewire_abst.prism	mlentropy	auto	axisonly	multisplit	multilabelentropy	0.00001	false	🗑 ▶

Results

Experiment #	Controller	Preset	Status	# Inner Nodes	# Leaf Nodes	Construction time	Actions
1	10rooms.scs	mlentropy	Completed	3	4	00:00:02.07	👁
2	cartpole.scs	maxfreq	Completed	5	6	00:00:02.47	👁

Fig. 3: Screenshot of the new web-based graphical user interface. It offers a sidebar for easy selection of the controller file and hyper-parameters, an experiments table where benchmarks can be queued, and a results table in which some statistics of the run are provided. Moreover, users can click on the 'eye' icon in the results table to inspect the built decision tree.

parameters, `dtControl` also offers extensive benchmarking functionality, allowing to specify multiple variants and reporting several statistics.

Technical notes `dtControl 2.0` is written in Python 3 following an architecture closely resembling the schema in Figure 2. The modularity, along with our technical documentation, allows users to easily extend the tool. For example, supporting another input format is only a matter of adding a parser.

`dtControl 2.0` works with Python version 3.7.9 or higher. The core of the tool which runs the learning algorithms requires `numpy` [23], `pandas` [36] and `scikit-learn` [41] and optionally the library for the heuristic `OC1` [39]. The algebraic predicates rely on `SymPy` [37] and `SciPy` [48]. The web user interface is powered by Flask [1] and D3.js [8].

4 Predicate domain

The domain of the predicates that we allow in the inner nodes of the DT is of key importance. As we saw in the motivating Example 1, allowing for more expressive predicates can dramatically reduce the size of the DT.

We assume that our state space is structured, i.e. it is a Cartesian product of the domain of the variables ($S = S_1 \times \cdots \times S_n$). We use s_i to refer to the i-th state-variable of a state $s \in S$. In Example 2, the three state-variables are the velocity of our car, the velocity of the front car, and the distance.

We first give an overview of the predicate domains `dtControl 2.0` supports, before discussing the details of the new ones.

Axis-aligned predicates [38] have the form $s_i \leq c$, where c is a rational constant. This is the easiest form of predicates, and they have the advantage that there are only finitely many, as the domain of every state-variable is bounded. However, they are also least expressive.

Linear predicates (also known as oblique [39]) have the form $\sum_i s_i \cdot a_i \leq c$, where a_i are rational coefficients and c is a rational constant. They have the advantage that they are able to combine several state-variables which can lead to saving linearly many splits, cf. [29, Fig. 5.2]. The disadvantage of these predicates is that there are infinitely many choices of coefficients, which is why heuristics were introduced to determine a good set of predicates to try out [39, 4]. However, heuristically determined coefficients and combinations of variables can impede explainability.

Algebraic predicates have the form $f(s) \leq c$, where f is any mathematical function over the state-variables and c is a rational constant. It can use elementary functions such as exponentiation, log, or even trigonometric functions. Example 1 illustrated how this can reduce the size and improve explainability. More discussion of these predicates follows in Section 4.2.

Categorical predicates are special predicates for categorical (enumeration-type) state-variables such as colour or protocol state, and they are discussed in Section 4.1.

4.1 Categorical predicates

Categorical state-variables do not have a numeric domain, but instead are unordered and qualitative. They commonly occur in the models coming from the tools `PRISM` and `STORM`.

Example 3. Let one state-variable be 'colour' with the domain $\{\text{red}, \text{blue}, \text{green}\}$. A simple approach is to assign numbers to every value, e.g. $\text{red} = 0, \text{blue} = 1, \text{green} = 2$, and treat this variable as numeric. However, a resulting predicate such as $\text{colour} \leq 2$ is hardly explainable and additionally depends on the assignment of numbers. For example, it would not be possible to single out $\text{colour} \in \{\text{red}, \text{green}\}$ using a single predicate, given the aforementioned numeric assignment. Using linear predicates, for example adding half of the colour to some other state-variable, is even more confusing and dependent on the numeric assignment. △

Instead of treating the categorical variables using their numeric encodings, `dtControl 2.0` supports specialized algorithms from literature, see e.g. [43, 44]. They work by labelling an inner node with a categorical variable and performing a (possibly non-binary) split according to the value of the categorical variable. The node can have at most one child for every possible value of the categorical variable, but it can also group together similarly behaving values, see Figure 4 for an example. For the grouping, `dtControl 2.0` uses the greedy algorithm from [44, Chapter 7] called attribute-value grouping. It proceeds by first considering to have a branch for every single possible value of the categorical variable, and then merging branches as long as it improves the predicate; see Appendix C for the full pseudocode of the algorithm.

In our experiments we found that the grouping algorithm sometimes did not merge branches in cases where it would actually have made the DT smaller or more explainable. This is because the resulting impurity, the goodness of a predicate, could be marginally worse due to floating-point inaccuracies. Thus, we introduce *tolerance*, a bias parameter in favour of larger value groups. When checking whether to merge branches, we do not require the impurity to improve, but we allow it to become worse up to our tolerance. Setting tolerance to 0 corresponds exactly to the algorithm from [44], while setting tolerance to ∞ results in merging branches until only two remain, thus producing binary predicates.

To allow `dtControl 2.0` to use categorical predicates, the user has to provide a metadata file, which tells the tool which variables are categorical and which are numeric; see Appendix B.1 for an example.

4.2 Algebraic predicates

It is impossible to try out every mathematical expression over the state-variables, and it would also not necessarily result in an explainable DT. Instead, we allow the user to enter *domain knowledge* to suggest templates of predicates that `dtControl 2.0` should try. See Appendix B.2 for a discussion of the format in which domain knowledge can be entered.

Providing the basic equations that govern the model behaviour can already help in finding a good predicate, and is easy to do for a domain expert. Additionally, `dtControl 2.0` offers several possibilities to further exploit the provided domain knowledge:

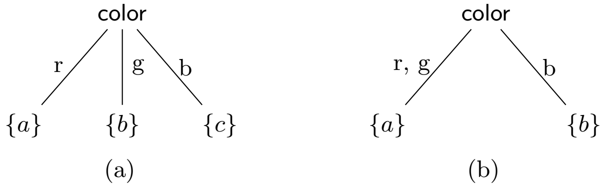


Fig. 4: Two examples of a categorical split. On the left, all possible values of the state-variable colour lead to a different child in a non-binary split. On the right, red and green lead to the same child, which is a result of grouping similar values together.

Firstly, the given predicates need not be exact, but may contain coefficients. These coefficients can be both completely arbitrary or may come from a finite set suggested by the user. For coefficients with finite domain, `dtControl 2.0` tries all possibilities; for arbitrary coefficients, it uses curve fitting to find a good value. For example, the user can specify a predicate such as $d + (v_o - v_f) \cdot c_0 > c_1$ with c_0 being an arbitrary rational number and $c_1 \in \{0, 5, 10\}$.

Secondly, the interactive predicate selection (see Section 5) allows the user to try out various predicates at once and observe their respective impurity in the current node. The user can then choose among them as well as iteratively suggest further predicates, inspired by those where the most promising results were observed.

Thirdly, the decisions given by a DT can be visualized in the simulator, possibly leading to better understanding the controller. Upon gaining any further insight, the user can directly edit any subtree of the result, possibly utilizing the interactive predicate selection again.

5 Predicate selection

The tool offers a range of options to affect the selection of the most appropriate predicate from a given domain.

Impurity measures: As mentioned in Section 2, the predicate selection is typically based on the lowest *impurity* induced. The most commonly used impurity measure (and the only one the first version of `dtControl` supported) is Shannon's entropy [46]. In `dtControl 2.0`, a number of other impurity measures from the literature [43, 12, 25, 39, 3] are available. However, our results indicate that entropy typically performs the best, and therefore it is used as the default option unless the user specifies otherwise. Due to lack of space, we delegate the details and experimental comparison between the impurity measures to Appendix D.

Priorities: `dtControl 2.0` also has the new functionality to assign *priorities* to the predicate generating algorithms. Priorities are rational numbers between 0 and 1. The impurity of every predicate is divided by the priority of the algorithm that generated it. For example, a user can use axis-aligned splits with priority 1 and a linear heuristic with priority 1/2. Then the more complicated linear predicate is only chosen if it is at least twice as good (in terms of impurity) as the easier-to-understand axis-aligned split. A predicate with priority 0 is only considered after all predicates with non-zero priority have failed to split the data. This allows the user to give just a few predicates from domain knowledge, which are then strictly preferred to the automatically generated ones, but which need not suffice to construct a complete DT for the controller.

Interactive predicate selection: `dtControl 2.0` offers the user the possibility to manually select the predicate in every split. This way, the user can prefer predicates that are explainable over those that optimize the impurity.

The screenshot of the interactive interface in Appendix F shows the information that dtControl 2.0 provides. The user is given some statistics and metadata, e.g. minimum, maximum and step size of the state-variables in the current node, a few automatically generated predicates for reference and all predicates generated from domain knowledge. The user can specify new predicates and is immediately informed about their impurity. Upon selecting a predicate, the split is performed and the user continues in the next node.

The user can also first construct a DT using some automatic algorithm and then restart the construction from an arbitrary node using the interactive predicate selection to handcraft an optimized representation, or at any point decide that the rest of the DT should be constructed automatically.

6 New insights about determinization

In our context, *determinization* denotes a procedure that, for some or all states, picks a subset of the allowed actions. Formally, a determinization function δ transforms a controller C into a "more determinized" C', such that for all states $s \in C$ we have $\emptyset \subsetneq C'(s) \subseteq C(s)$. This reduces the permissiveness, but often also reduces the size. Note that, for safety controllers, this always preserves the original guarantees of the controller. For other (non-safety) controllers, see Remark 1.

dtControl 2.0 supports three different general approaches to determinizing a controller: pre-processing, post-processing and safe early stopping. Pre-processing commits to a single determinization before constructing the DT. Post-processing prunes the DT after its construction, e.g. safe pruning in [6]. The basic idea of safe early stopping is already described in Section 2: if all states agree on at least one action, then instead of continuing to split the controller, stop early and return a leaf node with that common action. Alternatively, to preserve more permissiveness, one can return not only a single common action, but all common actions; formally, return the maximum set B such that for all states s in the node $B \subseteq C(s)$.

The results of [4] show that both pre-processing and post-processing are outperformed by an on-the-fly approach based on safe early stopping. This is because pre-processing discards a lot of information that could have been useful in the DT construction and post-processing can only affect the bottom-most nodes of the resulting DT, but usually not those close to the root.

We now give a new view on safe early stopping approaches for determinizing a controller that allows us to generalize the techniques of [4], reducing the size of the resulting DTs even more.

Example 4. Consider the following controller: $C(s_1) = \{a, b, c\}$, $C(s_2) = \{a, b, d\}$, $C(s_3) = \{x, y\}$. All three states map to different sets of actions, and thus an impurity measure like entropy penalizes grouping s_1 and s_2 the same as grouping s_1 and s_3. However, if determinization is allowed, grouping s_1 and s_2 need not be penalized at all, as these states agree on some actions, namely a and b. Grouping

s_1 and s_2 into the same child node thus allows the algorithm to stop early at that point and return a leaf node with $\{a, b\}$, in contrast to grouping s_1 and s_3. △

Knowing that we want to determinize by safe early stopping affects the predicate selection process. Intuitively, sets of states are more homogeneous the more actions they share. We want to take this into account when calculating the impurity of predicates. One way to do this would be to calculate the impurity of all possible determinization functions and pick the best one. This, however, is infeasible, hence we propose the heuristic of *multi-label impurity measures*. These impurity measures do not only consider the full set of allowed actions in their calculation, but instead they depend on the individual actions occurring in the set. This allows the DT construction to pick better predicates, namely those whose resulting children are more likely to be determinizable. In Appendix E we formally derive the multi-label variants of entropy and Gini-index.

To conclude this section, we point out the key difference between the new approach of multi-label impurity measures and the previous idea that was introduced in [4]. The approach from [4] does not evaluate the impurity of all possible determinization functions, but rather picks a smart one – that of maximum frequency (MaxFreq) – and evaluates according to that. MaxFreq determinizes in the following way: for every state, it selects from the allowed actions that action occurring most frequently throughout the whole controller. This way, many states share common actions. This is already better than pre-processing, as it does not determinize the controller a priori, but rather considers a different determinization function at every node. However, in every node we calculate the impurity for several different predicates, and the optimal choice of determinization function depends on the predicate. Thus, choosing a single determinization function for a whole node is still too coarse, as it is fixed independent of the considered predicate. We illustrate the arising problem in the following Example 5.

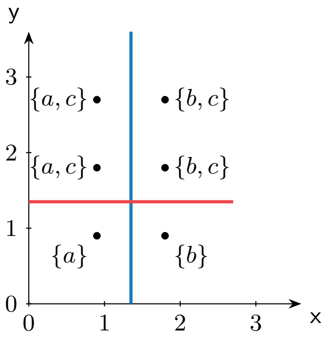


Fig. 5: A simple example of a dataset that is split suboptimally by the MaxFreq approach from [4], but optimally by the new multi-label entropy approach.

Example 5. Figure 5 shows a simple controller with a two-dimensional state space. Every point is labeled with its set of allowed actions.

As c is the most frequent action, MaxFreq determinizes the states $(1,2)$, $(1,3)$, $(2,2)$ and $(2,3)$ to action c. Hence the red split (predicate $y < 1.5$) is considered optimal, as it groups together all four states that map to c. The blue split (predicate $x < 1.5$) is considered suboptimal, as then the data still looks very heterogeneous. So, using MaxFreq, we need two splits for this controller; one to split of all the c's and one to split the two remaining states.

However, it is better to first choose a predicate and then determine a fitting determinization function. When calculating the impurity of the blue split, we can choose to determinize all states with $x = 1$ to $\{a\}$ and all states with $x = 2$ to $\{b\}$. Thus, in both resulting sub-controllers the impurity is 0 as all states agree on at least one action. This way, one split suffices to get a complete DT. Multi-label impurity measures notice when labels are shared between many (or all) states in a sub-controller, and thus they allow to prefer the optimal blue split. $\triangle$

7 Experiments

Experimental setup We compare three approaches: BDDs, the first version of `dtControl` from [4] and `dtControl 2.0`. For BDDs[4] the variable ordering is important, so we report the smallest of 20 BDDs that we constructed by starting with a random initial variable ordering and reordering until convergence. To determinize BDDs, we used the pre-processing approach, 10 times with the minimum norm and 10 times with MaxFreq. For the previous version of `dtControl`, we picked the smaller of either a DT with only axis-aligned predicates or a DT with linear predicates using the logistic regression heuristic that was typically best in [4]. Determinization uses safe early stopping with the MaxFreq approach. For `dtControl 2.0`, we use the multi-label entropy based determinization and utilize the categorical predicates for the case studies from probabilistic model checking. We ran all experiments on a server with operating system Ubuntu 19.10, a 2.2GHz Intel(R) Xeon(R) CPU E5-2630 v4 and 250 GB RAM.

Comparing determinization techniques on cyber-physical systems Table 1 shows the sizes of determinized BDDs and DTs on the permissive controllers of the tools `SCOTS` and `Uppaal Stratego` that were already used in [4]. We see that the new determinization approach is strictly better than the previous one, with only two DTs being of equal size, as the result of the previous method was already optimal. With the exception of the case studies helicopter and truck_trailer where BDDs are comparable or slightly better, both approaches using DTs are orders of magnitude smaller than BDDs or an explicit representation of the state-action mapping.

[4] Our implementation of BDDs is based on the `dd` python library `https://github.com/tulip-control/dd`.

Table 1: Controller sizes of different determinized representations of the controllers from SCOTS and Uppaal Stratego. "States" is the number of states in the controller, "BDD" the number of nodes of the smallest BDD from 20 tries, dtControl 1.0 [4] the smallest DT the previous version of dtControl could generate and dtControl 2.0 the smallest DT the new version can construct. "TO" denotes a failure to produce a result in 3 hours. The smallest numbers in each row are highlighted.

Case study	States	BDD	dtControl 1.0	dtControl 2.0
cartpole	271	127	11	**7**
10rooms	26,244	128	**7**	**7**
helicopter	280,539	870	221	**123**
cruise-latest	295,615	1,448	**3**	**3**
dcdc	593,089	381	9	**5**
truck_trailer	1,386,211	**18,186**	42,561	31,499
traffic_30m	16,639,662	TO	127	**97**

Case studies from probabilistic model checking For Table 2, we used case studies from the quantitative verification benchmark set [24], which includes models from the PRISM benchmark suite [33]. Note that these case studies contain unordered enumeration-type state-variables for which we utilize the new categorical predicates. To get the controllers, we solved the case study with STORM and exported the resulting controller. This export already eliminates unreachable states. The previous version of dtControl was not able to handle these case studies, so we only compare dtControl 2.0 to BDDs.

Table 2 shows that also for case studies from probabilistic model checking, DTs are a good way of representing controllers. The DT is the smallest representation on 13 out of 19 case studies, often reducing the size by an order of magnitude compared to BDDs or the explicit representation. On 3 case studies, BDDs are smallest, and on 2 case studies, both the DT and the BDD fail to reduce the size compared to the explicit representation. This happens if there are many different actions and thus states cannot be grouped together. A worst case example of this is a model where every state has a different action; then, a DT would have as many leaf nodes as there are states, and hence twice as many nodes in total.

Remark 2. Note that the controllers exported by STORM are deterministic, so no determinization approach can be utilized in the DT construction. We conjecture that if a permissive strategy was exported, dtControl 2.0 would benefit from the additional information and be able to reduce the controller size further as for the cyber-physical systems.

Table 2: Controller sizes of different representations of controllers from the quantitative verification benchmark set [24], i.e. from the tools `STORM` and `PRISM`. "States" is the number of states in the controller, "BDD" the number of nodes of the smallest BDD of 20 tries and `dtControl 2.0` the smallest DT we could construct. The smallest numbers in each row are highlighted.

Case study	States	BDD	`dtControl 2.0`
triangle-tireworld.9	48	51	**23**
pacman.5	232	330	**33**
rectangle-tireworld.11	**241**	498	373
philosophers-mdp.3	344	295	**181**
firewire_abst.3.rounds	610	61	**25**
rabin.3	704	303	**27**
ij.10	1,013	**436**	753
zeroconf.1000.4.true.correct_max	1,068	386	**63**
blocksworld.5	1,124	3,985	**855**
cdrive.10	**1,921**	5,134	2,401
consensus.2.disagree	2,064	138	**67**
beb.3-4.LineSeized	4,173	913	**59**
csma.2-4.some_before	7,472	1,059	**103**
eajs.2.100.5.ExpUtil	12,627	1,315	**153**
elevators.a-11-9	14,742	**6,750**	9,883
exploding-blocksworld.5	76,741	34,447	**1,777**
echoring.MaxOffline1	104,892	43,165	**1,543**
wlan_dl.0.80.deadline	189,641	5,738	**2,563**
pnueli-zuck.5	303,427	**50,128**	150,341

8 Conclusion

We have presented a radically new version of the tool `dtControl` for representing controllers by decision trees. The tool now features a graphical user interface, allowing both experts and non-experts to conveniently interact with the decision tree learning process as well as the resulting tree. There is now a range of possibilities on how the user can provide additional information. The algebraic predicates provide the means to capture the (often non-linear) relationships from the domain knowledge. The categorical predicates together with the interface to probabilistic model checkers allow for efficient representation of strategies for Markov decision processes, too. Finally, the more efficient determinization yields very small (possibly non-performant) controllers, which are particularly useful for debugging the model.

We see at least two major promising future directions. Firstly, synthesis of predicates could be made more automatic using mathematical reasoning on the domain knowledge, such as substituting expressions with a certain unit of measurement into other domain equations in the places with the same unit of measurement, e.g. to plug difference of two velocities into an equation for velocity. Secondly, one could transform the controllers into possibly entirely different controllers (not just less permissive) so that they still preserve optimality (or yield ε-optimality) but are smaller or simpler. Here, a closer interaction loop with the model checkers might lead to efficient heuristics.

References

1. Flask web development: developing web applications with python. https://pypi.org/project/Flask/, accessed: 14.10.2020
2. Adadi, A., Berrada, M.: Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). IEEE Access **6**, 52138–52160 (2018)
3. Ashok, P., Brázdil, T., Chatterjee, K., Křetínský, J., Lampert, C.H., Toman, V.: Strategy representation by decision trees with linear classifiers. In: QEST. Lecture Notes in Computer Science, vol. 11785, pp. 109–128. Springer (2019)
4. Ashok, P., Jackermeier, M., Jagtap, P., Křetínský, J., Weininger, M., Zamani, M.: dtcontrol: decision tree learning algorithms for controller representation. In: HSCC. pp. 17:1–17:7. ACM (2020)
5. Ashok, P., Jackermeier, M., Křetínský, J., Weinhuber, C., Weininger, M., Yadav, M.: dtControl 2.0: Explainable Strategy Representation via Decision Tree Learning Steered by Experts (TACAS 21 Artifact) (Jan 2021). https://doi.org/10.5281/zenodo.4437169, https://doi.org/10.5281/zenodo.4437169
6. Ashok, P., Křetínský, J., Larsen, K.G., Coënt, A.L., Taankvist, J.H., Weininger, M.: SOS: safe, optimal and small strategies for hybrid Markov decision processes. In: QEST. Lecture Notes in Computer Science, vol. 11785, pp. 147–164. Springer (2019)
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. Formal Methods Syst. Des. **10**(2/3), 171–206 (1997). https://doi.org/10.1023/A:1008699807402, https://doi.org/10.1023/A:1008699807402
8. Bostock, M., Ogievetsky, V., Heer, J.: D^3 data-driven documents. IEEE transactions on visualization and computer graphics **17**(12), 2301–2309 (2011)
9. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: IJCAI. pp. 1104–1113. Morgan Kaufmann (1995)
10. Brázdil, T., Chatterjee, K., Chmelik, M., Fellner, A., Křetínský, J.: Counterexample explanation by learning small strategies in Markov decision processes. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 158–177. Springer (2015)
11. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 385–407. Springer (2018)
12. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth (1984)
13. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819, https://doi.org/10.1109/TC.1986.1676819
14. Cimatti, A., Roveri, M., Traverso, P.: Automatic obdd-based generation of universal plans in non-deterministic domains. In: AAAI/IAAI. pp. 875–881. AAAI Press / The MIT Press (1998)
15. David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Uppaal stratego. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer (2015)
16. Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: CAV (2). Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer (2017)

17. Della Penna, G., Intrigila, B., Lauri, N., Magazzeni, D.: Fast and Compact Encoding of Numerical Controllers Using OBDDs, pp. 75–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
18. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 379–395. Springer (2011)
19. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods Syst. Des. **10**(2/3), 149–169 (1997). https://doi.org/10.1023/A:1008647823331, `https://doi.org/10.1023/A:1008647823331`
20. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL. pp. 499–512. ACM (2016)
21. Girard, A.: Low-complexity quantized switching controllers using approximate bisimulation. CoRR **abs/1209.4576** (2012)
22. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), `http://www.deeplearningbook.org`
23. Harris, C.R., Millman, K.J., van der Walt, S., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with numpy. CoRR **abs/2006.10256** (2020)
24. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS (1). Lecture Notes in Computer Science, vol. 11427, pp. 344–350. Springer (2019)
25. Heath, D.G., Kasif, S., Salzberg, S.: Induction of oblique decision trees. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993. pp. 1002–1007 (1993)
26. Hoey, J., St-Aubin, R., Hu, A.J., Boutilier, C.: SPUDD: stochastic planning using decision diagrams. In: UAI. pp. 279–288. Morgan Kaufmann (1999)
27. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is np-complete. Inf. Process. Lett. **5**(1), 15–17 (1976). https://doi.org/10.1016/0020-0190(76)90095-8, `https://doi.org/10.1016/0020-0190(76)90095-8`
28. Ittner, A., Schlosser, M.: Non-linear decision trees - NDT. In: ICML. pp. 252–257. Morgan Kaufmann (1996)
29. Jackermeier, M.: dtControl: Decision Tree Learning for Explainable Controller Representation. Bachelor's thesis, Technische Universität München (2020)
30. Jr., M.M., Davitian, A., Tabuada, P.: PESSOA: A tool for embedded controller synthesis. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 566–569. Springer (2010)
31. Kiesbye, J.: Private Communication (2020)
32. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)
33. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST. pp. 203–204. IEEE Computer Society (2012)
34. 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,

Oldenburg, Germany, September 8-9, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9360, pp. 260–277. Springer (2015). https://doi.org/10.1007/978-3-319-23506-6_17, https://doi.org/10.1007/978-3-319-23506-6_17

35. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020). https://doi.org/10.1007/s00236-019-00349-3, https://doi.org/10.1007/s00236-019-00349-3
36. Wes McKinney: Data Structures for Statistical Computing in Python. In: Stéfan van der Walt, Jarrod Millman (eds.) Proceedings of the 9th Python in Science Conference. pp. 56 – 61 (2010). https://doi.org/10.25080/Majora-92bf1922-00a
37. Meurer, A., Smith, C.P., Paprocki, M., Certík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., Rathnayake, T., Vig, S., Granger, B.E., Muller, R.P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M.J., Terrel, A.R., Roucka, S., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.M.: Sympy: symbolic computing in python. PeerJ Comput. Sci. **3**, e103 (2017)
38. Mitchell, T.M.: Machine learning. McGraw Hill series in computer science, McGraw-Hill (1997)
39. Murthy, S.K., Kasif, S., Salzberg, S., Beigel, R.: OC1: A randomized induction of oblique decision trees. In: AAAI. pp. 322–327. AAAI Press / The MIT Press (1993)
40. Neider, D., Markgraf, O.: Learning-based synthesis of safety controllers. In: FMCAD. pp. 120–128. IEEE (2019)
41. 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. Journal of Machine Learning Research **12**, 2825–2830 (2011)
42. Pyeatt, L.D., Howe, A.E., et al.: Decision tree function approximation in reinforcement learning. In: Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models. vol. 2, pp. 70–77. Cuba (2001)
43. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
44. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
45. Rungger, M., Zamani, M.: SCOTS: A tool for the synthesis of symbolic controllers. In: HSCC. pp. 99–104. ACM (2016)
46. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**(4), 623–656 (1948)
47. St-Aubin, R., Hoey, J., Boutilier, C.: APRICODD: approximate policy construction using decision diagrams. In: NIPS. pp. 1089–1095. MIT Press (2000)
48. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, I., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy: Scipy 1.0-fundamental algorithms for scientific computing in python. CoRR **abs/1907.10121** (2019)
49. Zapreev, I.S., Verdier, C., Jr., M.M.: Optimal symbolic controllers determinization for BDD storage. In: ADHS. IFAC-PapersOnLine, vol. 51, pp. 1–6. Elsevier (2018)

A Deriving algebraic predicates from domain knowledge for the cruise-control model

The cruise-control model is governed by the kinematic equations, i.e. the new distance between the cars is computed as follows:

$$d_{new} = d + (v_f - v_o) \cdot t + (a_f - a_o) \cdot t^2,$$

where t is a time span in seconds, d, v_f and v_o are distance between the cars, velocity of the front car and velocity of our car as before, and a_f and a_o are the acceleration that the cars use during the whole time span. The model restricts these accelerations to be from the set $\{-2, 0, 2\}$, which corresponds to the actions deceleration, neutral or acceleration.

We want that the distance between the cars always is greater than some threshold. The worst case behaviour of the front car is to always decelerate, corresponding to emergency braking. We can thus use $a_f = -2$ and the only unknown in the equations is our acceleration a_o. Now we can calculate the worst-case distance between the cars, assuming we accelerate for one time step (i.e. $a_o = 2$) and then brake ($a_o = -2$) until both cars are at minimum velocity. If that distance is greater than our threshold, we know that it is safe to accelerate in the next time step. Since the actions are ordered, in any state in which it is safe to accelerate, we can also stay neutral or decelerate (excluding the corner case of minimum velocity). Similarly, we can split of the states that allow to stay neutral or decelerate, but not to accelerate. All remaining states only allow deceleration.

See Figure 6 for the DT using algebraic predicates to represent the whole permissive controller with 11 nodes. t_f is the time until the front car reaches minimum velocity. The root node checks whether acceleration is safe in the next time step, the left child of the root checks whether staying neutral is safe. -6 and 14 are minimum respectively maximum velocity, and thus have to be treated separately after these two most important splits.

B Domain knowledge and metadata formats

B.1 Metadata file

An example of a metadata file that is used to inform `dtControl 2.0`, which variables are numeric and which are categorical. Giving the column names as well improves the graphical output, as the variables are not indexed (e.g. `x_0`), but rather named (e.g. `Host_1_ev`).

```
{
    "x_column_types":
    {
        "numeric": [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
        "categorical": [ 10, 11, 12, 13, 14, 15, 16 ]
```

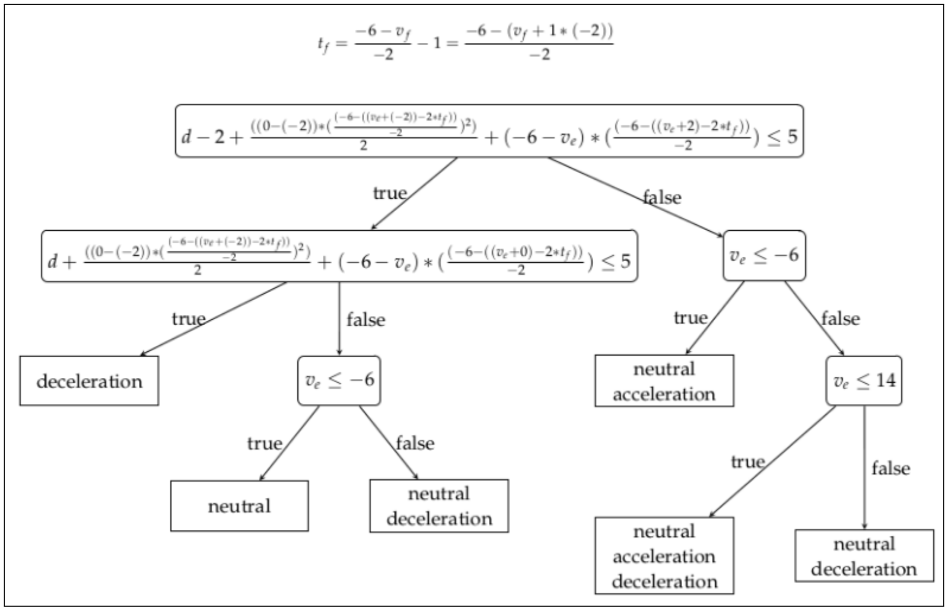

$t_f = \frac{-6 - v_f}{-2} - 1 = \frac{-6 - (v_f + 1 * (-2))}{-2}$
$d - 2 + \frac{((0-(-2))*(\frac{(-6-((v_e+(-2))-2*t_f))}{-2})^2)}{2} + (-6 - v_e) * (\frac{(-6-((v_e+2)-2*t_f))}{-2}) \leq 5$
true
false
$d + \frac{((0-(-2))*(\frac{(-6-((v_e+(-2))-2*t_f))}{-2})^2)}{2} + (-6 - v_e) * (\frac{(-6-((v_e+0)-2*t_f))}{-2}) \leq 5$
$v_e \leq -6$
true
false
true
false
deceleration
$v_e \leq -6$
neutral acceleration
$v_e \leq 14$
true
false
true
false
neutral
neutral deceleration
neutral acceleration deceleration
neutral deceleration

Fig. 6: DT using algebraic predicates to represent the whole permissive controller for the cruise-control model with 11 nodes.

```
    },
    "x_column_names": [
        "Host_1_ev",
        "Host_1_na",
        "Host_1_wt",
        "Host_2_ev",
        "Host_2_na",
        "Host_2_wt",
        "Host_ev",
        "Host_na",
        "Host_wt",
        "N",
        "_loc_Clock",
        "_loc_Host",
        "_loc_Host_1",
        "_loc_Host_2",
        "cr",
        "gave_up",
        "line_seized"
    ]
}
```

B.2 Domain knowledge format

All domain knowledge must take the form of predicates. Their structure can be summarized as follows:

$$\textit{term} \sim \textit{term}; \textit{def}$$

term is an arbitrary arithmetic term, using any elementary function that can be parsed by SymPy, any state-variable and coefficients that are defined in *def*; the exact format of the coefficient definition *def* is provided in our documentation, but most importantly, it allows you to specify finite sets of values or a completely arbitrary coefficient. $\sim$ is a standard comparator from the set {`<=`,`>=`,`<`,`>`,`=`}. An example of a predicate is

```
c_1 * x_1 - c_2 + 2 * x_2 <= c_3; c_1 in {1,2,3}; c_2 in {4,8}
```

Here, c_1 and c_2 are from a finite set and c_3 is completely arbitrary.

C Algorithm for categorical predicates

Let v be a categorical state-variable with possible values $v_1, \dots, v_m$. It is not feasible to simply try all different possible attribute value groupings for every categorical state-variable v, since the number of such groupings is exponential in the number of possible values of v [44, Ch. 7]. We instead use a greedy algorithm based on iterative merging of value groups suggested by [44, Ch. 7], which proceeds as follows:

1. Initially, create a single group for every possible value, i.e. set the inital grouping to $G = (\{v_1\}, \dots, \{v_m\})$. Let G_i denote the i-th set in the grouping.
2. If only two value groups remain, return those as the optimal grouping.
3. For every pair of value groups (G_i, G_j), compute the impurity of the new value grouping in which G_i and G_j are merged.
4. If the impurity has not decreased in any of the new groupings, return the original grouping. Otherwise, proceed to Step 2 with the best new grouping.

Our modification of tolerance that is discussed in Section 4.1 only modifies the stopping condition in Step 4: it replaces “has not decreased in any of the new groupings” with “has increased by more than the tolerance in all new groupings”.

D Other impurities

In the following, we give a description of all impurity measures that are supported by `dtControl 2.0`, and then evaluate them on some models from probabilistic model checking as well as some cyber-physical systems. The text is almost verbatim from the Bachelor’s thesis [29].

D.1 Description

Entropy A particularly well-known impurity measure is based on the concept of *entropy* from information theory, as introduced in the seminal work of Shannon [46]. Information theory is concerned with quantifying the amount of information the occurrence of a random event yields. If an event x occurs with probability p_x, its information content is defined to be

$$I(x) = -\log_2(p_x) \text{ bits.}$$

This definition has several desirable properties [22, Ch. 3]: first, we see that the information content of an event is inversely proportional to its probability. For instance, an event with probability 1 always occurs, and thus does not convey any information. Second, if two events are independent, the information content of both events occurring is the sum of the individual information contents, since

$$I(x, y) = -\log_2(p_{x,y}) = -\log_2(p_x p_y) = -\log_2(p_x) - \log_2(p_y).$$

In the context of impurity measures, the events that we are interested in are that a randomly picked data point from a sub-controller $C \subseteq S \times 2^A$ allows the action set $B \in 2^A$. Let n_B be the frequency of the set B in C, i.e. $n_B = |\{s \in S : C(x) = B\}|$. Then, such an event occurs with probability

$$\frac{n_B}{|C|}$$

and has an information content of

$$-\log_2\left(\frac{n_B}{|C|}\right) \text{ bits.}$$

The crucial insight that allows for the development of an impurity measure is the following: if the expected amount of information content of these events is low, we already have a lot of knowledge about the labels in the sub-controller. Thus, classifying this dataset probably requires less effort than classifying a dataset where the expected amount of information content from these events is high. The expected amount of information content is also known as the *entropy* of the controller C and is defined as

$$H(C) = -\sum_{B \in 2^A} \frac{n_B}{|C|} \log_2\left(\frac{n_B}{|C|}\right).$$

To illustrate two extreme cases, consider a dataset where every data point has a different label. This is extremely hard to classify since every data point has to be separated from all other points, and, correspondingly, the entropy of such a dataset is maximal. In contrast, the entropy of a pure dataset is always 0.

We now have a way to measure the difficulty of classifying a sub-controller. The entropy impurity measure then simply averages the difficulty of classifying the partitions $C_1, \ldots, C_m$ created by a predicate ρ. It is thus defined as follows:

$$\text{ent}(\rho, C) = \sum_{i \in [m]} \frac{|C_i|}{|C|} H(C_i).$$

Instead of entropy, a similar measure called *information gain* is sometimes used. The only real difference between the two is that information gain is a measure of *goodness*, i.e. we want to maximize the information gain in DT learning. This is however equivalent to minimizing entropy [43]. Entropy and information gain are some of the most common ways to determine the quality of predicates and have received a great deal of attention in the DT literature [12, 43, 44].

Entropy ratio An issue with entropy that is sometimes encountered with categorical features is that it favors multi-comparison predicates with a large number of branches [43]. Quinlan [43] thus introduces a normalization of the information gain criterion called the *gain ratio*. Since this is again a goodness measure, we modified it into an impurity measure named the *entropy ratio*.

We first introduce the quantity of the intrinsic information content of a split

$$\text{split info}(\rho, C) = -\sum_{i \in [m]} \frac{|C_i|}{|C|} \log_2 \left(\frac{|C_i|}{|C|} \right).$$

This measures the expected information content of the event that a randomly selected data point in C will be assigned the i^{th} branch, which corresponds to the information generated by the partitioning itself. Naturally, the more branches the predicate ρ creates, the higher this information content will be. In contrast, the entropy measures the amount of information *relevant to classification* from the same partitioning [44, Ch. 2].

The entropy ratio then simply normalizes the entropy with the intrinsic information content of a split, i.e.

$$\text{ent ratio}(\rho, C) = \frac{\text{ent}(\rho, C)}{\text{split info}(\rho, C)}.$$

It has to be noted that one of the primary reasons for using the entropy ratio instead of just the entropy is to prevent overfitting [44, Ch. 2]. In our setting, overfitting is desirable and there is hence less justification for the entropy ratio.

Gini index Another common impurity measure used in e.g. the `CART` system is the *Gini index* [12, Ch. 4]. It measures the probability of a data point being misclassified if we were to assign labels randomly based on the label distribution in the sub-controller. This probability is given by

$$G(C) = \sum_{\substack{y, z \in 2^A \\ y \neq z}} \frac{n_y}{|C|} \frac{n_z}{|C|}$$

and can equally be written as

$$G(C) = \left(\sum_{y \in 2^A} \frac{n_y}{|C|} \right)^2 - \sum_{y \in 2^A} \left(\frac{n_y}{|C|} \right)^2 = 1 - \sum_{y \in 2^A} \left(\frac{n_y}{|C|} \right)^2.$$

Similarly to entropy, the Gini index is then a weighted average of these values:

$$\mathrm{Gini}(\rho, C) = \sum_{i \in [m]} \frac{|C_i|}{|C|} G(C_i).$$

Twoing rule The `CART` system also defines another measure known as the *twoing rule*, which is a goodness measure only defined for binary predicates. It is based on transforming the problem of computing the impurity of a multi-class dataset into the task of computing the impurity of a *two-class* dataset. This transformed problem is then solved with the Gini index defined above.

Let l (r) be the number of examples on the left (right) side of the split and $n_{l,y}$ ($n_{r,y}$) be the number of examples with label y on the left (right) side of the split. Then, the twoing rule is defined as follows [12, Ch. 4]:

$$\mathrm{twoing}(\rho, C) = \frac{l}{|C|} \frac{r}{|C|} \left(\sum_{y \in 2^A} \left| \frac{n_{l,y}}{l} - \frac{n_{r,y}}{r} \right| \right)^2 .$$

Following [39], the impurity measure we minimize is then simply the reciprocal of the twoing rule.

Sum minority Probably the simplest way to calculate impurity is to count the number of misclassified instances if we were to assign the most frequent label in a partition to all of its data points. This impurity measure is called *sum minority* and due to Heath et al. [25].

Formally, for every partition C_i, let $|C_i|$ be the number of examples in that partition and $\gamma(C_i)$ be the label occurring most frequently in C_i. Then, define the *minority* μ_i as

$$\mu_i = |\{(x, y) \in C_i : y \neq \gamma(C_i)\}|.$$

The sum minority impurity measure is then simply the sum of these minorities:

$$\text{sum minority}(\rho, C) = \sum_{i \in [m]} \mu_i.$$

Max minority A very similar impurity measure is *max minority* [25], defined as

$$\text{max minority}(\rho, C) = \max_{i \in [m]} \mu_i.$$

It counts the number of misclassified instances in the "worst" partition with the maximum number of misclassifications.

Max minority has the theoretical advantage that it produces trees of depth at most $\log_2(|C|)$ [39]. However, note that it is the *depth* that is logarithmic in this expression – the number of *nodes* is linear in $|C|$. Thus, this theoretical insight is not very useful in practice.

Area under the receiver-operator curve The final impurity measure we discuss has been developed specifically for DTs with linear classifiers in the context of controller representation [3]. The underlying idea is simple: we want to exploit the knowledge that the controller will be split with a hyperplane, obtained from a linear classifier or a different heuristic. The impurity measure tries to estimate how well separable the controller is by a hyperplane after having been split with the predicate ρ.

For now, let us consider the case of only two actions in a controller. In order to estimate how well separable a sub-controller is by a hyperplane, we can again train a linear classifier on this sub-controller and report a metric that measures some quality of this classifier. The simplest such quality would probably be the accuracy, i.e. the fraction of data points classified correctly. However, this has the disadvantage that even trivial classifiers that assign the same label to every data point can achieve high accuracy in the case of an imbalanced label distribution. Ashok et al. [3] instead suggest the usage of the *area under the receiver-operator curve* (AUC), a well-known metric in statistics and machine learning that does not suffer from this drawback.

We thus proceed as follows to estimate the quality of a predicate ρ:

1. Train a linear classifier for every sub-dataset C_i.
2. Return the sum of the obtained AUC scores of the classifiers.

Note that this is again a goodness measure, which we can transform into an impurity measure by considering the reciprocal.

Ashok et al. [3] use a different data representation. Instead of mapping states to sets of actions, they map state-action pairs to $\{0, 1\}$, indicating whether an action is safe in a state or not. Hence they always deal with a binary classification problem. In order to utilize their idea with our different data representation, we again make use of the technique based on one-versus-the-rest classification, cf. [4, Sec. 4.1.3]. We train one linear classifier for every possible label, which tries to separate this label from the rest, and report a weighted average of the obtained AUC scores.

Note that this impurity measure has the practical disadvantage that we need to train several linear classifiers for every considered predicate, which can be very inefficient.

D.2 Evaluation

Models from probabilistic model checking Table 3 gives the results of attribute value grouping in combination with different impurity measures on some case studies from probabilistic model checking. It clearly shows that probabilistic impurity measures such as entropy and gini index perform far better than non-probabilistic impurity measures like sum- and max-minority. Furthermore, we see that the entropy ratio is strictly worse than the standard entropy. As discussed in Section D.1, this is expected, since the main reason for choosing the entropy ratio over just the entropy is normally to prevent overfitting. On the other hand,

Table 3: Number of nodes obtained with different impurity measures and attribute value grouping on models from probabilistic model checking.

Case study	Entropy	Entropy ratio	Gini index	Sum minority	Max minority
csma2_4	48	89	**43**	579	1,412
firewire_abst	**9**	18	12	110	36
firewire_impl	**77**	124	**77**	442	126
leader4	154	207	**150**	547	1,767
mer30	**158**	213	165	7,557	6,797
wlan2	222	416	**220**	495	3,723
zeroconf	**367**	456	374	7,243	14,624

gini index and entropy perform similarly well and are both viable choices. Note that the twoing rule is not applicable in this scenario, as it is limited to binary predicates.

We also experimented with our modified version of the AUC impurity measure, introduced in Section D.1. As previously noted, one of its drawbacks is that it is very expensive to compute. Indeed, it took more than 13 minutes to build a tree with AUC for the small `firewire_abst` controller – in comparison to roughly 0.2 seconds with the standard impurity measures. On the one hand, this is due to the fact that AUC requires the training of several linear classifiers for every considered predicate, which simply is computationally demanding. On the other hand, we notice that it also produces unnecessarily large trees, which in turn again increases the computational cost: we obtain 716 nodes in the tree for `firewire_abst`.

Why does AUC not work in our scenario? There are two factors that come into play: first, the impurity measure tries to estimate the linear separability of the sub-datasets resulting from a predicate. However, since many features in the examples are categorical and we only use oblique splits with numeric features because of explainability reasons, the measure itself is not that meaningful in our context. Second, we conjecture that our approach based on one-versus-the-rest classification, which we adopted due to our data representation, just is not well-suited as an impurity measure.

Cyber-physical systems For cyber-phyiscal systems (CPS), our experiments suggest that entropy is one of the strongest impurity measures overall in the case of controllers obtained from CPS synthesis. To illustrate, we list the number of nodes when learning DTs with axis-aligned predicates, no determinization, and varying impurity measures in Table 4.

The table clearly shows that sum- and max-minority perform far worse than the probabilistic impurity measures on many datasets and are overall not competitive. Entropy, gini index, and twoing rule usually perform similarly well, although entropy is slightly better in a number of cases. As expected, the entropy ratio is overall somewhat worse. We again encountered the same performance issues with AUC as before, and the numbers we could compute were not promising, which is why we did not include this impurity measure in the table.

Table 4: Effects of different impurity measures with axis-aligned predicates on decision tree sizes for synthesis of CPS. "∞" indicates failure to produce a result within three hours.

Case study	Entropy	Entropy ratio	Gini index	Sum minority	Max minority	Twoing rule
Single-output						
cartpole	**253**	257	255	259	277	**253**
tworooms	**27**	37	**27**	39	2,627	**27**
helicopter	**6,347**	7,363	7,177	31,835	125,727	6,429
cruise	**987**	1,161	1,065	11,131	89,503	1,043
dcdc	**271**	391	275	∞	2,429	277
Multi-output						
tenrooms	17,297	**15,951**	17,297	18,565	26,751	17,415
truck_trailer	338,389	348,959	**312,741**	442,013	561,083	316,457
traffic	**12,573**	∞	16,627	276,067	∞	15,319
vehicle	13,237	15,677	13,135	32,271	39,129	**13,109**
aircraft	**913,857**	932,625	923,709	∞	2,242,773	922,727

E More information on the better determinization

Here we give more information on the new insights about determinization. First we give the proof that it suffices to consider complete determinization functions. Then we give the derivation of multi-label entropy and afterwards multi-label Gini index. Lastly, we give another derivation of multi-label entropy and Gini-index, to provide more intuition and insight to their workings. Parts of this appendix are almost verbatim from the Bachelor's thesis [29].

E.1 Considering complete determinization functions suffices

We show that we can reduce the search space by only considering complete determinization functions, i.e. determinization functions such that for all states s we have $|\delta(C)(s)| = 1$. In particular, we prove that there always exists a complete determinization with minimal impurity in Proposition 1 and Theorem 1. We limit our discussion to the entropy and the Gini index, since these impurity measures are the most widely used and performed the best in our experiments. We shorten the terminology and in the following refer to determinization functions as determinizations.

Proposition 1. *Given an impurity measure $\phi \in \{\text{ent}, \text{Gini}\}$, a predicate ρ, and a controller D, for every incomplete determinization δ of D there exists a complete determinization $\bar{\delta}$ of D with $\phi(\rho, \bar{\delta}, D) \leq \phi(\rho, \delta, D)$.*

Proof. We outline a procedure that turns an incomplete determinization into a complete determinization with at most the same impurity. Let δ be an incomplete determinization of D with co-domain $c\mathbb{D}(\delta(D))$. Furthermore, let $n_{i,y}$ denote the number of data points that are assigned the (possibly non-deterministic) label $y \in c\mathbb{D}(\delta(D))$ under δ in the i^{th} sub-dataset D_i created by ρ.

Let us start with $\phi = \text{ent}$. We have that

$$\text{ent}(\rho, \delta, D) = \sum_{i \in [m]} \frac{|D_i|}{|D|} H(\delta, D_i),$$

where

$$H(\delta, D_i) = - \sum_{y \in c\mathbb{D}(\delta(D))} \frac{n_{i,y}}{|D_i|} \log_2 \left(\frac{n_{i,y}}{|D_i|} \right).$$

Since δ is incomplete, there is a label $q \in c\mathbb{D}(\delta(D))$ with $|q| > 1$. Consider the determinization $\bar{\delta}$ that is equivalent to δ, except that it assigns the single-label $r \in q$ to all data points $x \in X$ where $\delta(x) = q$. We define $\bar{n}_{i,y}$ for $\bar{\delta}$ equivalently to $n_{i,y}$ for δ. We fix a specific sub-dataset D_i and drop the corresponding index to simplify notation. Then,

$$\begin{aligned} H\left(\bar{\delta}\right) &= - \sum_{y \in c\mathbb{D}(\bar{\delta}(D))} \frac{\bar{n}_y}{|D|} \log_2 \left(\frac{\bar{n}_y}{|D|} \right) \\ &= - \sum_{y \in c\mathbb{D}(\delta(D)) \setminus \{q, r\}} \frac{n_y}{|D|} \log_2 \left(\frac{n_y}{|D|} \right) - \frac{\bar{n}_r}{|D|} \log_2 \left(\frac{\bar{n}_r}{|D|} \right). \end{aligned}$$

It follows that

$$\begin{gathered} H(\delta) - H\left(\bar{\delta}\right) \\ = - \frac{n_q}{|D|} \log_2 \left(\frac{n_q}{|D|} \right) - \frac{n_r}{|D|} \log_2 \left(\frac{n_r}{|D|} \right) + \frac{\bar{n}_r}{|D|} \log_2 \left(\frac{\bar{n}_r}{|D|} \right). \end{gathered}$$

With

$$\bar{n}_r = n_r + n_q,$$

we obtain:

$$\begin{gathered} H(\delta) - H\left(\bar{\delta}\right) \\ = - \frac{n_q}{|D|} \log_2 \left(\frac{n_q}{|D|} \right) - \frac{n_r}{|D|} \log_2 \left(\frac{n_r}{|D|} \right) + \frac{n_r + n_q}{|D|} \log_2 \left(\frac{n_r + n_q}{|D|} \right). \end{gathered}$$

First, consider the special case of $n_r = 0$. As is usual in information theory, we evaluate $0 \log_2(0)$ as $\lim_{x\to 0} x \log_2(x) = 0$, and thus arrive at

$$H(\delta) - H\left(\bar{\delta}\right) = 0.$$

If $n_r > 0$, we have

$$\begin{gathered} H(\delta) - H\left(\bar{\delta}\right) \\ = \frac{n_q}{|D|}\left(\log_2\left(\frac{n_r + n_q}{|D|}\right) - \log_2\left(\frac{n_q}{|D|}\right)\right) + \frac{n_r}{|D|}\left(\log_2\left(\frac{n_r + n_q}{|D|}\right) - \log_2\left(\frac{n_r}{|D|}\right)\right) \\ > 0, \end{gathered}$$

where the last step follows from the fact that $\log_2$ is strictly increasing.

Thus, for every sub-dataset D_i, we have $H\left(\bar{\delta}, D_i\right) \leq H(\delta, D_i)$, and consequently

$$\mathrm{ent}(\rho, \bar{\delta}, D) \leq \mathrm{ent}(\rho, \delta, D).$$

Note the following key point: $\bar{\delta}$ is "more deterministic" than δ as there are fewer states to which it assigns a label y with $|y| > 1$. If we thus continue this process of producing "more deterministic" determinizations (now starting with $\bar{\delta}$), we will eventually reach a complete determinization with an entropy less than or equal to the entropy of δ.

A similar analysis can be conducted for the case of $\phi = \mathrm{Gini}$. With the same definitions as above, we obtain

$$G\left(\bar{\delta}\right) = 1 - \sum_{y \in c\mathbb{D}(\delta(D)) \setminus \{q, r\}} \left(\frac{n_y}{|D|}\right)^2 - \left(\frac{\bar{n}_r}{|D|}\right)^2.$$

Then,

$$\begin{aligned} & G(\delta) - G\left(\bar{\delta}\right) \\ &= -\left(\frac{n_q}{|D|}\right)^2 - \left(\frac{n_r}{|D|}\right)^2 + \left(\frac{n_r + n_q}{|D|}\right)^2 \\ &= -\frac{n_q^2}{|D|^2} - \frac{n_r^2}{|D|^2} + \frac{n_r^2 + 2n_r n_q + n_q^2}{|D|^2} \\ &= \frac{2n_r n_q}{|D|^2} \\ &\geq 0. \end{aligned}$$

Therefore, similar as above, we have

$$\mathrm{Gini}(\rho, \bar{\delta}, D) \leq \mathrm{Gini}(\rho, \delta, D)$$

and can continue this process to eventually reach a complete determinization with Gini index less than or equal to the Gini index of δ.

Theorem 1. *Let Δ^* be the set of determinizations that achieve the minimal impurity with respect to an impurity measure $\phi \in \{\mathrm{ent}, \mathrm{Gini}\}$, a predicate ρ, and a dataset D. Then, there exists a $\delta^* \in \Delta^*$ that is complete.*

Proof. We give an indirect proof. Assume every determinization $\delta^* \in \Delta^*$ is incomplete. Then, by Proposition 1, we know that there exists a $\bar{\delta}^*$ that is complete and $\phi(\rho, \bar{\delta}^*, D) \leq \phi(\rho, \delta^*, D)$ for every $\delta^* \in \Delta^*$. However, this means that $\bar{\delta}^*$ achieves the minimal impurity and would have to be an element of Δ^*. Therefore, Δ^* cannot be the set of determinizations that achieve minimal impurity.

Theorem 1 shows that it suffices to consider all complete determinizations to determine the best predicate. However, the number of complete determinizations is still far too large to simply enumerate them all.

E.2 Multi-label entropy derivation

We start our derivation with the following general formulation for multi-label entropy of a controller C and a determinization δ. For a controller C, $c\mathbb{D}(C)$ denotes the codomain and $|C|$ denotes the size of the domain, and p_y is the empirical frequency of y in $\delta(C)$, formally $p_y = |\{s \in \delta(C) \mid \delta(C)(s) = y\}|$:

$$H(C, \delta) = - \sum_{y \in c\mathbb{D}(\delta(C))} \frac{p_y}{|C|} \log_2(\frac{p_y}{|C|})$$

If δ is the identity-function, the considered co-domain is 2^A, and we have the classic entropy. For any other determinization function, we get different p_y and thus different impurities. As mentioned earlier, trying every possible determinization function δ and calculating the precise p_y is optimal, but infeasible. We showed in Theorem 1 that for calculating impurities it suffices to consider determinization functions that map to singleton sets, i.e. for all states s we have $|\delta(C)(s)| = 1$. Then y is a singleton set $\{a\}$ for some action $a \in A$, and p_y is the frequency of $\{a\}$ in the determinized controller. Thus, we can over-approximate p_y by counting the occurrences of every action a in the controller, which is feasible and was already done for MaxFreq. The difference is that MaxFreq used this to explicitly calculate one fixed determinization function δ that was used for all predicates, while the new approach uses it to over-approximate the multi-label impurity, implicitly using a different determinization function for every considered predicate[5].

To complete the formulation of our heuristic, we add that in the corner case that all states agree on an action, the entropy should be 0, and thus

[5] Note that we formulate the multi-label impurity for a single sub-controller. As for all previous impurity measures, the impurity of a predicate then is the weighted average of the impurities of the sub-controllers. Thus, since the determinization comes into play after the split in every sub-controller, a different determinization is considered for every predicate.

get the following impurity measure, where for an action $a \in A$ we use p_a to denote the frequency of that action in the un-determinized controller, i.e. $p_a = |\{s \in (C) \mid a \in (C)(s)\}|$:

$$\mathrm{MLE}(C) = \begin{cases} 0 & \text{if } \exists a \in A, \forall s \in S : a \in C(s) \\ -\sum_{a \in A} \frac{p_a}{|C|} \log_2(\frac{p_a}{|C|}) & \text{otherwise} \end{cases}$$

E.3 Multi-label Gini index derivation

We proceed with the multi-label formulation of the Gini index, which can be derived similar to multi-label entropy. Applying Theorem 1, we obtain that

$$G(\delta^*_\rho, C) = 1 - \sum_{l \in L} \left(\frac{n_{i,l}}{|C|} \right)^2 . \tag{1}$$

We can again estimate the $n_{i,l}$ with the approximation of maximum frequency as in the entropy derivation. However, we need to be careful since we over-approximate and the value of the sum in Eq. 1 can therefore be greater than 1. In order to keep the impurity non-negative, we thus need to subtract the sum not from 1, but from its maximal value $|L|$. Finally, again treating the corner case of all labels agreeing, we get

$$\widehat{G}(C) = \begin{cases} 0, & \text{if } \exists l \in L : f_i(l) = |C| \\ |L| - \sum_{l \in L} \left(\frac{f_i(l)}{|C|} \right)^2, & \text{otherwise.} \end{cases}$$

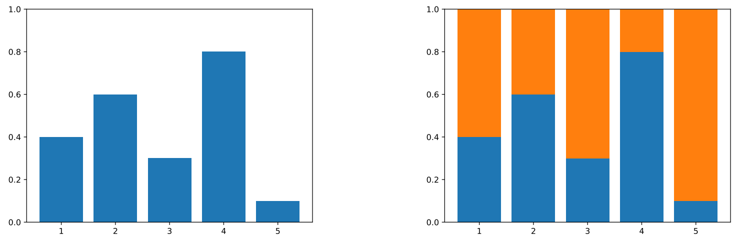
The complete multi-label Gini index is then again the weighted average of the estimated values $\widehat{G}(C)$.

E.4 An alternative point of view

Having derived multi-label impurity measures formally, we now want to point out an alternative, more intuitive point of view that may shed some light on their internal workings. Let us fix a specific sub-controller D_i with frequency function f_i. Plotting $f_i/|D_i|$, i.e. the fraction of data points that can be assigned a specific single-label, yields a bar chart that may look like the one depicted in Fig. 7 (left).

If we now had to construct an impurity measure solely from this bar chart, we might make the following observations:

1. The impurity should be 0 if all bars have a value of 1, since then every label can be assigned to every data point.
2. The impurity should be high if there are many bars with low values.
3. Generally, if there are fewer bars the impurity should be lower, because if there are fewer labels we will probably need fewer splits of the dataset.



(a)

(b)

Fig. 7: Bar chart of the frequency function. The bar chart (left) for a controller with actions $1, \ldots, 5$ and the error bars indicating the impurity (right).

This already rules out two simple ideas that come to mind: we cannot use the reciprocal of the sum of the bars, because this would violate point 2 if there is a great number of different labels. We also cannot use the reciprocal of the mean of the bars, since this would not take observation 3 into account. Instead, we could come up with the following impurity measure that satisfies all three desired properties: we measure how much is missing from each bar to get a value of 1 and return the sum of these values. This idea is depicted in Fig. 7 (right).

Formalizing this concept yields the following function to measure the impurity of a sub-dataset:

$$\begin{aligned} J(D_i) &= \sum_{l \in 2^A} 1 - \frac{f_i(l)}{|D_i|} && (2) \\ &= |L| - \sum_{l \in 2^A} \frac{f_i(l)}{|D_i|}. && (3) \end{aligned}$$

We could then compute the impurity of a predicate as the weighted average of the values $J(D_i)$ for every sub-dataset D_i.

Eq. 3 already looks surprisingly similar to the function $\widehat{G}(D_i)$ of the multi-label Gini index. Indeed, we see that $\widehat{G}(D_i)$ is merely a scaled version of $J(D_i)$: before computing the error bars, the bar chart is scaled with the function $s(x) = x^2$, which penalizes smaller bars more strongly.

With the same idea we can also work towards the multi-label entropy. Consider the scaling function $s(x) = 1 + x\log_2(x)$. We have

$$\begin{aligned} J(D_i, s) &= \sum_{l\in L} 1 - s\left(\frac{f_i(l)}{|D_i|}\right) \\ &= \sum_{l\in L} 1 - 1 - \frac{f_i(l)}{|D_i|}\log_2\left(\frac{f_i(l)}{|D_i|}\right) \\ &= -\sum_{l\in L} \frac{f_i(l)}{|D_i|}\log_2\left(\frac{f_i(l)}{|D_i|}\right), \end{aligned}$$

which matches the function $\widehat{H}(D_i)$ of the multi-label entropy.

The scaling functions of the multi-label entropy and Gini index are plotted in Fig. 8. As previously mentioned, the Gini index scaling function especially increases the impurity assigned to small bars. In contrast, the entropy scaling function penalizes bars with a value of approximately 0.37 the most heavily and assigns small impurity to bars with very low value. While not quite as intuitive at first glance, this also seems like a valid approach: small bars mean that only few data points can be assigned a particular label and we thus only have to separate those few data points. On the other hand, a label that can be assigned to around 40 percent of the examples means that we might have to split off a large fraction of the dataset.

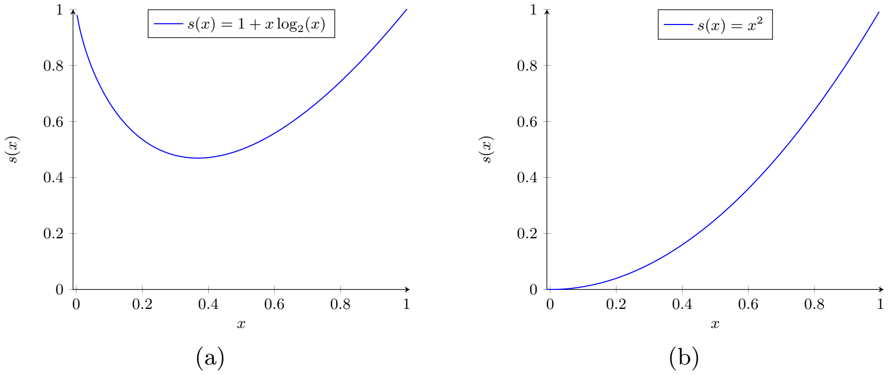
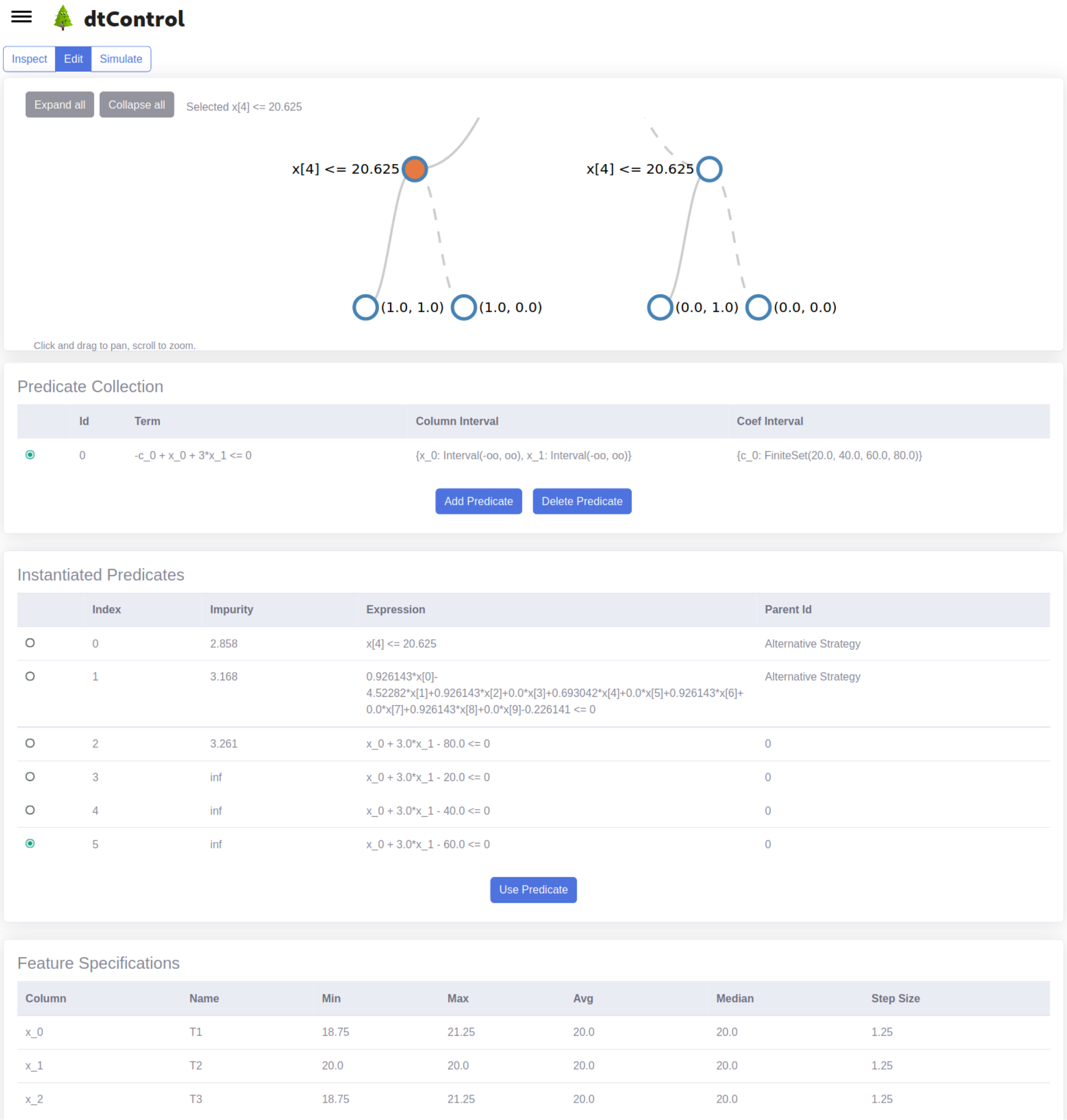


Fig. 8: Plots of the scaling functions arising out of multi-label entropy (left) and Gini index (right).

F Interactive Interface Screenshot



Selected x[4] <= 20.625

Click and drag to pan, scroll to zoom.

Predicate Collection

	Id	Term	Column Interval	Coef Interval
●	0	-c_0 + x_0 + 3*x_1 <= 0	{x_0: Interval(-oo, oo), x_1: Interval(-oo, oo)}	{c_0: FiniteSet(20.0, 40.0, 60.0, 80.0)}

Instantiated Predicates

	Index	Impurity	Expression	Parent Id
○	0	2.858	x[4] <= 20.625	Alternative Strategy
○	1	3.168	0.926143*x[0]-4.52282*x[1]+0.926143*x[2]+0.0*x[3]+0.693042*x[4]+0.0*x[5]+0.926143*x[6]+0.0*x[7]+0.926143*x[8]+0.0*x[9]-0.226141 <= 0	Alternative Strategy
○	2	3.261	x_0 + 3.0*x_1 - 80.0 <= 0	0
○	3	inf	x_0 + 3.0*x_1 - 20.0 <= 0	0
○	4	inf	x_0 + 3.0*x_1 - 40.0 <= 0	0
●	5	inf	x_0 + 3.0*x_1 - 60.0 <= 0	0

Feature Specifications

Column	Name	Min	Max	Avg	Median	Step Size
x_0	T1	18.75	21.25	20.0	20.0	1.25
x_1	T2	20.0	20.0	20.0	20.0	1.25
x_2	T3	18.75	21.25	20.0	20.0	1.25

Fig. 9: The interactive interface for predicate selection.