BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Static Analysis and Verification 2020/2021

http://fit.vut.cz

The Probabilistic Model Checker STORM

Šimon Stupinský, xstupi00@fit.vut.cz,
Faculty of Information Technology, Brno University of Technology

Abstract
In this paper, we introduce the probabilistic model checker STORM. It features the analysis of *discrete*-time and *continuous*-time variants of both *Markov chains* (MC) and *Markov decision processes* (MDP). It supports the *PRISM* and *JANI* modelling languages for Markov models. Further, STORM supports probabilistic programs, dynamic fault trees, generalised stochastic Petri nets and the probabilistic guarded command language. In Chapter 1 we introduce the fundamental description of STORM and its features. In Chapter 2 we will demonstrate behaviour of STORM on a extensive benchmark, and we will compare our achieved results with the referent results. Chapter 3 introduce our experiments with STORM and investigate the impacts of the model size to the model checking process.

Keywords: model checking — probability — rewards — engines — formal model —- formal specification — benchmark — binary tree — die

Supplementary Material: Web page — Repository

1. Introduction

STORM [8] is an extensible toolbox for analysing the systems involving probabilistic or random phenomena. Given an input formal model description and formula specification, it can determine whether the specification is holding within the model. It has been designed with performance and modularity in mind. Therefore, the main aim of the STORM is to provide a performant, easy-extendible platform, that supply various probabilistic model checking algorithms. STORM provides reusable models to promptly implementation of the new functionality for *probabilistic* model checking, and also a range of engines that make use of various back-end solvers. The core of the STORM is aiming for low memory requirements and high overall performance. On the highest level of this core are available the different sophisticated techniques, and that: *probabilistic program* verification, generation of *probabilistic counterexamples* (CEXs), *parameter synthesis*, *permissive scheduler* generation, and *dynamic fault trees* analysis [10]. Moreover, STORM also supports the development of new techniques in a variety of fields, since it accepts the wide range of the input languages. STORM can by compiled from source (Mac OS and Linux), installed via Homebrew (Mac OS), or used from a Docker container – all platforms, but slightly reduced performance.

Model Checking [3]. As have been said, the formal model of the system and the formal specification are the inputs of the model checker. It processes these inputs in some defined way and subsequently returns one from three possible outputs. When the property holds in the given system, then the output is SAT, when the property is violated the output is UNSAT. The last possible output is that the model checker ran out of computational resources. The complete overview of this process is shown in Figure 1. However, many systems that give rise to models involving *probabilistic* aspects, require a slightly different approach. These systems involve random phenomena or other forms of behaviour that can be approximated by randomisation. All such systems are naturally translating to *Markov* models, and they are as input for *probabilistic* model checking. It extends the traditional model checking described above, with techniques and tools for the analysis of all these systems.

Model types. One of two inputs of the *model checking* is the formal description of the system model.

STORM supports both *discrete*-time and *continuous*-time *Markov* models and also *non-deterministic* variants of thereof. According to this classification, it supports the following concrete model types: *discrete*-time *Markov* chains (DTMC), *continuous*-time *Markov* chains (CTMCs), *Markov* automata (MA) and *Markov* decision processes (MDPs). Furthermore, all listed model types can be enhanced with *reward* models.

	discrete-time	**continuous-time**
deterministic	DTMCs	CTMCs
non-deterministic	MDPs	MA

Table 1. Model types supported by STORM.

Modelling languages. When modelling a probabilistic system, we may describe it as, for example, a *Markov* chain. However, this approach is not applicable in practice due to the state space explosion problem. Therefore, we usually describe the system using a high-level programming language. Within the STORM can be specified the above-mentioned model types with using the several modelling languages. The PRISM [1] and JANI [5] input languages are supported, in an attempt to unify the probabilistic modelling language landscape. In compare to another model checkers, STORM also supports the modelling every *generalised stochastic Petri net*. This can be realised via encoding the *Petri net* in JANI or via the *dedicated model builder*. Finally, STORM also features the modelling of *probabilistic programs* in the *conditional probabilistic guarded command language*.

Properties. The second input to the process of *model checking* is the *property specification*. STORM focuses on reachability queries, respectively in more precise on probabilistic branching-time logics, i.e. *PCTL* [6] and *CSL* [2], for both *discrete*-time and *continuous*-time models. It supports the *reward* extensions – expected rewards, long-run average rewards – of these logics, to enable the treatment of *reward-objectives*. Additionally, within STORM are also supported the *conditional properties* and *conditional rewards*, because they could express the interesting properties [4].

Engines. In dependence on the characteristics of the input model, STORM uses various representations to store the probabilistic models in the memory, that differ between itself in the efficiency. When the models at the input are small and moderately size, the most suitable representation is the *sparse matrices*. The second internal representation within STORM is *multi-terminal decision diagrams* (MTBDDs), which are able to represent systems that have models with gigantic size. STORM features the several engines built around these two in-memory representations, which ensure the verification of a broader class of input models. Both first *sparse* and the *learning* engine purely use the *sparse matrix*-based representation. Two other engines, called *hybrid* and *dd*, use *MTBDDs* as their primary representation. The *hybrid* engine does not use exclusively only *decision diagrams*, as *dd* engine does, and it uses the *sparse matrices* representation for operation deemed more suitable on this data format.

Last but not least, STORM provides the *automatic* engine, which try to determine the reasonable settings for specific running of STORM. This engine will be use in the replication of the experiments and therefore we briefly introduce it. Its currently implementation uses a *decision tree* with specific number of leafs and specific value of height. The *decision tree* has been generated with the tool `scikit-learn` using training data from experiments data-set, which will be near described in the following section. In trying to avoid over-fitting of the selection, the *automatic* engine choice only from the following options. In dependence on the properties of the input instance, the *hybrid* or *sparse* engine can be selected, potentially also *sparse* engine with exact model checking and rational arithmetic. The last engine is *dd-to-parse* with symbolic bi-simulation minimisation, which is similar as *dd* engine, but performs the translation independent of the property.

Solvers. STORM features the various *solvers* built within its own infrastructure. For instance, there are available solvers for sets of *linear* or *Bellman* equations, *mixed-integer linear programming* (MILP) and *satisfiability modulo theories* (SMT) solving. The availability of these solvers provides a coherent and comfortable approach to the assignments commonly involved in the *probabilistic* model checking. Furthermore, the dedicated state-of-the-art high-performance libraries – e.g. *Eigen*, *CUDD*, *Z3*, *MathSAT*, *Gurobi*, etc. – can be used in this way, for such tasks of the model checking. STORM also offers the interface to easy implementation of new solver functionality, which can be realised without knowledge about the global code base. The individual implemented solvers are used in the many STORM components. For instance, *equation solvers* are commonly used to answer the standard queries of the verification process. From between the mentioned modules, the *generator of the counterexamples* and *permissive scheduler generation* use the following solvers: *SMT* and *MILP*.

Parametric models and exact arithmetic. STORM uses next dedicated library *CARL* to represent the

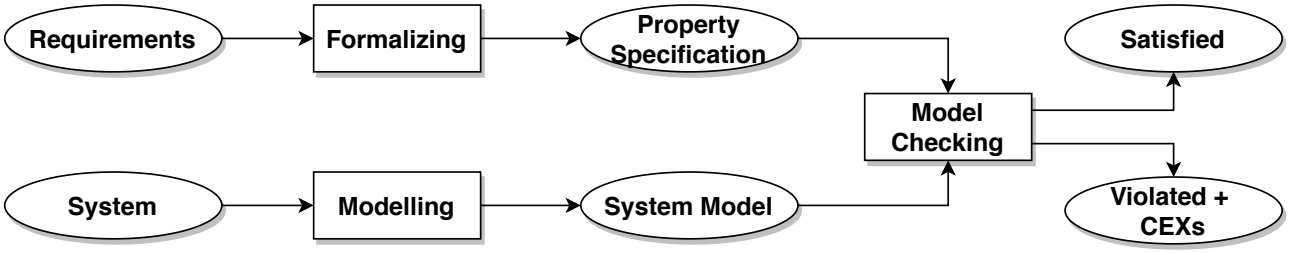


Figure 1. Overview of the model checking approach.

rational functions, and for analysing the *parametric* discrete-time models uses the novel algorithms. Thanks to these features, it significantly exceeds existing tools dealing with these areas. In addition, `STORM` does not use the floating-point arithmetic and therefore it is able to compute exact solution for these models.

Usage. `STORM` provides three different interfaces for usage, depending on the requirements of the use. The typical end-users can use the first of them, which is available via the *command-line* interface. It provides several binaries that feature the specialised access to the available configurations for different tasks. For instance, `STORM-CONV` offers the features and settings related to the conversions between individual model files. The advanced users can utilise one of the many settings for tuning the performance. Both other interfaces are target to the developers, which implements new features within `STORM` or use it as back-end. The first of them, `C++ API`, provides performance-oriented and fine-grained access to `STORM` functionality. The second one, `Python API`, allows users to profit from high-performance implementations within `STORM` and supports rapid prototyping.

2. Evaluation Results

This section contains an evaluation of the model checker `STORM` based on the replication results from the available package [9]. We decided to replicate the results from this package since it is the latest data-set for the competitiveness of `STORM`. It targets the key functionalities of `STORM`, covers the whole set of supported model types and also all kinds of the properties specifications. We emphasise, that it groups the benchmarks from two last *QComp* conferences – *QComp* 2019 and 2020 – in which `STORM` took part.

2.1 Benchmark Setup

This data-set includes the set of *96* benchmark instances, that were selected from the *Quantitative Verification Benchmark Set* (QVBS) [7]. Each instance is a combination of the *formal model* description, the *parameter assignment*, and the single *property specification*. Most of the model descriptions are implemented in the `PRISM` language, but when such formalism is not available, then the model is built from the `JANI` description. The individual benchmarks have a unique identifier consisting of the short model name, parameter values, and the name of the property – e.g. `zenotravel.4-2-2.goal`. When running the benchmark is needed to select a way to invoke the tool (*configuration*) by a selecting the specific *engine* or enabling the specific *features*. Subsequently, based on the chosen *configuration*, the set of the *invocations* is constructed. An *invocation* is a sequence of the commands that produces the results of the single given benchmark by invoking `STORM`. Before the execution of the tool itself are sometimes required additional operations – e.g. conversions to other formalism – and therefore it is needed to consider the sequences of the commands.

2.2 Replication Setup

For each instance, the task is to solve the corresponding *model checking* query within a time limit of *30* minutes and a memory limit of *12* GB. The obtained results are compared to the reference results provided within *QVBS*, and when the difference between these results is greater than the given threshold, then the result of the instance is marked as *incorrect*. According to the instructions to replicate the original experiments, we have run the `Python` script `run.py` and follow up the displayed instructions we created a list of invocations we wanted to measure. Initially, the list of 7 available configurations was printed out and was required to select the configurations for experiments. We decided to select only one configuration, specifically the *automatic* engine, which tries to automatically selects reasonable setting for `STORM`. In the next selection, we decided to select all models that are available in the benchmark instances: *DTMC*, *CTMC*, *MDP* and *MA*. Subsequently, from the offered property specifications were selected all kinds available within the specific instances: e.g. *probability reachability* (39), *expected rewards* (20), *probability reachability time-bounded* (15), *expected time* (7), *steady-state probability* (4), etc. We obtained a total of 96 instances for performing the experiments with this chosen configuration, and the constructed invocation file can be seen here.

2.3 Evaluation results

All our experiments were run on the faculty machine `pcpluhackova2`, thus the machine with 8 cores of an Intel® Core™ i7-4770 CPU. In all experiments, the *wall-clock* run-time is measured, so it includes both *model building* and *model checking*. For our evaluation we consider `STORM` version 1.6.0. The origin experiments were run on 4 cores of an Intel® Xeon® Platinum 8160 Processor and considered the newest version 1.6.2 of `STORM`. The following table shows the comparisons of the summary results of our replication experiments and the original experiments performed in the original paper:

	Solved	**TO**	**MO**	**Incorrect**	**Fast$_{+1\%}$**	**Fast$_{+50\%}$**
#1	85	8	1	2	56	83
#2	84	3	7	2	40	78

Table 2. Summary of experiments results.

The first line (*#1*) shows the results of our experiments running with the `STORM 1.6.0` and the second line (*#2*) the original results from `STORM 1.6.2`. The first column (*Solved*) indicates how many of the 96 considered instances were correctly solved for both versions of `STORM`. The subsequent columns indicate the number of times the time-limit (*TO*), respectively memory-limit (*MO*), was exceeded. The fourth column (*Incorrect*) indicates the number of obtained incorrect results, which exists due to the imprecise floating-point or algorithms that do not guarantee *sound* results. We note, that the sum of these columns in both cases sums to 96. The last two columns show how often the running configuration was either the fastest among the all available configurations or only 1% (50%) slower than the fastest one, i.e., terminated within 101% (150%) of the fastest configuration.

The complete results are summarised in the Table, which shows the detail information about the concrete instance, such as model name and type, modelling language, parameters value, property name, etc. After clicking on the model name in the first column, detailed information and a description of the model and its properties will be displayed. After clicking on the measured times in two last columns are displayed the basic benchmark info, invocation command, execution data (run-time, relative error, etc.) and whole log message from the execution itself. Figure 2 shows the comparison of the specific run-times measured on the individual instances with both analysed versions of `STORM`. The chart includes the 82 instance pairs because we compare only relevant pairs with the marked time in both versions and we also omitted the two longest runs due to the chart clarity.

3. Experiments

In this section, we introduce own experiments with the tool `STORM`, which were produced within the frame of this paper. They target to the better familiarisation with the *modelling language*, *property specifications* and also with the impacts of the different engines at analysis of the same instance.

3.1 Case study

The experiments are connected with the well-known model, which simulates the system of the *fair dice* using only *fair coins*. This probabilistic program starting at the root vertex (state **s0**), one repeatedly tosses a coin. Every time *heads* appears, one takes the *upper* branch and when *tails* appears, the *lower* branch. This continues until the value of the die is decided, the whole process is showed at the binary tree below. The individual values at the tree edges represents the probability of the transition firing when the system is in the specific state. The constructed code in the `PRISM` modelling language for this probabilistic program is available here. When we want to prove the correctness of this probabilistic program, we have to show, that the probability of reaching the state $\mathbf{d = k}$, **for** $\mathbf{k = 1,\ldots,6}$ is equal to $\mathbf{1/6}$. Therefore, we construct the following *probability formula* to verifying the correctness within `STORM`: $\mathbf{P =?\ [F\ s = 7 \& d = k]}$ **for** $\mathbf{k = 1,\ldots,6}$. Another interesting property for research in this model can be the *expected time* of coin tosses. This property can be represented in `STORM` through the following formula: $\mathbf{R =?\ [F\ s = 7]}$. The described model, and both *probability* and *expected-reward* specifications will be the basis of our experiments. This model for simulating the fair dice using only fair coins represents the *optimal* variant. By the optimal, we mean in this case, that the *expected number* of coins tosses is the lowest possible, and the model with the lower expected number does not exists. Performing the verification of the *expected-reward* property within `STORM`, we find that the expected number of coin tosses is $\mathbf{11/3}$. We note, that the binary tree representing the constructed model has height equal to **3**.

3.2 Template generator

The whole process of the `STORM` *model checking* over the given *formal model* description and *property specification* can be divided into a few sub-processes. Initially, the *input parsing* of the model template is performed followed by the relevant *model construction* according to the given description. This phase constructs the specific model type with the concrete numbers of *states* and *transitions*, eventually *actions* when the

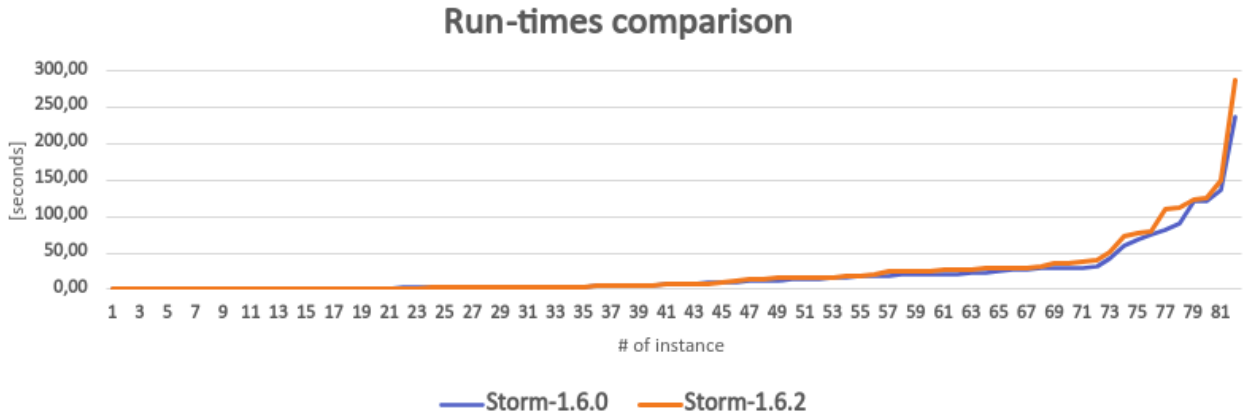
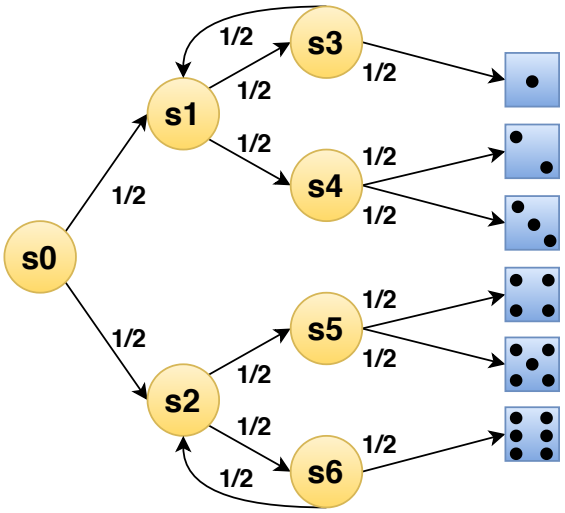
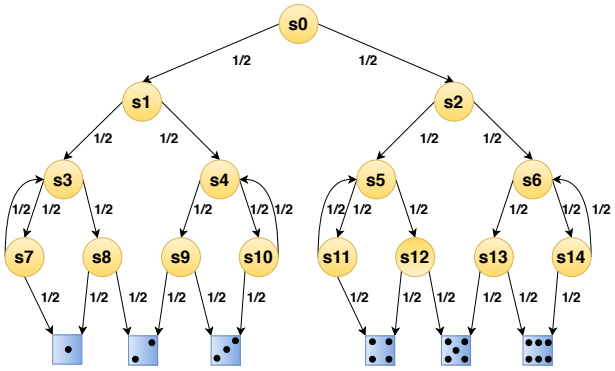


Figure 2. The comparison of the run-times of the individual benchmark instances between two versions of STORM. We can see that the run-time results are almost the identical at both versions, nevertheless, the experiments took place on different machines.



model is for example *MDP*. The last phase includes the itself *model checking* over the constructed model and given formula specification. Obviously, the run-time of all these phases depend on the complexity and size of the input model and also partially on the verified formula. Therefore, we decided to explore in our experiments how the different sizes of input models will affect the overall run-time of *model checking* process.

In the presented case study is the possibility to increase the *height* (level) of the binary tree, which will always meets the requirements of the modelled system. This means, that we want to simulate the fair dice using only fair coins at the binary tree with the arbitrary height, not only at the tree with height 3, as was presented. Firstly, we had to come up with a general procedure for what similar trees with higher heights should look like. According to the observed properties and features of the presented model for tree with height 3, we designed the automatic procedure for constructing the tree of the arbitrary height. At the figure below, we introduced the constructed binary tree of the height 4, which meets all required properties, including optimality for a given level. The optimality was proven by the verification of the following *expected-reward* property, whose result was equal to $\mathbf{14/3}$, which corresponding to raise height about one: $\mathbf{R} =? \ [\mathbf{F\ s} = \mathbf{15}]$. We will not go into the details of the next levels, which are constructed in a similar way as this level. Importantly, we were able to find a way to construct a tree of any height with the required properties. To do this automatically, however, we need some *template generator* of individual models, which will generating them written in the PRISM language.



To construct the models representing these binary trees with the arbitrary height we developed the script written in Python. This script requires the given height as its parameter and its output is the file that contains the *model template* in the PRISM language. It uses the general properties of the binary trees, but especially it uses the knowledge gained from researching the well-known tree of height 3. In this way, it constructs the individual *guard commands* to build the required model. Its output is the file with the suffix .templ and the name of this file is die_ln, where n represents the specific height. This script is mainly used in the next phase of our experiments, where it generates the concrete models, but it can be used also in-

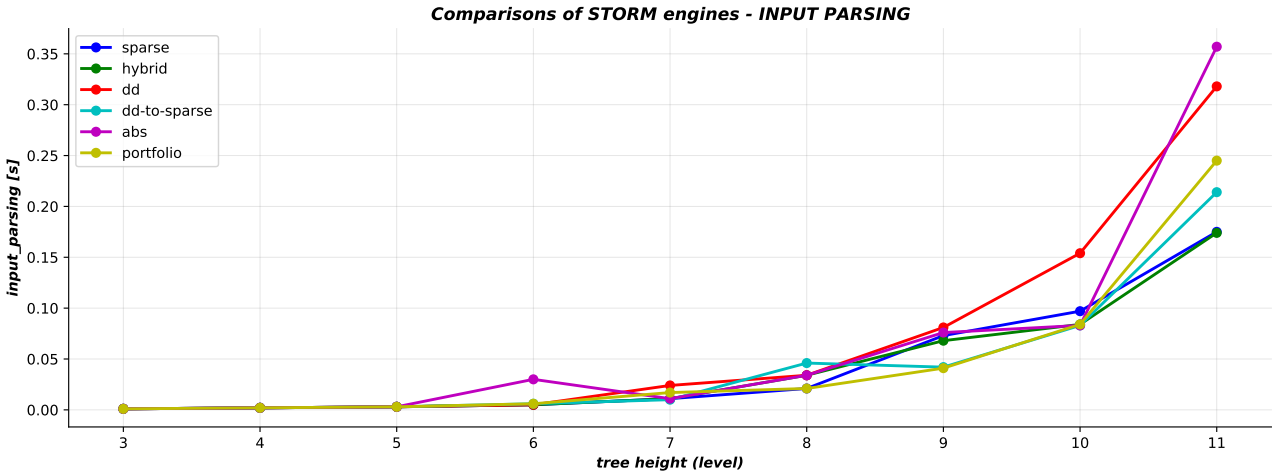


Figure 3. The comparison of the *input parsing* run-times on the individual benchmark instances of the trees with different heights. We can see that with the increasing complexity of the *input description* the period needed to its parsing also increasing. We can observe only minimal differences between the individual engines, because when parsing, there is a minimal number of differences in processing between them.

dividually for subsequent manual analysis. This script is again available to see[1].

3.3 Run Experiments

As we mentioned above, the target of our experiments was researched the behaviour of different metrics at *model checking* – e.g. input parsing run-time, construction model run-time, peak memory usage, etc. – of the model with same features, but different number of *states* and *transitions*. Furthermore, we researched the behaviour of these metrics across the available `STORM` engines. We implemented `Python` script to automatic run our experiments, that implements the following features.

The first step is to set the range of the tree height, for whom will be performed the experiments. According to the our observations, we choose the range $< \mathbf{3}, \mathbf{12} >$, because the minimal required height is **3** and the model checking for tree with height greater than **12** is too cost with some engines. In the next phase, the script constructs the corresponding *model template* for the given level using our script. Such constructed template is subsequently used as the input *model instance* to `STORM` *model checking* process with concrete selected engines. We have selected the engines that support *discrete-time* model, since we have *DTMC* model, and that also support the analysis of both *probability* and *expected-reward* properties. The following engines meet these requirements: *sparse*, *hybrid*, *dd*, *dd-to-sparse*, *abs* and *portfolio* (*automatic*). Subsequently, the script for each engine runs the `STORM` *model checking* over the constructed *model description* and *specified property*.

[1]From the internal server reasons, it is stored as `.txt` file and not `.py`.

We invoke the `STORM` with the following command:

$$storm\ \text{--}prism\ die\_lh.templ \\ \text{--}prop\ "P=?\ [\ F\ s=2^h-1\ \&\ d=1\ ]"\ \\ \text{--}engine\ e\ \text{--}timemem$$

where h represents the current tree height and e the selected `STORM` engine, the probability property can be replaced also with the *reward property*: "$R=?\ [\ F\ s=2^h-1]$". This command invoke `STORM` with its engine `e`, and with `PRISM` model description in `die_lh.t-empl`, and the properties listed after the option `-prop`. Algorithm 1 describes the whole process of running the experiments.

Method `RunExperiments` (Φ, H_{max})
  foreach $h \in < 3, H_{max} >$ **do**
    M = `TempConstruct` (h)
    foreach $e \in E$ **do**
      out = `RunStorm` (Φ, M)
      yield `ParseOutput` (out)
    end
  end

Algorithm 1: The pseudo-algorithm represents the running of our experiments. The inputs to the algorithm are the *property specifications* (Φ) and the maximal *tree height* (H_{max}) for which have to be experiments performed. The set E represents the set of supported engine within experiments.

The last phase of this script to automatic run the experiments performs the parse of the `STORM` output. We match the relevant parts of the output using the *regular expressions* and subsequently the useful information are parsed from them. In more detail, about the each run of the `STORM` model checking we store

the following information: *input parsing* time, *model construction* time, *model checking* time, number of *states*, number of *transitions*, quantitative *result*, *peak memory usage*, *CPU* time and *wall-clock* time. The result of this script is `JSON` file, which contains these statistics for each explored tree *height* and *engine*.

3.4 Experiment evaluation

The results of the previous phase are stored in the files results_probability.json and results_reward.json. We implemented the next one `Python` script that processes this results to the human-readable form. It constructs the charts which displays on the *x-axis* the heights of the tree within the experiments and on the *y-axis* the concrete collected metric at *model checking* process. Figure 3 represents the example of constructed charts, and the rest charts can be found via the references in Chapter 5 to the paper directory.

When we look at the charts that render the metrics connected to the some kind of run-time at analysis of the *expected-reward* property, then we can conclude the following. When the height of the tree is less than *9*, then all researched engines achieved the almost equal run-times at all phases of *model checking*. When the tree height is greater or equal than *9*, then the best results achieved the engines *portfolio* (automatic) and *sparse*, since the `STORM` chosen for *automatic* engine also the *sparse* engine without the *bisection*. For the remaining engines we can observe a significant increase in run-time in all sub-phases, which also caused the interruption of experiments at the value of the tree height of 12. The *model checking* at all these engines is too cost and takes a lot of time, when the height of the tree is greater than 12. Further, it has been observed that the engine *sparse* retains almost constant behaviour even at higher tree heights. The chart displaying the *peak memory usage* distinguishes the type of structure to store the models to the memory. We can see, that the engines that use the *sparse matrices* effectively reduce the needed memory space, whereas the engines using *MTBDDs* use an unnecessarily excessive amount of memory to store the relatively small model. The same trend, with the same argumentation, can be observed at the charts representing the memory metric, which was collected when analysing the *probability specification*. The same trend can be observed in graphs showing time metrics, with the only exception that the limit of significant deterioration of times at the same group of engines is at a value of 10.

4. Conclusion

In this report we focused on the model checker `STORM`. We started by illustrating its key features and exploring its rather unique approach for *model checking*. We then proceeded by testing the tool on the replication data set with 96 benchmark instances. The results of the replication were compared with the reference results presented in the most recent `STORM` paper. Further, we implemented the set of `Python` scripts to run the own experiments. They includes the construction of *model templates* for simulating the fair dice using only fair coin, which can be modelled by the binary tree with various height. The next two scripts ensures the automatic running of experiments and their subsequent evaluation. These experiments have shown that the model size has effect on the *model checking* and its various metrics, in dependent to selected `STORM` engine.

5. Directory Structure

We lists the whole structure of paper directory located at the faculty web, which includes the all relevant files – implemented scripts, tables of results, logs from runs and constructed charts. We emphasise, that the individual files can be showed by the open the reference to them:

```
xstupi00
└── SAV
    ├── die.templ
    ├── results_probability.json
    ├── results_reward.json
    ├── sav_invocations.json
    ├── charts
    │   ├── probability
    │   │   ├── cpu_time.pdf
    │   │   ├── input_parsing.pdf
    │   │   ├── model_construction.pdf
    │   │   ├── peak_memory_usage.pdf
    │   │   ├── states.pdf
    │   │   ├── transitions.pdf
    │   │   └── wallclock_time.pdf
    │   └── reward
    │       ├── cpu_time.pdf
    │       ├── input_parsing.pdf
    │       ├── model_construction.pdf
    │       ├── peak_memory_usage.pdf
    │       ├── states.pdf
    │       ├── transitions.pdf
    │       └── wallclock_time.pdf
    ├── scripts
    │   ├── run_experiments.py
    │   ├── run_postprocess.py
    │   └── template_generator.py
    └── table
        ├── logs
        │   └── ...
        ├── sav_logs
        │   └── ...
        ├── style.css
        └── table.html
```

References

[1] ALUR, R. and HENZINGER, T. A. Reactive modules. *Formal methods in system design*. Springer. 1999, vol. 15, no. 1, p. 7–48.

[2] AZIZ, A., SANWAL, K., SINGHAL, V. and BRAYTON, R. Verifying Continuous Time Markov Chains. In: ALUR, R. and HENZINGER, T., ed. *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*. Springer, 1996, p. 269–276. LNCS, vol. 1102.

[3] BAIER, C. and KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X.

[4] BAIER, C., KLEIN, J., KLÜPPELHOLZ, S. and MÄRCKER, S. Computing conditional probabilities in Markovian models efficiently. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, p. 515–530.

[5] BUDDE, C. E., DEHNERT, C., HAHN, E. M., HARTMANNS, A., JUNGES, S. et al. JANI: Quantitative Model and Tool Interaction. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 2017, p. 151–168. Available at: https://doi.org/10.1007/978-3-662-54580-5_9.

[6] HANSSON, H. and JONSSON, B. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*. february 1995, vol. 6.

[7] HARTMANNS, A., KLAUCK, M., PARKER, D., QUATMANN, T. and RUIJTERS, E. The Quantitative Verification Benchmark Set. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, p. 344–350. ISBN 978-3-030-17462-0.

[8] HENSEL, C., JUNGES, S., KATOEN, J.-P., QUATMANN, T. and VOLK, M. *The Probabilistic Model Checker Storm*. 2020.

[9] HENSEL, C., JUNGES, S., KATOEN, J.-P., QUATMANN, T. and VOLK, M. *The Probabilistic Model Checker Storm: Evaluation Results and Replication Package*. Zenodo, september 2020. Available at: https://doi.org/10.5281/zenodo.4017717.

[10] VOLK, M., JUNGES, S. and KATOEN, J.-P. Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates. *Computer Safety, Reliability, and Security*. Springer International Publishing. 2016, p. 253–265. Available at: http://dx.doi.org/10.1007/978-3-319-45477-1_20. ISSN 1611-3349.