dtcontrol

Release 2.0.1-6-g86b3e0e

Mathias Jackermeier, Pranav Ashok, Maximilian Weininger

Aug 25, 2022

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

dtControl is a tool for compressing memoryless controllers arising out of automatic controller synthesis of cyber-physical systems (CPS). dtControl takes as input a controller synthesised by various formal verification tools and represents them in the form of decision trees. In the process, the size of the controller is reduced greatly, and at the same time, it becomes more explainable. While in principle, memoryless strategies in any format can be handled by dtControl, currently it supports controllers output by three tools: SCOTS, Uppaal Stratego and Storm. Additionally, there is rudimentary support for strategies produced by PRISM Model Checker. Moreover, it also supports a CSV-based format which allows the user to quickly experiment with the techniques provided by dtControl.

We provide a *User Manual*, which gives information necessary to use dtControl and run the various decision tree learning algorithms implemented in it, as described in the paper "dtControl: Decision Tree Learning Algorithms for Controller Representation" appearing at the 23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2020).

An additional *Developer Manual* is made available for those who are interested in interfacing their own controller synthesis tools with dtControl and/or those interested in implementing their own strategy representation algorithms.

1.1 User Manual

This document equips the user with the information necessary to use dtControl and run the various decision tree learning algorithms implemented in it.

1.1.1 Capabilities

(Watch a 5 minute introductory video on YouTube)

dtControl is a tool to represent controllers using decision trees. It uses decision tree learning to achieve this. The 5-minute video above gives a quick introduction to why and how one may benefit from using dtControl.

The decision tree algorithm running inside dtControl is highly configurable. You may choose to represent a determinized controller, allow for more expressible decision predicates or even tune the heuristic used to pick the best predicate. See *Presets and configuration files* for more details.

While dtControl achieves best results with permissive or non-deterministic controllers with determinization enabled, it can also be used with deterministic controllers. However empirical results with such controllers are not as significant as with non-deterministic controllers.

1.1.2 Getting Started

A quick start installation guide is available in the README. In this section, we elaborate a little more on the installation process.

Installation

Getting dtControl from the Python Package Index (PyPI) is the recommended way to install it. Before running the `pip install` command, we recommend creating a virtual environment so that dependencies of dtControl do not conflict with any other packages already installed on your system. The official Python documentation for creating virtual environments may help you set it up. However, we provide the most essential information here.

Once you have a recent version of Python 3 installed, you may run:

```
$ python3 -m venv venv
```

to create the virtual environment in a folder called `venv` located in your current directory. You can enter the virtual environment by running:

```
$ source venv/bin/activate
```

Typically, your shell might indicate that the virtual environment is activated by changing the prompt symbol `$` to something like `(venv) $`. You can now proceed with installing dtControl from PyPI using `pip`:

```
(venv) $ pip install dtcontrol
```

Note: In case you want to get the development version of dtControl, you could instead clone the Git repository first:

```
(venv) $ git clone https://gitlab.lrz.de/i7/dtcontrol.git
```

And then run:

```
(venv) $ pip install .
```

from within the `dtcontrol` folder.

Once the dtControl package is installed, the command line interface can be accessed using the `dtcontrol` command. Try running:

```
(venv) $ dtcontrol -h
```

If your installation has run successfully, you will now see the help page detailing the usage and arguments.:

```
usage: dtcontrol [-h] [-v] [--input INPUT [INPUT ...]] [--output OUTPUT] [--benchmark-
↪file FILENAME] [--config CONFIGFILE] [--use-preset USE_PRESET [USE_PRESET ...]] [--
↪rerun]
              [--timeout TIMEOUT]
              {preset,clean} ...

 Scroll to the end of the help message for Quick Start.

 optional arguments:
   -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  -v, --version         show program's version number and exit

input/output:
  --input INPUT [INPUT ...], -i INPUT [INPUT ...]
                        The input switch takes in one or more space separated file␣
↪names or a folder name which contains valid controllers (.scs, .dump or .csv)
  --output OUTPUT, -o OUTPUT
                        The output switch takes in a path to a folder where the␣
↪constructed controller representation would be saved (c and dot)
  --benchmark-file FILENAME, -b FILENAME
                        Saves statistics pertaining the construction of the decision␣
↪trees and their sizes into a JSON file, and additionally allows to view it via an HTML␣
↪file.

run configurations:
  --config CONFIGFILE, -c CONFIGFILE
                        Specify location of a YAML file containing run configurarions.␣
↪Use along with the --use-preset switch. More details in the User Manual.
  --use-preset USE_PRESET [USE_PRESET ...], -p USE_PRESET [USE_PRESET ...]
                        Run one or more presets defined in the CONFIGFILE. If the --
↪config switch has not been used, then presets are chosen from the system-level␣
↪configuration file.
                        Special parameters for this switch include 'all', 'all-user',
↪'all-system'. Refer the User Manual for more details.
  --rerun, -r           Rerun the experiment for all input-method combinations.␣
↪Overrides the default behaviour of not running benchmarks for combinations which are␣
↪already present in
                        the benchmark file.
  --timeout TIMEOUT, -t TIMEOUT
                        Sets a timeout for each method. Can be specified in seconds,␣
↪minutes or hours (eg. 300s, 7m or 3h)

other commands:
  {preset,clean}        Run 'dtcontrol COMMAND --help' to see command specific help

Examples:
Create a file storing run configurations
    dtcontrol preset --sample > user-config.yml

Display all presets available with dtcontrol
    dtcontrol preset --config user-config.yml --list

Run the 'my-config' preset on the SCOTS model located at 'examples/cps/cartpole.scs'
    dtcontrol --input examples/cps/cartpole.scs --config user-config.yml --use-preset␣
↪my-config
```

Input

Supported tools

dtControl currently supports the file formats generated by the tools SCOTS, Uppaal Stratego, PRISM, and Storm. To see how to add suppport for new file formats to dtControl, refer to the Developer Manual.

SCOTS and Uppaal output `.scs` and `.dump` files, respectively, as the result of a controller synthesis process. These can directly be specified as input to dtControl.

For PRISM, dtControl expects a strategy file that maps state indices to actions and a states file that maps state indices to the corresponding values of state variables. These files can be generated by PRISM with the following options:

```
prism firewire_abst.nm liveness_abst.pctl -const 'delay=50,fast=0.5000' -prop 1 -
↪explicit -exportstrat 'firewire_abst.prism:type=actions' -exportstates 'firewire_abst_
↪states.prism'
```

It is important that both files have a `.prism` extension and the states file has the same name as the actions file with an `_states` suffix.

Storm has a JSON strategy export that can be invoked using, for example, the following command:

```
storm --jani firewire_abst.jani --janiproperty time_max --constants delay=3 --exact --
↪timemem --buildstateval --buildchoiceorig  --exportscheduler firewire_abst.3.time_max.
↪storm.json
```

or if using PRISM models

> storm –prism firewire_abst.nm –prop firewire.props –constants delay=50,fast=0.5 –buildstateval –buildchoiceorig –exportscheduler firewire_abst.storm.json

Note that `--exportscheduler` must be given an output file name that ends with `.storm.json`, both so that Storm knows that the strategy must be exported in the JSON format, and dtControl recognizes the exported strategy.

Note: We recommend using the Storm exporter even for PRISM models, unless every action in the PRISM model is named (`[action_name] guard -> update`). Typically, PRISM models contain anonymous actions and this causes the strategy export to give all such actions the name `null`. When using Storm, an exported action is comprised of: the action name (if available), the guard and the update.

It is also possible to run dtControl on all controllers in a given folder. The tool then simply looks for all files with one of the above extensions.

Specifying metadata

While dtControl tries to obtain as much information as possible from the controller directly, it is sometimes necessary to provide additional metadata to the tool. For example, dtControl cannot know which variables in the controller are categorical, which is necessary for some of the specialized algorithms. It is also possible to specify names, e.g. for variables, which are used in the DOT output.

This metadata can be given in a JSON file named `controller_name_config.json` (where `controller_name` must match the name of the controller file), which allows to set the following options:

- `x_column_types` is a dictionary with two entries, `numeric` and `categorical`. These entries are lists with indices specifying which variables are numeric or categorical, respectively.
- `y_column_types` provides the same information for the output variables.

- `x_column_names` is a list of variable names.
- `x_category_names` is a dictionary with one entry for every categorical variable (as specified in `x_column_types[categorical]`). This entry can either be an index or a name from `x_column_names` and maps to a list of category names for the variable. For instance, an entry of the form `"color": ["red", "green", "blue"]` would mean that a 0 in the `color` variable stands for `red`, a 1 means `green`, and a 2 denotes `blue`.
- `y_category_names` gives the same information for the output variables.

If any of the options in the metadata are not set, dtControl tries to fall back to reasonable defaults, such as `x[i]` for the column names or just integers `i` for the category names. By default, all variables are treated as numeric.

An example is given in form of the configuration file for the `firewire_abst.prism` case study:

```
{
  "x_column_types": {
    "numeric": [
      0
    ],
    "categorical": [
      1
    ]
  },
  "x_column_names": [
    "clock",
    "state"
  ],
  "x_category_names": {
    "state": [
      "start_start",
      "fast_start",
      "start_fast",
      "start_slow",
      "slow_start",
      "fast_fast",
      "fast_slow",
      "slow_fast",
      "slow_slow"
    ]
  }
}
```

This configuration provides the information that the two variables are `clock` and `state`, the first of which is numeric and the second of which is categorical. Furthermore, a `state` of 0 corresponds to `start_start`, a state of 1 to `fast_start`, and so on. Note that, for PRISM models, dtControl automatically parses the names of the actions and it is thus not necessary to provide a `y_category_names` entry.

The Command-line Interface

This section shows how to configure and run dtControl. For this purpose, we assume that you have an `examples` folder in your current directory containing `cartpole.scs`. You can choose to download all of our examples from our Gitlab repository via this zip archive or using *git*. Extract the contents of the archive into a folder called `examples` and unzip `cartpole.scs.zip`. Alternatively, you can run the following commands:

```
$ mkdir -p examples && cd examples
$ wget -P examples/cps https://gitlab.lrz.de/i7/dtcontrol-examples/-/raw/master/cps/
↪cartpole.scs.zip
$ unzip -d ./examples/cps ./examples/cps/cartpole.scs.zip
```

Next, activate the virtual environment you installed dtControl in:

```
$ source venv/bin/activate
```

Running your first command

Finally, you can run dtControl with the default parameters on the *cartpole* example (`cartpole.scs`), use the following command:

```
(venv) $ dtcontrol --input examples/cps/cartpole.scs
```

This will produce some new files and folders in the current folder:

```
decision_trees
|-- default
|   `-- cartpole
|       |-- default.c
|       |-- default.dot
benchmark.json
benchmark.html
```

Open `benchmark.html` in your favourite browser to view a summary of the results. For more details on what these files are, see *Understanding the output*.

Presets and configuration files

dtControl allows the user to configure the learning algorithm using "presets" defined in a "configuration file". The presets can be chosen using the `--use-preset` switch and the preset configuration file can be chosen using the `--config` switch. For your convenience, we have pre-defined a bunch of preset configurations that we believe are interesting. You can list the available presets by running:

```
(venv) $ dtcontrol preset --list
```

This should produce the following table of presets.

name	numeric-predicates	categorical-predicates	determinize	impurity	tolerance	safe-pruning
default	['axisonly']	['multisplit']	none	entropy	1e-05	False
cart	['axisonly']		none	entropy		
linsvm	['axisonly', 'linear-linsvm']		none	entropy		
logreg	['axisonly', 'linear-logreg']		none	entropy		
oc1	['oc1']		none	entropy		
avg	['axisonly']	['valuegrouping']	none	entropy		
singlesplit	['axisonly']	['singlesplit']	none	entropy		
maxfreq	['axisonly']		maxfreq	entropy		
maxfreqlc	['axisonly', 'linear-logreg']		maxfreq	entropy		
minnorm	['axisonly']		minnorm	entropy		
minnormlc	['axisonly', 'linear-logreg']		minnorm	entropy		
mlentropy	['axisonly']		auto	multilabelentropy		
sos	['axisonly']		none	entropy		
sos-safepruning	['axisonly']		none	entropy		True
linear-auroc	['axisonly', 'linear-logreg']		none	auroc		

The `--use-preset` argument takes in one or more preset names as argument. For each preset specified as argument, dtControl will run the learning algorithm configured as described in this table and produce results in the folder: `decision_trees/<preset_name>/<example_name>/`.

Configurable options

1. **numeric-predicates** can be used to configure the class of predicates that are considered for constructing the tree. It can take the values
   a. `axisonly` for predicates which compare a variable to a constant
   b. `linear-logreg` for predicates which compare a linear combination of variables to a constant (`ax + by < c`) obtained using Logistic Regression
   c. `linear-linsvm` for linear predicates obtained using linear Support Vector Machines, and finally
   d. `oc1` for predicates obtained from the tool of Murthy et. al
2. **categorical-predicates** determines how non-numeric or categorical variables (such as `color = blue`) should be dealt with. Currently, it only supports the option
   a. `multisplit` which creates a decision node with as many children as the number of possible categories the variable can take (e.g. `color = blue`, `color = green` and `color = red`).
   b. `singlesplit` which creates a decision node with just two children, one satisfying a categorical equality (`color = blue`) and the other that does not (`color != blue`).
   c. `valuegrouping` as described in M. Jackermeier's thesis (TODO link)
3. **determinize** determines the type of determinization used on permissive/non-deterministic controller when constructing the tree. Possible options are

   a. `none` to preserve permissiveness,
   b. `minnorm` to pick control inputs with the minimal norm,
   c. `maxnorm` to pick control inputs with the maximal norm,
   d. `random` to pick a control input uniformly at random,
   e. `maxfreq` to pick our in-house developed determinization strategy, details of which are available in M. Jackermeier's thesis.
   f. `auto` to let dtControl automatically choose a determinization strategy; currently defaults to `maxfreq`.
4. **impurity** allows users to choose the measure by which splitting predicates are evaluated. Possible options are
   a. `entropy`
   b. `gini`
   c. `auroc`
   d. `maxminority`
   e. `twoing`
   f. `multilabelentropy`
   g. `multilabelgini`
   h. `multilabeltwoing`
5. **tolerance** is a floating point value relevant only when choosing the `valuegrouping` categorical predicate.
6. **safe-pruning** decides whether to post-process the decision tree as specified in Ashok et. al. (2019).

Creating your own presets

As a user, you can define your own preset by mixing and matching the parameters from *Configurable options*. The presets must be defined inside a `.yml` file as follows:

```
presets:
  my-config:
    determinize: maxfreq
    numeric-predicates: ['axisonly']
    categorical-predicates: ['singlesplit']
    impurity: 'entropy'
    safe-pruning: False
  another-config:
    determinize: minnorm
    numeric-predicates: ['linear-logreg']
    categorical-predicates: ['valuegrouping']
    tolerance: 10e-4
    safe-pruning: False
```

Note: The values for the keys `numeric-predicates` and `categorical-predicates` are lists. If the list contain more than one elements, e.g. `numeric-predicates: ['axisonly', 'linear-svm']`, dtControl will construct predicates for each of the classes present (in this case, both axis-parallel and linear splits using a linear SVM) in the list and pick the best predicate amongst all the classes.

The above sample presets can be generated automatically and wrote into a `user-config.yml` file by running:

```
(venv) $ dtcontrol preset --sample > user-config.yml
```

Now, dtControl can be run on the *cartpole* example with the `my-config` preset by running:

```
(venv) $ dtcontrol --input examples/cps/cartpole.scs --config user-config.yml --use-
↪preset my-config
```

Understanding the output

Once dtControl is used to run some experiments, you may notice a bunch of new files and folders:

```
decision_trees
|-- default
|   `-- cartpole
|       |-- default.c
|       |-- default.dot
|       |-- default.json
|-- my-config
|   `-- cartpole
|       |-- my-config.c
|       |-- my-config.dot
|       |-- my-config.json
benchmark.json
benchmark.html
```

- `benchmark.html` is the central file, which summarizes all the results obtained by dtControl. It may be opened using a browser of your choice.
- `benchmark.json` is a JSON file containing all the statistics collected by the tool (tree size, bandwidth, construction time and other metadata). The `benchmark.html` file is rendered from this JSON file at the end of the experiments.
- `default.c` contains the C-code of the decision tree
- `default.dot` contains the DOT source code which can be compiled using the `dot -Tpdf default.dot -o default.pdf` command or viewed using a web-based tool
- `default.json` contains a decision tree in a machine-readable format (valid according to this JSON Schema). The schema is also given below as a (human-readable) Python code below.

```python
@dataclass
class LinearTerm:
    coeff: Optional[float] = None
    intercept: Optional[float] = None
    var: Optional[int] = None


@dataclass
class AxisParallelExpression:
    coeff: int
    var: int


@dataclass
```

(continues on next page)

(continued from previous page)

```
class Split:
    lhs: Union[List[LinearTerm], AxisParallelExpression]
    op: str
    rhs: str


@dataclass
class Node:
    actual_label: Union[List[str], None, str]
    children: List['Node']
    edge_label: Optional[Union[List[str], str]] = None
    split: Optional[Split] = None
```

where **actual_label** is a list of actions (only for leaf nodes) and **edge_label** is a string that defines the relation between the parent and the current node (for Boolean predicates, the **edge_label** marks the `true` and `false` branches; for categorical predicates, the **edge_label** gives the value or list of values being split on).

By default, the decision trees are stored in the `decision_trees` folder and the statistics are stored in the `benchmark.json` and `benchmark.html` files. This can however be customized with the help of the `--output` and the `--benchmark-file` switches. For example:

```
(venv) $ dtcontrol --input examples/cps/cartpole.scs \
                            --config user-config.yml \
                            --use-preset my-config \
                            --output cartpole_trees \
                            --benchmark-file cartpole_stats
```

Will produce the following files and directories:

```
cartpole_trees
|-- my-config
|   `-- cartpole
|       |-- my-config.c
|       |-- my-config.dot
|       |-- my-config.json
cartpole_stats.json
cartpole_stats.html
```

Timeout

Another useful feature is timeout which can be set with the `--timeout/-t` switch. For example,:

```
$ dtcontrol --input examples/truck_trailer.scs --timeout 3m
```

will run CART on the *truck_trailer* example, and time out if it is taking longer than 3 minutes to finish. The `--timeout/-t` switch can accept timeout in seconds, minutes and hours (`-t 42s` or `-t 30m` or `-t 1h`). The timeouts is applied for each preset individually, and not for the whole set of experiments.

Re-run

By default, new results are appended to `benchmark.json` (or the file passed to the `--benchmark-file` switch) and experiments are not re-run if results already exist. In case you want to re-run a method and overwrite existing results, use the `--rerun` flag.:

```
$ dtcontrol --input examples/cps/cartpole.scs --rerun
```

1.1.3 Quick Start with the Python Interface

More advanced users can use dtControl programmatically using Python or as part of a Jupyter notebook. Here is an example of the Python interface with comments that give guidance on what is happening:

```
# imports
# you might have to import additional classifiers
from sklearn.linear_model import LogisticRegression
from dtcontrol.benchmark_suite import BenchmarkSuite
from dtcontrol.decision_tree.decision_tree import DecisionTree
from dtcontrol.decision_tree.determinization.max_freq_determinizer import␣
↪MaxFreqDeterminizer
from dtcontrol.decision_tree.impurity.entropy import Entropy
from dtcontrol.decision_tree.impurity.multi_label_entropy import MultiLabelEntropy
from dtcontrol.decision_tree.splitting.axis_aligned import AxisAlignedSplittingStrategy
from dtcontrol.decision_tree.splitting.linear_classifier import␣
↪LinearClassifierSplittingStrategy

# instantiate the benchmark suite with a timeout of 2 hours
# rest of the parameters behave like in CLI
suite = BenchmarkSuite(timeout=60*60*2,
                       save_folder='saved_classifiers',
                       benchmark_file='benchmark',
                       rerun=False)

# Add the 'examples' directory as the base where
# the different controllers will be searched for
# You can also choose to only include specific files
# in the directory with the 'include' and 'exclude' list
suite.add_datasets('examples')

# setting up the predicates
aa = AxisAlignedSplittingStrategy()
logreg = LinearClassifierSplittingStrategy(LogisticRegression, solver='lbfgs', penalty=
↪'none')

# select the DT learning algorithms we want to run and give them names
classifiers = [
    DecisionTree([aa], Entropy(), 'CART'),
    DecisionTree([aa, logreg], Entropy(), 'LogReg'),
    DecisionTree([aa], Entropy(), 'Early-stopping', early_stopping=True),
    DecisionTree([aa], Entropy(MaxFreqDeterminizer()), 'MaxFreq', early_stopping=True),
    DecisionTree([aa], MultiLabelEntropy(), 'MultiLabelEntropy', early_stopping=True)
]
```

(continues on next page)

(continued from previous page)

```
# finally, execute the benchmark
suite.benchmark(classifiers)
# open the web browser and show the result
suite.display_html()
```

As you can see, the Python interface provides mostly the same parameters as the CLI, but gives you some additional control. In particular, the following functionality is currently only supported by the Python interface:

- Using `early_stopping` with the label powerset method
- Parameters for safe pruning and early stopping which control the amount of nondeterminism preserved
- Choosing any determinizer for oblique splits
- Only allowing oblique splits in leaf nodes
- Various parameters of the OC1 heuristic
- The `ScaledBincount` impurity measure with a custom scaling function

The easiest way to get more information on the methods available in the Python interface is to directly browse the source code of dtControl.

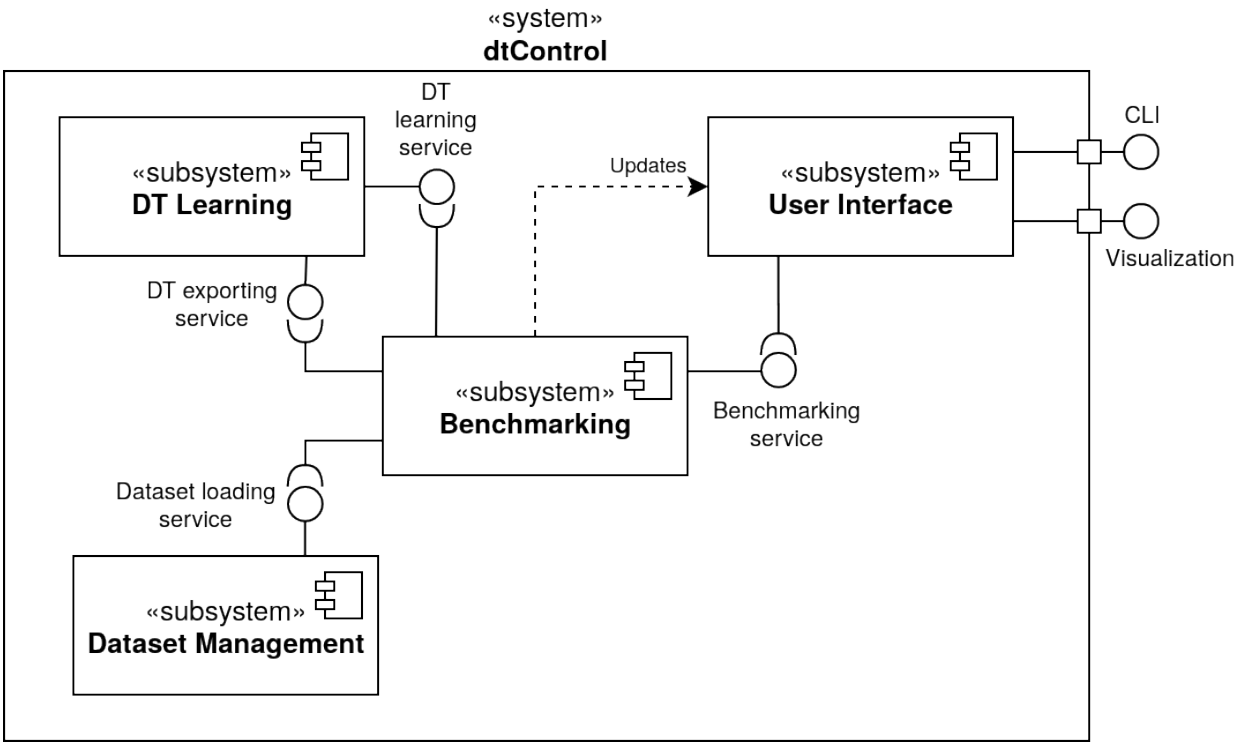
1.2 Developer Manual

This document aims to provide the reader with the necessary information to be able to extend or customize dtControl. We first give an overview of the general software architecture of the tool. Then, we show how support for controllers in new file formats can be added, explain the inner workings of the decision tree (DT) learning component, and explore how dtControl could be extended with new output formats.

1.2.1 Overview

dtControl is written entirely in Python and makes use of both the numpy and scikit-learn packages for data representation and manipulation. A basic familiarity with this programming environment is assumed throughout this manual. More information on dependencies can be found in the provided readme and setup.py files.

An overview of the software architecture of dtControl is given in the following UML component diagram:



The heart of the tool is formed by the *DT Learning* subsystem, which is responsible for the actual decision tree algorithms and representation. It provides all of the different options for training DTs and can export the learned trees in the DOT and C format.

To make the DT learning itself independent of any verification tool, 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, Uppaal Stratego, and PRISM.

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. The constructed HTML files with benchmark statistics also belong to the user interface.

1.2.2 Supporting new file formats

dtControl currently supports the file formats generated by the the tools SCOTS, Uppaal Stratego, and PRISM. There are two ways to make the tool work with other formats, as described in the following.

The CSV format

The first option is to convert the new file format to a custom CSV format that dtControl also supports. We now describe the specification of the custom CSV format.

The first two lines of the file are reserved for metadata. The first line must always reflect whether the controller is permissive (non-deterministic) or non-permissive (deterministic). This is done using either of the following lines:

```
#PERMISSIVE
```

or:

```
#NON-PERMISSIVE
```

The second line must reflect the number of state variables (or the state dimension) and the number of control input variables (or the input dimension). This line looks as follows:

```
#BEGIN N M
```

where N is the state dimension and M is the input dimension.

Every line after the 2nd line lists the state action/input pairs as a comma separated list:

```
x1,x2,...,xN,y1,y2,...,yM
```

if the controller prescribes the action `(y1,y2,...,yM)` for the state `(x1,x2,...,xN)`. If the state allows more actions, for example, `(y1',y2',...,yM')`, then this should be described on a new line:

```
x1,x2,...,xN,y1,y2,...,yM
x1,x2,...,xN,y1',y2',...,yM'
```

An excerpt of the `10rooms.scs` controller written in this CSV format would look as follows:

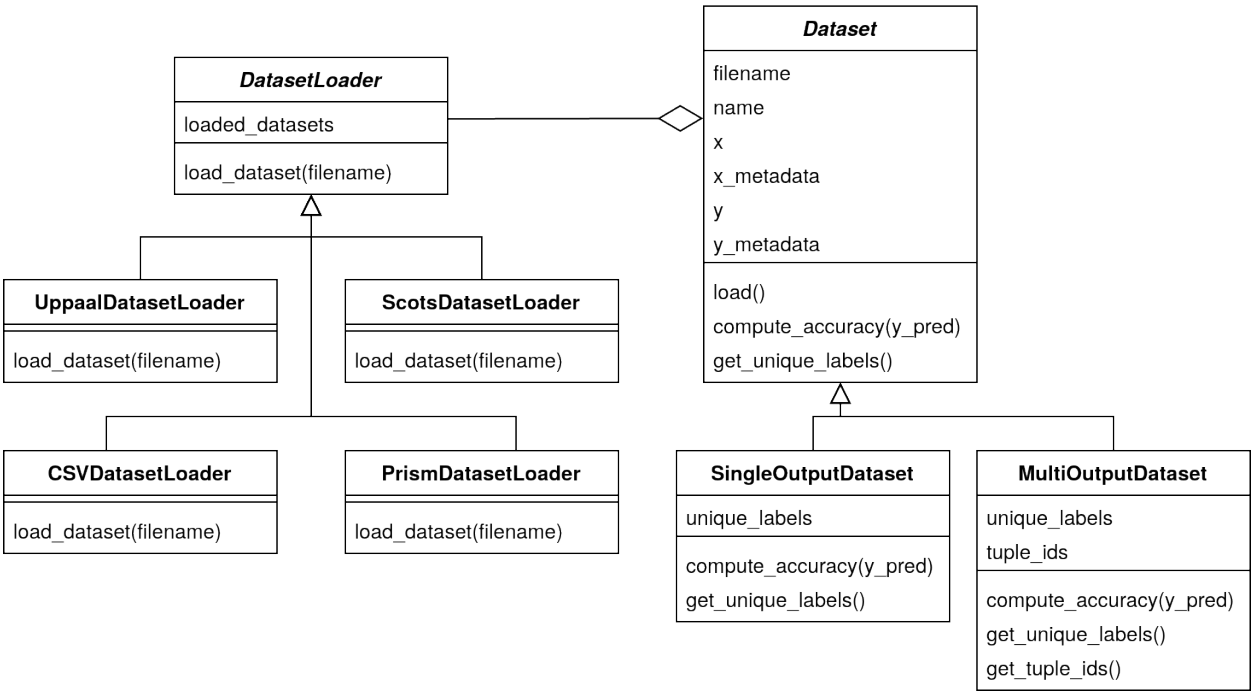
```
#PERMISSIVE
#BEGIN 10 2
18.75,20.0,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,1.0,1.0
20.0,20.0,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,1.0,1.0
21.25,20.0,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,1.0,1.0
18.75,21.25,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,0.0,1.0
18.75,21.25,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,0.5,1.0
18.75,21.25,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,1.0,1.0
20.0,21.25,18.75,18.75,20.0,18.75,18.75,18.75,18.75,18.75,0.0,1.0
```

dtControl will automatically look for files with a .csv extension and parse them with the assumption that they follow this format.

Implementing a new dataset loader – the dataset management subsystem

Additionally, it is also possible to integrate the new file format natively into dtControl by providing a dataset loader. For this, we will take a closer look at the *Dataset Management* subsystem.

An overview of the subsystem is given in the following UML class diagram:



The `Dataset` class is what the DT learning algorithm needs to train a decision tree. A `Dataset` can either be a `SingleOutputDataset` if it only has a single control input, or a `MultiOutputDataset` if there are multiple control inputs.

Note: Since the code focuses on the decision trees themselves, it refers to the *output* of those trees, which is the same as the action produced by the controller, i.e. the control *input*.

Its most important properties are detailed in the following:

- `x` is a simple NumPy array containing the values of the state variables. It has dimension #number of states in the controller $\times$ #number of state variables.
- `x_metadata` is a dictionary containing metadata about the array `x`. For instance, it contains the names of the variables (if available), the minimum and maximum value in `x`, and indicates which columns of `x` are categorical.
- `y` is a NumPy array containing the actions that can be performed for every state. Its format differs, depending on whether the dataset is single- or multi-output:
  - In the case of single-output datasets, `y` is a two-dimensional array where each row contains all (non-deterministic) actions that can be performed at the corresponding row of `x`. Instead of the actual (possibly) floating point values, we use integer indices representing those values throughout the code; the mapping of indices to the actual values can be found in `dataset.index_to_value`. Since NumPy usually cannot deal with rows of different sizes, but we have varying numbers of possible actions, some rows have to be filled with `-1` s. These `-1` s have to be ignored during tree construction.

  - In the case of multi-output datasets, `y` is a three-dimensional array whose first dimension (or axis) corresponds to the different control inputs. Thus, there is a two-dimensional array for each control-input, which exactly matches the structure outlined above. The DT learning algorithms implemented so far all convert from this representation to the *tuple ID* representation in which every action `(y1,y2,...,yM)` is replaced with a single tuple id. The method `get_tuple_ids()` returns an array of labels in this tuple ID representation, which again matches the structure of the `y` array in the single-output case.

- `y_metadata` is a dictionary containing metadata about the array `y`. Similarly to `x_metadata`, it indicates which columns of `y` are categorical outputs, and provides some other information such as the minimum and maximum value in `y`.

The `Dataset` class provides some auxiliary functionality for the DT learning algorithm, such as `compute_accuracy(y_pred)` and `get_unique_labels()`, as well as a `load()` method that uses a `DatasetLoader` to actually load the dataset from a file into the internal NumPy representation.

It is this `DatasetLoader` class that needs to be extended in order to add support for a new file format to dtControl. A new `DatasetLoader` must provide exactly one method: `_load_dataset()` parses a file in the new format and returns the tuple `(x, x_metadata, y, y_metadata, index_to_value)` which corresponds to the attributes of a `Dataset` as outlined above.

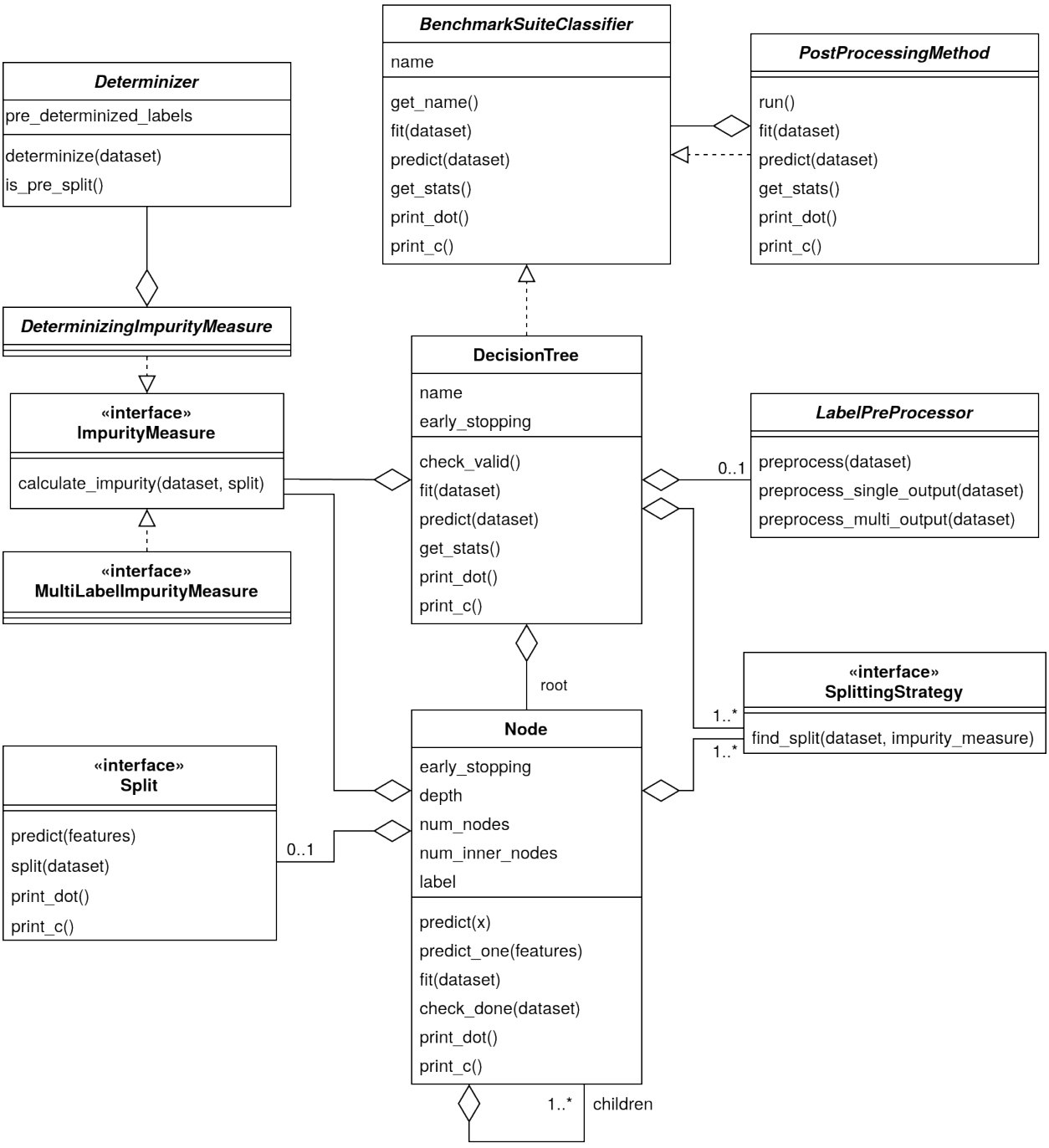
The new dataset loader can be registered in the `extension_to_loader` dictionary in the `Dataset` class. Now, if dtControl encounters a file with an extension of the new file format, it will attempt to load it using the registered loader.

You may find inspiration for writing your own dataset loader in some of the already existing ones such as the `UppaalDatasetLoader` or the relatively simple `PrismDatasetLoader`. However, they are very specific to their respective file format.

1.2.3 Extending dtControl with new algorithms - the decision tree learning subsystem

dtControl already supports a wide variety of decision tree construction algorithms. Furthermore, the tool can readily be extended with new algorithms, as we will see in this section.

We again start with a UML diagram of the DT learning subsystem. In order to keep it as flexible as possible, we use a composition-based approach that makes heavy use of interfaces. This has the advantage that you only need to develop against a specific interface if you want to only extend a part of the DT learning algorithm. For instance, if you want to add a new impurity measure, you just have to provide an implementation of the `ImpurityMeasure` interface and your code will immediately integrate with the rest of the learning algorithm.



As can be seen, the heart of the component is the `DecisionTree` class, which offers the core methods one would expect:

- `fit(dataset)` constructs a decision tree for a dataset.
- `predict(dataset)` returns a list of control inputs predicted for a dataset.
- `get_stats()` returns the statistics to be displayed in the benchmark results as a dictionary. This will mainly include the number of nodes and potentially some algorithm-specific statistics.
- `export_dot()` saves a representation of the decision tree in the DOT format.

- `export_c()` exports the decision tree to a C-file as a chain of if-else statements.

Most of these methods simply delegate to the `root` object of type `Node`, which implements the actual decision tree data structure. It has mostly the same attributes as a `DecisionTree`, as well as some statistics and either a list of children or a label. Depending on the dataset and algorithm, a label can be one of the following:

- A single integer (that appears in the `index_to_value` dictionary) corresponding to a single action
- A single tuple of integers corresponding to a single action with multiple outputs
- A list of integers corresponding to multiple possible actions
- A list of tuples corresponding to multiple possible actions with multiple outputs

We now examine the most important interfaces in detail.

Splitting strategies

A `SplittingStrategy` provides the method `find_split(dataset, impurity_measure)`, which returns the best predicate of a certain type, given a dataset and an impurity measure. For instance, the `AxisAlignedSplittingStrategy` searches through all possible axis-aligned splits for the given dataset and returns the one with lowest impurity.

The returned predicate is of type `Split` and must provide the following methods:

- `predict(features)` returns an index into the `children` list of a `Node` corresponding to the child that should be picked for the given NumPy array of features.
- `get_masks(dataset)` returns a list of NumPy masks indicating how the dataset is split. A mask is just a one-dimensional array of boolean values with a length of `len(dataset)`. A value of `True` in the jth row of the ith mask indicates that the jth row in `dataset.x` belongs to the ith sub-dataset created by the split.
- `print_dot()` returns the string that should be put in the node in the DOT format.
- `print_c()` returns the string that should be put in the corresponding if-statement in the C code.

The simplest example of a `Split` is probably the `AxisAlignedSplit`.

Impurity measures

An `ImpurityMeasure` needs to provide the `calculate_impurity(dataset, split)` method, which simply returns a float indicating the impurity. There are two types of impurity measures:

- `MultiLabelImpurityMeasures` directly compute the impurity from the nondeterministic labels. Examples include `MultiLabelEntropy` and `MultiLabelGiniIndex`.
- `DeterminizingImpurityMeasures` correspond mostly to the traditional impurity measures known from decision trees in machine learning. Examples include `Entropy` and `GiniIndex`. These impurity measures are called *determinizing* since they don't directly work on the nondeterministic labels. Instead, the use a `Determinizer` that first converts the labels to a new representation.

  By default, the `LabelPowersetDeterminizer` is used, which treats every combination of possible labels as a unique label and thus preserves all of the nondeterminism present in the original controller. Other options are for example the `MaxFreqDeterminizer`, which implements the maximum frequency determinization technique. Determinizers can either be applied before or after splitting, as indicated by the `is_pre_split()` method.

Determinization

The final ingredient of the DT learning algorithm - determinization - is mainly controlled by the `early_stopping` attribute of a `DecisionTree`. If it is set to `True`, early stopping is performed and the resulting DT is thus (possibly) smaller and more deterministic. This parameter should always be enabled if impurity measures that make use of determinization are used, such as the `MultiLabelEntropy` or any `DeterminizingImpurityMeasure` with the `MaxFreqDeterminizer`.

Instead, one can also choose to determinize the controller itself before DT learning with a `LabelPreProcessor`, such as the `NormPreProcessor`. For this, the methods `preprocess_single_output(dataset)` and `preprocess_multi_output(dataset)` that return a NumPy array of determinized labels must be provided.

Finally, decision trees can also be post-processed by a `PostProcessingMethod` such as safe pruning. The most important method of the class is `run()`, which runs the post-processing technique on its classifier, transforming the decision tree.

1.2.4 Supporting new output formats

As shown above, the core decision tree data structure is implemented in the `DecisionTree` and `Node` classes. These classes also offer functionality for DOT and C printing.

To add a new output format to dtControl, one thus would have to provide new exporting methods in the `DecisionTree` and `Node` classes. Furthermore, the `BenchmarkSuite` would have to be adapted to export the tree to the new output format once a DT has been constructed.