BNClassifier Manual

Contents

1 Intro to BNClassifier

BNClassifier is a tool for classifying Boolean models by dynamic properties expressed in the hybrid extension of the branching-time logic CTL (HCTL).

Specifically, BNClassifier provides a means to categorize various interpretations of partially specified Boolean networks (PSBNs). It enables users to first restrict the space of possible network interpretations to those that satisfy a set of given *required properties*. Then, it decomposes the remaining interpretations into classes based on the *classification properties* they satisfy. BNClassifier further provides an interactive interface to explore the relationships between the resulting classes and analyze necessary conditions for the properties to hold.

1.1 Tool workflow

BNClassifier consists of two main components (see Fig. 1): The model checking and *classification engine* (available as a Rust or a Python library) and an interactive *decision tree explorer* (a multiplatform desktop application).

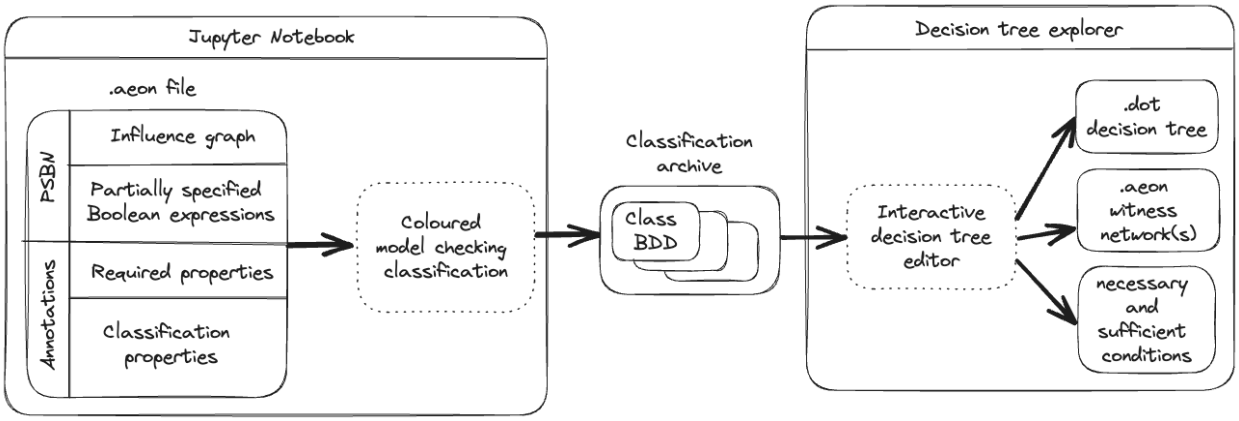


Figure 1: Visual overview of the BNClassifier components and functionality. The left part represents the *classification engine*, typically used through its Python API in Jupyter notebooks. The right part represents the *decision tree explorer*, which is used to explore the classification result interactively.

2 Running BNClassifier

This Section provides general instructions for installing and setting up both components of BNClassifier– the classification engine, and the desktop application for exploring decision trees. The tool's source code, licensed under MIT license, is freely available at `https://github.com/sybila/biodivine-bn-classifier`.

2.1 Classification engine

Since the functionality of the *classification engine* is available as a Python library, it can be integrated with Jupyter notebooks and other similar environments. A typical classification engine use occurs through the Python scripts or Jupyter Notebooks, so we focus on that aspect.

To use the Python bindings, one must activate the virtual Python environment and install all the dependencies. See `https://docs.python.org/3/library/venv.html` for more information on Python environments. For instance, on debial-based linux distributions, you need to execute the following:

```
sudo apt update
sudo apt install python3-venv
```

```
python3 -m venv ./env
source ./env/bin/activate
```

Once you have the environment set up and active, install dependencies using the following:

```
pip3 install pip --upgrade
pip3 install biodivine_aeon==0.3.0 notebook==7.0.6 graphviz==0.20.1
```

Finally, you can start a Jupyter session in which you will execute the experiments and generate results for BNClassifier to analyze. Note the environment must be active first.

```
python3 -m jupyter notebook
```

On most systems, this command will open a new browser window. However, in some cases, the browser cannot open the default URL provided by *jupyter*. Instead, you need to return to the terminal window and find a *"Or copy and paste one of these URLs:"* message. Then, copy the URL under this message and paste it into the browser window. The URL should look like this:

`http://localhost:8888/tree?token=SOME_RANDOM_STRING_OF_CHARACTERS`.

If you are unfamiliar with Jupyter Notebooks, you can execute a code cell by selecting it and pressing `Shift + Enter`. Alternatively, you can also press the `run` button in the top toolbar (using the *play* arrow icon). See Fig. 2 for an illustration.

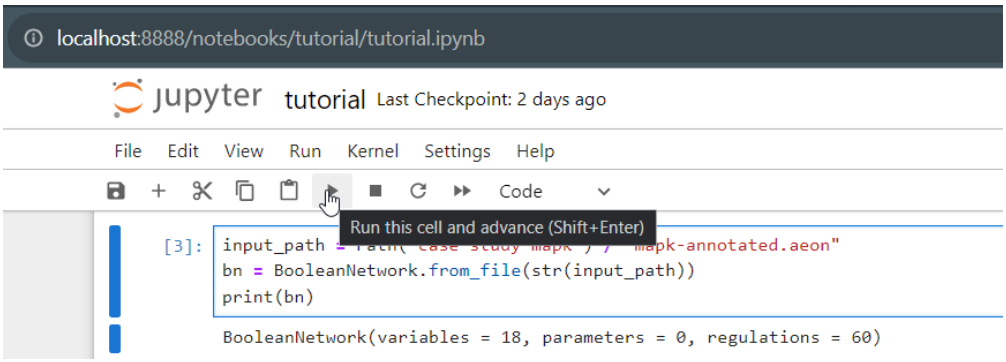


Figure 2: Illustration of executing code cells in Jupyter Notebooks.

2.2 Desktop application

You can download the pre-built installation packages for all major operating systems from the `GitHub releases` at `https://github.com/sybila/biodivine-bn-classifier/releases/tag/v0.2.3`. Choose an option relevant to your system, download it, and potentially install it (e.g. if you chose `.deb` or `.msi` installation package). We also provide a tutorial and example input files as part of the tool's repository.

Once you execute the binary (or run the application), you will be prompted to pick an archive representing the classification results. To test that everything works correctly, you can use the pre-computed results `./tutorial/classification-archive.zip` from the tool's repository. Once you select the archive, the application should show you a single `Mixed outcomes` node overlayed with a short help message.

The desktop application can also be built directly from the repository's source code. To do this, please consult the *Development guide* of the repository readme.

2.3 Tutorial

For a short step-by-step tutorial on how BNClassifier can be used, see the provided jupyter notebook at `tutorial/tutorial.ipynb` in the tool's repository. This tutorial illustrates the whole workflow on a MAPK pathway case study.

3 Input format

As shown in Fig. 1, the inputs of BNClassifier consist of a partially specified Boolean network, a corresponding influence graph, a set of required properties, and a set of classification properties. BNClassifier takes all its inputs via a single *.aeon* file. The text-based *.aeon* format was originally introduced to compactly describe PSBNs. Later, the format was extended with general-purpose annotations, which we use to specify the required and classification HCTL properties. This section describes the format and syntax of all the input components.

3.1 Partially specified Boolean network

First, the influence graph of the PSBN is described as a list of edges, where each edge is encoded as:

```
regulator edge_type target
```

Here, `regulator` and `target` are the names of the network variables, and the `edge_type` is the type of influence, corresponding to one of {`->`, `-|`, `-?`, `->?`, `-|?`, `-??`}. Arrow `->` denotes activation, `-|` inhibition and `-?` is for unspecified monotonicity. Finally, an extra `?` signifies a non-observable regulation (a regulation that may or may not affect the target).

A particular update function for variable `A` is written as

```
$A: function
```

The format of the actual update functions is given by the following grammar. Note that each *name* is a combination of alphanumeric characters and underscores.

$$
\begin{aligned}
function &= \texttt{true} \mid \texttt{false} \mid name \mid fnSymbol \mid \texttt{!}function \mid function\ OP\ function \\
OP &= \texttt{\&} \mid \texttt{|} \mid \texttt{=>} \mid \texttt{<=>} \\
fnSymbol &= name\texttt{(}args\texttt{)} \\
args &= name \mid args, args
\end{aligned}
$$

Additional information (model name, description and layout) is encoded in *comments* (lines starting with `#`). The order of declarations is not taken into account.

3.2 Properties

Both required and classification properties are given as HCTL formulae. The textual format we use for HCTL formulae is given later in this section. Below, we first show how to annotate a model file with (a) required properties and (b) named classification properties. A name of a classification property can be an arbitrary combination of alphanumeric characters and underscores.

a) `#! dynamic_assertion: #`HCTL_FORMULA`#`

b) `#! dynamic_property: NAME: #`HCTL_FORMULA`#`

3.2.1 Format for HCTL formulae

The following grammar defines our textual format for HCTL formulae:

$$
\begin{aligned}
\mathit{formula} &= \texttt{(}\ \mathit{formula}\ \texttt{)} \mid \mathit{const} \mid \mathit{propName} \mid \mathit{var} \mid \mathit{unaryOp}\ \mathit{formula} \\
&\mid \mathit{formula}\ \mathit{binaryOp}\ \mathit{formula} \\
&\mid \mathit{hybridOp}\ \mathit{var}\ \texttt{:}\ \mathit{formula} \\
\mathit{const} &= \texttt{true} \mid \texttt{false} \\
\mathit{var} &= \texttt{\{}\mathit{varName}\texttt{\}} \\
\mathit{unaryOp} &= \sim \mid \texttt{AX} \mid \texttt{EX} \mid \texttt{AF} \mid \texttt{EF} \mid \texttt{AG} \mid \texttt{EG} \\
\mathit{binaryOp} &= \texttt{\&} \mid \texttt{|} \mid \texttt{=>} \mid \texttt{<=>} \mid \texttt{\^{}} \mid \texttt{AU} \mid \texttt{EU} \\
\mathit{hybridOp} &= \texttt{!} \mid \texttt{3} \mid \texttt{V} \mid \texttt{@}
\end{aligned}
$$

Only alphanumeric characters and underscores can appear in *varName* and *propName*. Further, each *propName* must correspond to a valid BN variable name (this is checked). Note that $\sim$ corresponds to the negation, `^` to the xor operator, `!` to the bind operator, `3` to the existential quantifier, `V` to the universal quantifier, and `@` to the jump operator. Correspondence with other HCTL operators is straightforward.

The operator precedence is the following (the lower, the stronger):

- unary operators (both negation and temporal): 1
- binary temporal operators: 2
- Boolean binary operators: and=3, xor=4, or=5, imp=6, eq=7
- hybrid operators: 8

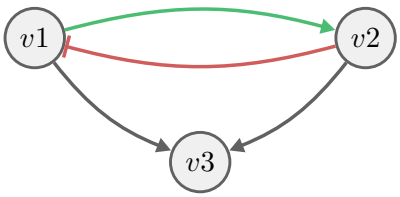
However, it is strongly recommended to use parentheses wherever possible. Note that formulae must not contain free variables.

For example, formula $\exists x.\forall y(@_{\mathrm{y}}.\,\mathbf{EF}(x \wedge (\downarrow z.\,\mathbf{AX}\,z)))$ would be written as:

```
3{x}: V{y}: (@{y}: EF ({x} & (!{z}: AX {z})))
```

3.3 Example of an annotated model

Let us have a PSBN model given by the following influence graph and update functions.



$$
\begin{aligned}
F_1(x) &= \neg v2 \\
F_2(x) &= v1 \\
F_3(x) &= \mathbf{f}(v1, v2)
\end{aligned}
$$

Variable `v1` positively influences (regulates) `v2`; `v1` negatively influences regulates `v2`; and both `v1` and `v2` influence `v3` (but the influences might be arbitrary). The update function for `v3` is not fully specified, but instead given by a *function* symbol **f** of arity two.

We further add the required property for the existence of (arbitrary) fixed-point attractor, $\exists x.\,@_{\mathrm{x}}(\mathbf{AX}\,x))$ and classification property for the existence of a fixed point reachable from any state, $\mathit{phi} = \exists x.\forall y(@_{\mathrm{y}}.\,\mathbf{EF}(x \wedge (\downarrow z.\,\mathbf{AX}\,z)))$.

The input file containing this whole model and properties may look like this:

```
#!  dynamic_assertion: #`3{x}: @{x}: (AX {x})`#
#!  dynamic_property:  phi: #`3{x}: V{y}: (@{y}: EF ({x} & (!{z}: AX {z})))`#
```

```
v2 -| v1
v1 -> v2
v2 -? v3
v1 -? v3
$v1: !v2
$v2: v1
$v3: f(v1, v2)
```

4 Classification engine

The task of the classification engine is to compute a *classification mapping*. Once you have completed the setup described in 2.1, it becomes very simple to run the computation. Fig. 3 shows an example of how to run the classification process in the jupyter notebook (or from a Python script). The tutorial jupyter notebook (`tutorial/tutorial.ipynb`) provides additional examples of what the user can do to analyze the input and output of the computation directly in the notebook environment.

```
from biodivine_aeon import *
input_model = "case-study-mapk/mapk-annotated.aeon"
output_zip = "classification-archive.zip"
run_hctl_classification(input_model, output_zip)
```

```
Loaded model and properties out of `case-study-mapk/mapk-annotated.aeon`.
Parsing formulae and generating symbolic representation...
Successfully parsed all 1 required property (assertion) and all 4 classification properties.
Successfully encoded model with 18 variables and 4 parameters.
Model admits 16 instances.
Evaluating required properties (this may take some time)...
Required properties successfully evaluated.
15 instances satisfy all required properties.
Evaluating classification properties (this may take some time)...
Classification properties successfully evaluated.
Generating classification mapping based on model-checking results...
Results saved to `classification-archive.zip`.
```

Figure 3: Illustration of how to run the classification from a jupyter notebook.

After finishing the computation, the classification engine produces a *.zip* archive containing the following:

- The result of classification (the *class* map) that consists of several raw BDD files, each representing the set of interpretations for one of the non-empty classes.
- A textual report of the whole classification procedure.
- The original annotated input model (to reconstruct the results if needed).

This classification archive is then used as an input for the *decision tree explorer* GUI.

5 Decision tree explorer

The main element of the application is an interactive decision tree editor. Here, the user can partition the classification map based on various attributes of the PSBN interpretations into a decision tree. Such a decision tree helps to explain the relationships between classes and enables user-guided search for necessary and sufficient conditions leading to the emergence of a particular class.

5.1 Decision trees

Decision trees [1] allow us to visualise custom scenarios showing the user how the particular settings of individual function symbols lead to desired classes. The fully expanded tree consists of two types of nodes:

- A *leaf node* represents interpretations exhibiting the same combination of properties.
- A *decision node* represents a choice on the concrete interpretation of a particular function symbol (a *decision attribute*). It has outgoing positive and negative edges, which lead to the sub-trees where the corresponding interpretations evaluate that particular function as true or false, respectively.

We build the decision tree in an interactive manner, allowing the user to choose which tree nodes to expand. Before the tree is fully expanded, it can also contain *mixed nodes* representing a combination of interpretations of several different classes.

Initially, the root is a *mixed node*, (unless there is only a single class – then it would be a *leaf node* from the start). Either manually, or automatically, a decision attribute is selected for each mixed node, making it into a decision node and generating two new (mixed or leaf) child nodes. Once all the terminal nodes become leaves, the tree is fully expanded.

In the GUI, a decision node is labelled by the name of its decision attribute. A leaf node representing instances of some class is labelled by the class name (the class name is given by properties that all the instances satisfy) and the number of instances the node represents. A mixed node is labelled by the number of different classes (outcomes) it represents. For a concrete example, see Fig. 4.

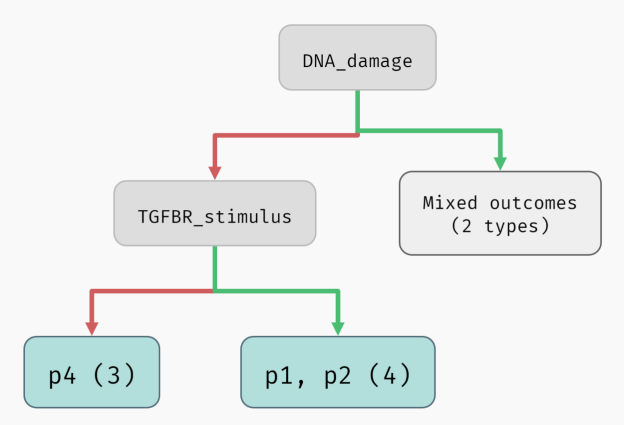


Figure 4: Example of a decision tree with all three types of nodes. There are two decision nodes labelled by their decision attributes – `DNA_damage` and `TGFBR_stimulus`. On the right is a mixed node representing interpretations of two different classes. Lastly, there are two leaf nodes, one representing 3 instances satisfying property `p4`, and the other representing 4 instances satisfying properties `p1` and `p2`.

5.1.1 Decision attributes

If the parameters of a network are function symbols of arity 0 (i.e., constants), that can be either `true` or `false`, it is simple to make decision nodes. We simply use the constant as a decision attribute.

If a network contains an uninterpreted function of higher arity as a parameter (for example, something like `f(X, Y)`), it is not very clear how such a function can appear in a decision tree. One option is to simply decide based on specific values in the function table. For example, `f(`$\neg x_1$`,`x_2`)` could be a decision attribute that fixes the value of `f(0,1)` to be either `true` or `false`. An example of such a decision attribute is given in Fig. 5 (left).

For function symbols of higher arity, we also admit different types of decision attributes that better translate to real-life conditions. Suppose we have a variable `V` with the update function given as `f(X, Y)` (function symbol `f` applied to variables `X` and `Y`). We say that an input `X` is *essential* in function `f` (or in `V`) if the value

of `f` depends on the value of `X`. We can then generalize this property further, and say that `X` is essential in `f` when `Y=false`. This means that not only `f` has to depend on the value of `X`, it has to depend on it when `Y=false`. An example of such a decision attribute is given in Fig. 5 (right).

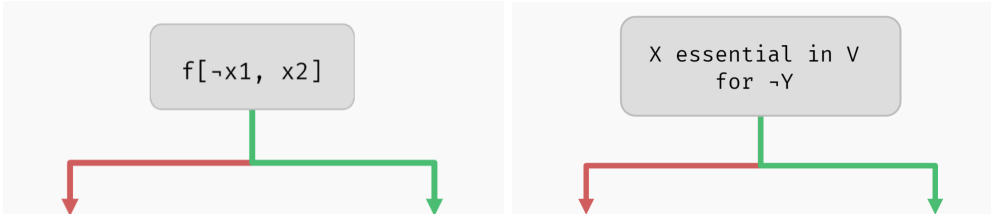
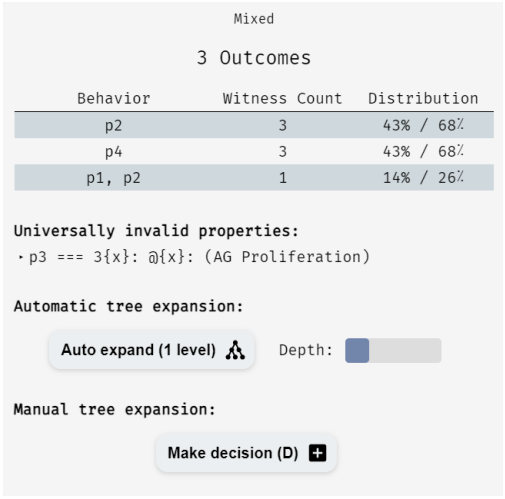


Figure 5: Examples of possible decision attributes for update function of variable `V` given by an uninterpreted function `f(X, Y)`. On the left, the decision attribute `f(`$\neg x_1$`,`x_2`)` represents a decision on the value of `f(0,1)`. On the right, the decision is on whether the variable `X` is *essential* in the update function of `V` in the context of `Y=false`.

5.2 Information panels

BNClassifier provides additional information regarding each node of the tree (and the interpretations it represents). By selecting a node, an information panel on the left appears. A different kind of panel appears for each node type, containing the information (and offering actions) relevant to that type of node.

Mixed node panel A mixed node represents interpretations of several classes. At the top, a panel for a mixed node contains a table summarizing these classes (outcomes). For each possible class, a number of represented instances and their proportions are listed (% for classical percentage and ‰ for log percentage). Then, there are lists of *Universally (in)valid properties*. The property is universally valid for a given node if it is satisfied by all the instances the node represents (and universally invalid if no instance satisfies it). After that, there are buttons for automatic and manual node expansion; more on that later. An example of a mixed node panel is shown in Fig. 6a.


Mixed

3 Outcomes

Behavior	Witness Count	Distribution
p2	3	43% / 682‰
p4	3	43% / 682‰
p1, p2	1	14% / 273‰

Universally invalid properties:

- p3 === 3{x}: @{x}: (AG Proliferation)

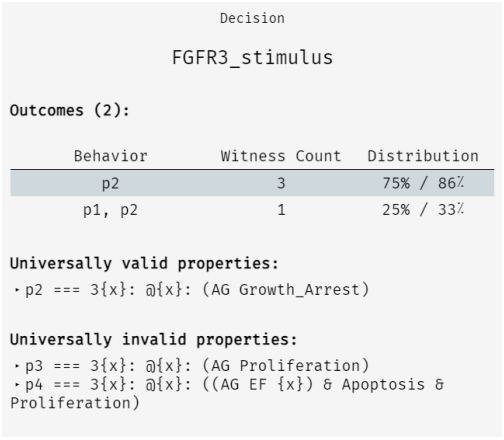
Automatic tree expansion:

Auto expand (1 level) Depth:

Manual tree expansion:

Make decision (D)

(a) An example of a panel for a mixed node.


Decision

FGFR3_stimulus

Outcomes (2):

Behavior	Witness Count	Distribution
p2	3	75% / 857‰
p1, p2	1	25% / 333‰

Universally valid properties:

- p2 === 3{x}: @{x}: (AG Growth_Arrest)

Universally invalid properties:

- p3 === 3{x}: @{x}: (AG Proliferation)
- p4 === 3{x}: @{x}: ((AG EF {x}) & Apoptosis & Proliferation)

(b) An example of a panel for a decision node.

Figure 6: An example of a mixed and decision node panels. The selected mixed node represents interpretations of three classes, and the selected decision node of two. The property `p3` is universally invalid for the mixed node, with no universally valid properties. The property `p2` is universally invalid for the decision node, while `p3` and `p4` are universally valid.

Decision node panel A panel for a decision node contains similar information as a mixed node – a table summarizing the classes of represented interpretations and their proportions, and lists of universally (in)valid

properties in the corresponding subtree. It does not offer the means for tree expansion. An example of a decision node panel is shown in Fig. 6b.

Leaf node panel Leaf node panels hold slightly different information. First, there is again a *witness count* with the number of instances represented by the node and their proportion with respect to *all* classified instances. A list of *necessary conditions* to reach the node follows. These correspond to the *choices* on network parameters one must make to get to this node (summarizing the decisions on a path from the root node). *Satisfied properties* again lists all properties satisfied by instances the node represents. Lastly, there are buttons for generating witness BN instances; more on that later. An example of a leaf node panel is shown in Fig. 7.

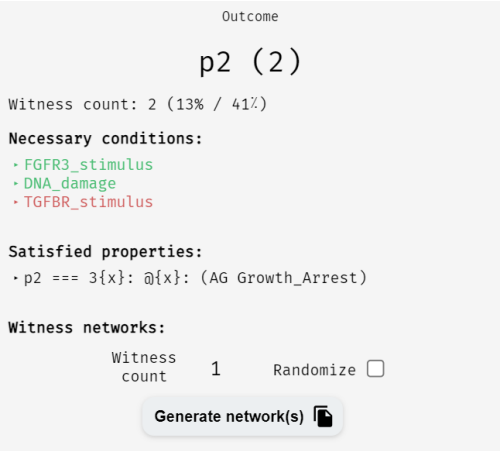


Figure 7: An example of a leaf node panel. This node represents 2 instances that satisfy only the property `p2`. To reach this node, we had to fix `FGFR3_stimulus=true`, `DNA_damage=true`, and `TGFBR_stimulus=false`.

5.3 Using the GUI

Navigation and shortcuts By scrolling, you can zoom the tree in or out. You can move the tree across the screen by dragging (at the empty space). To easily navigate in the tree, you can use the following shortcuts:

- Tree navigation: ↑ to select a parent node, ↓ to select a positive child node, ⇧+↓ to select a negative child node, ← or → to switch between positive and negative branches.
- Tree actions: backspace for removing a selected node, D for computing possible decisions for a selected node, H for displaying a help message

Manually expanding mixed nodes We can turn a mixed node into a decision node by selecting its decision attribute. Select a node, and a panel on the left appears. Click on the `Make Decision` button. This button will cause BNClassifier to compute the possible impact of picking different parameters as decision attributes. You will then be presented with a list of attributes (i.e. parameters) that you can use to split the mixed node into two. For each decision attribute, the application shows you the impact of that decision on the behaviour classes in the mixed node. You can sort the attributes by various standard metrics – *information gain* [3] is the primary sorting criterion, but you can also pick from other sorting heuristics. By selecting an attribute, the node expands. If the decision isolates one behaviour class from the rest, a leaf node with that class will appear. Otherwise, a new mixed node is created. Fig. 8 illustrates this process.

Automatically expanding mixed nodes. The explorer can also automatically generate an optimal decision tree with respect to the *information gain*, a standard metric in machine learning [3]. The procedure behind this second phase is explained in more detail in [1]. To automatically expand a mixed node, select it first. In the panel that appears on the right, select how many levels the tree should expand via the provided `Depth` toggle. Then click on the `Auto expand (N levels)` button. Fig. 9 illustrates this process.

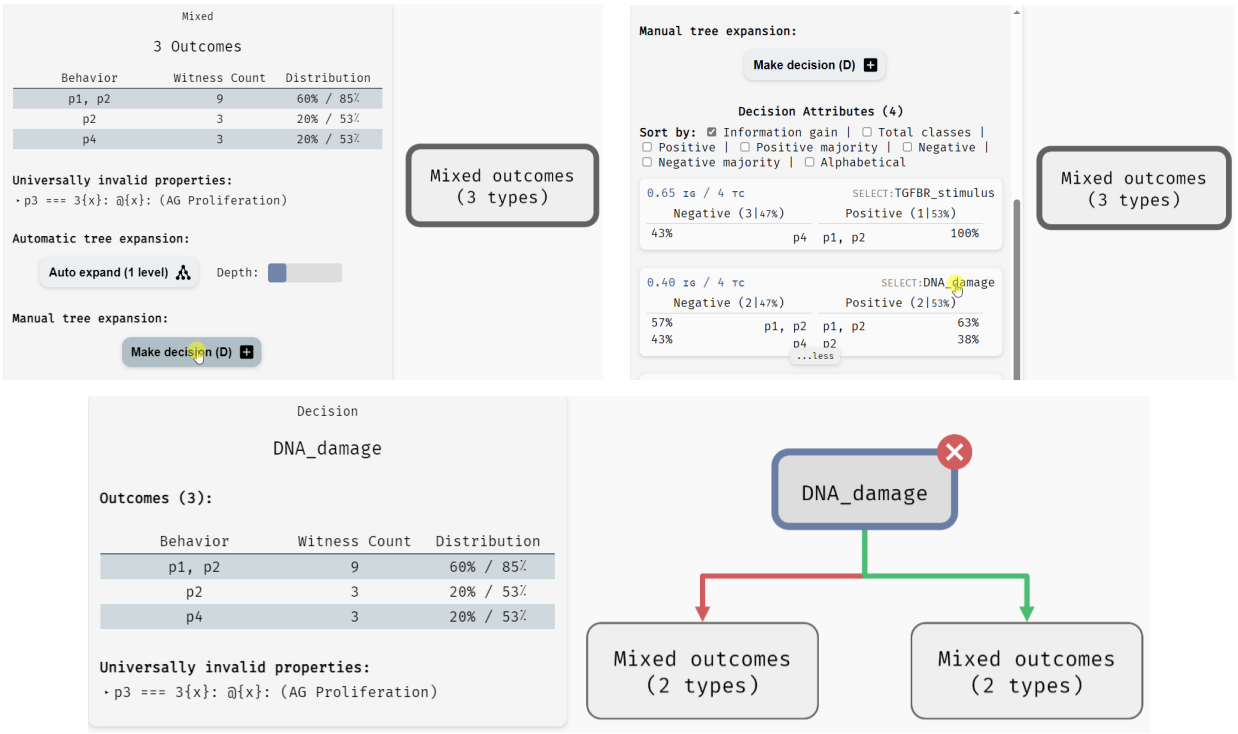


Figure 8: A guide on how to manually expand a selected node. Click on the `Make Decision` button, and then select an attribute (in our case, `DNA_damage`). This attribute then becomes a label of the decision node.

Removing tree nodes You can always delete the current decision nodes and then try a different combination of decision attributes. To remove a node, select it and click a red X in its corner, as shown in Fig. 10.

Exporting decision trees The user can then export the current tree for use in other materials as a standard `.dot` graph by clicking the `Export Decision Tree` button in the lower left corner. The `.dot` format, developed as part of the Graphviz software package [2], is a plain text graph description language used for visualizing structured data as diagrams. Relevant button and example of a tree and its `.dot` format is shown in Fig. 11.

Generating witness networks Finally, the user can randomly sample the fully specified witness networks (as `.aeon` files) for any fully expanded tree branch (representing a set of conditions sufficient to achieve a particular behaviour class). To generate witness networks, select a leaf node. You can select how many witnesses to generate in the node's information panel. You can also select a random sampling (the default sampling is deterministic). By clicking on the `Generate network(s)` button, the `File Save Dialog` will open. Witness files are saved in a `.zip` archive. Fig. 12 illustrates witness generating.

References

[1] Beneš, N., Brim, L., Pastva, S., Poláček, J., Šafránek, D.: Formal analysis of qualitative long-term behaviour in parametrised boolean networks. In: Formal Methods and Software Engineering (ICFEM 2019). Lecture Notes in Computer Science, vol. 11852, pp. 353–369. Springer (2019)

[2] Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz— open source graph drawing tools. In: Graph Drawing. pp. 483–484. Springer (2002)

[3] Quinlan, J.R.: Induction of decision trees. Machine learning **1**, 81–106 (1986)

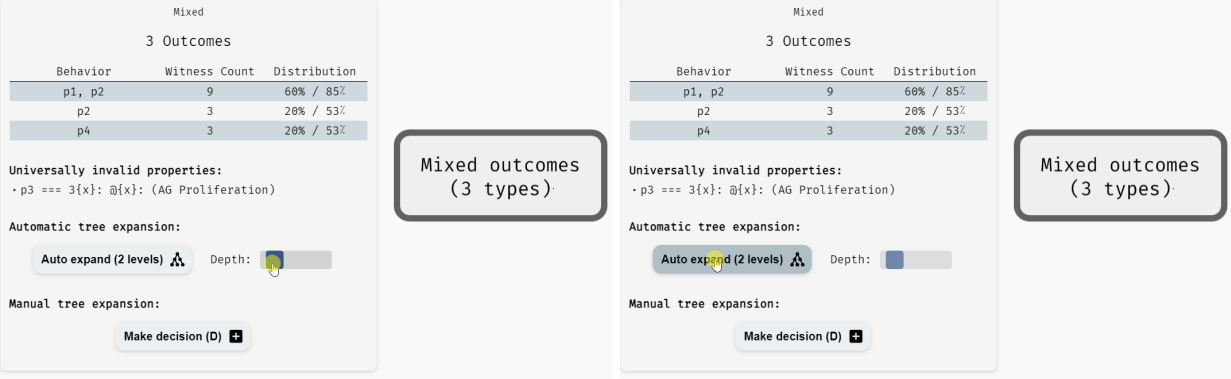


Figure 9: A guide on how to automatically expand a selected node.

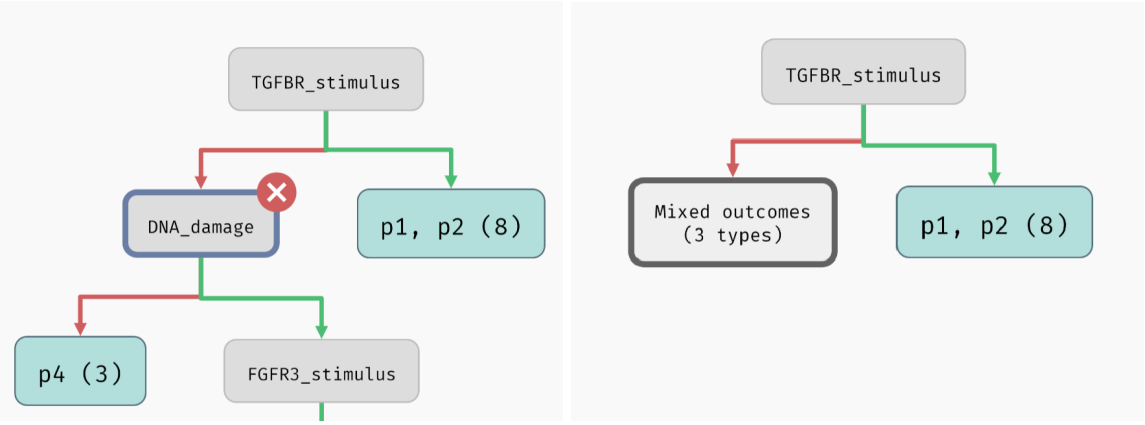
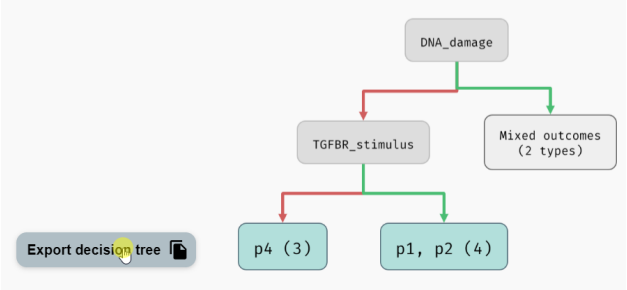


Figure 10: Example of removing a node from a decision tree and how the result might look.




```
digraph G {
init__ [label="", style=invis, height=0, width=0];
init__ -> 0;
0[label="DNA_damage"]
0 -> 3 [style=dotted];
0 -> 4 [style=filled];
3[label="TGFBR_stimulus"]
3 -> 5 [style=dotted];
3 -> 6 [style=filled];
5[label="p4 (3)(3)"];
6[label="p1, p2 (4)(4)"];
4[label="Unprocessed([(p2,3), (p1, p2,5)])"]
}
```

Figure 11: Example of a decision tree and its exporter `.dot` representation. The decision tree is exported via the `Export Decision Tree` button in the lower left corner.

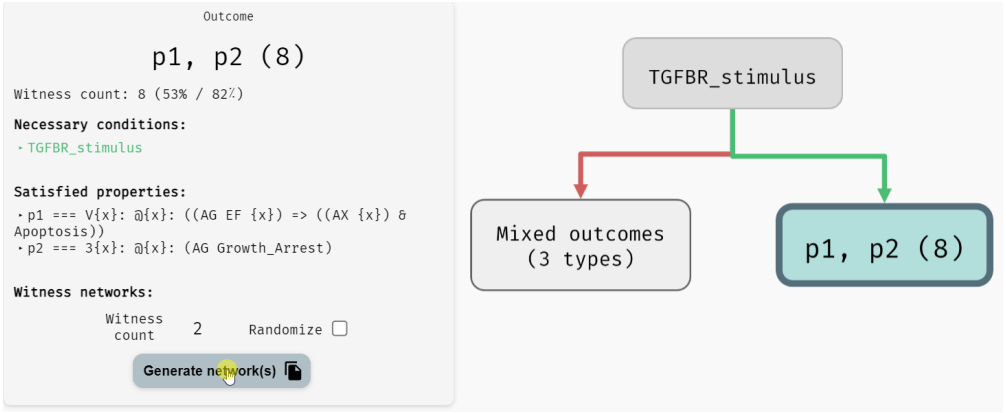


Figure 12: Illustration of how to generate witness networks. In this example, we generate 2 BN instances represented by the (selected) node on the right.