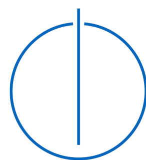
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

`dtControl`: Decision Tree Learning for Explainable Controller Representation

Mathias Jackermeier



DEPARTMENT OF INFORMATICS

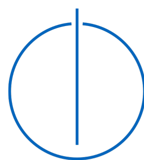
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

dtControl: Decision Tree Learning for Explainable Controller Representation

dtControl: Entscheidungsbaum-Lernen für erklärbare Repräsentation von Controllern

Author: Mathias Jackermeier
Supervisor: Prof. Dr. Jan Křetínský
Advisors: M.Sc. Maximilian Weininger
M.Sc. Pranav Ashok
Submission Date: 05.05.2020



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

Munich, 05.05.2020 Mathias Jackermeier

Acknowledgments

I thank my advisors M.Sc. Maximilian Weininger and M.Sc. Pranav Ashok for the many fruitful discussions and the continuous exchange of ideas over the last year and during my work on this thesis. Their willingness to meet frequently, often for several hours, has been greatly appreciated, as well as the opportunity to work from their office during the occasionally stressful weeks leading up to the submission of our joint conference paper. Maxi's valuable comments greatly improved the manuscript, especially the structure of Chapter 7.

I also want to express my deep gratitude to my supervisor Prof. Dr. Jan Křetínský, who introduced me to the exciting field of formal verification. Jan enabled me to gain first research experiences and contribute to the project that would ultimately result in my first academic publication and form the cornerstone of this thesis. I have always appreciated the time he devoted to our meetings as I know how busy he usually is.

Finally, I thank my family and friends for their continued support throughout my studies and during the writing of this thesis.

Abstract

Controllers are central objects in many model checking and synthesis processes. They can not only be used for implementation, but also have the potential to reveal insights about the system and validate the correctness of or highlight errors in the underlying model. To leverage this potential, succinct and easily explainable data structures for controller representation are required. Recent work has shown that decision trees are particularly well-suited to this task, providing much more concise and understandable representations than lookup tables or binary decision diagrams. This thesis presents an overview of decision tree learning for controller representation and proposes a variety of algorithmic improvements that make the resulting trees more efficient and explainable. These techniques have been implemented in the latest version of the tool `dtControl`, which enables the easy conversion of controllers from many verification tools to decision trees. We furthermore introduce a thoroughly redesigned software architecture for `dtControl`, allowing for much more flexibility and extensibility. We demonstrate the effectiveness of our methods on a variety of case studies, where we can in almost all cases reduce controller sizes by at least 96% compared to a lookup table representation.

Kurzfassung

Controller sind wesentliche Bestandteile vieler Modellprüfungs- und Syntheseverfahren. Sie werden einerseits zur Implementierung verwendet und erlauben andererseits Einblicke in das System zu gewinnen, sowie die Korrektheit beziehungsweise potenzielle Fehler des zugrundeliegenden Modells aufzuzeigen. Um dieses Potenzial nutzen zu können, werden kompakte und leicht verständliche Datenstrukturen für die Repräsentation von Controllern benötigt. Jüngste Forschungsergebnisse haben gezeigt, dass Entscheidungsbäume besonders gut für diese Aufgabe geeignet sind und sehr viel kleinere und eklärbarere Repräsentationen ergeben als Lookup-Tabellen oder binäre Entscheidungsdiagramme. Diese Bachelorarbeit gibt einen Überblick über Entscheidungsbaum-Lernen zur Repräsentation von Controllern und präsentiert verschiedene algorithmische Verbesserungen, die zu effizienteren und verständlicheren Bäumen führen. Diese Verfahren wurden in der neuesten Version des Programms `dtControl` implementiert, welches es ermöglicht, Controller von vielen Verifikationstools in Entscheidungsbäume umzuwandeln. Darüber hinaus wurde die Softwarearchitektur des Programms von Grund auf neu konzipiert, um mehr Flexibilität und Erweiterbarkeit zu ermöglichen. Die Effektivität unserer Methoden wird anhand einer Vielzahl von Fallstudien nachgewiesen, in denen unsere Algorithmen fast immer eine Größenreduktion von mindestens 96% gegenüber Lookup-Tabellen erreichen.

Contents

Acknowledgments v

Abstract vii

Kurzfassung ix

1 Introduction 1

2 Related Work 5

3 Preliminaries 7
3.1 Controllers 7
3.2 Machine learning 8
3.2.1 Learning algorithms and the classification task 8
3.2.2 Generalization and overfitting 9
3.3 Decision trees 10
3.3.1 Data structure 10
3.3.2 Decision tree learning 12

4 Tool 15
4.1 Overview 15
4.2 Workflow 15
4.2.1 Input formats 15
4.2.2 Parameters 16
4.2.3 Output formats 17

5 Decision Tree Learning for Controller Representation 21
5.1 Controller representation as a classification problem 21
5.1.1 The feature space 21
5.1.2 The labels 21
5.1.3 The classification problem 22
5.1.4 Other approaches 22
5.1.5 Comparison of the verification and the machine learning setting . 23
5.2 Predicates 24
5.2.1 Numeric features 24
5.2.2 Categorical features 27

5.3 Impurity measures 30
5.3.1 Entropy 30
5.3.2 Entropy ratio 31
5.3.3 Gini index 31
5.3.4 Twoing rule 32
5.3.5 Sum minority 32
5.3.6 Max minority 33
5.3.7 Area under the receiver-operator curve 33
5.4 Determinization 34
5.4.1 Determinizing before decision tree learning 34
5.4.2 Safe pruning 34
5.4.3 Early stopping 36
5.4.4 Improving impurity estimates in the context of determinization . 38
5.4.5 Tradeoff between decision tree size and optimality 45

6 Implementation **47**
6.1 Design goals 47
6.2 Overview of the software architecture 48
6.3 Object model 49
6.3.1 Dataset management 49
6.3.2 Decision tree learning 50
6.3.3 Benchmarking 52
6.3.4 User Interface 53

7 Evaluation **55**
7.1 Overall results 55
7.1.1 Model checking of Markov decision processes 55
7.1.2 Controller synthesis for cyber-physical systems 58
7.2 Detailed comparison of decision tree learning algorithms 61
7.2.1 Model checking of Markov decision processes 61
7.2.2 Controller synthesis for cyber-physical systems 64

8 Future Work **69**
8.1 Improving decision tree learning 69
8.2 Extending `dtControl` 70

9 Conclusion **71**

Acronyms **73**

Bibliography **75**

1 Introduction

In the modern world, computer systems are becoming ever more present in nearly every aspect of everyday life, as for instance in the form of driving assistants, smart home technology, or medical equipment. Designing and programming these devices is a profoundly difficult task, especially when they need to operate in safety-critical, but unreliable environments. It is therefore hardly surprising that such systems occasionally fail — often with dire consequences: for example, programming errors in the Therac-25 radiation therapy machine caused the device to sometimes give its patients massive overdoses in radiation, resulting in deaths and serious injuries [LT93].

A common and obvious technique to try to prevent such failures is *testing* systems before their deployment, whether that is automatically testing individual software components or performing end-user acceptance tests that treat the system as a black-box and only validate its external behavior. While tests should of course always be conducted, they are problematic in that they do not provide any guarantees for the system: if every test is successful, the system may indeed be correct — or the tests may simply be insufficient. To put it in the words of Dijkstra [Dij72]: "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

An alternative approach is provided by the field of *formal verification*, which is concerned with rigorously proving the correctness of systems using a variety of formal mathematical methods. In contrast to tests, these proofs guarantee that the system behaves as specified, which is of great benefit, especially in many safety-critical domains.

A particularly prominent formal verification technique is *model checking* [CGP01], in which an abstract mathematical model of a system, usually in the form of some finite state automaton, is first constructed and subsequently analyzed with regard to a set of formally specified properties. These properties could for example state that an airbag must deploy within 20 milliseconds upon detection of a crash in a car safety system or that the probability of successful message transmission must be greater than 99% in a communication protocol [NP14]. Model checking has been successfully employed in a wide variety of applications ranging from automotive systems to cloud computing [Alj+09; KM11; NP14].

Besides only determining whether the system satisfies the desired properties, the model checking process often also produces a *controller* that specifies exactly how the properties are satisfied; or in the case that they are not, a controller representing a counterexample to the properties. These controllers are useful in various ways: not only can they be used for implementation, e.g. of communication protocols, but they are also essential to understand the inner workings of the system and verify that the

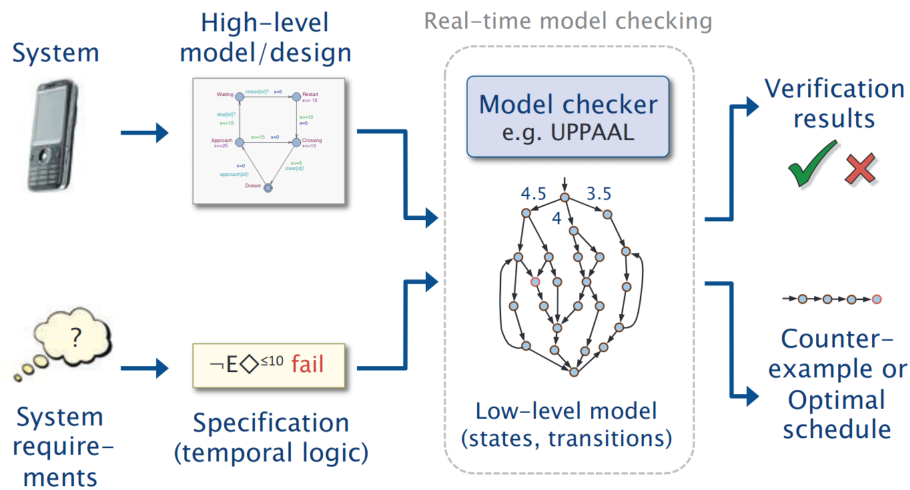


Figure 1.1: Overview of the controller synthesis model checking process [NP14].

mathematical model is correct. In the case of counterexamples, they can provide a means to identify and fix bugs.

Controller synthesis is a special form of model checking that emphasizes the construction of controllers even more: instead of merely verifying properties of a given system, this technique aims to directly construct a controller such that the resulting system provably satisfies a given specification over its behavior. A schematic overview of the process is given in Fig. 1.1. We initially have a system and a set of requirements that it needs to satisfy. In the first step, a high-level mathematical model of the system is constructed and the requirements are mapped to a formal specification, for instance in temporal logic. We then invoke the actual model checking process, which produces both the verification results — an answer to the question whether it is possible to fulfill the requirements — and a controller that either satisfies the formal specification or represents a counterexample.

To illustrate, consider the example given in [RZ16]: we provide a mathematical description of the bicycle dynamics of a robot and the environment in which it must operate, as well as a specification of the robot's behavior, i.e. that it must avoid the obstacles in the environment. Given this information, controller synthesis tools such as SCOTS [RZ16] are able to produce a provably-correct controller that moves the robot through the environment while avoiding the obstacles. Successful industrial applications of controller synthesis include for example temperature control [Jes+07] and operating hydraulic pumps [Cas+09].

The format in which most formal verification tools output controllers is usually just a simple list of state-action pairs (i.e. a lookup table), which specifies which actions the controller can take depending on the current state of the system. For instance, in the robot motion planning example given above, the states of the system correspond to the current position and orientation of the robot, and the actions correspond to setting

the velocity and steering angle. `SCOTS` simply produces a list of velocities and steering angles that the controller can set for every possible position and orientation of the robot.

As systems can often have thousands or even millions of states, such a lookup table representation of controllers is impractical and has several disadvantages: first, its sheer size makes it inherently incomprehensible, with the result that we cannot understand how the controller actually operates. Such a representation will not yield any insights about the semantics and inner workings of the system, nor is it suitable to verify the mathematical model itself. In the case of counterexamples, we will hardly be able to use it to find and fix bugs. Second, the embedded devices on which controllers are usually implemented often only have very limited amounts of memory and it is therefore practically not feasible to work with a lookup table of up to several hundred megabytes. It is thus evident that we need some other data structure for controller representation that is ideally both *succinct* and *explainable* [Brá+18].

While binary decision diagrams [Bry86] have been traditionally used for controller representation, a lot of progress in recent years has been made with *decision trees*, e.g. [Mit97], a data structure originally from the field of machine learning. They satisfy exactly the two key properties identified above: decision trees are commonly several orders of magnitude smaller than lookup tables [Ash+19b], making them much better suited for implementation, and represent the controller on a semantic level, which makes them easily explainable and understandable [Brá+15]. Recent work has investigated primarily how traditional machine learning algorithms for the construction of decision trees can be adapted to and extended for the problem of controller representation [Brá+15; Brá+18; Ash+19a; Ash+19b; Ash+20b].

Contribution. In this thesis, we aim to provide an overview of these approaches and refine them in defining ways to yield even smaller, more explainable decision trees. To this end, we

- explore different ways to handle controllers with states described by categorical variables, thereby generalizing from binary trees to arbitrary m-ary trees,
- investigate the impact of different impurity measures on decision tree sizes,
- develop a theoretical framework for determinization methods and introduce the concept of multi-label impurity measures.

We demonstrate the effectiveness of our methods on a variety of case studies, which show that particularly our specialized algorithms for categorical variables and novel determinization techniques indeed greatly reduce tree sizes. Generally, our best algorithms reduce the size of the lookup table by at least 96% on almost all of our case studies and usually perform significantly better than binary decision diagrams. Furthermore, we manage to produce easily understandable trees with only a single-digit number of nodes in many cases, and illustrate with an example that they indeed capture the intuitive semantics of the underlying system.

Our work has been implemented in the latest version of the tool `dtControl` [Ash+20b], which provides many algorithms and parameters to construct decision trees for controller representation. Parts of the software architecture of the tool have been thoroughly redesigned to allow for much more flexibility and extensibility. We have also added support for the probabilistic model checker PRISM [KNP11] to `dtControl`.

Structure. The rest of this thesis is structured as follows: we begin with a brief discussion of related work (Chapter 2), after which we introduce some fundamental preliminary concepts from machine learning and give precise definitions of controllers and decision trees for the purpose of this thesis (Chapter 3). The primary software artifact we develop and extend is the tool `dtControl`, which we describe in Chapter 4 from a user's point of view, thereby giving a high-level overview of the functionality we are able to provide. We proceed with Chapter 5, in which we examine at great length the theoretical details of how decision tree learning can be used for controller representation. It is in this chapter that we develop our primary algorithmic contributions, i.e. the handling of categorical variables and novel determinization techniques. With the theoretical background covered, we can then examine the concrete implementation of `dtControl` in Chapter 6, where we introduce its new software architecture. Chapter 7 demonstrates the performance of our methods in practice on a variety of case studies. Finally, we briefly consider possible directions of future research in Chapter 8, before concluding the thesis in Chapter 9.

2 Related Work

We give a brief overview of previous work related to this thesis. We begin with a general review of decision trees and their use in formal methods, subsequently consider alternative data structures for controller representation, and finally give an overview of related tools.

Decision trees. Decision trees, e.g. [Mit97, Ch. 3], are well-known and widely-used data structures for classification and regression tasks in the field of machine learning. They have several advantages compared to other methods from the field, such as their non-parametric nature and explainability [Mur98]. Our work builds upon some of the most fundamental decision tree learning algorithms, such as `CART` [Bre+84], `ID3` [Qui86], and `C4.5` [Qui93]. A survey on the use of decision trees in multiple disciplines is given in [Mur98].

Several extensions of the basic algorithms have been investigated, mainly to allow for more complex, *oblique* predicates in the decision nodes, either by the use of some heuristic [Mur+93], or by combining decision trees with linear classifiers [Utg88; LHF03; CE07]. We investigate the effect of these ideas on tree sizes in the verification setting.

Decision trees have also been adapted to the problem of *multi-label* classification, which is similar to the setting of nondeterministic controller representation, e.g. [CK01; CHC03; Zha+10]. The fundamental difference between those techniques and our approach is that we need to ensure that a subset of the possible labels is always exactly represented by the decision tree, while multi-label classification in machine learning can allow small errors in the form of predicting wrong labels. Interestingly, the impurity measure developed by Clare and King [CK01] is fairly similar to our multi-label entropy formulation (Section 5.4.4), although they give less theoretical justification. An overview of multi-label classification techniques can be found in [TK07; Mad+12].

Decision trees in formal methods. In the verification setting, decision trees have previously been used for the representation of controllers or strategies, for example in Markov decision processes (MDP) [Brá+15] or graph games [Brá+18; NM19]. They have also been combined with linear classifiers [Ash+19a] and been investigated with regards to optimality [Ash+19b]. The primary difference between those approaches and ours is that we mostly use a different, more intuitive, data representation and provide specialized techniques for categorical variables that lead to much more explainable decision trees. Furthermore, we allow linear classifiers also in inner nodes and thoroughly investigate determinization and impurity measures as key components of the learning algorithm. Some of the techniques we use are based on our previous work in [Ash+20b].

Neider et al. [NSM16] use decision trees in the domain of program synthesis for the representation of piecewise functions and formulate a multi-label learning problem very similar to our own. Their developed impurity measure based on minimal hitting sets is quite different from our approaches, but could in principle also be used in the setting of controller representation.

Decision diagrams. Binary decision diagrams (BDD) [Bry86] are an alternative, widely used data structure for controller representation. However, they have several drawbacks in comparison with decision trees [Brá+15; Brá+18]: first, they operate on a bit-level representation of state-action pairs, which makes them barely comprehensible at all. Second, they are usually only small if a good variable ordering is chosen, which is a notoriously difficult problem. In contrast, decision trees are easily understandable as they directly work on the semantic level of the variables and there exist a plethora of sophisticated algorithms for their construction.

Algebraic decision diagrams (ADD) [Bah+97] are an extension of BDDs that allow for leaf nodes from an arbitrary, not necessarily Boolean, domain. Furthermore, they are often used not with a bit-level representation, but operate on Boolean vectors that describe the outcome of several tests. Any such ADD can straightforwardly be converted to a decision tree by inserting the tests themselves into the inner nodes and splitting leaves to create an acyclic graph. They have been used in controller representation for example by Girard [Gir12], although he does not provide any concrete algorithm.

Tools. We build upon and extend the tool `dtControl` [Ash+20b], which was developed specifically for the construction of decision trees for controller representation. All alternative decision tree building tools we are aware of, such as `CART` [Bre+84], `C4.5` [Qui93], OC1 [Mur+93] or WEKA [Fra+10], are geared towards the machine learning usecase, which often makes it harder to obtain decision trees precisely representing controllers. They also do not implement our specialized algorithms, nor do they offer close integration with verification tools. MEKA [Rea+16] is an extension of WEKA allowing for multi-label classification, although it is also primarily a machine learning tool. A variety of programs exist for the construction and manipulation of BDDs and ADDs, e.g. [Som01; Gos+19].

3 Preliminaries

In this chapter, we introduce the basic terms and definitions, as well as the fundamental algorithms our work builds upon. We start by providing a precise definition of controllers for the purpose of this thesis and examine different properties they may have. We proceed by giving a brief introduction to some concepts from the field of machine learning that are especially important in our context. Finally, we introduce decision trees and the abstract decision tree learning algorithm, which forms the basis of most of our work.

3.1 Controllers

Recall that the starting point for the model checking and synthesis problems is some kind of abstract mathematical model of a system and a set of properties that it should satisfy. The verification of this model produces a controller that specifies the behavior of the system in such a way that the properties are guaranteed to be satisfied. Controllers can be used both to deepen one's understanding of a system and directly for implementation.

There exist a great deal of mathematical models that can be used to describe a system. For instance, Markov decision processes [Put94], timed automata [AD90], and symbolic models of hybrid systems [Tab09] have all been used successfully in various verification contexts. While these are different models used in different scenarios, they still share some key concepts:

- They define a notion of *states* of the system, which depend on a number of *state variables*.
- They define *actions* that the system can perform in each state.
- Executing a particular action in a particular state changes the state of the system; it *transitions* into a new state.

To specify the behavior of a system, a controller then needs to exactly describe which action can be taken in which state in order to satisfy the given properties. These actions are often referred to as *safe*.

Inspired by [Brá+15; Ash+19b], we formally define a controller as follows:

Definition 1. Given a model $\mathcal{M}$ with states S and actions A, a *controller* $C \subseteq S \times A$ for that model is a (left-total) relation specifying safe actions for a state. We assume that $\mathcal{M}$'s states are determined by n state variables v_i with domain $\mathrm{Dom}(v_i)$, i.e. $S \subseteq \prod_{i=1}^{n} \mathrm{Dom}(v_i)$.

Remark 1. In general, the actions that a controller deems safe in a particular state could depend on the previous actions taken by the controller. However, these controllers with *memory* are often not required for optimal behavior [Brá+15; EJ91]. In this thesis, we only consider the *memoryless* controllers defined above.

In some models, the actions of the controller correspond to the setting of one or more *output variables*. For instance, a room temperature control system might be able to simultaneously manage heaters in multiple rooms. This would still be considered a single action in the definition above; we do not distinguish between these models and models with single-output actions.

Remark 2. The terminology of input and output can become quite confusing in the discussion of controllers, since the *output* of the controller often acts as the *input* of the system to be controlled (e.g. in the temperature control system given above). In the context of this thesis, we use the term *output*, since we are mainly interested in the controllers themselves, and efficient representations thereof.

An important characteristic of controllers is whether they are *deterministic* or *nondeterministic*. We say that a controller C is deterministic if and only if it only assigns one action to each state, i.e. for all $s \in S$ we have that $|\{(s, a) : a \in A, (s, a) \in C\}| = 1$. On the other hand, controllers that allow more than one action per state are called nondeterministic (sometimes also *liberal* or *permissive*).

Remark 3. In literature, controllers are also called *strategies*, *policies*, or *schedulers*. In the context of this thesis, these terms are equivalent.

3.2 Machine learning

Decision trees have originally been developed in the field of machine learning. Therefore, we need a basic understanding of some of the main concepts from that field to grasp their underlying ideas in detail. Moreover, such an understanding will allow us to realize the fundamental differences between the verification and the machine learning setting and the resulting challenges that we will need to overcome.

3.2.1 Learning algorithms and the classification task

Machine learning is concerned with the development of *learning algorithms* — algorithms whose performance at some class of tasks improves with some form of experience [Mit97, Ch. 1]. In this thesis, we will focus on the particular task of *classification*, where a computer program needs to assign *labels* to some input. For example, in image recognition, the input is an image of an object and the program tries to label it with the corresponding name of the object.

The following discussion is mainly based on [GBC16, Ch. 5]. In classification, the experience an algorithm can utilize is usually present in the form of a *training dataset* of examples (also known as *data points*) and their labels. For instance, we might have a dataset of hand-labeled images that we provide to our learning algorithm.

We also need to specify how we represent the input to the algorithm. Usually, this is done by extracting a set of *features* or *attributes* from an example that measure important properties. In the case of image recognition, these features would mostly be the color values of the individual pixels of the image, but could for example also include metadata such as whether the image was taken inside or outside. We differentiate between *numeric*, i.e. real-valued features such as the color values of the pixels, and *categorical* features, which can only take on one of a specific set of possible values, e.g. *inside* or *outside*.

Formally, the task of classification is defined as follows:

Definition 2. Given a training dataset $D\colon X \to Y$, explicitly represented as $D \subseteq X \times Y$, where $X \subseteq \mathcal{F}$ is a set of examples represented as feature vectors from some feature space $\mathcal{F}$, and Y is the set of possible labels, the task of *classification* requires learning a function $f\colon \mathcal{F} \to Y$ that should match the training dataset D as closely as possible. The learned function f is called a *classifier*.

One special case of classification important in our setting is *multi-label* classification, e.g. [TK07]. Here, we are given a set $L = \{l_1, \dots, l_n\}$ of single-labels l_i and have that $Y \subseteq 2^L \setminus \{\emptyset\}$. Thus, there exist potentially multiple single-labels that can be assigned to a single data point. This is not to be confused with *multi-class* classification, which simply requires that $|Y| > 2$.

The process of learning the function f is often referred to as *training* a classifier. Once a classifier is trained, the label of a data point $x \in \mathcal{F}$ can be retrieved by evaluating $f(x)$, a process known as *inference* or *prediction*.

In Section 5.1 we will see how we can frame the problem of controller representation as a classification task, which will allow us to utilize learning algorithms such as decision tree learning.

3.2.2 Generalization and overfitting

As is evident from Def. 2, a classifier f is an approximation of the real, unknown function $f^*\colon \mathcal{F} \to Y$, learned with only the limited information from the training dataset D. It is crucial that f also performs well on the data points in $\mathcal{F} \setminus X$, and not just on those on which it has been trained. This ability is called *generalization* [GBC16, Ch. 5].

A key challenge in machine learning is preventing *overfitting*, which means that a classifier performs very well on the training data, but poorly generalizes to previously unseen examples [Mit97, Ch. 3]. This phenomenon can occur if the classifier only learns concepts (or noise) that are specific to the training dataset, but do not extend to the general feature space. In the worst case, a classifier simply memorizes the training dataset, leading to perfect performance on the training data, but very poor predictions for new examples. The concepts of generalization and overfitting are illustrated in Fig. 3.1.

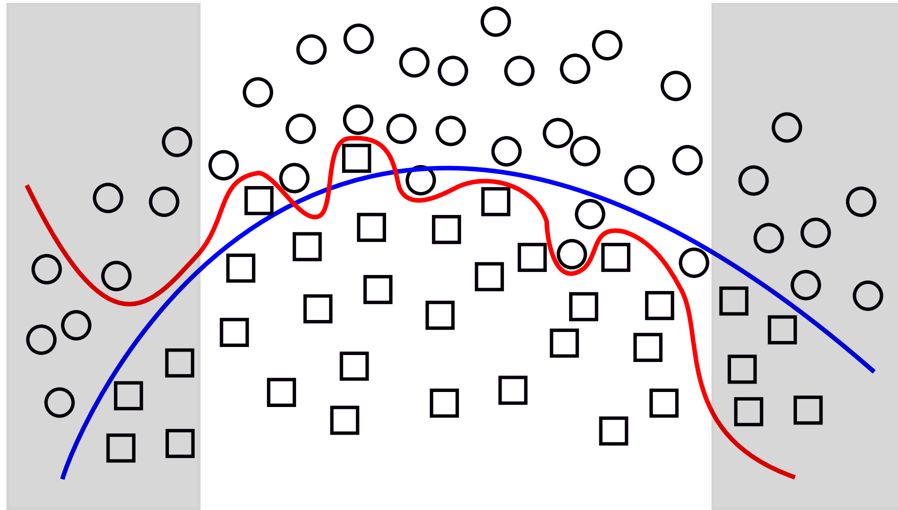


Figure 3.1: The concepts of overfitting and generalization. The data are points in the feature space $\mathcal{F} = \mathbb{R}^2$ with different labels represented as squares and circles. The shaded area indicates previously unseen examples not present in the unshaded training data. The blue line shows a classifier that makes small errors on the training data, but generalizes well. In contrast, the red line represents an overfitting classifier that perfectly recalls the training data, but poorly generalizes.

3.3 Decision trees

We now turn our attention to a particular classifier – decision trees – and a related learning algorithm that have received much attention in a variety of fields [Mur98].

3.3.1 Data structure

The underlying idea of classification with decision trees (DT), e.g. [Mit97, Ch. 3], is simple. In order to classify a data point, we simply ask a number of questions about the features of that data point. Depending on the answers to our questions, we return a different label. A typical question we could ask is "Is the value of the ith feature of the data point ≤ 5?" or "What is the value of the jth feature of the data point?".

It seems natural that the questions we want to ask depend on the answers to the questions we have already asked. This process of asking questions can be conveniently represented in a tree structure: starting from the first question in the root node, we select a subtree depending on the answer to that question, and then proceed recursively until we have enough information to confidently assign a label to the data point.

To illustrate, consider the task of classifying a day as suitable or not for playing tennis based on several attributes, such as weather and wind strength, as introduced in [Qui86; Mit97, Ch. 3]. We might not be willing to play if it rains; if it is overcast, our decision might depend on the temperature; and if it is sunny, we might only want to play if

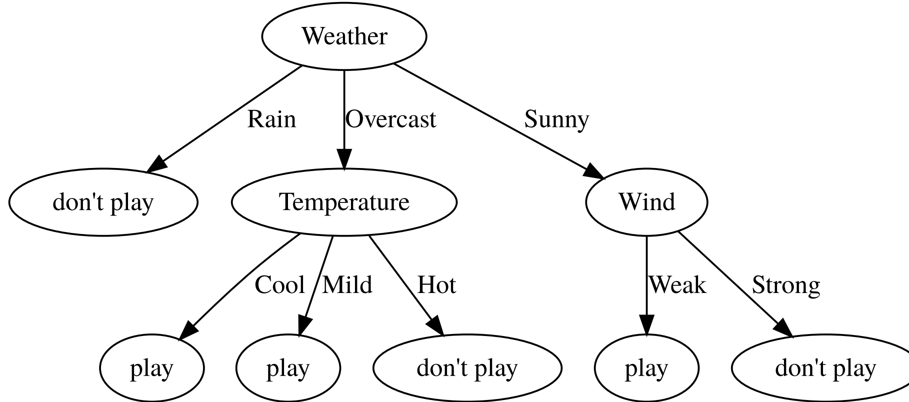


Figure 3.2: A sample decision tree that determines if a day is suitable for playing tennis. In this case, the classes are either *play* or *don't play*. Adapted from [Qui86].

the wind is not too strong. A sample decision tree corresponding to this classification process is depicted in Fig. 3.2. For example, this DT would classify the day

$$\langle \mathit{Weather} = \mathit{Overcast}, \mathit{Temperature} = \mathit{Cool}, \mathit{Wind} = \mathit{Strong}, \mathit{Humidity} = \mathit{Normal} \rangle$$

as *play*, since the instance would take the second branch of the root node and subsequently the first branch of the child node. Note that the DT does not have to consider all attributes — e.g. in this example, it would ignore *Humidity*.

Formally, we give the following definition of DTs:

Definition 3. A *decision tree* (DT) T over a domain $\mathcal{F}$ with a set of labels Y is defined recursively as follows:

1. A DT of height 0 is a label $y \in Y$.
2. A DT of height $h+1$ is a tuple $(\mathcal{T}, \rho)$, where
   - $\mathcal{T} = (T_1, \ldots, T_n)$ is an ordered list of sub-decision trees with height $\leq h$, with at least one $T_i \in \mathcal{T}$ of exactly height h,
   - $\rho \colon \mathcal{F} \to [n]$ is a *predicate* assigning an index corresponding to a subtree $T_i \in \mathcal{T}$ to every data point $x \in \mathcal{F}$.

Given a DT T and a data point $x \in \mathcal{F}$, predicting the label $T_i(x)$ is simple: if T is a leaf, its label y is returned. If T is not a leaf, we evaluate $\rho(x)$ to obtain an index corresponding to a subtree $T_i \in \mathcal{T}$. We then return the result $T_i(x)$ of classifying x with the DT T_i.

An interesting perspective on DTs is that they recursively partition the feature space according to their predicates [Bre+84, Ch. 2]. In the example above, the feature space is first partitioned into the three subspaces containing the points where *Weather* equals *Rain*, *Overcast*, or *Sunny*, respectively. Then those subspaces are again partitioned by the respective subtrees. Finally, when reaching a leaf, we stop partitioning and assign a label to the corresponding subspace.

3.3.2 Decision tree learning

We have seen that DTs are useful structures for representing a classification process. But how do we construct a DT for a given dataset $D \subseteq X \times Y$? This amounts to the problem of training a classifier as defined in Def. 2.

There exist numerous learning algorithms to solve this task, such as `CART` [Bre+84], `ID3` [Qui86], and its successor, `C4.5` [Qui93]. The general principle behind these algorithms is similar: they greedily search through the space of possible DTs in a top-down manner [Mit97, Ch. 3]. In particular, they start with a single root node containing the whole training dataset D, and proceed with the following steps [Qui93, Ch. 2]:

1. If D is *pure*, i.e. there exists a $y \in Y$ such that for all $x \in X$, we have $D(x) = y$, then return the leaf node y.
2. Otherwise, pick the best possible predicate $\rho^* \in \Pi$ out of the set of possible predicates Π. Partition D into subsets $D_1, \ldots, D_n$ using ρ^*, recursively train DTs T_i on those subsets, and return the tuple $(\mathcal{T}, \rho^*)$ with $\mathcal{T} = (T_1, \ldots, T_n)$.

The set of possible predicates Π is a central parameter of the learning algorithm that we will discuss in Section 5.2.

We see that the algorithm is greedy as it immediately picks the best predicate ρ^* at a node. But how do we determine the quality of a predicate? For this, we introduce the concept of impurity measures.

Definition 4. An *impurity measure* ϕ is a function that assigns a non-negative impurity value to a predicate ρ with respect to a (sub-)dataset D.

Intuitively, an impurity measure estimates how "impure" the dataset is after having been split with ρ. For instance, if ρ perfectly separates the data, i.e. the resulting partitions of D are all pure, the impurity $\phi(\rho, D)$ is 0. On the other hand, if there is still a lot of variety in the labels in the partitions after the split, the impurity is large. We discuss several concrete impurity measures in Section 5.3.

The DT learning algorithm is specified formally in Algorithm 1.

Avoiding overfitting. DTs are classifiers that are especially prone to overfit [Mit97, Ch. 3]. To see why, notice that Algorithm 1 is guaranteed to grow a tree that perfectly classifies the training dataset (assuming the considered predicates are expressive enough to completely separate the data). If there is any noise in the training data, or concepts that do not generalize, the DT will memorize this information, and thus perform poorly on unseen examples.

A variety of techniques have been developed to overcome this problem, which can be grouped into two main classes [Mit97, Ch. 3]:

- *early stopping* approaches that stop growing the tree before it perfectly separates the training data,
- *pruning* methods that post-process the tree in some manner.

Algorithm 1 Decision tree learning

Require:
A dataset $D \subseteq X \times Y$, a set of possible predicates Π, and an impurity measure ϕ.

1: **procedure** LEARN-DT(D, Π, ϕ)
2:     **if** D is pure **then**
3:         **return** some $y \in Y$
4:     **end if**
5:     $\rho^* \leftarrow \arg\min_{\rho \in \Pi} \phi(\rho, D)$
6:     **for all** i in $[n]$ **do**
7:         $D_i \leftarrow \{(x,y) \in D : \rho^*(x) = i\}$
8:         $T_i \leftarrow$ LEARN-DT(D_i, Π, ϕ)
9:     **end for**
10:     $\mathcal{T} \leftarrow \{T_1, \ldots, T_n\}$
11:     **return** $(\mathcal{T}, \rho^*)$
12: **end procedure**

Furthermore, an extension of DTs called *random forests* [Bre01] has been developed. These classifiers consist of multiple decision trees and deal with the problem of overfitting by introducing randomness to the tree growing process.

With these adaptations of the tree building algorithm, DTs and random forests have been used successfully in machine learning for decades. Especially random forests have been shown to produce very good results on a variety of datasets [Del+14]. However, we will not investigate these techniques further since overfitting is not a problem in our setting — in fact, it is an essential requirement as we will see in Section 5.1.5.

4 Tool

All techniques for controller representation with decision trees presented in this thesis have been implemented in the latest version of the tool dtControl [Ash+20b]. In this chapter, we give a short overview of the tool from a user's perspective. The interested reader is referred to the more extensive documentation in the user manual, which is available at the official website[1].

4.1 Overview

dtControl is an open-source tool that can convert memoryless controllers from a variety of verification tools to a decision tree representation. It is written in Python, supporting versions 3.6.7+. The only dependencies of dtControl are standard Python packages such as NumPy [Oli06] and scikit-learn [Ped+11], which are automatically installed alongside dtControl if the tool is installed as a pip[2] package. Detailed installation instructions can be found on the official website of the tool.

There are two main ways to interact with dtControl. The command-line interface (CLI) provides convenient access to all of the core functionality and should satisfy the needs of most users who simply want to run dtControl to convert controllers to DTs. We also provide a Python interface for users who want to integrate dtControl into their own Python programs and Jupyter notebooks.

4.2 Workflow

To convert a controller represented as a list of state–action pairs to a DT, the user has to (i) provide the controller as input to dtControl in one of the supported formats, (ii) choose one of the predefined algorithms or specify a custom combination of algorithm parameters, and (iii) select one of the various output formats. A schematic overview of this process is given in Fig. 4.1.

4.2.1 Input formats

dtControl currently supports the following verification tools:

[1] https://dtcontrol.model.in.tum.de/

[2] pip is a standard package-management system for Python.

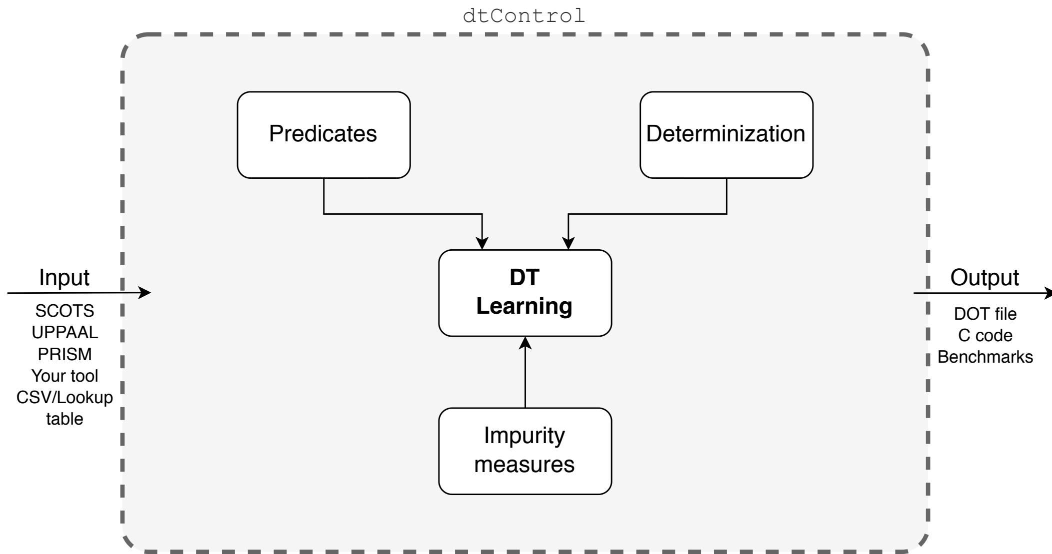


Figure 4.1: A schematic overview of converting controllers with `dtControl`. Adapted from [Ash+20a].

- `SCOTS` [RZ16], an open-source symbolic controller synthesis tool for nonlinear control systems. It can output the synthesized controllers in a sparse matrix format that can directly be used as input for `dtControl`.
- Uppaal Stratego [Dav+15], a model checking and synthesis tool for timed games. `dtControl` can read the controller produced by Uppaal Stratego in its raw format.
- PRISM [KNP11], a probabilistic model checker supporting a variety of models such as discrete- and continuous-time Markov chains, MDPs, and probabilistic timed automata. In our context, not every model is relevant, since some are entirely probabilistic and do not have a concept of actions. `dtControl` currently only supports controllers for MDPs given in PRISM's textual format.

Furthermore, `dtControl` supports a custom comma-separated values (CSV) format that simply lists values of state variables and the corresponding values of output variables. This format can be useful when dealing with output from unsupported verification tools. Another option in that case would be to directly add support for the new tool to `dtControl`, a straightforward process detailed in the developer manual[3].

4.2.2 Parameters

As shown in Fig. 4.1, the DT learning algorithm implemented in `dtControl` has three main parameters. We briefly describe their effect on the learned decision trees. For the theoretical background of the learning algorithm, see Chapter 5.

[3] Also available at `https://dtcontrol.model.in.tum.de/`

- **Predicates**. For a vector of *numeric* features $x^{(n)}$, `dtControl` supports axis-aligned predicates of the form $x_i^{(n)} \leq t$, where $t \in \mathbb{R}$ is an arbitrary threshold, as well as oblique splits of the form $w^\top x^{(n)} \leq t$, where $w \in \mathbb{R}^{|x^{(n)}|}$, $t \in \mathbb{R}$. Oblique splits have the advantage that they can incorporate information from more than one state variable, although they can be harder to interpret.
  For a vector of *categorical* features $x^{(c)}$, `dtControl` supports both binary predicates $x_i^{(c)} = a$, where a is one of the possible values of the feature $x_i^{(c)}$, as well as multi-valued splits with one branch for every possible value the feature $x_i^{(c)}$ can take. If the functionality is enabled, some of the branches of multi-valued splits can also be merged automatically to improve the DT representation.
- **Determinization**. The user has the option to let `dtControl` (partly) determinize nondeterministic controllers, i.e. remove some of the nondeterministic actions, which can significantly reduce tree sizes [Ash+20b]. If determinization is not wanted, `dtControl` retains all information present in the original controller.
- **Impurity measures**. `dtControl` offers a range of different impurity measures, which affect the inner workings of the DT learning algorithm (and thus, potentially, the size of the learned DTs). The default impurity measure, *entropy*, should be kept in most cases. Advanced users can also try other options, such as the *gini index*. `dtControl` furthermore offers extensions of the impurity measures meant to be used in conjunction with determinization.

In order to facilitate the usage of `dtControl` as much as possible, the CLI ships with a number of reasonable default parameter configurations that allow for quick experimentation. Many users will find these default configurations sufficient for their needs.

4.2.3 Output formats

`dtControl` outputs learned decision trees in two formats: in the DOT graph description language [GN00], for visual inspection of the trees, and as C code, which can directly be used for implementation. The C code is simply a chain of if-then-else statements corresponding to the DT. Examples for these output formats are given in Fig. 4.2 and Listing 4.1, respectively.

There is a great number of parameters that can be tweaked in the DT learning algorithm, and it can be fruitful to try many different combinations thereof. To this end, `dtControl` offers benchmarking functionality that allows for the easy training and comparison of differently configured decision trees. The results of a benchmark are displayed in an HTML file, as shown in Fig. 4.3.

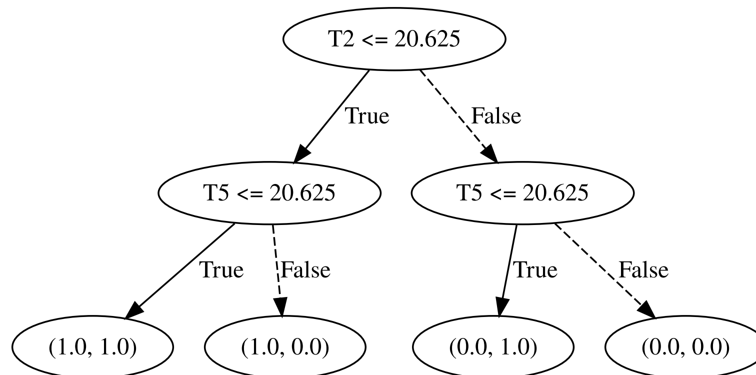


Figure 4.2: A sample DOT output of `dtControl`, as displayed by the `dot` program [GN00].

Listing 4.1: The C code produced by `dtControl` corresponding to the decision tree in Fig. 4.2.

```
void classify(const float x[], float result[]) {
    if (x[1] <= 20.625) {
        if (x[4] <= 20.625) {
            result[0] = 1.0f; result[1] = 1.0f;
        }
        else {
            result[0] = 1.0f; result[1] = 0.0f;
        }
    }
    else {
        if (x[4] <= 20.625) {
            result[0] = 0.0f; result[1] = 1.0f;
        }
        else {
            result[0] = 0.0f; result[1] = 0.0f;
        }
    }
}
```

	Axis	LogReg	Multi-label
10rooms `#(s,a)`: 52488 #doc: 26244	nodes: 17297 inner nodes: 8648 paths: 8649 bandwidth: 14 time: 00:00:07.667 DOT / C	nodes: 147 inner nodes: 73 paths: 74 bandwidth: 7 time: 00:00:11.920 DOT / C	nodes: 7 inner nodes: 3 paths: 4 bandwidth: 2 time: 00:00:00.269 DOT / C
cartpole `#(s,a)`: 21951 #doc: 271	nodes: 253 inner nodes: 126 paths: 127 bandwidth: 7 time: 00:00:00.240 DOT / C	nodes: 199 inner nodes: 99 paths: 100 bandwidth: 7 time: 00:00:03.294 DOT / C	nodes: 7 inner nodes: 3 paths: 4 bandwidth: 2 time: 00:00:00.018 DOT / C
helicopter `#(s,a)`: 561078 #doc: 280539	nodes: 6347 inner nodes: 3173 paths: 3174 bandwidth: 12 time: 00:00:47.359 DOT / C	nodes: 3753 inner nodes: 1876 paths: 1877 bandwidth: 11 time: 00:07:53.336 DOT / C	nodes: 123 inner nodes: 61 paths: 62 bandwidth: 6 time: 00:00:36.030 DOT / C

Figure 4.3: Part of an HTML file created by `dtControl` containing a table with benchmark results.

5 Decision Tree Learning for Controller Representation

With the preliminaries as well as the functionality of the tool `dtControl` covered, we now show how the basic DT learning algorithm described in Section 3.3.2 can be utilized for the problem of controller representation. We begin by examining how this problem can be framed as a classification task and what some of the key challenges are when doing so. Subsequently, we discuss the resulting algorithm and all its parameters in comprehensive detail.

5.1 Controller representation as a classification problem

In order to use DT learning algorithms to build decision trees for controller representation, we first need an instance of a classification problem. The main idea we employ is to treat a controller as a training dataset, which can then be used as input for the learning algorithm. Our approach has previously been described in [Ash+20b] and is similar to the one of [Ash+19b].

Recall that a training dataset is a function $D\colon X \to Y$ that maps feature vectors to labels. A controller $C \subseteq S \times A$ is a relation specifying safe actions for each state. We want to construct a DT that, given a state, returns the corresponding safe actions.

5.1.1 The feature space

With the above definitions, it is immediately obvious how we should define the feature space $\mathcal{F} \supseteq X$ of the classification problem. Since we want to map *states* to *actions*, the feature vectors must be able to represent states. Therefore, for the classification task, we define

$$X = S$$

and

$$\mathcal{F} = \prod_{i=1}^{n} \mathrm{Dom}(v_i),$$

where $v_1, \ldots, v_n$ are again the state variables of the model.

5.1.2 The labels

The set of labels Y is determined by the set of actions A. We distinguish two cases:

Deterministic controllers

If C is deterministic, we can directly set $Y = A$. Then the learned DT will represent a function $f\colon S \to A$ that assigns the (only) safe action $a \in A$ to a state $s \in S$.

Nondeterministic controllers

In the case of a nondeterministic controller C, the classification problem becomes an instance of a *multi-label* classification task (see Def. 2). We set

$$L = A$$

and

$$Y = \{y \in 2^L : \exists s \in S.\, \forall l \in y.\, (s,l) \in C\}.$$

In this case, the learned DT will map from a state s to the set of possible actions $y \in 2^A$ for that state.

Note that the standard DT learning algorithm will simply treat every label in Y as a unique label, disregarding the information that we are actually dealing with a multi-label classification problem. This approach is often called the *label powerset method* [Mad+12]. It has the advantage that the learned DT will retain all of the nondeterminism present in the original controller. Other approaches for dealing with nondeterminism will be discussed in Section 5.4.

5.1.3 The classification problem

We can use the defined sets X and Y to construct a training dataset: In the case of a deterministic controller C, we have

$$D = \{(x \in X, y \in Y) : (x,y) \in C\}.$$

If C is nondeterministic,

$$\begin{aligned} D &= \{(x \in X, y \in Y) : \forall a \in A.\, (x,a) \in C \iff a \in y\} \\ &= \{(x \in X, \{a \in A : (x,a) \in C\})\}. \end{aligned}$$

This training dataset D is the only ingredient we need to now actually run the DT learning algorithm described in Section 3.3.2 to construct a decision tree representing the controller C. With D as input, the algorithm will produce a tree that correctly maps from states to (sets of) possible actions.

5.1.4 Other approaches

Another approach to frame the problem of controller representation as a classification task is to construct a training dataset with only the two classes *Good* and *Bad*, where *Good* contains all state–action pairs that are allowed by the controller, and *Bad* contains

those pairs that are not allowed. A feature vector then consists of both the state variables as well as the suggested action, and the DT can determine whether this combination is safe. This approach has been used in [Brá+15; Brá+18; Ash+19a].

While this idea has been shown to also work rather well, it has a few disadvantages in comparison to our strategy. The first disadvantage is conceptual: we want an understandable representation of a controller, which assigns actions to states. It is thus more natural to learn DTs that also assign actions to states, instead of DTs that assign a label $\in \{Good, Bad\}$ to a combination of state and action. We think that a DT with actions only in its leaf nodes is more interpretable than a DT with actions in inner nodes.

Furthermore, the alternative approach poses two practical problems: first, it tremendously increases the size of the training dataset, since it must also include state–action pairs that are not safe, and thus slows the learning process. Second, if we want to actually compute safe actions for a state as part of the implementation of the controller, we now have to potentially query the DT with many state–action combinations, until we finally try one that is indeed safe.

5.1.5 Comparison of the verification and the machine learning setting

Because we have transformed the problem of controller representation into the machine learning problem of classification, it is important to be aware of the fundamental differences between the verification and the machine learning setting, and why we can still use DT learning algorithms.

As outlined in Section 3.2.2, in machine learning, one of the most important characteristics we want a classifier to have is the ability to generalize. In order to achieve this, various techniques to prevent overfitting are often employed.

In contrast, in the setting of verification, we want our controllers to be *safe*. Thus, we cannot allow any errors in the classification of the training dataset. Even if a single instance of the training data is misclassified, the controller representation is not guaranteed to be safe anymore. This means that we do not wish to prevent the overfitting of the classifier on the training dataset; in fact, it is now an essential requirement [Brá+18].

Generalization, on the other hand, does not apply in the verification setting. The controller we are given already exhaustively lists all relevant states the modeled system can reach, and provides the corresponding actions. Because we then map this controller to the training dataset, there simply is no state in $\mathcal{F} \setminus X$ that would ever be reached by the system. Therefore, the ability of the resulting DT to generalize is entirely irrelevant.

With all these differences, how can it be that DT learning algorithms can still be used in our setting? The answer to this question lies in the fact that the standard DT learning algorithm already trains DTs that completely overfit the training data by continuing with the tree building process until all leaves are pure. We simply do not apply any of the measures generally taken to prevent overfitting, outlined in Section 3.3.2, but appreciatively accept the overfitting DT. Consequently, we must be very careful to never instantiate the learning algorithm in such a way that it is unable to overfit, e.g. when using predicates not expressive enough to completely separate the training dataset.

5.2 Predicates

We now discuss the central parameter of the DT learning algorithm: the set of possible predicates Π. Our discussion is split into two parts: we first examine predicates for numeric features and secondly consider categorical attributes.

5.2.1 Numeric features

For this section, if $x \in X$ is a vector of features, let $x^{(n)}$ denote the features of x which are numeric.

Axis-aligned predicates

The simplest predicate on numeric features is to simply compare one feature against a fixed threshold. This results in so-called *axis-aligned* predicates of the form $x_i^{(n)} \leq t$, where $t \in \mathbb{R}$ is the arbitrary threshold. Formally, the predicate assigns 1 to the data points where $x_i^{(n)} \leq t$, and 2 to the data points where $x_i^{(n)} > t$.

Let us now fix a specific numeric feature λ. How do we choose appropriate thresholds from the infinitely many possible values t? Breiman et al. [Bre+84, Ch. 2] note that, in fact, there are only finitely many thresholds one has to consider and suggest the following procedure:

Let V_λ be the set of possible values for the feature λ in D. We can sort this set to obtain a sequence $v_{\lambda,1}, \dots, v_{\lambda,m}$ of values. Now, note that all thresholds $t \in [v_{\lambda,j}, v_{\lambda,j+1})$ produce the same partition of D: all data points with $\lambda \leq v_{\lambda,j}$ are separated from the data points with $\lambda \geq v_{\lambda,j+1}$. Thus, there are only $m-1$ thresholds we have to consider, where the jth threshold is simply the midpoint, i.e.

$$t_j = \frac{v_{\lambda,j} + v_{\lambda,j+1}}{2}.$$

In order to find the best predicate, we exhaustively consider all such axis-aligned predicates for every numeric feature $x_i^{(n)}$ and pick the one with lowest impurity.

Why are these predicates called *axis-aligned*? Consider the numeric part $\mathcal{F}^{(n)}$ of the feature space with dimension $d := |x^{(n)}|$. A predicate of the form $x_i^{(n)} \leq t$ partitions this feature space along a $d-1$ dimensional, axis-aligned hyperplane. For instance, if $\mathcal{F}^{(n)} = \mathbb{R}^2$, axis-aligned predicates corresponds to lines parallel to either the x- or the y-axis. A DT with axis-aligned predicates and the corresponding partitioning of the feature space is shown in Fig. 5.1.

Oblique predicates

An inherent disadvantage of axis-aligned predicates is that they are only able to utilize information from a single feature at once. However, in many cases we might be interested in a combination of different features. For example, suppose that x_1 and x_2 measure the

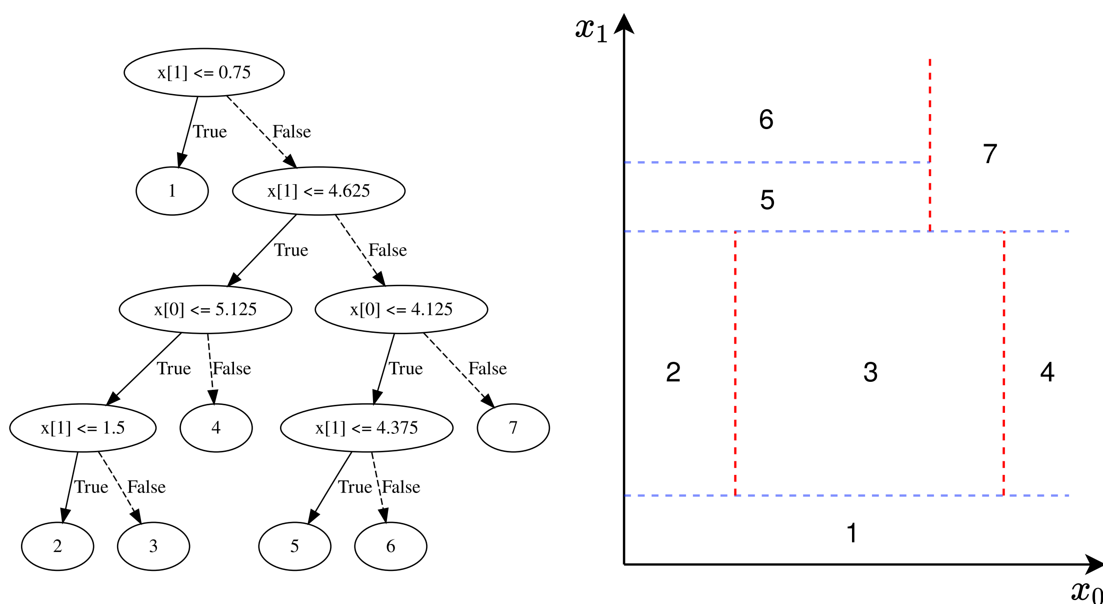


Figure 5.1: A DT with axis-aligned predicates (left) and its representation as a partition of the feature space (right). Concept from [Bre+84, Ch. 2].

distances that two cars have traveled. A controller for some kind of collision avoidance system could probably be represented much more concisely if we could ask questions about the distance *between* the cars, which would correspond to predicates of the form $x_1 - x_2 \leq t$.

In addition to axis-aligned predicates, we thus also consider *oblique* predicates, which are linear combinations of the numeric features. They have the following form:

$$\sum_{i=0}^{d} w_i x_i^{(n)} \leq t,$$

which can be represented more compactly in vector notation:

$$w^\top x \leq t.$$

The vector $w \in \mathbb{R}^d$ specifies the weights used in the linear combination, and $t \in \mathbb{R}$ is again an arbitrary threshold.

From the geometric point of view, these oblique predicates correspond to arbitrary $d - 1$ dimensional hyperplanes in the numeric part of the feature space. As illustrated in Fig. 5.2, the removal of the restriction that the hyperplanes be axis-aligned often allows to partition a dataset with fewer predicates.

The problem of finding the optimal hyperplane $\mathcal{H}$ to separate the dataset is much harder than the problem of finding the best axis-aligned predicate. Simply enumerating all possible hyperplanes, as we did for axis-aligned predicates, is not feasible, since there are exponentially many ways the examples could be partitioned with an oblique

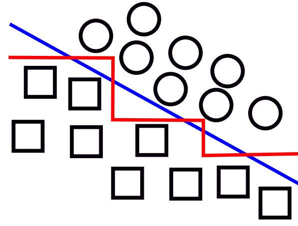


Figure 5.2: Oblique predicates. The blue line illustrates how a dataset in the feature space $\mathcal{F} = \mathbb{R}^2$ can be separated with just one oblique predicate. In contrast, five axis-aligned predicates would be required to partition the same dataset, as indicated by the red line. Concept from [Ash+19a].

predicate [Mur+93]. As such, some kind of heuristic to restrict the set of considered hyperplanes is required.

In this thesis, we investigate the two methods for finding oblique predicates that are currently implemented in `dtControl`:

OC1. The oblique classifier system OC1 [Mur+93] uses a combination of a local search and randomization to find a hyperplane with low impurity. The algorithm roughly works as follows:

1. Choose a random hyperplane $\mathcal{H}$.
2. For every coefficient c_i of $\mathcal{H}$, find a locally optimal value that minimizes the impurity.
3. Perturb $\mathcal{H}$ in a random direction. If this improves the impurity, go to step 2. Otherwise, try a different random direction until a stopping criterion is met.

This random search is carried out multiple times, and finally the hyperplane with the lowest impurity is returned.

It is obvious that this approach is entirely heuristic and does not guarantee in any way that the returned predicate is optimal. However, even if there might be another hyperplane with even lower impurity, tree sizes can already be significantly reduced with any hyperplane that performs better than the standard axis-aligned predicates.

Linear classifiers. An alternative to the random search approach of OC1 is using binary linear classifiers from machine learning to obtain oblique predicates.

Definition 5. A *binary linear classifier* f with labels *Pos* and *Neg*, e.g. [Bis07, Ch. 4], consists of a weight vector $w \in \mathbb{R}^d$ and a threshold $t \in \mathbb{R}$. A feature vector $x \in \mathcal{F}$ is classified as follows:

$$f(x) = \begin{cases} Pos, & \text{if } w^\top x \geq t \\ Neg, & \text{otherwise.} \end{cases}$$

In the machine learning literature, there exist a wide variety of binary linear classifiers and related learning algorithms. Some of the most prominent ones include Logistic Regression [Bis07, Ch. 2], linear Support Vector Machines (SVM) [Bis07, Ch. 7], Perceptrons [Bis07, Ch. 2], and Naive Bayes [For19, Ch. 2].

Now, assume for the moment that there are only two actions in our training dataset. In order to obtain an oblique predicate, we could train a linear classifier on the subset of training data at a particular node. The obtained classifier would yield a hyperplane, given by the classifier's weights w, that tries to best separate the dataset into the two classes. This approach has been used successfully with different linear classifiers in some machine learning problems, e.g. [Utg88; LHF03; CE07]. In the verification setting, Ashok et al. [Ash+19a] first explored using linear classifiers in DTs. By using the *Good/Bad* data representation outlined in Section 5.1.4, they ensure that they always have a binary classification problem.

In contrast, with our data representation, we cannot simply train a binary classifier, since we have a multi-class (and not binary) classification problem. As a remedy, [Ash+20b] introduces the following technique based on the classical *one-versus-the-rest* classification known from machine learning [Bis07, Ch. 4]: for every possible label y that appears in the subset of a particular node, train a classifier f_y that tries to separate the points with label y from all other points. Finally, pick the one classifier f_y^* with the lowest impurity.

It is again clear that there is no guarantee that the oblique predicates obtained this way are optimal — especially since with the one-versus-the-rest approach, the training of the linear classifiers only takes one particular label into account. Linear classifiers are thus simply another heuristic that can be used to obtain oblique predicates. In the end, the impurity of those predicates determines how useful they actually are.

With the concept of linear classifiers in the nodes of the DT introduced, the idea of using more powerful, non-linear classifiers might also occur. However, recall that one of our main goals is to create *explainable* controller representations. If we were to use e.g. a neural network inside the DT to partition the dataset, we would lose all of the interpretability of decision trees. The oblique predicates covered in this section strike the balance: they are more powerful than standard axis-aligned predicates, but are in principle still easy to understand.

5.2.2 Categorical features

We now turn our attention to categorical features, support for which has been added to `dtControl` as part of this thesis.

Treating categorical data as numeric

A very simple initial idea to deal with categorical features is to just treat them as numeric by mapping each value to an arbitrary integer and using axis-aligned or oblique

predicates as discussed above. This method has been used previously in verification with some success, e.g. [Ash⁺19a].

However, this approach has several obvious disadvantages: first, the resulting DTs are harder to interpret since relations such as "less than" hardly make sense on categorical features. Second, the arbitrary mapping from categorical values to integers can have an effect on the size of the learned DT. Clearly, we need more sophisticated techniques to deal with categorical features.

Single-comparison predicates

Let λ be a categorical feature with possible (discrete) values $V_\lambda = \{v_{\lambda,1}, \ldots, v_{\lambda,m}\}$. A very simple type of predicate we can construct are *single-comparison* predicates of the form $\lambda = v_{\lambda,i}$, which are for instance used in [HMS66]. Formally, these predicates assign 1 to the data points where $\lambda = v_{\lambda,i}$ and 2 to the data points where $\lambda \neq v_{\lambda,i}$.

Note the similarity of single-comparison predicates to axis-aligned predicates: they both have a finite set of values or thresholds to consider and compare a feature to a value or threshold with an (in-)equality. Using single-comparison or axis-aligned predicates always yields a binary tree.

Multi-comparison predicates

Another common way to partition the data at a node with a categorical feature is to create one branch for each possible value the feature can take [Qui86; Qui93]. Such a *multi-comparison* predicate assigns the integer i to a data point if and only if $\lambda = v_{\lambda,i}$ for that data point. These predicates were used in the first example of a DT in Fig. 3.2, Section 3.3.

In practice, multi-comparison predicates often produce trees that are larger than necessary. For example, consider the DT depicted in Fig. 5.3. Merging the branches with values *red* and *green*, as well as the branches with values *blue* and *yellow* would yield a much smaller tree. In general, we thus want to find multi-comparison predicates that do not assign a single feature value to a branch, but instead allow a *group* of attribute values at every branch.

It is not feasible to simply try all different possible attribute value groupings for every categorical feature λ, since the number of such groupings is exponential in the number of possible values $|V_\lambda|$ [Qui93, Ch. 7]. We instead use a greedy algorithm based on iterative merging of value groups suggested by [Qui93, Ch. 7], which proceeds as follows:

1. Set the initial value groups G_i to $\{v_{\lambda,i}\}$. This value grouping corresponds to the standard multi-comparison predicates as introduced above.
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.

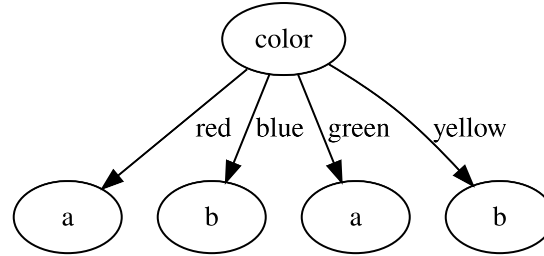


Figure 5.3: Merging branches of nodes with multi-comparison predicates can reduce tree sizes. In this case, the branches with values *red* and *green*, as well as the branches with values *blue* and *yellow* could be merged.

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.

In our experiments we found that the grouping algorithm sometimes did not merge branches where it would actually have made the tree more explainable, because the resulting impurity was marginally higher. We thus introduce a bias parameter in favor of larger value groups: instead of only proceeding to Step 2 if the impurity of the new groups has strictly decreased, we set a tolerance τ of an acceptable increase in impurity with which we still choose the new grouping. Note that, if we set $\tau = \infty$, the algorithm will continue merging branches until only two remain and thus produce binary predicates. The procedure is formalized in Algorithm 2.

Algorithm 2 Attribute value grouping

Require:
The currently considered categorical feature λ, the tolerance τ, a dataset D, and an impurity measure ϕ.

1: **procedure** AVG(λ, τ, D, ϕ)
2: $\quad \mathcal{G}' \leftarrow \mathcal{G} \leftarrow \{G_1, \ldots, G_m\} = \{\{v\} : v \in V_\lambda\}$
3: $\quad$ **while** $\phi(\rho_{\mathcal{G}'}, D) \leq \phi(\rho_{\mathcal{G}}, D) + \tau$ **do** $\qquad \triangleright$ $\rho_{\mathcal{G}}$ is the predicate induced by $\mathcal{G}$
4: $\qquad \mathcal{G} \leftarrow \mathcal{G}'$
5: $\qquad$ **if** $|\mathcal{G}| = 2$ **then**
6: $\qquad\quad$ **return** $\mathcal{G}$
7: $\qquad$ **end if**
8: $\qquad$ **for all** $(G_i, G_j) \in \mathcal{G}^2$ **do**
9: $\qquad\quad \mathcal{G}_{ij} \leftarrow \{G_i \cup G_j\} \cup (\mathcal{G} \setminus \{G_i, G_j\})$
10: $\qquad$ **end for**
11: $\qquad \mathcal{G}' \leftarrow \arg\min_{\mathcal{G}_{ij}} \phi(\rho_{\mathcal{G}_{ij}}, D)$
12: $\quad$ **end while**
13: $\quad$ **return** $\mathcal{G}$
14: **end procedure**

5.3 Impurity measures

The second parameter of the DT learning algorithm we discuss is the impurity measure ϕ, which measures the quality of a predicate with respect to a dataset.

5.3.1 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 [Sha48]. 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 [GBC16, 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-dataset $D \subseteq X \times Y$ has the label $y \in Y$. Let n_y be the frequency of the label y in D, i.e. $n_y = |\{x \in X : D(x) = y\}|$. Then, such an event occurs with probability

$$\frac{n_y}{|D|}$$

and has an information content of

$$-\log_2\left(\frac{n_y}{|D|}\right) \text{ bits.}$$

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-dataset. 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 dataset D and is defined as

$$H(D) = -\sum_{y \in Y} \frac{n_y}{|D|} \log_2\left(\frac{n_y}{|D|}\right).$$

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-dataset. The entropy impurity measure then simply averages the difficulty of classifying the partitions $D_1, \dots, D_m$ created by a predicate ρ. It is thus defined as follows:

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

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 [Qui86]. 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 [Bre+84; Qui86; Qui93].

5.3.2 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 [Qui86]. Quinlan [Qui86] 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

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

This measures the expected information content of the event that a randomly selected data point in D will be assigned the ith 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 [Qui93, Ch. 2].

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

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

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 [Qui93, Ch. 2]. In our setting, overfitting is desirable and there is hence less justification for the entropy ratio.

5.3.3 Gini index

Another common impurity measure used in e.g. the `CART` system is the *gini index* [Bre+84, 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-dataset. This probability is given by

$$G(D) = \sum_{\substack{y,z \in Y \\ y \neq z}} \frac{n_y}{|D|} \frac{n_z}{|D|}$$

and can equally be written as

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

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

$$\text{gini}(\rho, D) = \sum_{i \in [m]} \frac{|D_i|}{|D|} G(D_i).$$

5.3.4 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 [Bre+84, Ch. 4]:

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

Following [Mur+93], the impurity measure we minimize is then simply the reciprocal of the twoing rule.

5.3.5 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. [HKS93; Hea93].

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

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

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

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

5.3.6 Max minority

A very similar impurity measure is *max minority* [HKS93; Hea93], defined as

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

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 $\log_2(|D|)$ [Mur+93]. However, note that it is the *depth* that is logarithmic in this expression — the number of *nodes* is linear in $|D|$. Thus, this theoretical insight is not very useful in practice.

5.3.7 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 [Ash+19a]. The underlying idea is simple: we want to exploit the knowledge that the dataset 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 dataset 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 dataset. In order to estimate how well separable a sub-dataset is by a hyperplane, we can again train a linear classifier on this sub-dataset 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. [Ash+19a] 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 D_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. [Ash+19a] use the *Good*/*Bad* data representation outlined in Section 5.1.4 and hence 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 introduced in Section 5.2.1: 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.

5.4 Determinization

Determinization is a technique that can tremendously decrease the size of DTs for nondeterministic controllers. In this section, we first describe three simple ideas that already yield surprisingly good results. We then develop a theoretical framework for the improvement of impurity estimates in the context of determinization and introduce two techniques based on this framework that enable us to obtain even smaller decision trees.

Up to now, we have treated nondeterministic controllers $C \subseteq S \times A$ very similarly to deterministic controllers. The label powerset method employed so far simply treats every combination of safe actions as a unique label and preserves all possible safe actions for every state. The key idea of determinization is the following: since we know that all actions $C(s) := \{a : (s,a) \in C\}$ for a particular state s are safe, it suffices if the DT represents only a subset thereof. This reduces the amount of information the DT has to encode and consequently also its size.

5.4.1 Determinizing before decision tree learning

The simplest idea for determinization is to directly alter the set of labels Y used in the representation of the controller as a training dataset. For instance, we could randomly pick a single label $a \in C(s)$ for every state $s \in S$ and proceed with the usual DT learning algorithm as in the deterministic case.

A slightly more advanced technique is to determinize in a way that guarantees desirable properties of the controller. For example, to obtain the most energy-efficient controller, one can always pick the action $a \in C(s)$ with minimum norm (assuming the actions are numeric) [Mey+17]. Similarly, the tool Uppaal Stratego allows for the (partial) determinization of controllers to minimize a given cost function [Dav+14; Dav+15]. This determinized controller can then be used as input to the DT learning algorithm.

The disadvantage of this approach is that since we determinize before decision tree learning, we cannot use the information that there are multiple possible actions for a single state in the learning algorithm itself.

5.4.2 Safe pruning

An alternative is to determinize the controller represented by the DT *after* the learning process. To illustrate, consider the DT depicted in Fig. 5.4. The left child of the root node has two children with a common subset of labels. We can thus replace this subtree with a new leaf node that only contains the common subset. If we recursively apply this idea of merging leaf nodes, we obtain the *safe pruning* algorithm [Ash+19b] formalized in Algorithm 3.

Safe pruning clearly preserves the safety properties of the controller due to the fact that nodes are only merged if they share a common subset of safe labels. If all leaf nodes

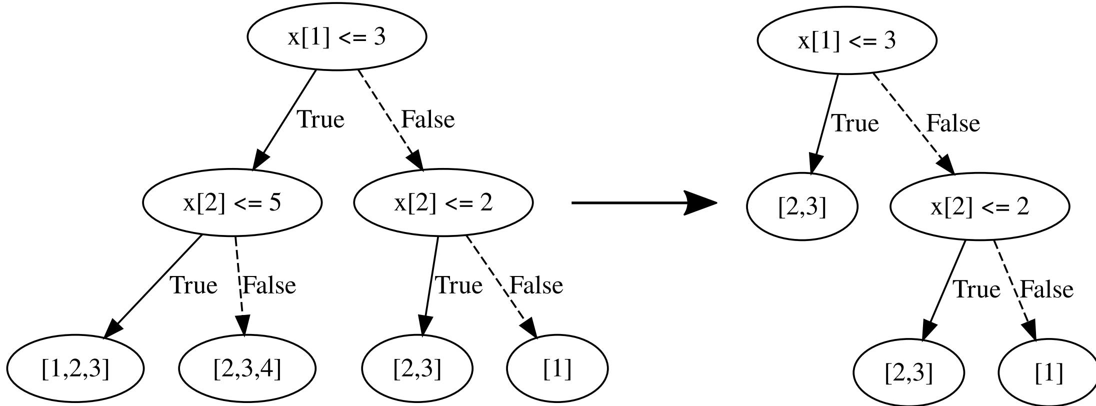


Figure 5.4: The safe pruning algorithm. If multiple children of a node share a common subset of possible labels, the node can be replaced with a leaf node containing only that subset. Concept from [Ash+19b].

Algorithm 3 Safe pruning

Require:
A decision tree T.

1: **procedure** SAFE-PRUNING(T)
2: **if** T is a leaf node **then**
3: **return** T
4: **end if**
5: $(\mathcal{T}, \rho) \leftarrow T$
6: **for all** $T_i \in \mathcal{T}$ **do**
7: $T'_i \leftarrow$ SAFE-PRUNING(T_i)
8: **end for**
9: **if** all T'_i are leaf nodes **then**
10: $y \leftarrow \bigcap_i T'_i$
11: **if** $y \neq \emptyset$ **then**
12: **return** y
13: **end if**
14: **end if**
15: $\mathcal{T}' \leftarrow (T'_1, \ldots, T'_m)$
16: **return** $(\mathcal{T}', \rho)$
17: **end procedure**

reachable from a node n can be assigned the set of labels y, it is safe to replace that node directly with a leaf node containing y.

Ashok et al. [Ash+19b] furthermore introduce a parameter to control the amount of nondeterminism preserved in the pruned tree. In particular, if the parameter p is supplied, only *p rounds* of safe pruning are performed. This means that leaf nodes are only merged at the deepest p levels of the tree.

We briefly want to contrast this *safe* pruning algorithm to the usual pruning performed in machine learning. In that setting, the DT is also pruned by merging and removing branches. However, the decision which branches to prune is made on the basis of some statistical measure with the aim to reduce overfitting [Min89]. In contrast, safe pruning only merges branches if the resulting tree is still guaranteed to overfit, as this is the primary requirement in the context of controller representation.

5.4.3 Early stopping

Instead of first constructing the whole DT and subsequently pruning it, techniques based on *early stopping* aim to directly learn a determinized tree. The general procedure is as follows: during standard DT learning, always check if there is a (maximal) subset $y \subseteq L$ of labels that are safe for all data points before training a node. If there exists such a y, directly return the leaf node y instead of training the node.

We can again introduce a variety of parameters to control the amount of nondeterminism preserved. For example, we can only perform this check for a common subset if there are fewer than k data points at a node or if the depth of the node is greater than d. Early stopping with the former parameter has been introduced in [Ash+19b] under the name of *minimum split size*.

We now show that, if no parameters are specified, early stopping and safe pruning produce equivalent trees:

Theorem 1. *Given a (multi-label) dataset $D \subseteq X \times Y$, a set of predicates Π, and an impurity measure ϕ, standard DT learning with early stopping returns the same decision tree as the safe pruning algorithm applied to the output of standard DT learning, assuming both techniques are used without parameters.*

Proof. First, note that neither early stopping nor safe pruning affect the predicate selection of the DT learning algorithm. Therefore, the inner nodes produced will always be the same. It remains to show that both algorithms return trees with the same leaf nodes.

Recall from Def. 2 that in multi-label classification we have a set L of single-labels and $Y \subseteq 2^L \setminus \{\emptyset\}$. Let us formally define the maximal subset $\sigma(D)$ of safe labels for a dataset D: we have that $\sigma(D) \subseteq L$ such that for every $x \in X$, we have $\sigma(D) \subseteq D(x)$. Furthermore, for all different $\sigma'(D) \subseteq L$ with the same property, $|\sigma(D)| > |\sigma'(D)|$.

An interesting property of $\sigma(D)$ arising directly from its definition is that for any partition $D_1, \ldots, D_m$ of D, we have that $\bigcap_i \sigma(D_i) = \sigma(D)$. This is due to the fact that a label that is safe in D must also be safe in any subset of D.

Early stopping then simply trains a DT as in the standard algorithm, but stops at a node n with sub-dataset $D_n \subseteq X_n \times Y$ if and only if $\sigma(D_n) \neq \emptyset$ and returns $\sigma(D_n)$.

We now prove the following two claims about safe pruning: (1) if $\sigma(D_n) \neq \emptyset$ then Safe-Pruning(n) $= \sigma(D_n)$ and (2) if Safe-Pruning(n) is a leaf, then $\sigma(D_n) \neq \emptyset$. We use structural induction on the DT n:

- **Base case:** n is a leaf. Then, by definition, $n = y$ for some $y \in Y$ (and $y \subseteq L \setminus \{\emptyset\}$) and for all $x \in X_n$, we have that $D(x) = y$. Thus, $\sigma(D_n)$ must be equal to y. Furthermore, we have that Safe-Pruning(n) $= n = y = \sigma(D_n)$. With this, both claims from above hold.
- **Inductive hypothesis:** Suppose n is a DT of the form $n = (\mathcal{T}, \rho)$. Then, for every DT $T_i \in \mathcal{T}$ that is assigned the sub-dataset $D_i \subseteq D_n$ by the predicate ρ, the following claims hold: (1) if $\sigma(D_i) \neq \emptyset$ then Safe-Pruning(T_i) $= \sigma(D_i)$ and (2) if Safe-Pruning(T_i) is a leaf, then $\sigma(D_i) \neq \emptyset$.
- **Inductive step:** $n = (\mathcal{T}, \rho)$.
  (1) Assume $\sigma(D_n) \neq \emptyset$. Then, for every sub-dataset D_i created by ρ, we also have $\sigma(D_i) \neq \emptyset$. Thus, in the recursive calls of safe pruning (lines 6–8), by the inductive hypothesis, all children of n are replaced with the leaf nodes $\sigma(D_i)$. Then, the condition in line 9 will be true. Furthermore, we have that $\bigcap_i \sigma(D_i) = \sigma(D_n)$ since the D_i partition D_n. Line 10 of the algorithm computes exactly this intersection, which is nonempty by assumption, and replaces n by it. Thus, Safe-Pruning(n) $= \sigma(D_n)$.
  (2) Assume Safe-Pruning(n) is a leaf node. This means that all children of n are replaced by leaf nodes in the recursive calls (see line 9). By the inductive hypothesis, we thus have that $\sigma(D_i) \neq \emptyset$ for all D_i created by ρ. Now, again applying the inductive hypothesis, this time with Claim 1, we get that all children were replaced with $\sigma(D_i)$. Thus, the intersection in line 10, which by assumption cannot be empty, is equal to $\bigcap_i \sigma(D_i)$. We again have $\bigcap_i \sigma(D_i) = \sigma(D_n)$ because the D_i form a partition of D_n. Therefore, $\sigma(D_n) \neq \emptyset$. ■

Taking claims 1 and 2 together, we get Claim 3: if Safe-Pruning(n) is a leaf, Safe-Pruning(n) $= \sigma(D_n)$. Thus, if safe pruning replaces a node n, it always replaces it with $\sigma(D_n)$.

Now, let T_{sp} be the tree obtained by safe pruning and T_{es} be the tree obtained by the early stopping algorithm.

Let n be a leaf node in T_{es}. By definition, $n = \sigma(D_n) \neq \emptyset$. Therefore, by Claim 1, the safe pruning algorithm will replace n with $\sigma(D_n)$ when it recursively reaches n. The same leaf node thus also occurs in the same place in T_{sp}.

On the other hand, let n be a leaf node in T_{sp}. If n has been replaced by safe pruning, by claims 2 and 3, we have $n = \sigma(D_n) \neq \emptyset$. If n was already a leaf node in the tree before pruning, the same property trivially holds. Therefore, when the early stopping algorithm reaches the corresponding node, it is immediately replaced with $\sigma(D_n)$. □

The obvious advantage of early stopping over safe pruning is that only the determinized part of the tree has to be built. Safe pruning, on the other hand, first has to construct the whole nondeterministic DT, which is less efficient both runtime- and memory-wise. However, the safe pruning parameter p allows for the exact specification of how many levels of the tree should be pruned, while the parameters of early stopping, such as the number of examples at a node, depend strongly on the specific dataset and are thus harder to choose.

5.4.4 Improving impurity estimates in the context of determinization

The determinization techniques covered so far have not used the information that there are multiple safe actions in a single state to improve the selection of predicates, i.e. the impurity measures. To see why this may be suboptimal, consider a predicate splitting a dataset into the two sub-datasets D_1 and D_2, where every data point in those sub-datasets has a different combination of safe labels. Furthermore, assume that there exist single-labels l_1 (l_2) that can be assigned to every data point in D_1 (D_2). Since after splitting, every data point has a different combination of safe labels, the impurity measures we have used so far will estimate the impurity of this predicate as very high. However, we know that we can determinize the labels in D_1 and D_2 such that both datasets are pure (with the determinization that always determinizes to l_1 respectively l_2). Therefore, the impurity of such a split should actually be 0.

We now develop improvements of some of the impurity measures introduced in Section 5.3 to take determinization into account. The setting in which these impurity measures are meant to be used is the following: we want to reduce tree sizes as much as possible and are willing to accept that the resulting DT might be heavily determinized, as long as it is still safe. To this end, the early stopping algorithm is employed.

Let us first formally define the notion of a determinization of a multi-label dataset:

Definition 6. A *determinization* of a multi-label dataset $D \subseteq X \times Y$, where $Y \subseteq 2^L \setminus \{\emptyset\}$, is a function $\delta\colon X \to 2^L \setminus \{\emptyset\}$, such that for all $x \in X$ we have that $\delta(x) \subseteq D(x)$. δ is called *complete* if and only if $|\delta(x)| = 1$ for all $x \in X$, otherwise it is called *incomplete*.

Now, note that in the case of nondeterminism, an impurity measure is actually not only a function of a predicate and a dataset, but also of a determinization. The determinization specifies which labels are used for every data point to compute the impurity. Up to now, we have always implicitly used the trivial determinization $x \mapsto D(x)$. However, the minimization problem we have in general is the following:

$$\rho^* = \operatorname*{arg\,min}_{\rho \in \Pi} \phi(\rho, \delta^*_\rho, D),$$

where δ^*_ρ denotes the best possible determinization of D for the predicate ρ. We thus want to pick the predicate with the lowest impurity *regardless of the determinization required to achieve that impurity*.

It is clearly not feasible to simply iterate over all possible determinizations for every predicate, since the number of possible determinizations is in $\mathcal{O}(2^{|L|^{|X|}})$. We thus need a heuristic to approximate δ_ρ^*.

First, we show that we can reduce the search space by only considering complete determinizations. In particular, we prove that there always exists a complete determinization with minimal impurity in Proposition 1 and Theorem 2. 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.

Proposition 1. *Given an impurity measure $\phi \in \{\text{ent}, \text{gini}\}$, a predicate ρ, and a dataset 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 image $\text{Im}(\delta)$. Furthermore, let $n_{i,y}$ denote the number of data points that are assigned the (possibly non-deterministic) label $y \in \text{Im}(\delta)$ under δ in the ith 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

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

Since δ is incomplete, there is a label $q \in \text{Im}(\delta)$ 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,

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

It follows that

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

With

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

we obtain:

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

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

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

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

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

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 2. *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 2 shows that it would suffice 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. We thus present two heuristics allowing us to efficiently approximate $\phi(\rho, \delta^*_\rho, D)$ by making use of the fact that the best determinization is complete.

Maximum frequencies

The maximum frequency approach, introduced in [Ash+20b] (albeit without the theoretical justification given above), explicitly approximates the complete determinization δ^*_ρ with lowest impurity. The idea works as follows: consider a dataset D at a particular node and a predicate ρ that partitions D into $D_1, \ldots, D_m$. In order to make the partitions D_i as pure as possible, we greedily assign the label occurring most frequently in D_i to all data points where it is safe. We then recursively continue this process with the remaining data points, until every point is assigned a determinized label.

Formally, let $f_i \colon L \to \mathbb{N}$ be the function that assigns to every single-label the number of states in D_i for which it is safe, i.e. $f_i(l) = |\{x \in X_i : l \in D_i(x)\}|$. Then, assign the safe single-label with maximum frequency in the corresponding sub-dataset to every data point $x \in X$: if x belongs to the ith partition D_i, we have

$$\delta^*_\rho(x) = \left\{ \operatorname*{arg\,max}_{l \in D(x)} f_i(l) \right\}.$$

Note that $\delta^*_\rho(x)$ is a set (with only a single element) to be consistent with Def. 6.

In practice, computing a new determinization δ^*_ρ for every predicate ρ often is computationally infeasible. Instead, one can compute the function f, assigning to every action the number of states in D for which it is safe, and use the resulting determinization δ^* as an approximation for the δ^*_ρ. This approach tries to make D itself as pure as possible, instead of the sub-datasets D_i. It is this approximation that was originally introduced in [Ash+20b].

Multi-label impurity measures

We now introduce a different approach — multi-label impurity measures — that allows us to approximate $\phi(\rho, \delta^*_\rho, D)$ without computing the explicit determinization δ^*_ρ. In principle, any standard impurity measure can be converted into a multi-label impurity measure, but we again focus on the entropy and the gini index.

Let us first examine the multi-label formulation of the entropy impurity measure. Recall that $\text{ent}(\rho, \delta^*_\rho, D)$ averages the entropies of the sub-datasets D_i, defined as

$$H(\delta^*_\rho, D_i) = -\sum_{y \in \text{Im}(\delta^*_\rho)} \frac{n_{i,y}}{|D_i|} \log_2\left(\frac{n_{i,y}}{|D_i|}\right). \tag{5.1}$$

From Theorem 2 we know that it suffices to consider complete determinizations to minimize the entropy, and can thus rewrite Eq. 5.1 to

$$H(\delta^*_\rho, D_i) = -\sum_{l \in L} \frac{n_{i,l}}{|D_i|} \log_2\left(\frac{n_{i,l}}{|D_i|}\right). \tag{5.2}$$

The only values we do not know in Eq. 5.2 are the $n_{i,l}$ — the number of data points in D_i that are assigned single-label l under δ^*_ρ. We can, however, provide the following rough (over-)approximation:

$$\begin{aligned} n_{i,l} &= |\{x \in X_i : \delta^*_\rho(x) = l\}| \\ &\leq |\{x \in X_i : l \in D_i(x)\}| \\ &= f_i(l). \end{aligned} \tag{5.3}$$

While we previously used the values $f_i(l)$ to approximate the best determinization δ^*_ρ, we now use them as an approximation of the $n_{i,l}$ to directly compute the impurity.

To complete the formulation of the multi-label entropy, we use one further insight: if there exists a label $l \in L$ such that $f_i(l) = |D_i|$, we explicitly know the actual best determinization δ^*_ρ: it is simply the determinization that assigns l to every data point $x \in X_i$, producing an entropy of 0.

We thus obtain the following approximations for the entropies:

$$\widehat{H}(D_i) = \begin{cases} 0, & \text{if } \exists l \in L : f_i(l) = |D_i| \\ -\sum_{l \in L} \frac{f_i(l)}{|D_i|} \log_2\left(\frac{f_i(l)}{|D_i|}\right), & \text{otherwise.} \end{cases}$$

The multi-label entropy is then simply the weighted average of the estimated dataset entropies $\widehat{H}(D_i)$.

We proceed with the multi-label formulation of the gini index, which can be derived similarly. Applying Theorem 2, we obtain that

$$G(\delta^*_\rho, D_i) = 1 - \sum_{l \in L} \left(\frac{n_{i,l}}{|D_i|}\right)^2. \tag{5.4}$$

We can again estimate the $n_{i,l}$ with the approximation given in Eq. 5.3. However, we need to be careful since we over-approximate and the value of the sum in Eq. 5.4 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, applying the same insight we had for the entropy yields

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

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

The approximation in Eq. 5.3 is of course very rough and there is no guarantee that it will be close to the actual values of $n_{i,l}$. For instance, imagine a sub-dataset where eight data points can be assigned either label 1 or 2, one data point can only be assigned label 1, and one data point can only be assigned label 2. Obviously, the best determinization assigns either label 1 or label 2 to all eight nondeterministic data points. Thus, the real values (n_1, n_2) would be either $(0.9, 0.1)$ or $(0.1, 0.9)$. In contrast, the approximation given above produces the values $(0.9, 0.9)$ — we inevitably make a relative error of 800% in one of the values.

Despite being based on rough approximations, multi-label impurity measures prove to be very useful in practice. Partly, this can be attributed to the fact that the DT learning process is entirely heuristic anyway, and the approximation errors therefore do not matter as much. Furthermore, because we do not have to construct the actual determinizations δ_ρ^*, multi-label impurity measures are often much more efficient than the maximum frequency approach — especially in comparison to the more precise version that approximates a determinization for every predicate.

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-dataset 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. 5.5 (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.

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

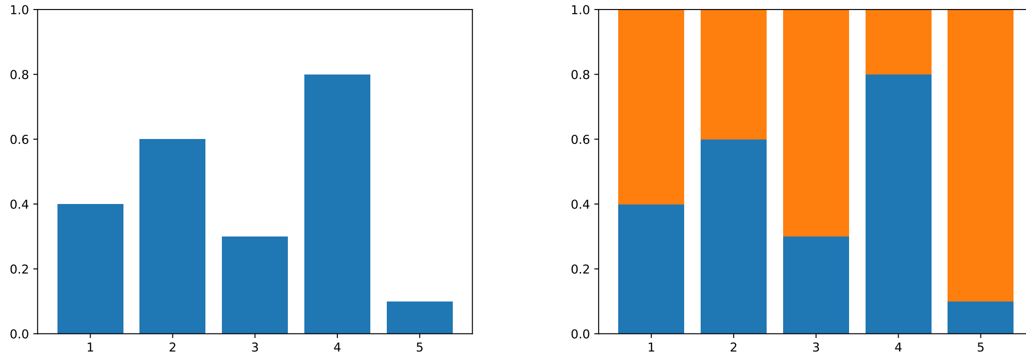


Figure 5.5: Bar chart of the frequency function. The bar chart (left) for a dataset with labels $1, \ldots, 5$ and the error bars indicating the impurity (right).

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. 5.5 (right).

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

$$J(D_i) = \sum_{l \in L} 1 - \frac{f_i(l)}{|D_i|} \tag{5.5}$$

$$= |L| - \sum_{l \in L} \frac{f_i(l)}{|D_i|}. \tag{5.6}$$

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. 5.6 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

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

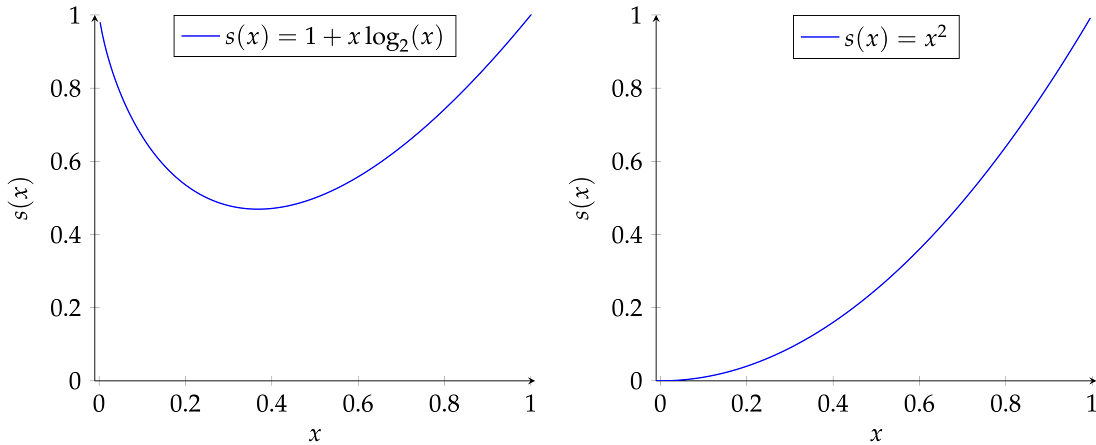


Figure 5.6: Plots of the scaling functions arising out of multi-label entropy (left) and gini index (right).

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. 5.6. 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.

5.4.5 Tradeoff between decision tree size and optimality

To conclude the determinization section, we briefly want to mention the tradeoff between DT size and optimality that is often encountered in practice. By construction, the determinization algorithms introduced preserve the safety guarantees of the controller — this is the essential property that we must never violate in any controller representation. However, there are often additional desirable properties besides safety that we want the controller to satisfy, such as resource-efficiency or the quality of service, as measured by some metric [Ash+19b]. We thus want the controller to not only be *safe*, but furthermore also *optimal* with respect to some optimality criteria.

The determinization algorithms we have covered only guarantee the safety of the determinized controller and throw away much of the nondeterminism that could be used to also increase the optimality. For example, consider the problem of constructing an automatic cruise control system, as introduced in [LMT15]. We want to produce a controller that follows the car in front of it as closely as possible (optimality criterion) while keeping a minimum safe distance (safety property).

Standard verification tools such as Uppaal Tiga [Beh+07] can synthesize a nondeterministic controller satisfying the safety property [Lar+18]. To make it also behave optimally, we have to intelligently choose one of the safe actions for each state, i.e. determinize the controller, so that it drives as close as possible to the car ahead. However, when applying for instance the maximum frequency determinization strategy, we obtain a very small decision tree with only three nodes that simply always decelerates [Ash+20b]. Clearly, this strategy is safe and we have succeeded in training a very small DT, but it performs very badly regarding the optimality criterion.

Ashok et al. [Ash+19b] have investigated various ideas to learn small, yet still almost optimal DTs. For instance, they have introduced the various discussed parameters for the safe pruning and early stopping algorithms that allow keeping some amount of nondeterminism in the DT. We can of course also determinize the controller optimally before DT learning, as discussed in Section 5.4.1, with the caveat that the resulting DT will likely be much larger than if a more sophisticated determinization technique is applied. Other approaches to retain optimality and make it an integral part of the tree building process, while keeping tree sizes as small as possible, remain an active area of further research.

6 Implementation

As described in Chapter 4, the algorithms and techniques presented in this thesis have been implemented in the latest version of the tool `dtControl` [Ash+20b]. While we previously reviewed the tool from a user's perspective, we now have enough theoretical background to examine it from a developer's point of view. We start with a brief discussion of the primary design goals that guided the entire development process. Afterwards, we give a high-level overview of the software architecture from the system design perspective, and then describe the details of the individual subsystems and the object model. Readers who are interested in the concrete implementation are encouraged to read the extensive developer manual[1] or directly explore the source code[2].

The general high-level structure of `dtControl` as a number of distinct interacting subsystems had already been implemented in the initial version of the tool. As part of this thesis, the DT learning component has been thoroughly redesigned to allow for much more flexibility and the integration of a variety of new algorithms. Furthermore, we added support for the probabilistic model checker PRISM [KNP11] to `dtControl`.

6.1 Design goals

The ability to deal with frequent changes in the requirements is one of the cornerstones of modern software engineering. While the requirements of academic tools are usually less prone to change than those of commercial software, it is equally important to keep the software architecture *flexible*: often, during the lifetime of an academic product a variety of new discoveries are made which will need to be integrated with the existing code. Furthermore, we want to be able to quickly try out new ideas and see whether they are worthwhile to pursue further from a theoretical perspective.

To make the tool useful to the research community as a whole, it is also essential that it is easily *extensible*. Other people might want to tightly integrate it into their existing workflows, for example by supporting new input and output formats.

Finally, any tool dealing with a large amount of data — in our case controllers with a size of several hundreds of megabytes — should be *efficient*. The smallest and most explainable DT is worth nothing if it cannot be found in a reasonable amount of time.

[1] Available at the official website of `dtControl`: `https://dtcontrol.model.in.tum.de/`

[2] Available at `https://gitlab.lrz.de/i7/dtcontrol`

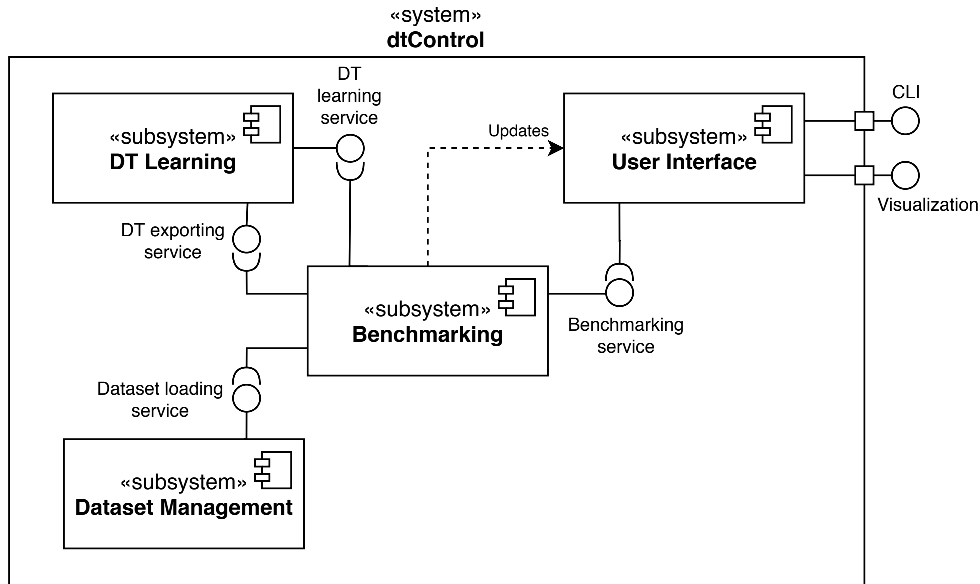


Figure 6.1: Component diagram showing the subsystem decomposition of `dtControl`.

6.2 Overview of the software architecture

The functionality of `dtControl` is distributed among four distinct subsystems, whose high-level functionality and interaction we briefly describe. The subsystem decomposition is illustrated in the the Unified Modeling Language (UML) [RJB04] component diagram in Fig. 6.1.

The heart of the tool is formed by the *DT Learning* subsystem, which is responsible for the actual decision tree learning algorithms and representation. It provides a great deal of different options for training DTs (as covered in Chapter 5) and can export the learned trees in the DOT and C format.

The decision tree learning itself should undoubtedly be independent of any verification tool that produces the synthesized controllers. Therefore, all of the *Dataset Management* has been extracted into a separate subsystem, which mainly provides functionality for loading and converting controllers from many different sources such as `SCOTS` [RZ16], UPPAAL STRATEGO [Dav+15], and PRISM [KNP11].

The *Benchmarking* component is responsible for running a set of different given DT learning configurations on a number of specified controllers. It thus uses both the functionality provided by the Dataset Management and the DT Learning subsystems.

Finally, the *User Interface* serves as the main entry point for user interaction with the tool, in the form of a CLI. Furthermore, once decision trees have been constructed, `dtControl` reports several statistics and gives the DOT and C output in a HTML file that can be inspected by the user.

As shown in Fig. 6.1, the subsystems communicate solely over a few, well-defined interfaces. This ensures that *coupling* between subsystems is reduced as much as

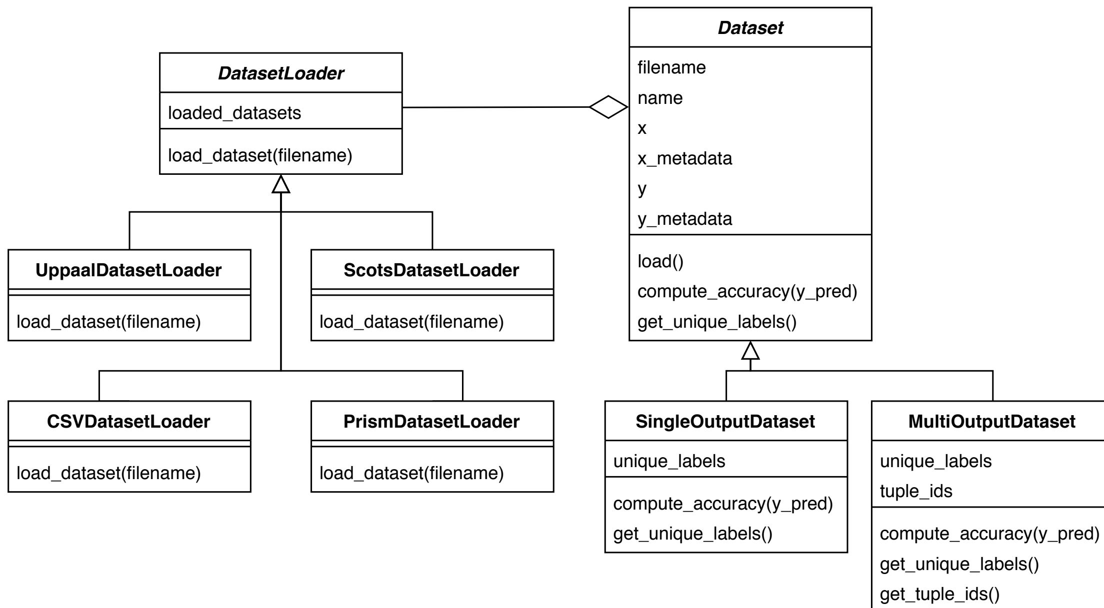


Figure 6.2: Class diagram of the dataset management subsystem.

possible, which means that changes to one subsystem generally do not affect the other components [SMC74]. For example, we could easily add support for new verification tools to the dataset management subsystem without making any modifications to the other components. Similarly, we could easily add a different user interface such as a desktop GUI without touching any of the core entities and algorithms.

6.3 Object model

With the high-level overview in mind, we now examine the individual subsystems of `dtControl` in detail.

6.3.1 Dataset management

The core of the dataset management subsystem is the abstract `DatasetLoader` class, which provides the abstract `load_dataset(filename)` method that loads a controller from a file and returns its states, actions, and relevant metadata. For efficiency reasons, we chose `NumPy` [Oli06] arrays as the primary data structure for this information. Furthermore, the `DatasetLoader` keeps track of which datasets have already been loaded in the dictionary `loaded_datasets`, and stores converted controllers in `dtControl`'s internal representation in the file system, which decreases future loading times.

The concrete instantiations of this class are then the dataset loaders responsible for a specific verification tool, such as the `ScotsDatasetLoader`. Their implementations vary

widely depending on the file format they need to parse. In order to add support for a new verification tool, one thus simply has to provide an instantiation of a `DatasetLoader` that satisfies the required interface.

The abstract `Dataset` class is a wrapper for the data returned by a dataset loader. It also provides some methods convenient for DT learning, such as a procedure to convert nondeterministic labels to a unique label representation with the label powerset method. *Caching* of the result ensures that such a representation is only computed once for every dataset and thus decreases runtime.

A simple implementation of the abstract methods in the `Dataset` class is given in the `SingleOutputDataset`, which represents controllers with a single output variable. To keep `dtControl` flexible and allow for potential future algorithms operating on multi-output controllers, these are represented separately with the `MultiOutputDataset` class. Its most important method is `get_tuple_ids()`, which converts the multi-output actions to single tuples that can be used equivalently to `SingleOutputDatasets`.

An overview of the different classes that make up the dataset management component is given in the UML class diagram in Fig. 6.2.

6.3.2 Decision tree learning

The central choice that has to be made for the DT learning subsystem is how all of the different possible configurations of the learning algorithm can be represented in a clean way. For instance, the first version of `dtControl` used an inheritance-based approach: there was one central decision tree class that implemented the main algorithm. To alter the parameters, one had to extend this class and override the required functionality; e.g. there was a separate class for DT learning with linear classifiers and a separate class that implemented the maximum frequency determinization technique. The problem with this approach is its inflexibility: what if we want a DT that both uses linear classifiers as well as the maximum frequency technique? We would again have to create a new class that extends both corresponding DT classes and carefully ensure that it uses the correct functionality from the respective superclass. This leads not only to an explosion in the number of classes, but also to possible code duplication to circumvent the problems of multiple inheritance.

For the newest version of `dtControl`, we instead chose to follow the well-known principle of composition over inheritance [Gam+95] and implemented a composition-based approach that allows for much more flexibility and extensibility. The resulting design is shown in the UML class diagram in Fig. 6.3, which omits many concrete implementations of interfaces to avoid unnecessary complexity.

The heart of the DT learning subsystem is the `DecisionTree` class, which implements the interface required to be used with the benchmarking component. It provides functions such as `fit(dataset)`, which trains the DT on a dataset, `predict(dataset)`, which returns the labels the DT predicts for a dataset, `get_stats()`, providing several statistics such as the number of nodes in the tree, as well as `print_dot()` and `print_c()`, which return the DT in a DOT or C representation, respectively. Most of these methods

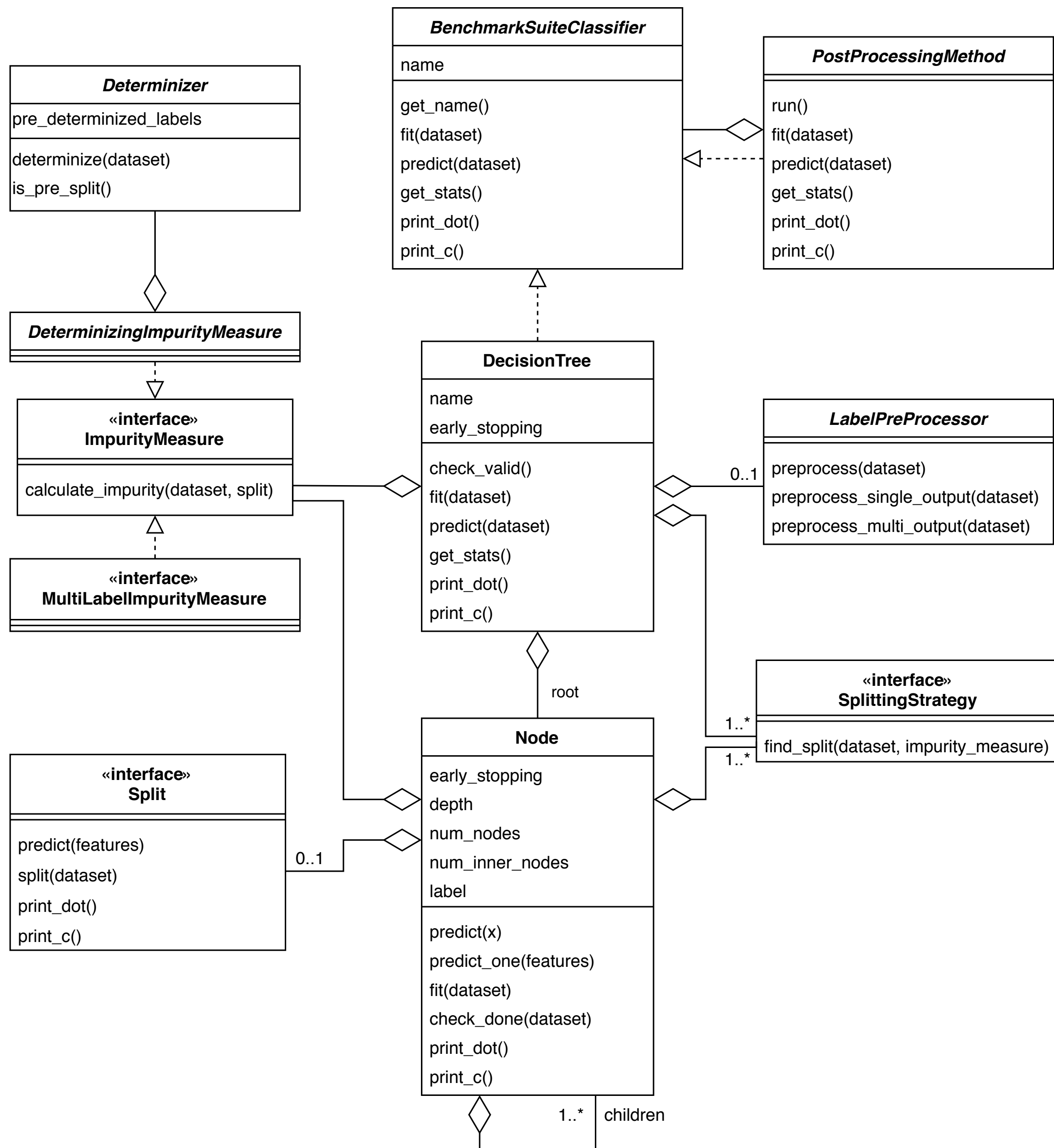


Figure 6.3: Class diagram of the decision tree learning subsystem.

simply delegate the call to the tree's root node, an instance of the Node class that represents the actual DT data structure. It has mostly the same attributes as the DecisionTree class, as well as certain statistics and either a list of child nodes (if it is an inner node) or a label (if it is a leaf node).

The considered predicates in the DT learning algorithm are provided by several splitting strategies of type SplittingStrategy, implementations of which include the AxisAlignedSplittingStrategy and the LinearClassifierSplittingStrategy. The predicates they return are of type Split and provide methods to determine the child node an example belongs to and to partition a dataset into the sub-datasets that can be used to continue the learning process.

A DecisionTree furthermore has an instance of an ImpurityMeasure, which can compute the impurity for a given dataset and predicate. There exist both multi-label impurity measures, such as the multi-label entropy, as well as determinizing impurity measures, such as standard entropy. Determinizing impurity measures furthermore have a Determinizer that determines with which labels the impurity is computed. This can for example be the LabelPowersetDeterminizer, if all nondeterminism should be preserved, or the MaxFreqDeterminizer, which can be used in conjunction with early stopping.

If determinization before DT learning is wanted, an optional LabelPreProcessor can be supplied to the decision tree. This interface, which is for example implemented by the NormPreProcessor, allows for the determinization of a dataset by returning a new dataset with modified labels.

Finally, a PostProcessingMethod both has a BenchmarkSuiteClassifier and satisfies this interface itself. Its most important method is run(), which runs the post-processing technique on the classifier. This interface is currently only implemented by the SafePruning class, which transforms a trained DecisionTree using the safe pruning algorithm.

This composition-based software architecture is clearly much more flexible than an approach based on inheritance. For instance, we can now combine splitting strategies and determinization approaches arbitrarily, simply by setting the corresponding attributes in the DecisionTree class. It also easily extensible: if we want to add a new impurity measure or splitting strategy, we just have to provide a class that implements the respective interface.

6.3.3 Benchmarking

The benchmarking subsystem is relatively simple. Its core class is the BenchmarkSuite, which has a variety of configuration attributes such as the file the benchmark is saved to, the folder the resulting DTs are saved to, and an optional timeout parameter. Furthermore, the class provides methods to add datasets to a benchmark and then execute a benchmark with given classifiers on these datasets. The classifiers need not necessarily be DecisionTrees, they just have to satisfy the BenchmarkSuiteClassifier interface.

The results of a benchmark are saved in an internal JSON format, which is processed by the `TableController` to update the HTML file that is part of the user interface.

6.3.4 User Interface

Finally, the user interface subsystem consists mainly of the CLI, which is a script that parses user input and runs the corresponding benchmarks using the benchmarking component. Furthermore, the HTML, CSS, and JavaScript files for the benchmark output belong to the user interface.

7 Evaluation

In this chapter, we examine how well the DT learning techniques we have covered perform in practice. We first give a general overview of our findings on a variety of case studies on controllers obtained from either the probabilistic model checker PRISM [KNP11] or the controller synthesis tools SCOTS [RZ16] or UPPAAL STRATEGO [Dav+15]. We split our discussion into two sections due to significant differences between these case studies: the former controllers arise out of standard model checking of MDPs, are deterministic, and include many categorical variables, while the latter arise out of controller synthesis for cyber-physical systems (CPS), are often nondeterministic, and only consist of numeric variables. Afterwards, we present a more detailed comparison of our results with different choices for the parameters of the decision tree learning algorithm. Some of our results for CPS have previously been reported in [Ash+20b]. All experiments were conducted on a machine with an Intel Xeon W-2123 processor with a clock speed of 3.60GHz and 64 GB of RAM.

7.1 Overall results

7.1.1 Model checking of Markov decision processes

We give a brief description of the case studies on which we ran our methods and subsequently present our results in comparison to other controller representation techniques.

Case studies

We evaluated our algorithms on the following case studies, most of which are available in the PRISM benchmark suite [KNP12]:

- **CSMA/CD communication protocol.** The CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol can be employed in Ethernet technology for media access control. We investigate the probabilistic model introduced by [Kwi+04].
- **FireWire root contention protocol.** The FireWire root contention protocol selects a leader as part of the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus, which is used for the transportation of video and audio signals in multimedia networks. A PRISM model of the protocol is given in [KNS03].
- **Leader.** The problem of choosing a leader in a ring of processors, as introduced in [IR90].

- **Mars exploration rovers (mer).** A probabilistic model of resource sharing among threads in mars rovers [Fen14].
- **WLAN.** The WLAN (Wireless Local Area Network) protocol uses a variant of the CSMA/CA (Carrier Sense, Multiple Access with Collision Avoidance) scheme for media access control. We examine the probabilistic model of [KNS02].
- **Zeroconf.** A randomized protocol for the dynamic configuration of IPv4 addresses, a PRISM model of which is given in [Kwi+06].

We used PRISM to construct an almost-sure winning strategy for the MDPs given by the case studies introduced above. This winning strategy is a deterministic controller where the state variables corresponds to the variables in the implicit PRISM model description and the actions consist of both a module and an action performed by that module [Ash+19a]. The controllers we obtain are all deterministic and mostly dominated by categorical variables, such as protocol states, but usually also have a few numeric variables, as for example clocks.

Results

Table 7.1 lists the number of nodes in the DTs constructed with our most performant and explainable method — attribute value grouping with varying tolerance settings and entropy as impurity measure — on the case studies introduced above. A comparison of our other methods is provided in Section 7.2.1. We compare our approach to the results obtained with other controller representation techniques, namely BDDs and the DTs with linear classifiers from [Ash+19a], which use the *Good/Bad* data representation from Section 5.1.4 and treat categorical data as numeric.

The BDDs were constructed with the Python library `dd`[1] and minimized by calling reordering heuristics until convergence. The numbers we report from [Ash+19a] are the overall best results they have achieved on the case studies. In order to provide a fair comparison, we converted the number of inner nodes in the DTs that [Ash+19a] report to the number of total nodes[2]. The results of our own method can be found in the *AVG* column of the table.

First, we notice that our decision tree learning algorithm is a very effective technique to concisely represent controllers arising out of PRISM model checking. Compared to a naive lookup table representation, our algorithm results in a size reduction of more than 96% on all of our case studies. On the `csma` and `firewire` examples, we obtain trees with only a double-digit number of nodes.

Comparing to the previous decision tree learning approach of Ashok et al. [Ash+19a], we find that attribute value grouping produces smaller trees by a factor of about $\frac{1}{2}$ on all but one dataset. Therefore, it is apparent that our different data representation in combination with techniques specific to categorical features has a positive impact on the size of the learned decision trees.

[1] `https://pypi.org/project/dd/`

[2] Since their trees are binary, the number of total nodes is two times the number of inner nodes plus one.

Table 7.1: Results on PRISM case studies. We report the number of states (i.e. entries in the lookup table), the number of nodes with a BDD representation, the number of nodes obtained with attribute value grouping with varying tolerance values and entropy, and the best results from [Ash+19a].

Case study	|States|	BDD	AVG	[Ash+19a]
csma2_4	7,958	1,236	**41**	83
firewire_abst	845	52	**9**	17
firewire_impl	6,953	1,721	**71**	145
leader4	3,168	1,240	119	**91**
mer30	67,092	874	**115**	253
wlan2	24,514	1,220	**201**	413
zeroconf	29,814	1,893	**367**	661

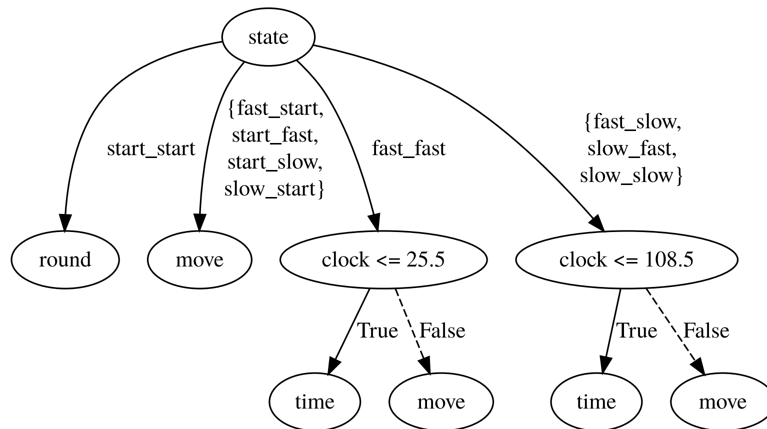


Figure 7.1: Decision tree learned for the `firewire_abst` example.

BDDs are also a more efficient controller representation than naive lookup tables. However, the DT-based representations always perform much better than BDDs and additionally have the potential to be explainable, since they do not operate on a binary level.

To illustrate this last point, consider the `firewire_abst` example. Applying our algorithm to the rather incomprehensible list of 845 state-action pairs produced by PRISM yields the small and easily understandable decision tree depicted in Fig. 7.1. In this representation, the semantics of the controller are immediately obvious: for many protocol states it simply always picks the *move* action, for others, it picks either *move* or *time* depending on the *clock* variable, and *round* is only chosen if the protocol state is *start_start*.

7.1.2 Controller synthesis for cyber-physical systems

Case studies

We evaluated our methods on ten different case studies arising out of controller synthesis for CPS with the tools `SCOTS` or Uppaal Stratego:

- **Cartpole.** This is a standard example in CPS, in which a cart must balance an inverted pendulum. We consider the model given in [Jag+18].
- **Temperature control.** The problem of regulating the temperature in connected rooms using the available heaters. We investigate a model with two rooms and one heater [Gir12] and a model with ten rooms and two heaters [JZ17].
- **Flight dynamics.** We consider models describing the flight dynamics of a 3-DOF tandem rotor helicopter [Jag+18] and an aircraft landing maneuver [RWR15].
- **Cruise control.** The problem of following a car in front as closely as possible. We examine the model introduced in Section 5.4.5 [LMT15] and an example involving a truck and a trailer [KZ19].
- **DC-DC boost converter.** The controller for a DC-DC boost converter from [RZ16].
- **Vehicle.** The path planning problem for the bicycle dynamics of a vehicle introduced in [RZ16] and discussed in Chapter 1.
- **Traffic planning.** A model of a road traffic planning system [SZ19].

The synthesized controllers for all ten case studies consist only of numeric variables. For all but the `aircraft` and `vehicle` datasets, the verification tools yield nondeterministic controllers. Half of the examples have multiple output variables, while the other half only has a single output variable, as indicated in the table below.

Results

Table 7.2 shows the overall best results we have achieved on the CPS case studies. We distinguish between the nondeterminism-preserving case, in which the smallest trees were obtained with different techniques for choosing oblique predicates and entropy as impurity measure, and the deterministic case, where multi-label entropy with early stopping *always* performed the best; sometimes with and sometimes without oblique predicates.

We compare our algorithm against BDDs, which were either again constructed with the `dd` Python library and minimized using reordering heuristics until convergence, or directly obtained from `SCOTS`. The deterministic BDDs were generated from a controller pre-determinized with minimum norm. Due to the different tools and reordering heuristics that were available, for `cruise` and `truck_trailer` the deterministic BDDs were actually worse than the nondeterministic BDDs; to provide a fair comparison, we report the lower number.

Table 7.2: Results on SCOTS and Uppaal Stratego case studies. We report the overall best results for our nondeterminism-preserving and determinizing methods in comparison to BDDs. "∞" indicates failure to produce a result within three hours; "n/a" denotes that the approach is not applicable, since the controller is already deterministic.

Case study	|States|	Nondeterministic		Deterministic	
		BDD	DT	BDD	DT
Single-output					
cartpole	271	363	**183**	169	**7**
tworooms	40,311	269	**15**	158	**7**
helicopter	280,539	**1,753**	3,753	1,572	**123**
cruise	295,615	1,815	**747**	1,815	**3**
dcdc	593,089	∞	**139**	440	**5**
Multi-output					
tenrooms	26,244	328	**133**	130	**7**
truck_trailer	1,386,211	**10,574**	338,389	**10,574**	20,781
traffic	16,639,662	∞	**8,953**	∞	**97**
vehicle	48,018	**4,087**	9,771	n/a	n/a
aircraft	2,135,056	**177,332**	815,045	n/a	n/a

It is again apparent that DTs are an effective data structure to represent controllers concisely. In the nondeterminism-preserving case, the number of nodes in the DT is usually comparable to the number of nodes in the BDD; sometimes higher, sometimes lower. However, we were always able to compute a DT for every dataset, while BDDs timed out on `dcdc` and `traffic`. Note again that a small BDD is desirable from an efficiency perspective, but — in contrast to a small DT — does not aid with understanding the controller.

The effectiveness of DT learning is greatly amplified when we allow determinization: in this case, our algorithms manage to learn DTs with fewer than ten nodes on more than half of our case studies, in which we initially started with a lookup table of up to almost 600,000 entries. These trees are so small that they can easily be drawn on a single sheet of paper! For `helicopter` and `traffic` the DT is slightly larger with about 100 nodes — which is still a size reduction of more than 99.9% from the original controller. The only example where DTs perform not quite as well is `truck_trailer`, where we can nevertheless reduce the size of the naive lookup table by more than 90%. On all but this last dataset, determinized DTs are also far more concise representations than determinized BDDs.

We want to exemplify the explainability of the DT representation and the resulting benefits with the `tenrooms` example, which has previously been discussed in [Ash+20b].

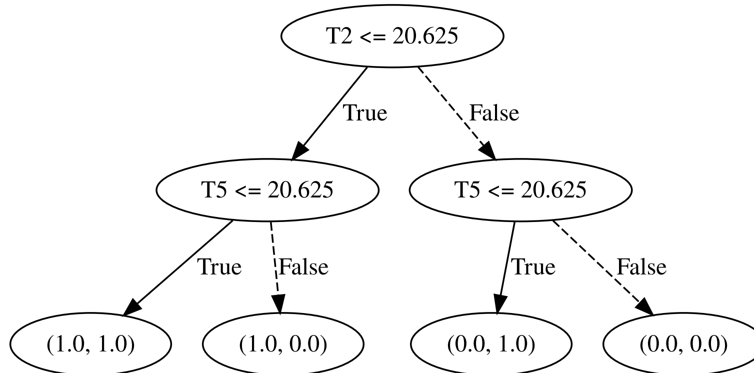


Figure 7.2: The decision tree learned with multi-label entropy and axis-aligned predicates on `tenrooms`.

The original controller for `tenrooms` output by `SCOTS` is a lookup table with 26,244 states and multiple potential actions per state. This is a large, inefficient, and obscure representation that makes it impossible to understand what the controller is actually doing.

In contrast, applying DT learning with multi-label entropy and axis-aligned splits, we obtain the extremely small decision tree with just seven nodes depicted in Fig. 7.2. Clearly, this representation is much more succinct and efficient: instead of having to store thousands of lookup table entries in memory, we now only have to store a small chain of if-then-else statements corresponding to the tree structure.

Furthermore, the concise DT representation allows us to actually understand the internal workings of (a determinized version of) the controller. We see that it simply compares the temperature in rooms two and five — the rooms with the heaters — against a threshold; if the temperature is below that threshold, it turns the respective heater on, if it is above the threshold, it turns the heater off. This immediately makes intuitive sense, and we have thus gained confidence in the correctness of the model and the controller, which goes beyond merely relying only on the output of some hard-to-understand verification process.

Additionally, the DT also reveals that temperature measurements are only needed in rooms two and five — we can therefore improve upon the implementation of the controller and only install temperature sensors in those two rooms.

We thus have come from a large, inefficient, and obscure controller representation as a lookup table to a small, memory-efficient, and explainable decision tree representation that helps us understand the controller and reduce its deployment cost by improving upon its implementation.

Table 7.3: Number of nodes obtained with different predicates on PRISM controllers.

Case study	Single	Multi	LogReg	OC1	AVG
csma2_4	**41**	51	45	63	48
firewire_abst	15	18	18	18	**9**
firewire_impl	79	94	95	93	**77**
leader4	**121**	174	174	174	154
mer30	**115**	202	202	202	158
wlan2	265	272	275	265	**222**
zeroconf	379	370	341	**285**	367

7.2 Detailed comparison of decision tree learning algorithms

7.2.1 Model checking of Markov decision processes

Since the PRISM controllers we work with are all deterministic, there are only two main parameters of the decision tree learning algorithm: the considered predicates and the impurity measure.

Predicates. We first set entropy as the impurity measure for our initial experiments and compare the following algorithms: single-comparison predicates in combination with axis-aligned predicates (*Single*), multi-comparison predicates with axis-aligned predicates (*Multi*), multi-comparison predicates with either logistic regression (*LogReg*) or OC1 (*OC1*), and attribute value grouping with tolerance 0 in combination with axis-aligned predicates (*AVG*). The effect of other tolerance values on attribute value grouping will be discussed shortly. The number of nodes in the resulting DTs is listed in Table 7.3.

We see that the different DT algorithms all perform roughly equally well. In many cases, the simple single-comparison predicates produce the trees with the smallest number of nodes. However, we find that this number is slightly misleading since multi-comparison predicates inherently produce more new nodes with a single split than single-comparison predicates; inspecting the trees shows that the multi-comparison approach often leads to more explainable trees.

Oblique predicates can sometimes further reduce tree sizes. However, on the PRISM controllers the effects are rather weak, which can partly be attributed to the fact that categorical variables dominate in the datasets. Recall that we ignore categorical variables in oblique splits for explainability reasons.

As expected, attribute value grouping always performs strictly better than standard multi-comparison predicates. The resulting decision trees are often the easiest to interpret. To illustrate, compare the trees for `firewire_abst` with single-comparison predicates and standard multi-comparison predicates in Fig. 7.3 to the tree obtained

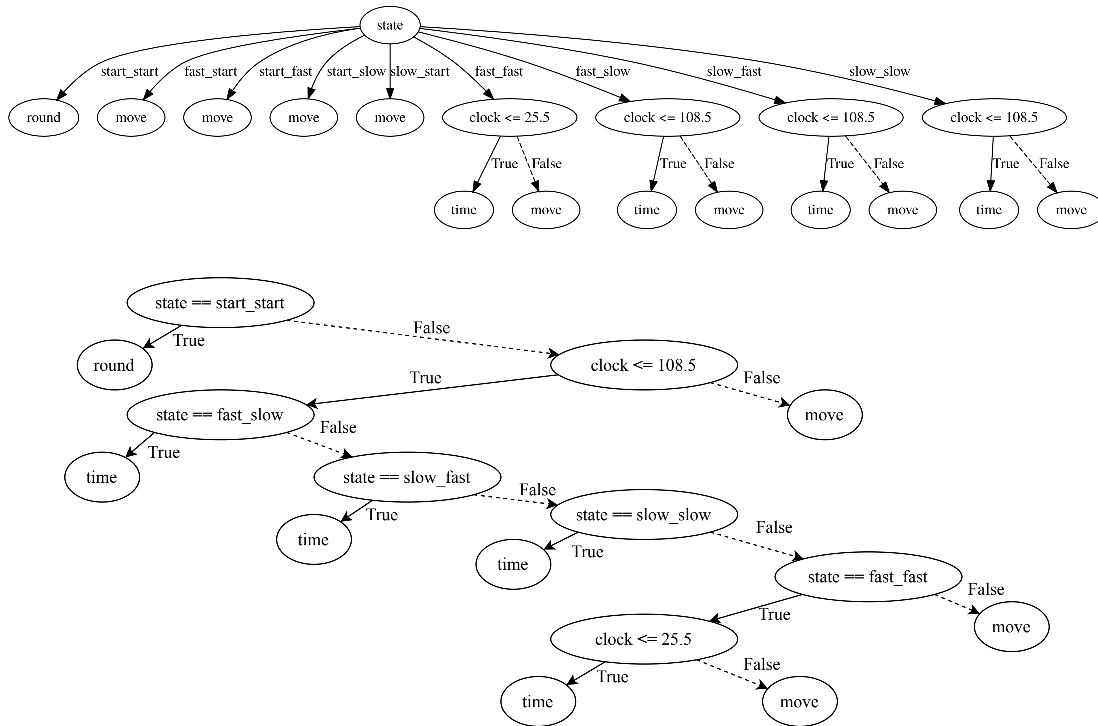


Figure 7.3: Decision trees with multi-comparison (top) and single-comparison (bottom) predicates for `firewire_abst`.

with attribute value grouping depicted in Fig. 7.1. Clearly, the latter is much more concise and easier to grasp.

The effects of different tolerance values τ on the attribute value grouping algorithm are shown in Table 7.4. We see that a value of $\tau = 10^{-5}$, which is merely used to handle floating point inaccuracies in the impurity calculation, only produces a different tree for the `mer30` example. A higher tolerance value can help to build smaller trees, but sometimes also increases tree size. Interestingly, the case where $\tau = \infty$, which guarantees that branches are merged until the resulting predicate is binary, often leads to the fewest number of nodes.

Impurity measures. The second parameter we need to consider is the impurity measure used to determine the quality of predicates. Table 7.5 gives the results of attribute value grouping in combination with different impurity measures on the case studies. We chose to provide the numbers for attribute value grouping since, as outlined above, it appears to us as the most sensible default predicate mechanism; the outcome of the same benchmark with other predicates is comparable.

Table 7.5 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

Table 7.4: Effects of different tolerance values on the performance of attribute value grouping on PRISM case studies.

Case study	$\tau = 0$	$\tau = 10^{-5}$	$\tau = 0.2$	$\tau = \infty$
csma2_4	48	48	**41**	**41**
firewire_abst	**9**	**9**	10	11
firewire_impl	77	77	**71**	**71**
leader4	154	154	120	**119**
mer30	158	149	120	**115**
wlan2	222	222	232	**201**
zeroconf	**367**	**367**	379	379

Table 7.5: Number of nodes obtained with different impurity measures and attribute value grouping on PRISM models.

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

standard entropy. As discussed in Section 5.3.2, 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, 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 5.3.7. 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`.

Table 7.6: 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

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 in general.

7.2.2 Controller synthesis for cyber-physical systems

Impurity measures. Similar to before, 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 7.6.

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 7.7: Number of nodes in the DTs produced by different predicates with entropy on controller synthesis case studies when retaining nondeterminism. "∞" indicates failure to produce a result within three hours.

Case study	Axis	LinSVM	LogReg	OC1
Single-output				
cartpole	253	251	199	**183**
tworooms	27	27	**15**	23
helicopter	6,347	5,789	**3,753**	∞
cruise	987	1,085	783	**747**
dcdc	271	279	**139**	179
Multi-output				
tenrooms	17,297	**133**	147	4,525
truck_trailer	**338,389**	∞	∞	∞
traffic	12,573	∞	**8,953**	∞
vehicle	13,237	13,183	10,389	**9,771**
aircraft	913,857	∞	**815,045**	∞

Now that we have established which impurity measure to use, we can examine the effect of oblique predicates and determinization strategies.

Preserving nondeterminism. In the case of not determinizing the controllers, we need to compare the performance of our different approaches to generate oblique predicates. We thus list the number of nodes obtained with the following algorithms in Table 7.7: standard axis-aligned predicates (*Axis*), linear SVMs (*LinSVM*), logistic regression (*LogReg*), and OC1 (*OC1*).

Clearly, oblique predicates are an effective way to reduce tree sizes on controllers dominated by numeric variables: DTs with oblique splits perform strictly better than those with only axis-aligned splits. However, the resulting reduction in the number of nodes varies from dataset to dataset. For instance, on `tenrooms`, oblique predicates can achieve a reduction of about 99%, on `tworooms` and `dcdc`, tree sizes are roughly halved, and the effect on `cruise` is much less pronounced.

There is no single strategy for generating oblique predicates that is superior on every example, although logistic regression and OC1 often perform comparatively well. We also notice that computing oblique predicates can be rather demanding, especially in the case of OC1 and linear SVMs.

Determinizing. We first compare the different determinization techniques with only axis-aligned predicates. Table 7.8 reports the number of nodes when determinizing with minimum norm before DT learning (*MinNorm*) and when using safe pruning / early

stopping (*ES*). We also include the effects of combining early stopping with maximum frequencies, either with the approximation introduced in Section 5.4.4 (*MaxFreq*) or without (*MaxFreq-exact*), and with multi-label entropy (*Multi-label*).

While all determinization techniques greatly reduce tree sizes, simply determinizing before DT learning is clearly not as effective as the other approaches. Once we integrate determinization into the actual DT learning algorithm with early stopping, we can produce a DT with fewer than 20 nodes on more than half of the datasets. Taking determinization into account when computing impurities further decreases tree sizes, as shown in the last three columns. As expected, the exact computation of maximum frequencies performs better than the approximation, but is more computationally expensive and cannot always produce a result within three hours. Multi-label entropy is clearly the overall best determinization approach as it is *always* better than or equally good as the other techniques.

The results of combining determinization with oblique predicates generated by logistic regression are listed in Table 7.9. The effects are overall less pronounced than in the nondeterminism-preserving case; probably because many of the DTs were already very small without oblique predicates. On the larger `truck_trailer` and `traffic` datasets, allowing linear splits did improve our results quite a bit — at least for maximum frequencies and multi-label entropy — although oblique predicates can also have a negative effect in some cases.

Table 7.8: Effects of different determinization methods with axis-aligned predicates and entropy on tree sizes in controller synthesis case studies. "∞" indicates failure to produce a result within three hours.

Case study	MinNorm	ES	MaxFreq	MaxFreq-exact	Multi-label
Single-output					
cartpole	111	11	11	9	**7**
tworooms	15	**7**	9	**7**	**7**
helicopter	1,353	447	229	185	**123**
cruise	563	**3**	**3**	**3**	**3**
dcdc	21	19	9	∞	**5**
Multi-output					
tenrooms	5,407	**7**	**7**	**7**	**7**
truck_trailer	190,833	49,431	43,195	∞	**31,499**
traffic	1,379	327	195	∞	**151**

Table 7.9: Effects of different determinization methods with logistic regression and entropy on tree sizes in controller synthesis case studies. "∞" indicates failure to produce a result within three hours.

Case study	MinNorm + LogReg	ES + LogReg	MaxFreq + LogReg	MaxFreq-exact + LogReg	Multi-label + LogReg
Single-output					
cartpole	77	**7**	13	**7**	**7**
tworooms	9	9	**7**	**7**	**7**
helicopter	1,051	437	267	207	**127**
cruise	393	**3**	**3**	**3**	**3**
dcdc	21	19	9	∞	**7**
Multi-output					
tenrooms	55	**7**	19	**7**	**7**
truck_trailer	61,775	∞	25,221	∞	**20,781**
traffic	∞	1,391	159	∞	**97**

8 Future Work

We now briefly examine possible directions for future research. We first consider how the decision tree learning algorithm itself might be improved, and then suggest potential additions to the tool `dtControl` that could make working with decision trees in practice much easier.

8.1 Improving decision tree learning

Globally optimal trees. All DT learning algorithms considered in this thesis are variants of the standard top-down, recursive algorithm used in e.g. `CART` [Bre+84], `ID3` [Qui86], and `C4.5` [Qui93]. A major drawback of this approach is its greedy search strategy, which immediately picks the predicate with lowest impurity at every node. It is easy to see that this procedure will often get stuck in local minima of the search space and thus result in globally sub-optimal trees.

One approach to possibly circumvent this issue is the introduction of a lookahead: instead of immediately picking the predicate with lowest impurity at every node, the algorithm could conduct a more extensive search by first exploring how well the best predicates for its children would perform. This idea has for example been investigated in [EM03; Brá+18].

A different approach is based on *evolutionary algorithms*, e.g. [LL02, Ch. 1], which have been used in different ways in DT learning [Bar+12]. Their randomized nature guarantees a robust global search, but could also pose a challenge in controller representation, since we need to guarantee overfitting of the training data.

Predicates. The predicates for numeric features we have considered have been limited to standard axis-aligned splits and oblique predicates. It might be fruitful to allow more expressive predicates involving nonlinear combinations of features to obtain even smaller decision trees. For instance, Ashok et al. [Ash+19b] show that crafting predicates from domain knowledge — in their case kinematic equations — can significantly reduce tree sizes. A potential starting point could be the work of Akmese [Akm19], who uses a grammar-based approach for the automatic generation of rich predicates.

Optimality. As discussed in Section 5.4.5, the determinization techniques we have introduced often suffer from the drawback that they produce non-optimal controllers. The question how optimality can be preserved, while still keeping the decision trees small and explainable, is an ongoing research issue. For instance, we could imagine that

some sort of refinement procedure that allows to interactively expand nodes to increase the amount of nondeterminism in combination with our determinization techniques could be a worthwhile pursuit. Additionally, one could try to adapt techniques from standard multi-label machine learning, such as PPT [Rea08] or HOMER [TKV08], which try to keep as many labels as possible by design, to the controller representation problem.

8.2 Extending `dtControl`

Visualization. The DOT format `dtControl` currently outputs is well-suited for the visual inspection of small decision trees. However, once DT sizes reach several hundred nodes, a more interactive way to visualize the trees would be desirable. For example, the ability to expand or collapse certain nodes of the tree would make it much easier to get an overview of how the DT operates. Similarly, the tool could highlight important nodes, i.e. nodes with large corresponding sub-datasets, and deemphasize nodes only applicable to a few states of the controller. Other ideas for the visualization of large decision trees have for example been presented in [NHS00].

Input and output formats. To facilitate the usage of `dtControl` and make it suitable for a wider range of applications, support for many other verification tools, such as `pFaces` [KZ19], `QUEST` [JZ17], `CoSyMA` [MGG13], or Storm [Deh+17] could be added to the tool. It would also be interesting to investigate the performance of our algorithms in the domain of program synthesis, where DTs have previously been used for the representation of piecewise functions [NSM16]. To this end, `dtControl` would for example need to provide support for the program synthesis framework developed in [NSM16]. Furthermore, additional output formats such as VHDL could simplify the implementation of the controllers produced by the tool.

Performance improvements. Although `dtControl` already is relatively performant, it still often needs several hours to learn decision trees for large controllers. To improve its efficiency, one could for example parallelize the tree building process and investigate whether there is room for optimization in the current implementation.

9 Conclusion

We have investigated a variety of techniques for controller representation with decision trees. We first demonstrated how the controller representation problem can be framed as a classification task, which allows for the usage of standard DT learning algorithms from the field of machine learning. However, fundamental differences between the verification and the machine learning setting made it necessary to adapt those algorithms to ensure that the resulting decision trees retain the safety guarantees of the controller.

We then examined the three main building blocks of the DT learning algorithm in detail: *predicates* partition the controller into several sub-controllers and make up the nodes of the decision tree. *Impurity measures* are used to evaluate the quality of predicates and select the best one at every node. Finally, various *determinization techniques* aim to reduce tree sizes by representing only a determinized version of the controller that allows a subset of the safe actions for every state.

Our results on numerous case studies demonstrate the effectiveness of decision trees for controller representation in general and our approach to DT learning in particular. Novel techniques for dealing with categorical state variables and new determinization approaches allow us to obtain DTs that are tremendously smaller than naive lookup tables and also perform significantly better than BDDs. Additionally, we have seen that the resulting decision trees are easily explainable as they often capture the semantics of the underlying system, which allows for understanding, validating, and potentially correcting the mathematical model.

All algorithms and techniques presented in this thesis are available in latest version of the open-source tool `dtControl`, which supports the easy conversion of controllers from a variety of formal verification tools to a decision tree representation in either the DOT format or as C code. As shown in Chapter 6, the new software architecture we developed for the tool is highly flexible and easily extensible, in that it allows for the straightforward integration of new algorithms and verification tools.

We anticipate that the decision tree learning algorithms we have covered and their implementation in the easy-to-use tool `dtControl` will allow a wider audience of researchers and practitioners of formal verification to represent their controllers concisely. Not only will this simplify the implementation of these controllers, but the understanding that can be gained from the decision tree representation also has the potential to help in validating and improving upon the underlying model. We expect that these advances will considerably facilitate the development of safe and reliable computer systems in the future.

Acronyms

ADD Algebraic Decision Diagram

AUC Area Under Receiver-Operator Curve

BDD Binary Decision Diagram

CLI Command-Line Interface

CPS Cyber-Physical System

DT Decision Tree

MDP Markov Decision Process

SVM Support Vector Machine

UML Unified Modeling Language

Bibliography

[AD90] R. Alur and D. L. Dill. "Automata For Modeling Real-Time Systems". In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*. Ed. by M. Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 322–335. ISBN: 3-540-52826-1. DOI: 10.1007/BFb0032042.

[Akm19] S. M. Akmese. "Generating Richer Predicates for Decision Trees". Bachelor's Thesis. Technical University of Munich, 2019.

[Alj+09] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue. "Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples". In: *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*. IEEE Computer Society, 2009, pp. 299–308. ISBN: 978-0-7695-3808-2. DOI: 10.1109/QEST.2009.8. URL: https://ieeexplore.ieee.org/xpl/conhome/5290656/proceeding.

[Ash+19a] P. Ashok, T. Brázdil, K. Chatterjee, J. Křetínský, C. H. Lampert, and V. Toman. "Strategy Representation by Decision Trees with Linear Classifiers". In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*. 2019, pp. 109–128. DOI: 10.1007/978-3-030-30281-8_7.

[Ash+19b] P. Ashok, J. Křetínský, K. G. Larsen, A. L. Coënt, J. H. Taankvist, and M. Weininger. "SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes". In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*. 2019, pp. 147–164. DOI: 10.1007/978-3-030-30281-8_9.

[Ash+20a] P. Ashok, M. Jackermeier, P. Jagtap, J. Křetínský, M. Weininger, and M. Zamani. "Demo: dtControl: Decision Tree Learning Algorithms for Controller Representation". In: *23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '20), April 22—24, 2020, Sydney, NSW, Australia*. ACM, New York, NY, USA, 2020. ISBN: 978-1-4503-7018-9. DOI: 10.1145/3365365.3383468.

[Ash+20b] P. Ashok, M. Jackermeier, P. Jagtap, J. Křetínský, M. Weininger, and M. Zamani. "dtControl: Decision Tree Learning Algorithms for Controller Representation". In: *23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '20), April 22—24, 2020, Sydney, NSW, Aus-*

tralia. ACM, New York, NY, USA, 2020. ISBN: 978-1-4503-7018-9. DOI: 10.1145/3365365.3382220.

[Bah+97] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. “Algebraic Decision Diagrams and Their Applications”. In: *Formal Methods in System Design* 10.2/3 (1997), pp. 171–206. DOI: 10.1023/A:1008699807402.

[Bar+12] R. C. Barros, M. P. Basgalupp, A. C. P. de Leon Ferreira de Carvalho, and A. A. Freitas. “A Survey of Evolutionary Algorithms for Decision-Tree Induction”. In: *IEEE Trans. Systems, Man, and Cybernetics, Part C* 42.3 (2012), pp. 291–312. DOI: 10.1109/TSMCC.2011.2157494.

[Beh+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. “UPPAAL-Tiga: Time for Playing Games!” In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by W. Damm and H. Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 121–125. ISBN: 978-3-540-73367-6. DOI: 10.1007/978-3-540-73368-3_14. URL: -.

[Bis07] C. M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN: 9780387310732. URL: http://www.worldcat.org/oclc/71008143.

[Brá+15] T. Brázdil, K. Chatterjee, M. Chmelik, A. Fellner, and J. Křetínský. “Counterexample Explanation by Learning Small Strategies in Markov Decision Processes”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 158–177. ISBN: 978-3-319-21689-8. DOI: 10.1007/978-3-319-21690-4_10.

[Brá+18] T. Brázdil, K. Chatterjee, J. Křetínský, and V. Toman. “Strategy Representation by Decision Trees in Reactive Synthesis”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by D. Beyer and M. Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 385–407. ISBN: 978-3-319-89959-6. DOI: 10.1007/978-3-319-89960-2_21.

[Bre+84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN: 0-534-98053-8.

[Bre01] L. Breiman. “Random Forests”. In: *Mach. Learn.* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324.

[Bry86] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[Cas+09] F. Cassez, J. J. Jessen, K. G. Larsen, J. Raskin, and P. Reynier. "Automatic Synthesis of Robust and Optimal Controllers - An Industrial Case Study". In: *Hybrid Systems: Computation and Control, 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings*. Ed. by R. Majumdar and P. Tabuada. Vol. 5469. Lecture Notes in Computer Science. Springer, 2009, pp. 90–104. ISBN: 978-3-642-00601-2. DOI: 10.1007/978-3-642-00602-9_7.

[CE07] I. T. Christou and S. Efremidis. "An Evolving Oblique Decision Tree Ensemble Architecture for Continuous Learning Applications". In: *Artificial Intelligence and Innovations 2007: from Theory to Applications, Proceedings of the 4th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI 2007), 19-21 September 2007, Peania, Athens, Greece*. Ed. by C. Boukis, A. Pnevmatikakis, and L. Polymenakos. Vol. 247. IFIP. Springer, 2007, pp. 3–11. ISBN: 978-0-387-74160-4. DOI: 10.1007/978-0-387-74161-1_1.

[CGP01] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC.

[CHC03] Y. Chen, C. Hsu, and S. Chou. "Constructing a multi-valued and multi-labeled decision tree". In: *Expert Syst. Appl.* 25.2 (2003), pp. 199–209. DOI: 10.1016/S0957-4174(03)00047-2.

[CK01] A. Clare and R. D. King. "Knowledge Discovery in Multi-label Phenotype Data". In: *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD 2001, Freiburg, Germany, September 3-5, 2001, Proceedings*. Ed. by L. D. Raedt and A. Siebes. Vol. 2168. Lecture Notes in Computer Science. Springer, 2001, pp. 42–53. ISBN: 3-540-42534-9. DOI: 10.1007/3-540-44794-6_4.

[Dav+14] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist. "On Time with Minimal Expected Cost!" In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. Ed. by F. Cassez and J. Raskin. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 129–145. ISBN: 978-3-319-11935-9. DOI: 10.1007/978-3-319-11936-6_10.

[Dav+15] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. "Uppaal Stratego". In: *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,*

London, UK, April 11-18, 2015. Proceedings. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 206–211. ISBN: 978-3-662-46680-3. DOI: 10.1007/978-3-662-46681-0_16.

[Deh+17] C. Dehnert, S. Junges, J. Katoen, and M. Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by R. Majumdar and V. Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 592–600. ISBN: 978-3-319-63389-3. DOI: 10.1007/978-3-319-63390-9_31.

[Del+14] M. F. Delgado, E. Cernadas, S. Barro, and D. G. Amorim. "Do we need hundreds of classifiers to solve real world classification problems?" In: *J. Mach. Learn. Res.* 15.1 (2014), pp. 3133–3181. URL: http://dl.acm.org/citation.cfm?id=2697065.

[Dij72] E. W. Dijkstra. "The Humble Programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[EJ91] E. A. Emerson and C. S. Jutla. "Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)". In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 368–377. ISBN: 0-8186-2445-0. DOI: 10.1109/SFCS.1991.185392. URL: https://ieeexplore.ieee.org/xpl/conhome/379/proceeding.

[EM03] T. Elomaa and T. Malinen. "On Lookahead Heuristics in Decision Tree Learning". In: *Foundations of Intelligent Systems, 14th International Symposium, ISMIS 2003, Maebashi City, Japan, October 28-31, 2003, Proceedings*. Ed. by N. Zhong, Z. W. Ras, S. Tsumoto, and E. Suzuki. Vol. 2871. Lecture Notes in Computer Science. Springer, 2003, pp. 445–453. ISBN: 3-540-20256-0. DOI: 10.1007/978-3-540-39592-8_63.

[Fen14] L. Feng. "On learning assumptions for compositional verification of probabilistic systems". PhD thesis. University of Oxford, UK, 2014. URL: http://ora.ox.ac.uk/objects/uuid:12502ba2-478f-429a-a250-6590c43a8e8a.

[For19] D. A. Forsyth. *Applied Machine Learning*. Springer, 2019. ISBN: 978-3-030-18113-0. DOI: 10.1007/978-3-030-18114-7.

[Fra+10] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg. "Weka-A Machine Learning Workbench for Data Mining". In: *Data Mining and Knowledge Discovery Handbook, 2nd ed*. Ed. by O. Maimon and L. Rokach. Springer, 2010, pp. 1269–1277. ISBN: 978-0-387-09822-7. DOI: 10.1007/978-0-387-09823-4_66. URL: http://www.springerlink.com/content/978-0-387-09822-7.

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gir12] A. Girard. "Low-Complexity Quantized Switching Controllers using Approximate Bisimulation". In: *CoRR* abs/1209.4576 (2012). arXiv: 1209.4576.

[GN00] E. R. Gansner and S. C. North. "An open graph visualization system and its applications to software engineering". In: *Softw., Pract. Exper.* 30.11 (2000), pp. 1203–1233. DOI: 10.1002/1097-024X(200009)30:11\<1203::AID-SPE338\>3.0.CO;2-N.

[Gos+19] F. Gossen, A. Murtovi, P. Zweihoff, and B. Steffen. "ADD-Lib: Decision Diagrams in Practice". In: *CoRR* abs/1912.11308 (2019). arXiv: 1912.11308.

[Hea93] D. G. Heath. "A Geometric Framework for Machine Learning". UMI Order No. GAX93-13375. PhD thesis. USA, 1993.

[HKS93] D. G. Heath, S. Kasif, and S. Salzberg. "Induction of Oblique Decision Trees". In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*. Ed. by R. Bajcsy. Morgan Kaufmann, 1993, pp. 1002–1007. ISBN: 1-55860-300-X. URL: http://ijcai.org/proceedings/1993-1.

[HMS66] E. B. Hunt, J. Marin, and P. J. Stone. *Experiments in Induction*. New York: Academic Press, 1966.

[IR90] A. Itai and M. Rodeh. "Symmetry breaking in distributed networks". In: *Inf. Comput.* 88.1 (1990), pp. 60–87. DOI: 10.1016/0890-5401(90)90004-2.

[Jag+18] P. Jagtap, F. Abdi, M. Rungger, M. Zamani, and M. Caccamo. "Software Fault Tolerance for Cyber-Physical Systems via Full System Restart". In: *CoRR* abs/1812.03546 (2018). arXiv: 1812.03546. URL: http://arxiv.org/abs/1812.03546.

[Jes+07] J. J. Jessen, J. I. Rasmussen, K. G. Larsen, and A. David. "Guided Controller Synthesis for Climate Controller Using Uppaal Tiga". In: *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*. Ed. by J. Raskin and P. S. Thiagarajan. Vol. 4763. Lecture Notes in Computer Science. Springer, 2007, pp. 227–240. ISBN: 978-3-540-75453-4. DOI: 10.1007/978-3-540-75454-1_17.

[JZ17] P. Jagtap and M. Zamani. "QUEST: A Tool for State-Space Quantization-Free Synthesis of Symbolic Controllers". In: *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*. Ed. by N. Bertrand and L. Bortolussi. Vol. 10503. Lecture Notes in Computer Science. Springer, 2017, pp. 309–313. ISBN: 978-3-319-66334-0. DOI: 10.1007/978-3-319-66335-7_21.

[KM11] S. Kikuchi and Y. Matsumoto. “Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker”. In: *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*. Ed. by L. Liu and M. Parashar. IEEE Computer Society, 2011, pp. 49–56. ISBN: 978-1-4577-0836-7. DOI: 10.1109/CLOUD.2011.48. URL: https://ieeexplore.ieee.org/xpl/conhome/6008653/proceeding.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_47.

[KNP12] M. Kwiatkowska, G. Norman, and D. Parker. “The PRISM Benchmark Suite”. In: *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST’12)*. IEEE CS Press, 2012, pp. 203–204.

[KNS02] M. Z. Kwiatkowska, G. Norman, and J. Sproston. “Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol”. In: *Process Algebra and Probabilistic Methods, Performance Modeling and Verification, Second Joint International Workshop PAPM-PROBMIV 2002, Copenhagen, Denmark, July 25-26, 2002, Proceedings*. Ed. by H. Hermanns and R. Segala. Vol. 2399. Lecture Notes in Computer Science. Springer, 2002, pp. 169–187. ISBN: 3-540-43913-7. DOI: 10.1007/3-540-45605-8_11.

[KNS03] M. Kwiatkowska, G. Norman, and J. Sproston. “Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol”. In: *Formal Aspects of Computing* 14.3 (2003), pp. 295–318.

[Kwi+04] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. “Symbolic Model Checking for Probabilistic Timed Automata”. In: *Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant Systems (FORMATS/FTRTFT’04)*. Ed. by Y. Lakhnech and S. Yovine. Vol. 3253. LNCS. Springer, 2004, pp. 293–308.

[Kwi+06] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. “Performance Analysis of Probabilistic Timed Automata using Digital Clocks”. In: *Formal Methods in System Design* 29 (2006), pp. 33–78.

[KZ19] M. Khaled and M. Zamani. “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*. Ed. by N. Ozay and P. Prabhakar. ACM, 2019, pp. 252–257. ISBN: 978-1-4503-6282-5. DOI: 10.1145/3302504.3311798. URL: https://dl.acm.org/citation.cfm?id=3302504.

[Lar+18] K. G. Larsen, A. L. Coënt, M. Mikucionis, and J. H. Taankvist. "Guaranteed Control Synthesis for Continuous Systems in Uppaal Tiga". In: *Cyber Physical Systems. Model-Based Design - 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Turin, Italy, October 4-5, 2018, Revised Selected Papers*. Ed. by R. D. Chamberlain, W. Taha, and M. Törngren. Vol. 11615. Lecture Notes in Computer Science. Springer, 2018, pp. 113–133. ISBN: 978-3-030-23702-8. DOI: 10.1007/978-3-030-23703-5_6.

[LHF03] N. Landwehr, M. A. Hall, and E. Frank. "Logistic Model Trees". In: *Machine Learning: ECML 2003, 14th European Conference on Machine Learning, Cavtat-Dubrovnik, Croatia, September 22-26, 2003, Proceedings*. Ed. by N. Lavrac, D. Gamberger, L. Todorovski, and H. Blockeel. Vol. 2837. Lecture Notes in Computer Science. Springer, 2003, pp. 241–252. ISBN: 3-540-20121-1. DOI: 10.1007/978-3-540-39857-8_23.

[LL02] P. Larrañaga and J. A. Lozano, eds. *Estimation of Distribution Algorithms*. Genetic Algorithms and Evolutionary Computation. Springer, 2002. ISBN: 978-1-4613-5604-2. DOI: 10.1007/978-1-4615-1539-5.

[LMT15] K. G. Larsen, M. Mikucionis, and J. H. Taankvist. "Safe and Optimal Adaptive Cruise Control". In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. Ed. by R. Meyer, A. Platzer, and H. Wehrheim. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 260–277. ISBN: 978-3-319-23505-9. DOI: 10.1007/978-3-319-23506-6_17.

[LT93] N. G. Leveson and C. S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41.

[Mad+12] G. Madjarov, D. Kocev, D. Gjorgjevikj, and S. Dzeroski. "An extensive experimental comparison of methods for multi-label learning". In: *Pattern Recognit.* 45.9 (2012), pp. 3084–3104. DOI: 10.1016/j.patcog.2012.03.004.

[Mey+17] P. J. Meyer, M. Rungger, M. Luttenberger, J. Esparza, and M. Zamani. "Quantitative Implementation Strategies for Safety Controllers". In: *CoRR* abs/1712.05278 (2017). arXiv: 1712.05278. URL: http://arxiv.org/abs/1712.05278.

[MGG13] S. Mouelhi, A. Girard, and G. Gößler. "CoSyMA: a tool for controller synthesis using multi-scale abstractions". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*. Ed. by C. Belta and F. Ivancic. ACM, 2013, pp. 83–88. ISBN: 978-1-4503-1567-8. DOI: 10.1145/2461328.2461343. URL: http://dl.acm.org/citation.cfm?id=2461328.

[Min89] J. Mingers. "An Empirical Comparison of Pruning Methods for Decision Tree Induction". In: *Mach. Learn.* 4 (1989), pp. 227–243. DOI: 10.1023/A:1022604100933.

[Mit97] T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN: 978-0-07-042807-2. URL: http://www.worldcat.org/oclc/61321007.

[Mur+93] S. K. Murthy, S. Kasif, S. Salzberg, and R. Beigel. "OC1: A Randomized Induction of Oblique Decision Trees". In: *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*. Ed. by R. Fikes and W. G. Lehnert. AAAI Press / The MIT Press, 1993, pp. 322–327. ISBN: 0-262-51071-5. URL: http://www.aaai.org/Library/AAAI/1993/aaai93-049.php.

[Mur98] S. K. Murthy. "Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey". In: *Data Min. Knowl. Discov.* 2.4 (1998), pp. 345–389. DOI: 10.1023/A:1009744630224.

[NHS00] T. D. Nguyen, T. B. Ho, and H. Shimodaira. "A visualization tool for interactive learning of large decision trees". In: *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2000), 13-15 November 2000, Vancouver, BC, Canada*. IEEE Computer Society, 2000, pp. 28–35. ISBN: 0-7695-0909-6. DOI: 10.1109/TAI.2000.889842. URL: https://ieeexplore.ieee.org/xpl/conhome/7148/proceeding.

[NM19] D. Neider and O. Markgraf. "Learning-Based Synthesis of Safety Controllers". In: *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. 2019, pp. 120–128. DOI: 10.23919/FMCAD.2019.8894254.

[NP14] G. Norman and D. Parker. *Quantitative Verification: Formal Guarantees for Timeliness, Reliability and Performance*. Tech. rep. The London Mathematical Society and the Smith Institute, 2014.

[NSM16] D. Neider, S. Saha, and P. Madhusudan. "Synthesizing Piece-Wise Functions by Learning Classifiers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 186–203. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_11.

[Oli06] T. Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: 10.1002/9780470316887.

[Qui86] J. R. Quinlan. "Induction of Decision Trees". In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: 10.1023/A:1022643204877.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

[Rea+16] J. Read, P. Reutemann, B. Pfahringer, and G. Holmes. "MEKA: A Multi-label/Multi-target Extension to WEKA". In: *J. Mach. Learn. Res.* 17 (2016), 21:1–21:5. URL: http://jmlr.org/papers/v17/12-164.html.

[Rea08] J. Read. "A pruned problem transformation method for multi-label classification". In: *Proc. 2008 New Zealand Computer Science Research Student Conference (NZCSRS 2008)*. 2008, pp. 143–150.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[RWR15] M. Rungger, A. Weber, and G. Reissig. "State space grids for low complexity abstractions". In: *54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*. IEEE, 2015, pp. 6139–6146. ISBN: 978-1-4799-7886-1. DOI: 10.1109/CDC.2015.7403185. URL: https://ieeexplore.ieee.org/xpl/conhome/7396016/proceeding.

[RZ16] M. Rungger and M. Zamani. "SCOTS: A Tool for the Synthesis of Symbolic Controllers". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. Ed. by A. Abate and G. E. Fainekos. ACM, 2016, pp. 99–104. ISBN: 978-1-4503-3955-1. DOI: 10.1145/2883817.2883834.

[Sha48] C. E. Shannon. "A mathematical theory of communication". In: *Bell Syst. Tech. J.* 27.4 (1948), pp. 623–656. DOI: 10.1002/j.1538-7305.1948.tb00917.x.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured Design". In: *IBM Syst. J.* 13.2 (1974), pp. 115–139. DOI: 10.1147/sj.132.0115.

[Som01] F. Somenzi. "Efficient manipulation of decision diagrams". In: *Int. J. Softw. Tools Technol. Transf.* 3.2 (2001), pp. 171–181. DOI: 10.1007/s100090100042.

[SZ19] A. Swikir and M. Zamani. "Compositional Synthesis of Symbolic Models for Networks of Switched Systems". In: *IEEE Control. Syst. Lett.* 3.4 (2019), pp. 1056–1061. DOI: 10.1109/LCSYS.2019.2920766.

[Tab09] P. Tabuada. *Verification and Control of Hybrid Systems - A Symbolic Approach*. Springer, 2009. ISBN: 978-1-4419-0223-8. URL: http://www.springer.com/mathematics/applications/book/978-1-4419-0223-8.

[TK07] G. Tsoumakas and I. Katakis. "Multi-Label Classification: An Overview". In: *IJDWM* 3.3 (2007), pp. 1–13. DOI: 10.4018/jdwm.2007070101.

[TKV08] G. Tsoumakas, I. Katakis, and I. Vlahavas. "Effective and efficient multilabel classification in domains with large number of labels". In: *Proc. ECML/PKDD 2008 Workshop on Mining Multidimensional Data (MMD'08)*. Vol. 21. sn. 2008, pp. 30–44.

[Utg88] P. E. Utgoff. "Perceptron Trees: A Case Study In Hybrid Concept Representations". In: *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*. Ed. by H. E. Shrobe, T. M. Mitchell, and R. G. Smith. AAAI Press / The MIT Press, 1988, pp. 601–606. ISBN: 0-262-51055-3. URL: http://www.aaai.org/Library/AAAI/1988/aaai88-107.php.

[Zha+10] X. Zhang, Q. Yuan, S. Zhao, W. Fan, W. Zheng, and Z. Wang. "Multi-label Classification without the Multi-label Cost". In: *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*. SIAM, 2010, pp. 778–789. ISBN: 978-0-89871-703-7. DOI: 10.1137/1.9781611972801.68.