COOL Model Checking

Getting Started

We provide a batteries included docker image cool-docker.tar.gz to enable easy replication of the benchmarking results reported in the paper 'Generic Model Checking for Modal Fixpoint Logics in COOL-MC'. The setup of our artifact is simply loading the docker container into docker by typing

```
docker load -i cool-docker.tar.gz
```

After this, the image can be run with

```
docker run -it cooldocker:2.5
```

which drops the user into a shell with all necessary executables in the path and the working directory being the root of the benchmarks. The benchmark folder contains an additional Readme file with further details on the indvidual benchmarks.

Step-by-Step Instructions for replicating the results from the paper

This artifact contains all the benchmarking scripts that were used to measure the performance of the model checker COOL-MC and create all the figures and tables (showing runtimes and graph sizes) from the paper.

After starting the docker container as described in the previous section, the benchmarks can be executed in any desired order. Error messages or warnings that might show up during the execution of the scripts can safely be ignored.

The default timeout of all the benchmarks can be changed by setting the `TIMEOUT` environment variable. Typing

```
export TIMEOUT=1s
```

for example allows for a quick test of whether the benchmarking process works; using a timeout of 1s, the script takes about 20 minutes to complete, but produces output only for a limited set of experiments.

- To execute the parity game benchmarks (replicating experiments reported on in Figures 1 through 5), simply type

```
cd parity-mod
```

and then

```
./bench.py
```

The script will execute the experiments and populate the `results` and `stats` sub-folders with the measurements. This experiment takes about 4 hours with the default timeout.

- For the experiment on the alternating-time mu-calculus (modulo games, results shown in Figures 6 and 7), type

`cd modulos`

followed by

`./genModulos.sh`.

Afterwards the `models` and `formulas` sub-folders contain the model checking instances. Run

`benchCOOL.sh`

and/or

`benchMCMAS.sh`

to perform the actual model checking and populate the `results` sub-folder with the mesurements for COOL and/or MCMAS.

The scripts create CSV files in individual `results` and `stats` folders, containing the measured values. Within the CSV files, the `size` column corresponds to the size parameter of our parity game experiments in the paper and the `mean` value is the average of all the executions of that single experiment with the given input (at least 3 runs are performed, more if the runs are fast). For the modulo game, the column `parameter_moves` takes the role of the size parameter. Apart from that, the created CVS files contain additional measurements like the standard deviation `stddev` or the minimum and maximum runtime as well as the exact command that has been executed by hyperfine.

Fig. 1 is plotted from the `laddergame` results generated as part of the `parity-mod` benchmarks Fig. 2 is plotted from the `langincl` results generated as part of the `parity-mod` benchmarks Fig. 3 is plotted from the `towersofhanoi` results and stats generated as part of the `parity-mod` benchmarks Fig. 4 is plotted from the `easy-hanoi` results and stats generated as part of the `parity-mod` benchmarks Fig. 5 presents additional results from the `langincl` and `easy-hanoi` results generated as part of the `parity-mod` benchmarks Fig. 6 and 7 are plotted from the results of the `modulos` benchmarks

(The additional Figures 8 and 9 found in the appendix of the paper are plotted from the `cliquegame' and`jurdzinskigame`results generated as part of the`parity-mod‘ benchmarks)

The names of the CSV files consist of the experiment name given above, followed by a value identifying the logic, and finally and indentifier for the tool,

The relevant parameters for the tool are `COOL`, corresponding to the local solver (indicated by subscript ‘l’ in the figures in the paper) and `PG`, corresponding to the game-based solver (indicated by subscript ‘g’ in the figures in the paper) and, for the modulos game, `MCMAS`.

Logics are identified as follows:

- `K` - standard mu-calculus
- `Monotone` - monotone mu-calculus
- `CL` - alternating-time mu-calculus
- `GML` - graded mu-calculus
- `PML` - probabilistic mu-calculus.

So to verify the runtimes reported for instance in Figure 1 (ladder game runtimes), run the `parity-mod` script and afterwards inspect the `mean` columns in the files

`laddergame-K-COOL.csv` for plot `standard_l` `laddergame-K-PG.csv` for plot `standard_g` `laddergame-PML-COOL.csv` for plot `probabilistic_l` `laddergame-PML-PG.csv` for plot `probabilistic_g` `laddergame-GML-COOL.csv` for plot `graded_l` `laddergame-GML-PG.csv` for plot `graded_g`

The results corresponding to the table in Figure 5 can be found in the `stats` sub-folder, within the files `langincl-Monotone.csv`, `easy-hanoi-K.csv` and `easy-hanoi-GML.csv`, respectively. Columns in the files and the table in the paper are related as follows:

- column `size` corresponds to column `parameter` in the table,
- column `full` of lines with first value `COOL` corresponds `full graph` in the table,
- column `visited` of lines with first value `COOL` corresponds `lazy graph` in the table,
- column `full` of lines with first value `PG` corresponds `game size` in the table.

All our experiments are deterministic so all reported results should be reproducible up to scaling due to different machine performance.

The results in the paper have been measured on a machine with an AMD Ryzen 7 2700 CPU and 32GB of RAM; the number of cores is not that important as everything runs mostly in a single thread.

Reuse

To enable experiments and usage beyond the results reported in the paper, the artifact also contains statically linked executables of the model checker COOL-MC. We briefly describe how the tool can be used for model checking; more documentation and the source code can be found in the modcheck branch of the repository.

To solve a single instance of the model checking problem by reduction to parity games (using PGSolver for parity game solving) run

```
cool-coalg modcheck LOGIC -m MODEL -p s0 --gameSolver pgsolver
<<< FORMULA
```

To use the local solver instead, use

```
cool-coalg modcheck LOGIC -m MODEL -p s0 --gameSolver cool <<<
FORMULA
```

where LOGIC selects one of the supported logics from the following list.

- `K` - standard mu-calculus
- `KD` - standard mu-calculus interpreted over serial models
- `CL` - alternating-time mu-calculus
- `GML` - graded mu-calculus
- `PML_gen` and `PML_react`- (generative and reactive variants of the) probabilistic mu-calculus.

MODEL is a text file containing the description of a model; more information on the structure of models in COOL is provided below.

FORMULA is formula belonging to the selected logic, given in COOL syntax (see https://git8.cs.fau.de/software/cool)

Furthermore, we also include an auxilliary experiment, called `castles game` on that we do not report in the paper. To execute this experiment, run

```
cd castles
```

followed by

```
./genCastles.sh
```

which generates a series of models and formulas in the sub-folders `models` and `formulas`. Run `benchCOOL.sh` and/or `benchMCMAS.sh` to populate the `results` sub-folder with the mesurements for COOL and/or MCMAS on this experiment.

Model Structure

COOL-MC expects a model as input. An easy explorative way to understand the syntax for models is to run `cool-model-gen -frame K`. This generates a random model for the standard mu-calculus (that is, a transition system).

The syntax for the individual parts of models is as follows:

- Functions are given as `[ domain_element: codomain_element, domain_element: codomain_element, ...]`
- Sets are given as `{ element, element, element, ... }`
- Tuples are given as `( element, element, element, ... )`
- Rationals are given as `numerator/nominator`
- Integers and IDs (worlds, atoms, relations) are just written as is

All models are functions at the top level where the domain is the set of worlds in the model, and the codomain is the functor specific structure of each world. In each functor this structure is equal at the outer level as this is a tuple where the first element is the set of atoms satisfied at the given world. The second tuple element is the functor specific structure:

- For `K` we have a function from relation label to a set of successor worlds.
- For `KD` we have the same as for `K` but the set of successors must always not be empty.
- For `CL` with x many agents, have a (x+1) element tuple were the initial x elements are integers assigning the moves for each agent and the last element is a function from x element integer tuples of combinations of the agents' individual moves to a single successor world.
- For `GML`, we have a function from relation label to a function assigning integer weights to the worlds (worlds with weight 0 can be left out).
- For `PML_react` we have a function from relation label to a function assigning rational probabilities to the worlds (worlds with probability 0 can be left out). The sum of all worlds per relation label should be one.
- For `PML_gen` we have a function from tuples of relation label and a world to a rational probabilities (tuples with probability 0 can be left out). The sum of all such successor tuples should be one.