# A Model Checker Collection for the Model Checking Contest using Docker and Machine Learning

Didier Buchs, Stefan Klikovits, Alban Linard,
Romain Mencattini, and Dimitri Racordon ⋆

*Software Modeling and Verification (SMV) Group*, *Faculty of Science, University of Geneva*
Geneva, Switzerland
firstname.lastname@unige.ch

**Abstract.** This paper introduces mcc4mcc, the Model Checker Collection for the Model Checking Contest, a tool that wraps multiple model checking solutions, and applies the most appropriate one based on the characteristics of the model it is given. It leverages machine learning algorithms to carry out this selection, based on the results gathered from the 2017 edition of the Model Checking Contest, an annual event in which multiple tools compete to verify different properties on a large variety of models. Our approach brings two important contributions. First, our tool offers the opportunity to further investigate on the relation between model characteristics and verification techniques. Second, it lays out the groundwork for a unified way to distribute model checking software using virtual containers.

## 1 Introduction

The Model Checking Contest @ Petri Nets [**?**] (MCC) is an annual event, held during the Petri Nets Conference. Initially launched in 2011, its objective is to investigate on the relation between model characteristics and verification techniques. Over the years, the MCC has proven successful to compare the performance of model checking tools with respect to different problems, and increase the confidence in the results they produce. For perspective, the 2017 edition saw 10 tools competing in 9 categories on 77 models. Despite the results being interesting and valuable to the model checking community, they do not clearly address the primary objective of the MCC. Indeed, a clear relation between model characteristics and verification techniques has still to be found.

Although a compelling theoretical answer to this problem may still be out of reach, seven years of MCC results opened the door to an empirical study. As each tool exports the list of employed verification techniques (e.g. decision diagrams, explicit or bounded model checking, etc.), it is possible to use the results of previous iterations of the MCC to shed light on this relation. We therefore created the *Model Checker Collection for the Model Checking Contest* (mcc4mcc), a tool that aims at establishing a relation between

⋆ This project is supported by: FNRS STRATOS : Strategy based Term Rewriting for Analysis and Testing Of Software, the Hasler Foundation, 1604 CPS-Move and COST IC1404: MPM4CPS

model characteristics and verification techniques. The tool serves as a wrapper around the model checking tools that competed in past iterations of the MCC, and leverages machine learning algorithms to select the most appropriate one, given a model and examination type. `mcc4mcc` uses virtual containers to provide homogeneous packaging and execution.

Our contributions to the research community are two-fold:

- While the current way of participation to the MCC (submission of virtual machines) is a valid means to enter, the re-use of tools can be burdensome and requires a lot of resources. We show that virtual container systems such as Docker[1] can be leveraged to create lightweight, uniform distribution systems that invite re-use of the tools.
- Based on the availability of such lightweight tools, we propose the Model Checker Collection as a means to empirically evaluate the relation between model characteristics and verification techniques. We show that `mcc4mcc` is capable of finding the best-suited tool amongst a range of candidates and provide details about machine learning algorithms we use for decision making.

This paper describes the methodology we have used to create Docker images containing the MCC model checkers, as well as the machine learning process that is implemented in `mcc4mcc`. Our tool is available at `https://github.com/cui-unige/mcc4mcc`, and is released under the open source MIT license.

## 2 Creation of Docker Containers

Tool authors who wish to compete in the MCC have to provide a virtual machine (VM) for their tool. Submissions have to be completely self-contained, i.e. contain the tool, its dependencies, models and their precomputed equivalents (for some tools). In the context of the MCC, this approach presents two significant advantages. First, examiners do not need to struggle with installation procedures, and authors can ensure their tool is run under the best configuration possible. Second, tools are guaranteed to be executed in identical, reproducible environments, which is obviously desirable for a competition.

However, when the tool is to be used as a classic software, such VM compartmentalization might present some inconveniences. Exchanging data with a virtual machine can only be done by the means of virtual networks, usually through a secure shell[2]. This limits, or at least significantly complicates the options a final user has to run the tool in a pipelined process. Disk space can be listed as another major inconvenience. Table 1 shows the size of each tool's virtual machine, submitted to the MCC. A virtual machine for a MCC tool typically uses at least 2 Gigabytes with some requiring up to 8.5 Gigabytes of disk space. In a setting such as our research goal, where multiple tools need to be used, this may quickly reach problematic proportions.

This caveat lead us to question the means of packaging model checkers within VMs. In general, we see three possibilities to produce an easily distributable, reusable `mcc4mcc` wrapper that incorporates the MCC submissions:

[1] `https://www.docker.com/`
[2] `https://en.wikipedia.org/wiki/Secure_Shell`

| Tool | | License | granted | Virtual Machine Size | System | Docker Size | System |
|---|---|---|---|---|---|---|---|
| GreatSPN | [?] | Closed | ✔ | 2.9 Gb | Debian | 278 Mb | Debian |
| ITS-Tools | [?] | GNU GPL v3 | ✔ | 3.3 Gb | Debian | 843 Mb | Debian |
| LoLA | [?] | GNU APL v3 | ✔ | 8.5 Gb | Debian | 14 Mb | Alpine |
| LTSMin | [?] | BSD 3 Clause | ✔ | 3.4 Gb | Debian | 668 Mb | Debian |
| Marcie | [?] | Non commercial | ✔ | 2.3 Gb | Debian | 21 Mb | Alpine |
| Smart | [?] | Closed | ✖ | 2.7 Gb | Debian | 284 Mb | Debian |
| Spot | [?] | GNU GPL v3 | ✔ | 7.0 Gb | Debian | 2 Gb | Debian |
| Tapaal | [?] | GNU GPL v2 | ✔ | 2.3 Gb | Debian | 69 Mb | Alpine |
| Tina | [?] | Freeware | ✔ | 2.6 Gb | Debian | 830 Mb | Debian |
| Total | | | | 35 Gb | | 4.9 Gb | |

**Table 1.** MCC tools and their licenses, virtual machine sizes and docker image sizes. Docker images include the models translated into the tool-specific format taken from the virtual machines.

**Embedding existing virtual machines.** One possibility is to create one monolithic VM that wraps around the VMs that were submitted by the developers. This requires the ability to run nested virtualization (i.e. virtual machines within virtual machines). While it is technically possible, this technique usually results in poor performances [**?**]. Moreover, the result would suffer from the disk space issue mentioned above, as the enclosing virtual machine would weigh around 40 Gigabytes. This approach clearly violates the constraint of facilitated distribution.

**Merging of virtual machines.** This technique addresses some of the performance issues. In addition, its also leads to a far lighter virtual disk, as it does not involve the duplication of many shared files and binaries. Unfortunately, such a virtual machine is very susceptible to possible configuration conflicts between the dependencies of the different tools. This approach heavily increases maintenance, modification and update complexity.

**Packaging tools in virtual containers.** The third approach relies on virtual *containers* [**?**], rather than virtual *machines*. Contrary to VMs, virtual containers do not provide their own operating system and do not require hardware support. They share central resources such as a file system or a network interface with their host machine, if required. Due to their isolated virtual-memory region they remain independent. Virtual containers cause less system overhead and have the potential for increased performance. They are usually lightweight and include only the bare necessities for program execution. This means that most of them abstain from including a user interface and unnecessary libraries.

We chose the third approach to create the `mcc4mcc`. We use Docker[3] to compose and execute the virtual containers. The tools inside the containers are built either from the tool sources when available, or by copying the required files from the virtual machines or binary distributions of the tools.

Docker containers[**?**] are instantiated virtual containers that package programs and files. These instances are created based on "container snapshots" called docker *images*.

[3] https://www.docker.com

Images store a container's file system state and, upon running, create exact runtime copies of the snapshot. For our purposes we create Docker images that contain the tools and any other files which are required for the tool execution. Additionally, certain Linux files and programs are provided by the *base image*. Base images are docker images that contain a minimal execution environment and are extended by our images.

The disk space required by the tool docker images is stated in the last two columns of Table 1 (alongside the base image). The severe reduction results in images sizes of less than 400 megabytes, which facilitates distribution. On average, we achieved a reduction of over 90 %. For LoLA and and Marcie we achieved drastic size reductions of 99.9% and 99.1%, respectively. This is due to the fact that we could use the *Alpine* base image, which measures only around 4 megabytes.

In general, Docker container images can be stored within private or public repositories for easy distribution. The usual naming scheme `<organization>/<tool>:<tag>`, `organization` is e.g. the university, laboratory or team name, `tool` is the tool name, and `tag` is an optional tag, such as version number or variant, facilitates the identification of each specific image. We follow this naming scheme in this article, and propose a standardized way to provide Docker images for tools, designed to ease both the diffusion of the tools, and their wrapping for the MCC. We split each tool into two Docker images:

- `<organization>/<tool>`, an image that contains only the tool, with the manual page and examples if needed, for instance `unirostock/LoLA`. This image should ideally be provided and maintained by the tool developers, and may be tagged for instance with the tool version.
- `mccpetrinets/<tool>`, a wrapper dedicated to the Model Checking Contest, that is built on top of `<organization>/<tool>` and creates an entry point (the default executable script in the image) conforming to the Model Checking Contest specifications. This image may also embed additional data, such as precomputed data for known models, and may tagged, for instance with the year of the participation.

The two images are built using docker's configuration files, called *Dockerfiles*[4]. These typically describe the build process (e.g. how to build a tool from source), various configuration settings and more importantly an entry point that points to the command (or set thereof) that is ran when the container starts. The Dockerfiles we used for the build are available in the respective subfolders of our tool repository[5].

Once the Docker images are configured, a tool can be run within its confined environment. By default, tools run in a completely isolated environment, and do not have access to the host file system. Hence it is necessary to explicitly mount a path to the host directory holding the models' data. We communicate other information by the use of environment variables, as it is the MCC's convention.

```
$ docker run --volume=<path to model>:/mcc-data \
             --env BK_TOOL="${BK_TOOL}" \
             --env BK_EXAMINATION="${BK_EXAMINATION}" \
             mccpetrinets/<tool>:<year>
```

[4] https://docs.docker.com/engine/reference/builder/
[5] https://github.com/cui-unige/mcc4mcc

| Tool | State Space | Upper Bounds | Reachability | | | CTL | | LTL | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Cardinality | Deadlock | Fireability | Cardinality | Fireability | Cardinality | Fireability |
| GreatSPN | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✖ | ✖ |
| ITS-Tools | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| LoLA | ✖ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| LTSMin | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Marcie | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✖ | ✖ |
| Smart | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ |
| Spot | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✔ |
| Tapaal | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✖ | ✖ |
| Tina | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ |

**Table 2.** Examinations performed by the tools

Running a container will try to find the image within the local repository, and – if not available – search in online repositories (e.g. docker hub[6]) for the image and download it if found. Online distribution is useful for tool developers to share their tool more easily than using binary archives. Uploading an image does not distribute the tool sources, as it contains the result of executing the Dockerfile, and thus the binaries.

This is especially important for tool developers who wish to keep the tool's source code private, as is the case for some MCC competitors. While most tools are open-source, such as LoLA or Spot, closed-source but freeware, such as Tina, a few programs explicitly require licenses to be used (e.g. GreatSPN, Smart). Table 1 lists the individual licenses for the tools that competed in the 2017 MCC. The table also states a license has been granted, if necessary.

## 3 Using Machine Learning to choose the Right Tool

Our second research question examines whether we can use empirical methods to choose a well-suited tool for a model checking problem. This evaluation is based on the results of the 2017 MCC. In the MCC, tools participate in different *examinations* such as state space exploration or deadlock reachability. Table 2 cross-references the MCC's competing tools and the respective examinations they participated in.

In each examination the tools try to solve their objective on three categories of models: *known*, *stripped* and *surprise*. Known models are taken from a catalog and known before the competition. Tool developers can hence optimize their tools towards solving them. Stripped models, similar to known models are taken from a catalog, however the precise model will not be identified beforehand. The challenge lies in performing the correct optimizations and try to provide a performance as if the model were known. Surprise models are new models that have not been used before. It is the hardest category, as tool developers cannot create optimizations for their tools.

[6] `https://hub.docker.com/`

| Model | Ordinary | Strongly Connected | Reversible | Colored | Place/Transition | Sink Place | Simple Free Choice | Sub-Conservative | Deadlock | State Machine | Marked Graph | Source Transition | Extended Free Choice | Source Place | Quasi Live | Parameterized | Live | Safe | Sink Transition | Connected | Loop Free | Conservative | Nested Units |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Eratosthenes | ✔ | ✖ | ✖ | ✖ | ✔ | ✔ | ✖ | ✔ | ? | ✖ | ✖ | ✖ | ✖ | ✔ | ✔ | ✔ | ✖ | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ |
| CSRepetitions | ✔ | ✖ | ? | ✔ | ✔ | ✖ | ✖ | ✔ | ? | ✖ | ✖ | ✖ | ✖ | ✖ | ? | ✔ | ? | ✖ | ✔ | ✔ | ✖ | ✖ | ✖ |
| TokenRing | ✔ | ✔ | ✖ | ✔ | ✔ | ✖ | ✖ | ✔ | ? | ✖ | ✖ | ✖ | ✖ | ✖ | ✔ | ✔ | ✔ | ? | ✖ | ✔ | ✖ | ✔ | ✖ |
| IBM703 | ✔ | ✖ | ✖ | ✖ | ✔ | ✔ | ✔ | ✖ | ? | ✖ | ✖ | ✖ | ✔ | ✔ | ✔ | ✖ | ✖ | ✔ | ✖ | ✔ | ✔ | ✖ | ✖ |
| PhilosophersDyn | ✖ | ✔ | ? | ✔ | ✔ | ✖ | ✖ | ✖ | ? | ✖ | ✖ | ✖ | ✖ | ✖ | ? | ✔ | ? | ? | ✖ | ✔ | ✖ | ✖ | ✖ |

**Table 3.** Characteristics of some models

Each model might also be parameterized, meaning that there are different configurations for this model. An example is the model *philosophers*, which can specify the number of dining philosophers as parameter. This information is published in a supporting document alongside the respective model, which contains all other model characteristics. These characteristics can be e.g. whether the model is *reversible*, contains *sink places* or *quasi liveness*. Table 3 lists the model characteristics for a few models. Note that the characteristics are Boolean, but unknown for certain models (marked with "?").

The MCC results are provided in the form *Tool*, *Instance*, *Examination* → *Time*, *Memory*, where *Instance* is the combination of a *Model* and its *Parameter*. Using these results and the information about model characteristics, we can evaluate which tools are best-suited for a model with certain characteristics. Self-evidently the term "best-suited" varies depending on the application. We might e.g. be interested in choosing the tool that computes the highest number of instances for a model, requires the least amount of time or resources, or produces the fewest of errors.

To perform this selection empirically `mcc4mcc` employs machine learning techniques to choose the best-suited tool. The principle of machine learning is "to predict the future based on the past" [**?**]. This means that existing data is analyzed and used to draw conclusions about new data.

There are numerous different analysis techniques, of which we use a subset that is commonly referred to as *classification algorithms*. These kinds of problems typically involve an existing data set providing observations and their classification, i.e. their correlated value. This data set is referred to as *learning* or *training set*. Through analysis of this data set, classification predictions can be made for new data, i.e. the *test set*.

For `mcc4mcc` five of the algorithms provided in the `scikit-learn` [**?**] Python library were evaluated:

*k* **nearest neighbors [?]** This algorithm places the data points in a multidimensional coordinate system. New data is classified by choosing a predefined number (i.e. *k*) of neighbours based on the Euclidean distance (or different distance measure) and choosing the best-suited classification using this information – usually the most common class amongst the neighbors.

**Support vector machines [?]** represent data as points in space, divided by a clear gap that is as wide as possible. This algorithm builds a hyper-plane based on the inner product of the new data's vectors and all observation's vectors.

**Neural network classification [?]** uses a learning algorithm that establishes a layered graph. The data is mapped onto the input layer which represents the data's features. Each node (*neuron*) in the graph takes input from the previous layer, performs a small calculation and offers this information to the following layer adding a weight measure. The final (output) layer is the classification. Using the learning set it is possible to adjust the weight of individual neurons to guide the classification.

**Naive Bayes classifier [?]** This method uses the Bayes theorem [**?**] to calculate the new data's membership probability for each class. The class with the highest probability is chosen.

**Decision Trees [?]** The creation of a decision tree is a supervised learning method where characteristics are used to create a tree structure. The nodes in this tree are decision points on individual characteristics, the classes are the leaf nodes of the tree. Decision trees can be represented simply using highly nested if-then-else nodes. The goal of the algorithm is to find data characteristics that serve as discriminators for the classification.

In order to evaluate the algorithms' efficiency, it is interesting to compute the score that `mcc4mcc` would have obtained during the Model Checking Contest. Table 4 provides these results, where the score is computed by awarding 12 points to a tool whenever it can find an answer for an instance and examination, and 2 points if it uses the least amount of time or memory. Note that the computation of the scores differs slightly from the rules of the Model Checking Contest for simplicity of implementation. For instance, the score is computed for each tool, instance and examination, whereas the Model Checking Contest computes a score for each formula contained in the examination. Multipliers depending on the status of models (known, stripped or unknown) are also not applied in the custom scoring function of this paper. Thus the reader can observe that the scores provided in this article do not correspond to the scores available on the webpage[7] of the contest.

## 4 Conclusion and Future Works

This article presents the Model Checker Collection for the Model Checking Contest (`mcc4mcc`). Our tool acts as a wrapper around the tools that participate in the Model Checking Contest (MCC), hosted annually at the Petri Nets conference. Using the tools of the 2017 edition and knowledge about their performance, we employ empirical evaluation to choose the most-suited tool for a problem.

Our research contributions are two-fold: First, we present how the packaging of tools in virtual containers is beneficial in terms of disk space and orchestration. `mcc4mcc` relies on Docker images, which are light-weight and closer to the execution hardware, to provide the execution environment. We provide proof-of-evidence of this efficiency by reducing the required disk size of the distributed files by over 85% on average. This allows easier distribution and execution. Docker has been further shown to be more

[7] `https://mcc.lip6.fr/2017/results.php`

| | Total | State Space | Upper Bounds | Reachability | | | CTL | | LTL | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Cardinality | Deadlock | Fireability | Cardinality | Fireability | Cardinality | Fireability |
| mcc4mcc – Decision Tree | 11.173 | 1.171 | 1.266 | 1.228 | 1.241 | 1.235 | 1.240 | 1.262 | 1.233 | 1.293 |
| mcc4mcc – SVM | 10.919 | 566 | 1.287 | 1.260 | 1.253 | 1.269 | 1.281 | 1.324 | 1.316 | 1.360 |
| mcc4mcc – Neural Network | 10.552 | 426 | 1.224 | 1.268 | 1.220 | 1.253 | 1.270 | 1.241 | 1.319 | 1.237 |
| LoLA | 10.540 | 0 | 1.287 | 1.335 | 1.253 | 1.345 | 1.318 | 1.324 | 1.316 | 1.360 |
| mcc4mcc – KNN | 9.889 | 584 | 1.259 | 1.232 | 962 | 1.243 | 1.245 | 1.245 | 1.062 | 1.053 |
| LTSMin | 8.680 | 718 | 1.051 | 1.039 | 594 | 1.044 | 1.043 | 1.043 | 1.088 | 1.057 |
| ITS-Tools | 6.746 | 989 | 680 | 747 | 851 | 745 | 522 | 594 | 830 | 785 |
| mcc4mcc – Naive Bayes | 6.406 | 975 | 632 | 743 | 824 | 775 | 749 | 750 | 599 | 355 |
| Marcie | 4.392 | 907 | 671 | 539 | 642 | 577 | 514 | 539 | 0 | 0 |
| Tapaal | 4.391 | 435 | 325 | 768 | 761 | 612 | 837 | 650 | 0 | 0 |
| GreatSPN | 4.211 | 1.052 | 742 | 438 | 645 | 441 | 464 | 424 | 0 | 0 |
| Tina | 1.096 | 1.096 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Smart | 714 | 714 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Spot | 632 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 632 | 0 |

**Table 4.** Comparison of scores for tools and machine learning algorithms

resource efficient as it does not need to emulate an entire operating system, but merely the necessary applications.

Our second contribution, the choice of tool based on an empirical evaluation of existing model checking results uses this framework. This paper introduces the five machine learning algorithms employed for the evaluation and shows that three algorithms have the potential to achieve a higher overall score than the 2017 MCC winners. While mcc4mcc still has potential for improvement, it already shows promising results, choosing well-performing tools amongst ten competitors. Our tool has been published online[8] so that others might use information for own analyses. To prove the efficacy of our tool, mcc4mcc participates in the 2018 Model Checking Contest.

Although our research shows promising results, we do plan on adding further improvements. We subsequently list the most important ones:

Firstly we plan to augment the prediction capabilities of our machine learning engine, by using additional machine learning algorithms, and adding domain-specific customizations. Additionally we plan on adding the results of all previous MCC competitions to increase the learning data set. To ease the storage and analysis of results we are planning a homogeneous data format that can be used with a standardized API. We propose a collaboration with the Petri nets repository [**?**], which already contains all the MCC models of the alongside their characteristics.

Second, mcc4mcc allows us to discover which model characteristics are most important when selecting the best tool. We can further use the data to propose new characteristics to the MCC organizing committee.

[8] https://github.com/cui-unige/mcc4mcc

Third, due to specific requirements we were not capable of creating virtual container images for all 2017 MCC tools. We propose a collaboration with tool developers in order to help them create virtual container images themselves. Next to the facilitated distribution, tool developers can then rest assured knowing that the tools are correctly installed and executed in their intended environment. We believe that this will ease the requirements to support different operating systems and configurations. We also plan to create containers for tools that have been developed within our team, and have competed in previous editions of the MCC: AlPiNA [**?**], StrataGEM [**?**], and Yadd [**?**]. The CosyVerif [**?**] and Ardoises [**?**] projects, that aim at creating an environment for formal modeling and verification should also use the containers of the various tools.

Lastly, we propose to change the MCC's submission format to virtual container images. The use of such containers requires less effort, disk space and system knowledge, while maintaining the same advantages of virtual machines. As this change could have a great impact on the infrastructure of the contest, it should be carefully discussed and tested with the organizers.