1 Getting Started

This section will provide a quick explanation for how to get started with BehaVerify. To do this, we will walk through the installation process and several examples.

1.1 Installation

BehaVerify can be installed locally or used through Docker. For now, we will use BehaVerify through Docker. Local installation instructions can be found at TODO: UPDATE THIS. Regardless of which method is used, you will need to acquire a nuXmv executable. We cannot distribute it with our software; however, if pointed to the appropriate website out scripts will automatically download it and move it to the appropriate locations. These instructions assume you have git, Python3, and docker installed. The general procedure for installation follows these steps

1. Ensure you can run docker without sudo.
2. Setup Python virtual environment.
3. Download and installation of BehaVerify.

1.1.1 Prerequisites — Docker without sudo As mentioned, the instructions assume you have git, Python3, and docker installed. Additionally, please follow the instructions at [1].T This will enable you to run docker without sudo. This is because docker-py, the Python package for interacting with docker requires the ability to run docker without sudo.

1.1.2 Python Virtual Environment First, we will create a virtual environment and install docker and Py-Trees.

```
python3 -m venv /some/path/here
/some/path/here/bin/python3 -m pip install docker py_trees
```

Please replace 'some/path/here' with a suitable location for your python virtual environment. From here on out, commands using python3 will assume that either this virtual environment has been activated or is being called directly.

1.2 Download and installation of BehaVerify

Next, we will download and install BehaVerify.

```
git clone https://github.com/verivital/behaverify
cd behaverify
python3 docker/BehaVerify/python_script/reinstall.py \
  docker/BehaVerify/ \
  nuxmv.fbk.eu/theme/download.php?file=nuXmv-2.1.0-linux64.tar.xz
```

[1] https://docs.docker.com/engine/install/linux-postinstall/

Again, please make sure you have either activated the appropriate python virtual environment or replace the 'python3' with the correct direct call. This command clones the BehaVerify repository, moves you into the top level of the repository, and then calls the reinstallation script. The reinstallation script will delete any previous installations; this means any images named behaverify_docker_img will be deleted by this action, as will containers names behaverify_docker. It will then create the appropriate docker image and container from the Dockerfile at 'behaverify/docker/BehaVerify/Dockerfile'. Finally, the container will be started and it will download nuXmv from the provided url, extract it, and ensure it is runnable. See [2] for more details on nuXmv.

We will now test the installation through the following steps

```
python3 docker/BehaVerify/python_script/generate.py \
  demos/GCD/GCD.tree - ./test nuXmv ltl
tar -xf ./test.tar
cat ./output/output/nuxmv_ltl_results.txt
```

The last line should list something like '–specification F (G . . .) is true'.

1.3 Runnning BehaVerify

Assuming the installation test worked correctly, then you've actually already run BehaVerify! When running BehaVerify through docker, we use generate.py to interface with the docker. TODO: ADD LINK HERE The general form of this is

```
python3 PATH/TO/TREE PATH/TO/NN/DIR \
  PATH/TO/OUTPUT GENERATE EXECUTE
```

We include a more complete break down of all the options for how to call generate.py later, but for now we'll stick with the basic arguments, all of which are required

1. **PATH/TO/TREE -** This is the path to the .tree file containing the Behavior Tree specification. It can be a relative or absolute path.
2. **PATH/TO/NN/DIR -** This is the path to the directory containing neural networks used by the tree, if any. If none, should be -. It can be relative or absolute. If multiple networks are being used, all must be in the same directory, though they need not be on the same level. Additionally, the directory should be in the same directory as the .tree file.
3. **PATH/TO/OUTPUT -** This is the path to where the output should be placed. (.tar will be appended to the path provided).
4. **GENERATE -** This is what type of 'code' BehaVerify will generate. It must be one of nuXmv, Python, or Haskell (case sensitive). For nuXmv, a .smv file will be generated which can be used as input for nuXmv; this will allow for verification of LTL, CTL, and Invariant properties. For Python, a series of .py files will be generated; these will allow the user to run the

[2] https://nuxmv.fbk.eu

BT using Python and Py-Trees. Similarly for Haskell, but with different file types.

5. **EXECUTE -** This is what BehaVerify should 'do' with what is generated. Must be one of ltl, ctl, invar, generate, simulate.
   - **ltl** check LTL specifications in the .tree (nuXmv only).
   - **ctl** check CTL specifications in the .tree (nuXmv only).
   - **invar** check INVAR specifications in the .tree (nuXmv only).
   - **generate** just create the relevant code/model.
   - **simulate** generate and run the code/model (nuXmv and python).

So, looking at the command we used during the test, we told BehaVerify to use GCD.tree, use no neural networks, output to ./test.tar, create nuXmv code, and verify the LTL specifications. Note that the 'ltl, ctl, and invar' options are only available when generating nuXmv.

```
python3 docker/BehaVerify/python_script/generate.py \
  ./demos/GCD/GCD.tree - ./test Python simulate
tar -xf ./test.tar
cat ./output/output/python_simulation.txt
```

This command generated Python code and ran it within the docker. The Python code can still be accessed in ./output/app. To run it locally, pleaes do the following

```
python3 ./output/app/GCD_runner.py
```

2 DSL Introduction

Now that we've installed and run BehaVerify, it's time to start creating Behavior Trees (*BT*). BehaVerify uses a custom Domain Specific Language (*DSL*) to specify *BT*s. A detailed breakdown of the *DSL* can be found at TODO: INSERT REFERENCE HERE. In order to introduce the *DSL* and provide an introduction to *BT*s, we will consider a simple example where we compute the greatest common divisor (*GCD*) of two numbers. Formally, this means that given two integers, a and b, we want to find $c \in \mathbb{Z}$ s.t. $\frac{a}{c} \in \mathbb{Z} \wedge \frac{b}{c} \in \mathbb{Z}$. Furthermore, $\forall d \in \mathbb{Z}, d \leq c \vee \frac{a}{d} \notin \mathbb{Z} \vee \frac{b}{d} \notin \mathbb{Z}$. (Here $\mathbb{Z}$ represents the set of all integers). To accomplish this task, we will utilize the Euclidean algorithm, presented as pseudo code.

```
 def gcd(a, b):
   if b >= a:
     temp = a
     a = b
     b = temp
   if a % b == 0:
     return b
   return gcd(b, a % b)
```

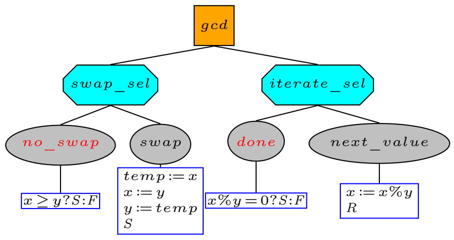

Fig. 1: The BT for the GCD example. We use the form $p?1{:}2$ to mean if p then 1 else 2. $:=$ represents assignment. S means success, F means failure, and R means running; these are explained in detail within the text.

Here % stands for modulo, thus $a\%b$ yields the remainder when dividing a by b. The BT shown in Figure 1 accomplishes this task. To understand how this works, we must understand the following rules:

1. Nodes return S, F, or R (success, failure, or running).
2. Children are ordered from left to right.
3. Sequence nodes (rectangles) try to execute their children in sequence with early termination conditions if a child returns F (failure) or R (running).
4. Selector nodes (octagons) try to 'select' a child that succeeded. As such, they terminate when a child returns S (success) or R (running).

Let's walk through how this works. We're going to start with $a = 14$ and $b = 21$. We initialize x to a and y to b.

1. We start at the root (gcd), which calls its left most child, $swap\_sel$.
2. $swap\_sel$ calls its left most child, no_swap.
3. no_swap executes its code; $x \geq y?S{:}F$ returns $Failure$, because 14 is less than 21.
4. Because no_swap returned F, $swap\_sel$ (a selector) calls the next child, $swap$.
5. $swap$ executes its code; at the end of this, we've swaped the values of x and y. Furthermore, it returns S.
6. Because $swap$ returned S, $swap\_sel$ (a selector) returns S.
7. Because $swap\_sel$ returned S, gcd (a sequence) continues the sequence by calling the next child, $iterate\_sel$.
8. $iterate\_sel$ calls its elft most child, $done$.
9. $done$ executes its code; 21 mod 14 is clearly not 0 so it returns F.
10. Because $done$ returned F, $iterate\_sel$ (a seleector) calls the next child $next\_value$.
11. $next\_value$ executes its code; x is now equal to 7. Furthermore, it returns R.
12. Because $next\_value$ returned R, $iterate\_sel$ returns R.
13. Because $\_sel$ returned R, gcd returns R.
14. Eventually, we start the next cycle (x is 7, y is 14).

15. *gcd* is where we start, again, and it begins by calling its left most child, *swap_sel*.
16. *swap_sel* calls its left most child, *no_swap*.
17. *no_swap* executes its code and returns F.
18. Because *no_swap* returned F, *swap_sel* (a selector) calls the next child, *swap*.
19. *swap* executes its code; at the end of this, we've swaped the values of x and y. Furthermore, it returns S.
20. Because *swap* returned S, *swap_sel* (a selector) returns S.
21. Because *swap_sel* returned S, *gcd* (a sequence) continues the sequence by calling the next child, *iterate_sel*.
22. *iterate_sel* calls its left most child, *done*.
23. *done* executes its code; since 14 mod 7 is 0, it returns S.
24. Since *done* returned S, *iterate_sel* terminates early and returns S. **We do not run *next_value* this time!**
25. Since *iterate_sel* returned S, *gcd* returns S.
26. And, indeed, the value of y is now 7, which is the greatest common divisor of 21 and 14.

See TODO: REFERENCE HERE for a glossary of all terms used in BehaVerify and *BT*s. For now, we will introduce terms as we use them. We implemented this tree for you at [3]. This example is fully commented and explains how elements of the *DSL* works. In fact, there's a series of examples for the *GCD* which can be found at [4].

1. **GCD.tree.** This is the most basic example. Very limited possible starting values, only checks that the final value is a common divisor, not the greatest common divisor.
2. **GCD_all_values.tree.** This example builds on the original example by allowing the initial values to be any integer between 1 and 100. It introduces the concept of loops.
3. **GCD_all_values_all_specs.tree.** Now we also check that the final value is the greatest common divisor (or at least, that there is no greater common divisor between 1 and 100).
4. **GCD_shrink_1.tree.** This example demonstrates that it is up to the user to determine how the tree should be structured. You can make very complex action nodes or have a vast tree; both are valid.
5. **GCD_shrink_2.tree.** As with the other shrink example, but even more extreme. This is a one node tree.
6. **GCD_array.tree.** Now we will find the *GCD* of more than two numbers. To accomplish this, we introduce arrays. This first attempt is very inefficient, and might not verify.
7. **GCD_array_fast.tree.** A more efficient implementation using arrays.

We suggest going through these in order to learn about the various aspects of the *DSL*.

[3] https://github.com/verivital/behaverify/demos/GCD/GCD.tree
[4] https://github.com/verivital/behaverify/demos/GCD

2.1 Neural Networks

So far, our examples have ignored Neural Networks. We are now going to change that. TODO: ADD DETAILED SCENARIO DESCRIPTIONS HERE.

```
python3 docker/BehaVerify/python_script/generate.py \
  demos/moving_target/moving_target.tree \
  demos/moving_target/networks/ \
  ./moving_output nuXmv ctl
tar -xf moving_output.tar
cat output/output/nuxmv_ctl_results.txt
```

Here the output will depend on the values of the constants declared in 'moving_target.tree'. The default values will result in the specification being true. If the value of 'movement_cooldown' is instead set to 5, it will be false. The 'networks' in this case are 'regression networks'; they provide an exact value rather than a classification. However, they are also simple in the extreme, hand coded, and not very 'real'. They are used to enforce a lawnmower pattern search; in practice this means one network outputs if we should go 'across' the map while the other determines if we should change which row we are on. Furthermore, because it is a regression network, though very simple, we must use a 'table'. That is to say, we run the network for all possible inputs, record the outputs, and encode that into nuXmv.

For a more practical example, we will consider the example in 'demos/-grid_net'. Here you will have to edit 'template.tree' or 'template_hole.tree' to select your network and the exact parameters to be used. For a start, we suggest editing 'template.tree' with the following changes

– **REPLACE_MODE -** Replace this with fixed (no quotation marks).
– **REPLACE_TOTAL -** Replace this with 100 (no quotation marks).
– **REPLACE_INT -** Replace this with 16 (no quotation marks).
– **REPLACE_FLOAT -** Replace this with 44 (no quotation marks).
– **REPLACE_SOURCE -** Replace this with './networks/1000__064_1.onnx' (YES quotation marks).

See test.tree for an example of how the replacements should look. template.tree includes information on what each of these parameters does. TODO: ADD FORMAL WRITE UP HERE.

Options As mentioned, you have many options here. For instance, you have 10 neural networks to choose from. They are named using the following form 'A__B_C.onnx', where A is the accuracy of the network, B is the number of neurons per hidden layer, and C is the number of hidden layers. Should you choose a network with less than an accuracy of 1, verification tasks will fail and produce counterexamples. Additionally, if you choose one of the larger networks, the verification task will be slower (probably). Finally, it is important to note that if you decrease the total size or float size too much, you will likely suffer overflow errors causing verification to fail. Using test.tree as an example, we would run

```
python3 docker/BehaVerify/python_script/generate.py \
  demos/grid_net/test.tree \
  demos/grid_net/networks/ \
  ./grid_net nuXmv ctl
tar -xf grid_net.tar
cat output/output/nuxmv_ctl_results.txt
```

Note that this might take some time, as it is significantly more complex than most of the other examples so far.