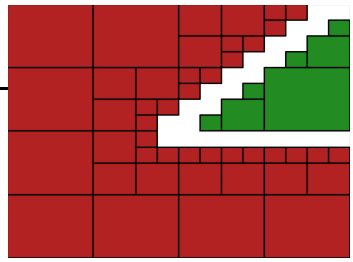
PaSyPy Documentation



This python-based tool uses parameter synthesis to find safe and unsafe regions of the parameter space.

Table of contents

1. Explanation of all files

__main__.py
The main entry point for starting this tool. Prepares the logs and starts the main loop of the GUI.

__init__.py
Creates this package.

pasypy.py
The Main module that processes and computes the queue. When trying to find safe and unsafe regions of the parameter space on an interval there are three different outcomes:

- If the viewed area is **sat** and the negated constraint is **unsat**, the area is considered safe (green).
- If the viewed area is **unsat**, the area is considered unsafe (red).
- If the viewed area is both **sat** and the negated constraint is **sat**, the area contains both safe and unsafe regions (white).

visualize.py
Visualizes all areas by drawing safe (green) and unsafe (red) regions with `matplotlib` library.

gui.py
Builds the complete graphical user interface (GUI) with `tkinter` library.

variables.py
Contains all variables used across different modules.

constraints_parser.py
Handles the parsing of the given constraints. The constraints can either come from an `SMT-LIB` file or directly from the text field.

splitting_heuristic.py
If the viewed area is both **sat** and the negated constraint is **sat**, the area contains both safe and unsafe regions (white). The region then has to be split into smaller sub-regions. For this there are different splitting heuristics available:

- **Default:** Splits every region in exactly $2^{Dimensions}$ equally large new regions.
- **Simple:** Splits every region into two equally large new regions changing the axis after every split.
- **Extended:** Automatically tries to find the most suitable candidate on each axis. Operates similar to the *Default* heuristic.
- **Random:** Operates similar to the *Default* heuristic but uses random candidates on each axis for splitting.

sampling.py
Sampling is used to generate more suitable splitting candidates and optimize performance. **Pre-Sampling** is used prior to computation and **Sampling** in between each split.

area_calculation.py
Calculates safe, unsafe and unknown areas in percentage.

settings.py
Contains adjustable settings:

- **Pre-Sampling:** Optimizes performance prior to computation by generating better candidates.
- **Sampling:** Optimizes performance prior to each splitting step by generating better candidates.
- **Hyperplane:** Approximation curve (hyperplane) with safe and unsafe regions as training sets.
- **White Boxes:** Also draws unknown (white) region borders.
- **Hatch Pattern:** Pattern for better differentiation between safe and unsafe regions.
- **Colorblind:** In case of color blindness, changes colors from green/red to blue/yellow.
- **Skip Visual:** Completely skips the visualization part and only executes the computation.

logger.py
Creates log files containing safe and unsafe regions.

timer.py
Calculates time for computation and visualization.

color.py
Contains different colors for safe and unsafe regions.

test
Contains unit tests.

images
Contains all icons used by this tool.

Documentation
Contains documentation.

requirements.txt
Includes all dependencies needed by this tool. Can be invoked with `pip install -r requirements.txt`.

.gitignore
A file used by git to ignore selected files on committing.

.pylintrc
Configuration file for the static code analysis tool `Pylint`.

.coveragerc
Configuration file for the measuring code coverage tool `Coverage.py`.

CITATION.cff
The file used by GitHub to facilitate citing this repository.

LICENSE
A license file.

2. Platform Support

This tool works on every platform (**Windows**, **MacOS**, **Linux**) with full functionality.

3. Known challenges

- **.smt2** files usually contain a **set-logic** line, which helps the solver to apply the correct tactic. When setting the logic to quantifier-free non-linear real arithmetic (QF NRA), using existential quantifiers for the constraints produces an error on reading the file, although existential quantification is defined for the quantifier-free non-linear real arithmetic (QF NRA) logic. This seems like a general problem with the `z3 theorem prover` and its function to read **.smt2** files.
- In general, the Python interface of the `z3 theorem prover` sometimes has difficulties automatically select the correct theory solver needed for the specific problem. This often results in a timeout even on relatively easy constraints. As a solution multiple theory solvers are used in parallel in case the default one fails to process the given constraints. This does not have any effect on the performance or time needed to solve the problem.
- The performance of Python in comparison to hardware-related programming languages (f.e. C and C++) is very weak. One approach to benefit from the simplicity of Python and not dispense with high performance, is to integrate `Cython` an optimizing static compiler. This would give us the combined power of Python and the programming language C, which is well known for high performance due to hardware-related functionality to speed up the execution of our Python code.
- Performance is also highly dependant on the splitting heuristic and sampling regarding the computation step and array processing regarding the visualization step. This can be further worked on to achieve optimal results.
- While **Pre-Sampling** is really efficient and often leads to a huge performance boost, **Sampling** lacks efficiency and needs a rework.