1 Introduction

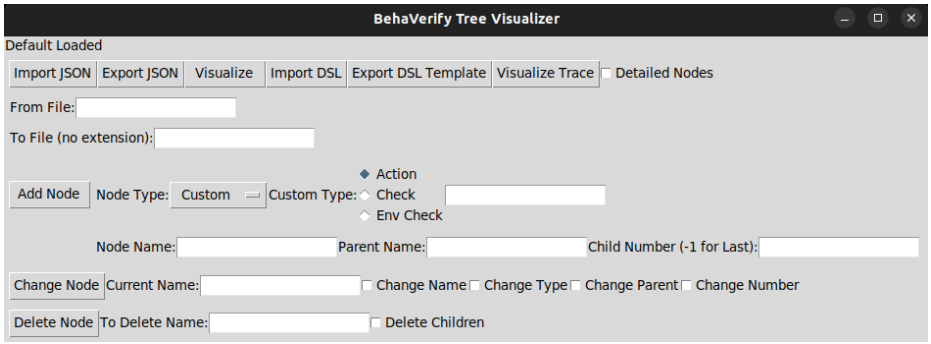

Fig. 1. The BehaVerify GUI

The BehaVerify GUI (Figure 1) is a lightweight GUI for the visualization of Behavior Trees. It is built in python and utilizes tkinter for the interface and graphiz for the creation of images.

The GUI can be run by the following command: python3 behaverify_gui.py

The remainder of this document will go over

- Tree Importing and Exporting (2)
- Tree Editing (3)
- The JSON Format (4)
- Tree Visualization (5)
- Trace Visualization (6)

2 Tree Importing and Exporting

There are 2 formats that can be imported and exported to: JSON and DSL. The JSON format is meant to be very general, while the BehaVerify DSL is a format specific to BehaVerify.

JSON Importing To import an existing tree in a JSON file f, type the file path of f in the 'From File' box and press the 'Import JSON' button (see Figure 1). The file path can either be absolute or relative.

JSON Exporting To export the current tree to a JSON file f, type the file path of f in the 'To File (no extension)' box and press the 'Export JSON' button (see Figure 1). The file path can either be absolute or relative. DO NOT INCLUDE THE FILE EXTENSION.

DSL Importing To import an existing tree in a DSL file f, type the file path of f in the 'From File' box and press the 'Import DSL' button (see Figure 1). The file path can either be absolute or relative.

DSL Exporting To export the current tree to a DSL file f, type the file path of f in the 'To File (no extension)' box and press the 'Export DSL Template' button (see Figure 1). The file path can either be absolute or relative. DO NOT INCLUDE THE FILE EXTENSION. This will create a fairly bare-bones DSL file for use with BehaVerify. BehaVerify should be capable of using the created file as is, but without additional information about variables, nodes, and specifications, the utility will be limited.

3 Tree Editing

You may edit the currently loaded tree by adding, changing, or deleting nodes.

Adding Nodes To add a node to the tree, press the 'Add Node' button (see Figure 1). If the Node Type Menu is set to custom, then a leaf node of the custom type will be added. Otherwise, the type selected in the Node Type menu will be used. The node will be named according to the 'Node Name' field. If a node with the specified name already exists, the new node will automatically be renamed. The new node must have a valid parent, as specified by the 'Parent Name' field. Note that composite nodes are always valid parents, decorators are valid parents only if they have no children, and leaf nodes are never valid parents. The 'Child Number' field allows you to specify where in the Parent's list of children this node will be inserted.

Changing Nodes To change a node in the tree, press the 'Change Node' button. The 'Current Name' must refer to an existing node in the tree. The options 'Change Name', 'Change Type', 'Change Parent', and 'Change Number' will determine what to change about the node. The change will pull the values from the fields as described in Adding Nodes.

Deleting Nodes To delete a node in the tree, press the 'Delete Node' button. The 'To Delete Name' must refer to an existing node in the tree. If you also check the 'Delete Children' option, all the children will be deleted as well.

NOTE: the GUI will do some basic checking and refuse to delete or change nodes if they result in trees that do not make sense (contain cycles, decorators with multiple children, etc).

4 JSON Format

Listing 1. Example JSON File

```
{
    "type": "selector",
    "children": [
        {
            "type": "selector",
            "children": [
                {"type": "checkSome"},
                {"type": "doSome"}
            ]
        },
        {"type": "a"}
    ]
}
```

A JSON object is an unordered set of name/value pairs. We consider a JSON object j to be valid for use with the BehaVerify GUI if it obeys the following properties:

- j contains a name/value pair such that the name is 'type' and the value v is a string such that each character c in v belongs [a-z] or [A-Z] or [0-9] or _.
- If v is the value associated with the name 'type' in j, then let $v' = lower(trim(v))$. If v' in {'selector', 'fallback', 'sequence', 'parallel', 'inverter', 'X_is_Y'} where X and Y are success, failure, or running and not equal, then j contains a name/value pair such that the name is 'children' and the value jc is an array of JSON objects all of which are valid for use with the BehaVerify GUI.

No additional information is required or used by the BehaVerify GUI when loading or saving to the JSON format. The BehaVerify GUI internally utilizes a different format.

5 Tree Visualization

To visualize the current tree, enter a file path in the 'To File (no extension)' field and press the 'Visualize' button (see Figure 1). The GUI will save and open the resulting .png file. If the 'Detailed Nodes' option is checked, then leaf nodes will contain additional information, marking them as Action, Check, or Environment Check nodes. Additionally, since Leaf Nodes can be reused, but names must be unique, Leaf Nodes will indicate both their name and precise type when using 'Detailed Nodes'. Note: 'Detailed Nodes' is mostly useful when visualizing a tree loaded from the DSL or created by hand.

6 Trace Visualization

The GUI can also be used to create a visualization of a verbose trace generated by nuXmv (must use the -v file when printing the trace). The trace should be stored in a text file. The trace can be the result of simulation or a counterexample of a specification violation.

To generate the trace visualization, enter a *folder* path in the 'To file (no extension)' field, enter the path to the trace file in the 'From File' field, and press the 'Visualize Trace' button. For each state in the trace, an image will be generated at the specified location. Each image should contain the tree with

additional ‘nodes’ indicating the return status and what variables, if any, were modified by the node.

Limitations There are several limitations with trace visualization.

- DEFINE Variable are not associated with nodes, so even if a node causes their value to change, they will not be attached.
- If a node changes multiple variables, they will be ordered alphabetically, rather than in the order they were changed in.
- If a node writes changes to the environment which take place after the tick finishes, the variable will still be attached to the node.
- Stage based opitmizations removes certain stages which are ‘unused’, making parsing the information more difficult.