Can a Tool Learn its Own Settings?

Dennis de Weerdt
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.deweerdt@student.utwente.nl

ABSTRACT

In the field of model checking, a number of tools has been developed. LTSmin, created at the Formal Methods and Tools group of the University of Twente, is a collection of such tools. While these tools are very useful for verifying the correctness of software and/or physical systems, they have a large number of switches and settings to choose from. The many settings have a significant impact on performance, and the choice of settings to use for any particular problem can be difficult to make. This paper explores the possibility of statically analysing the structure of model checking problems in order to predict whether or not Partial Order Reduction would be an effective way of limiting the time and memory required to solve queries. Surprisingly little correlation was found between certain structural properties and benchmark results, but nevertheless a tool has been developed which can offer reasonable advice.

1. INTRODUCTION

Model checking is a technique used for verifying the correctness software or (cyber-)physical systems. Over time, a number of different model checking languages have been developed. The Formal Methods and Tools group at the University of Twente has developed the LTSmin toolset, which provides a common interface (the *Partitioned Next-State Interface*, *PINS*) for a number of such languages [5]. LTSmin can be used to answer, for example, questions of *reachability*: Will the system ever be in state X? For example, given a model of a car's electrical system, we might ask whether or not the airbags will always deploy in the event of a crash. Another query which may be executed on a model is *deadlock*: will the system ever be in a state where it is stuck, and cannot proceed? In addition to simple yes/no answers to these questions, the modeling tools also provide a specific counterexample which leads to the queried state if such a sequence exists.
LTSmin offers a plethora of options and switches to customise the verification of queries [15]. Choosing the right options is essential to the execution speed and memory requirements, as this research will demonstrate. To aid in making this choice, the developed tool gathers structural properties of models and then employs a neural network to analyse these. This data is used to estimate whether or not *Partial Order Reduction* (POR) will be effective. The result (phrased as the answer to the question "Should I use POR on this model?") is either NO or MAYBE, as no strong positive correlations were discovered.

The research focuses on answering the following questions:

1. With what accuracy can the developed algorithm predict whether or not applying POR would be a sound investment?
   a. How well do experimental results generalise to new models?
   b. What gain can be achieved in time and/or memory usage when analysing a model using the algorithm prior to executing queries on it?
2. What structural properties of models exhibit correlation with the effectiveness of POR, and to what degree?

1.1 Background

This section will describe some of the background information required for both executing and understanding the proposed research.

1.1.1 Model Checking

The models used for model checking provide an abstract description of the system they represent. In order to perform any meaningful queries on these, they are instantiated to a simpler but far larger labeled transition system (LTS). The amount of states in these systems can vary from a few thousand for simple models to many millions for more complex ones. Experimental results show (see section 4) that choosing different options for running the same query on the same model can have a drastic effect on the execution time of the query. Picking the right ones, therefore, is essential if one wishes to run many such experiments. Doing so is very difficult, as the sheer number of options results in a nearly insurmountable amount of combinations to try.

1.1.2 LTSmin

LTSmin is a comprehensive set of tools developed by the Formal Methods and Tools group at the University of Twente [5]. Its central architecture consists of three layers: the frontend, PINS and the backend. The different frontend modules take a model in one of various modeling languages like mCRL2, Promela or DVE, and process these to match the PINS interface. PINS provides access to a next-state function as well as a number of statically calculated matrices. This information is then used by one of the available backends, such as the symbolic analysis

25^{th} *Twente Student Conference on IT* July 1^{st}, 2016, Enschede, The Netherlands.

and multi-core algorithms. It is these backends which perform the actual analysis. Before using them, however, it is possible to employ one or more so-called PINS2PINS wrappers, which perform some transformation on a PINS model. Partial Order Reduction, discussed below, is implemented as one such PINS2PINS wrapper.

Since using POR is central to this research, it is vital that the chosen backend supports it. Due to the fact that POR is incompatible with the symbolic tools and only limited support exists for the distributed ones [5], our only choice is between the multi-core and sequential backends. The sequential version is likely to be considerably slower, as the multi-core backend can more fully utilize available resources. As is further discussed in Section 3.2, we will use models written in the DVE language. The obvious choice of LTSmin tool, then, is *dve2lts-mc*.

1.1.3 Partial Order Reduction

Partial Order Reduction is an optimisation technique used in model checking for reducing the size of state spaces [1]. It is based in the observation that, when modeling concurrent systems, there are often many ways to order certain actions. This ordering, however, does not necessarily affect the eventual outcome. If two transitions can be executed in any order, resulting in the same outcome, these transitions are *independent*[3]. Such transitions can neither enable nor disable one another. This notion can be expanded to that of *stubborn sets*, which are sets of transitions which are independent of transitions not in the set. The goal of using stubborn sets in POR is to use only those transitions which are part of stubborn sets, thereby reducing the total amount of transitions[17]. Stubborn sets are not the only technique POR may use [14, 4], but it is the one used by LTSmin [5] and therefore deemed the most relevant to this research.

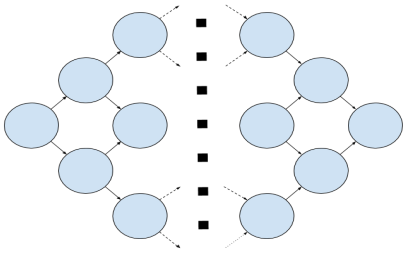
One particularly useful structure which may be encountered in state spaces is the *d*iamond. This is a group of transition which always lead from one state to another, along a potentially large amount of paths. An example of such a structure may be found in Figure 1. One observation is vital here: no matter what path one takes along the diamond, one always ends up in the right-most state. Intuitively, it would seem that such diamonds are very useful for POR, since the transitions within it are independent of each other. Pelánek, whose research will be explored more in Section 2, also notes this in [11], Section 6.2.1. So how are these diamonds formed? Consider the following pair of parallel processes:

```
i = 1, sum1 = 0;
while (i <= 10) {
  sum1 = sum1 + i;
  i = i + 1;
}
return sum1;
```

```
j = 1, sum2 = 0;
while (j <= 10) {
  sum2 = sum2 + j;
  j = j + 1;
}
return sum2;
```

These two processes are completely independent of each other. After all, they operate on different pairs of variables. When executing them, therefore, it does not matter how the system chooses to interleave them; they will always return the same values. This would give rise to a diamond structure in which each 'branching' represents a choice between executing a line from the left or right process. When one of the processes is done, we have reached the right side of the diamond and have only a single path left until we reach the end node: executing the process

Figure 1: A diamond structure



which has not yet finished. Now, consider an alternative:

```
i = 1, sum1 = 0;
while (i <= 10) {
  sum1 = sum1 + i;
  i = i + 1;
}
return sum1;
```

```
i = 1, sum2 = 0;
while (i <= 10) {
  sum2 = sum2 + i;
  i = i + 1;
}
return sum2;
```

In this scenario, both processes use the variable *i*. As they both increment and check *i*, they are no longer independent: the steps that the left process takes influence the steps the right one can take, and vice versa. Crucially, the processes are not guaranteed to always produce the same answer; it depends on the interleaving. Because of this, there is no single end node. Rather, there is one for each possible outcome. The internal structure differs as well. This second pair of processes, therefore, does not result in a diamond.
The intuition here, then, is that parallelism causes diamonds, and diamonds are good for POR. Section 4.1 will provide data disputing this intuition.

1.1.4 Machine Learning - General

In order to transform the benchmark data into recommendations as to whether or not to employ POR, an algorithm is used to discover relations between the model parameters and the eventual outcome. This algorithm is one of the many found in the field of *machine learning*. Mitchell defined machine learning as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E"[7]. Less formally, a machine learning algorithm adapts its parameters based on new data, in such a way that it is more effective after the adaptation than before. By doing so, these algorithms can 'learn' patterns which occur in data.

1.1.5 Machine Learning - Neural Networks

Artificial neural networks (ANNs) are mathematical abstractions of the structures that make up brains. They consist of a number of *neurons* organised into a number of layers, which may or may not be connected to other neurons. Data is presented to a network as a list of numbers, which are passed on along weighted connections to other neurons. Each neuron applies a certain function to the sum of its inputs, and passes the result of that function to all of its outputs. An example of an ANN is given in Figure 2.

2. RELATED WORK

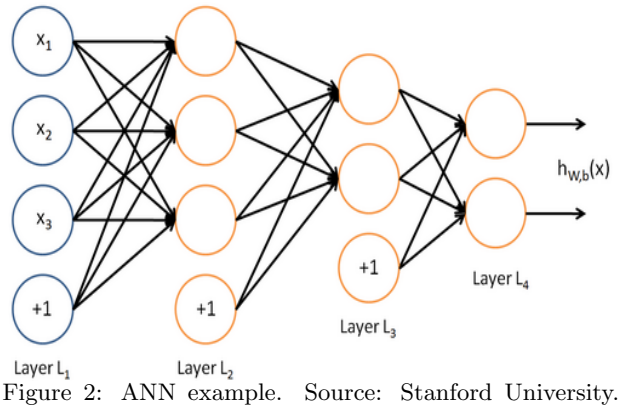


Figure 2: ANN example. Source: Stanford University. Here x_1 to x_3 are input values, the '+1' nodes are so-called bias, and h is the function representing the network as a whole (taking the input vector x as its parameter).

The proposed method of finding settings for models does not seem to have been attempted before. At least, the results of any such undertaking do not seem to have been published. Much of the techniques which have been used in this research are based on papers written (jointly) by Radek Pelánek [13, 12, 11]. In particular, Pelánek has devised and evaluated techniques to estimate structural parameters of state spaces [12]. Pelánek proposes several different techniques for estimating the size of state spaces, something which is normally only known *after* processing the model. An estimate for the state space size might very well be used to estimate the time required to execute a query, which would, as Pelánek points out by referring to Maister [6], make users more amenable to waiting for it to complete.
Initially, however, the parameters we are most interested in - for the purpose of predicting POR's efficiency - are related to the notion of *according transition groups* as outlined in [5]. These are structures where - very simply put - there are multiple paths leading from one state to another. Specifically, Pelánek suggests that POR will be more effective on models with more such transition groups [11]. Applying this knowledge will be the primary focus of this research, certainly early on. If more parameters are found to be useful and feasible to extract, they can be added at a later date.

3. METHOD

3.1 Data Gathering

No machine learning algorithm can work without data, so extracting useful information about the models is key. To conduct preliminary research, a program which extracts data related to POR has been developed. This program outputs, for a pre-compiled model in the DiVinE language [2], information on four matrices as specified by LTSmin. These are:

- Do-Not-Accord
  Two transition groups are according if - essentially - they create a diamond-shaped structure in the state space. A formal definition may be found in [5], section 4.3.
- Necessary Enabling
  A transition group is necessary-enabling for a guard if it writes to a variable that the guard depends on. This matrix gives the transition group/guard pairs for which this is the case.
- Necessary Disabling
  This matrix is similar to the previous one, but it instead details transition groups which lead to a guard being *disabled*. This matrix and the previous one are often very similar.
- May-Be-Coenabled
  Two guards may be coenabled (that is, enabled at the same time) if that does not cause a contradiction. For example, two guards $\{i > 5\}$ and $\{i < 3\}$ can never be true at the same time and thus must be indicated as *false* in this matrix.

In addition, it gives the number of global variables in the model. This is related to the previously mentioned intuition that parallelism causes POR to be effective: If there are no global variables, then it is impossible for two processes to influence one another. The presence of many global variables does not automatically mean that they cause dependencies between processes, however.

3.2 Machine Learning

The question of whether or not to apply POR presents a *classification* problem. All models can be divided into two disjoint classes: NO or MAYBE. One central problem in the application of machine learning for classification is a dichotomy expressed in different terms depending on where it is encountered, among which are exploration vs. exploitation and bias vs. variance. These all indicate the same things: a high bias means that the algorithm is using little data, and may be unable to clearly distinguish different inputs. A high variance, on the other hand, indicates that the algorithm is trained very precisely on the training data, but may very well fail when presented with new data. An ideal situation is one where there is neither high bias nor high variance.

The models used in the research were taken from the BEEM set [10]. These are models in the DiVinE language. The set contains a number of categories of models, essentially types of problems. Each category contains a number of variations upon the problem, with the differences mostly being in the scale of the model. To construct and use the neural network, the FANN library (Fast Artificial Neural Networks) is used [9]. This is a high-performance library written in C which is easy to use and very fast to execute. As hinted at in research subquestion 1A, it is expected that some difficulty may be found in generalising the results of experiments on this dataset to other, new models. It is hoped that such problems *(i.e. bias, variance)* can be alleviated to a large degree by employing K-fold cross-validation, a technique for the verification of a machine learning algorithm's output in which the algorithm is trained on K-1 slices of equal size, and then validated on the K'th slice. The intent is that by varying the 'verification slice' over different runs, most problems of bias and variance can be eliminated.

The process of gathering and analysing the data and training the network is comprised of these steps:

1 Use *dve2lts-mc* (part of LTSmin) to benchmark the models

2 Collect the structural data using a utility built for this paper

3 Combine these two datasets into the format expected by FANN

4 Train the network using FANN's SARPROP implementation[16]

After this process has been completed, new models can be assessed by computing the structural parameters and entering these into the network.

3.3 Evaluation

To evaluate the predictions made by the tool, the results will be scored as follows: First, we run the model both with and without POR enabled. Next, we invoke the tool, which calculates the required parameters and runs the neural network. For each process (POR, no POR, tool) we measure the time required to complete it. Finally, we calculate a score. This score is based on the measured times, and is defined in Equation 1. If the advice was NO, then the score is negated, because the worse POR performs, the better the advice was. Regardless of (the sign of) the score, if the advice was MAYBE, then the score is halved because it is not a strong advice either way. In the absence of strong positive advice, this is deemed acceptable. Should the tool be improved at a later point and positive advice made possible, then the scores for models with that advice would not be negated. The table below lists some (fictional) example models.

Model	POR	No POR	Alg. Time	Score
A	10	17	2	50
B	30	38	3	17
C	20	14	4	-50

The POR and No POR columns reflect the execution time in seconds. Alg. Time is the time required for the developed algorithm to give its recommendation. The score is calculated by the following formula:

$$Score = 100 * \frac{NoPOR - POR - AlgTime}{POR} \quad (1)$$

The final score is calculated by summing the scores for individual models. If this final score is positive, this indicates that on average the tool yields a good result.

4. RESULTS

4.1 Lack of Correlation

In Section 1.1.3, we outlined an intuition pertaining to correlation between the effectiveness of POR and parallel execution. In the figures at the end of this paper, we present data gathered in experiments which seem to contradict this intuition. In the first four graphs in Figures 3 and 4, we plot the ratios of true vs. false values in the four matrices against (Figure 3) the fraction of time required for the query using POR vs. without POR (that is, a value of 2 indicates that the process took twice as long when using POR), or (Figure 4) the size of the POR-reduced state space as a fraction of the original. In the fifth graphs of both figures, we plot the amount of global variables on the horizontal axis.

From these plots, it appears that there is little correlation between the explored factors and the effectiveness of the reduction. In most cases, the more effective cases (which are low on the vertical axes) are not clearly grouped on the horizontal axes. This is discouraging, as obviously the machine learning algorithm can only find patterns if they actually exist. That said, there are at least some observations to be made. Primarily, The 'good' values usually appear limited to certain x-values. Though too many 'bad' values exist in the same ranges, at first glance it should at least be possible to give a negative advice if a new value lies outside of these ranges. Even this seems a bit haphazard, however. The most surprising result is perhaps the apparent randomness of the global variable plots. Given the fact that only the presence of global variables can force processes to wait for one another, we would have expected a greater correlation.

Unfortunately, at this time we can see no proper explanation for this discrepancy. The most likely scenario is that the specific metrics chosen to analyse the different inputs are incorrect. The intuition given in Section 1.1.3 still seems sound, even if the experiments show different results.

4.2 Prediction

Despite the lack of clear correlation detailed in the previous section, we continued work on the neural network in the hope that it could combine inputs in such a way that useful patterns could be discovered. Several network structures were attempted. The most successful one was generated using the Cascade algorithm as implemented by FANN. This algorithm builds the network itself by iteratively trying new neurons and seeing whether or not they improve the network [8]. This results in a very peculiar network; one in which the thirteen inputs (three for each of the four tables, plus the global variable count) connect to a single neuron, which then starts a chain of single neurons up to the two outputs. Because neural networks are in essence black boxes in the sense that, even though we might design them with specific relations in mind, once trained it is next to impossible to understand the reasoning behind the weights it ends up with. The two outputs in the training examples are the relative runtime and state space size plotted in Figures 3, 4 and 5, but there is not necessarily any connection to this in the trained network.

Figure 5 shows how the outputs of the neural network relate to changes in execution time and state space caused by applying POR. As before, low values on the vertical axes indicate that the model could be solved faster or with fewer states with POR than without. The models for which this is the case are clearly grouped in small ranges of network output values. Unfortunately, these same ranges also contain many models for which POR was counterproductive. We can see, however, that there are only a few cases in which POR worked well on models outside of these ranges. This suggests that it is possible to provide a negative advice: If, for a new model, its parameters result in the neural network returning values outside of these small ranges, then POR is unlikely to work well. If the outputs *do* fall within the ranges, then the result is inconclusive: In most cases, the performance with POR is marginally worse than the unreduced model, in others it does yield an improvement, and in a few applying POR leads to the time required more than doubling. It should be noted that there are exceptions to all of these cases, so the tool's output can never be fully relied on.

Comparing the advice values against the measured effectiveness of POR, we found that the tool outputs MAYBE approximately 52% of the time. If we define *hard misses* as cases where POR reduced runtime by at least 20%, but the tool advised NO, then we find that these hard misses occur in about 8% of cases. Further, we define *soft misses* as the less severe case where the tool advised MAYBE when POR increased the total runtime. Such soft misses occur in approximately 23% of cases. Combining these

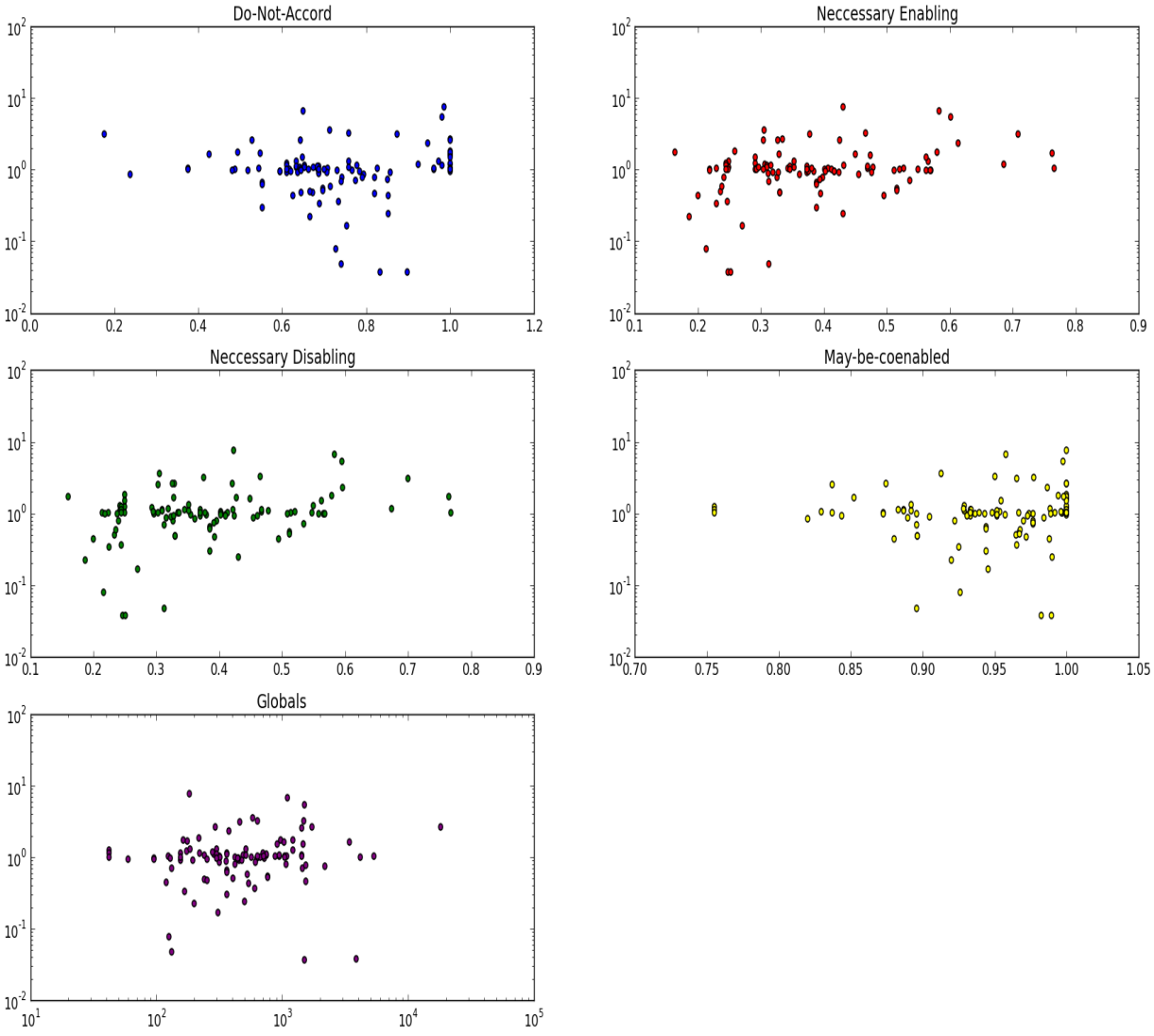


Figure 3: **Time Ratio Benchmark** The horizontal axis represents the fraction of values in the relevant table which is true. The vertical axis is the time required to explore the POR-reduced model, expressed as a fraction of the time required without POR. That is, a value of 1 indicates no change, 0.2 means an 80% reduction and a value of 3 indicates that the query took three times as long when using POR. The "Globals" graph displays the amount of global variables instead. *Note the logarithmic scales.*

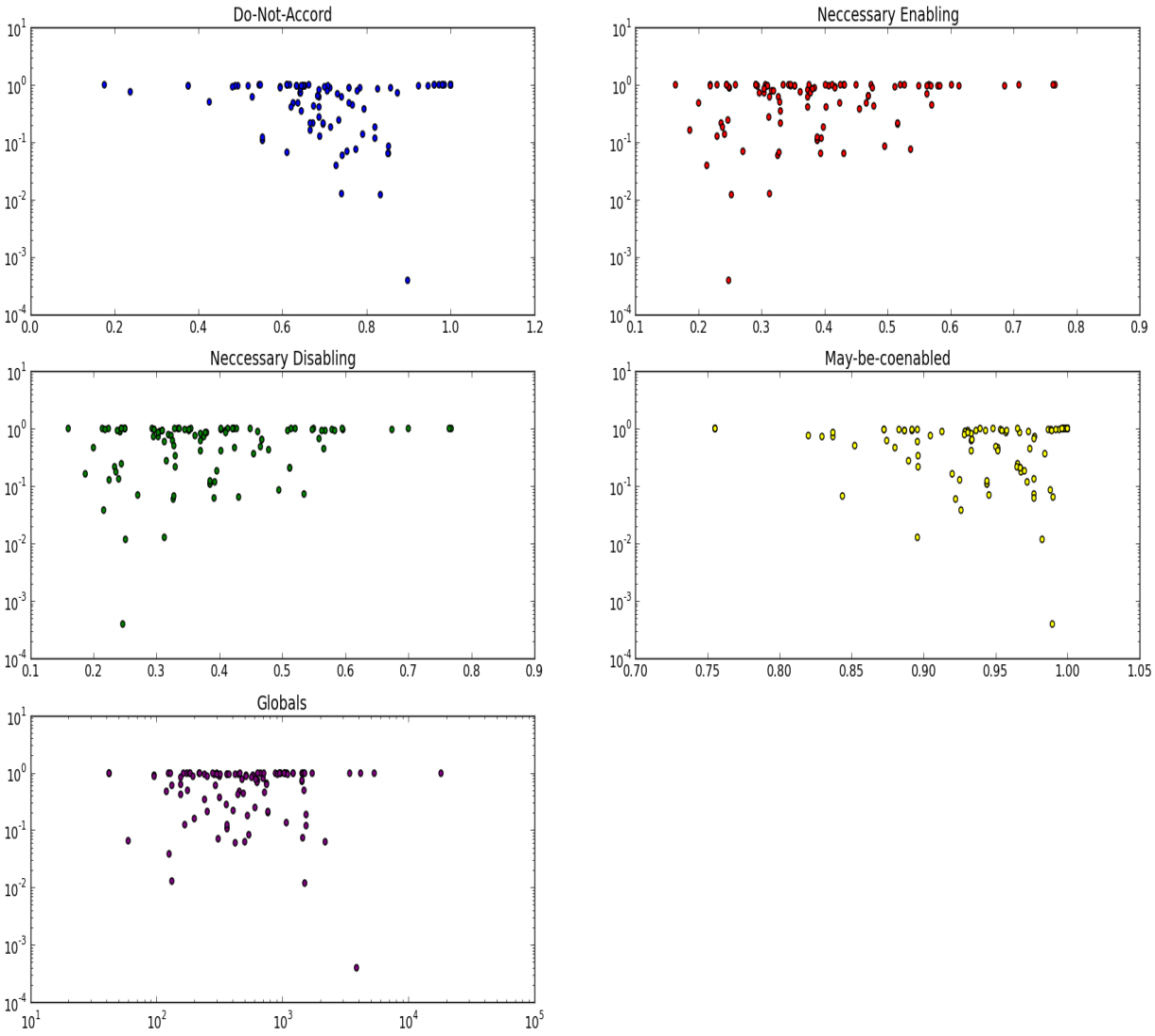


Figure 4: **State Space Ratio Benchmark** Again, the horizontal axis indicates the fraction of true values in the matrix (except in the case of the "Globals" plot). On the vertical axis, we see the amount of states in the reduced state space as a fraction of the original amount. A value of 0.5, then, indicates that POR reduced the size of the state space by half. *Note the logarithmic scales.*

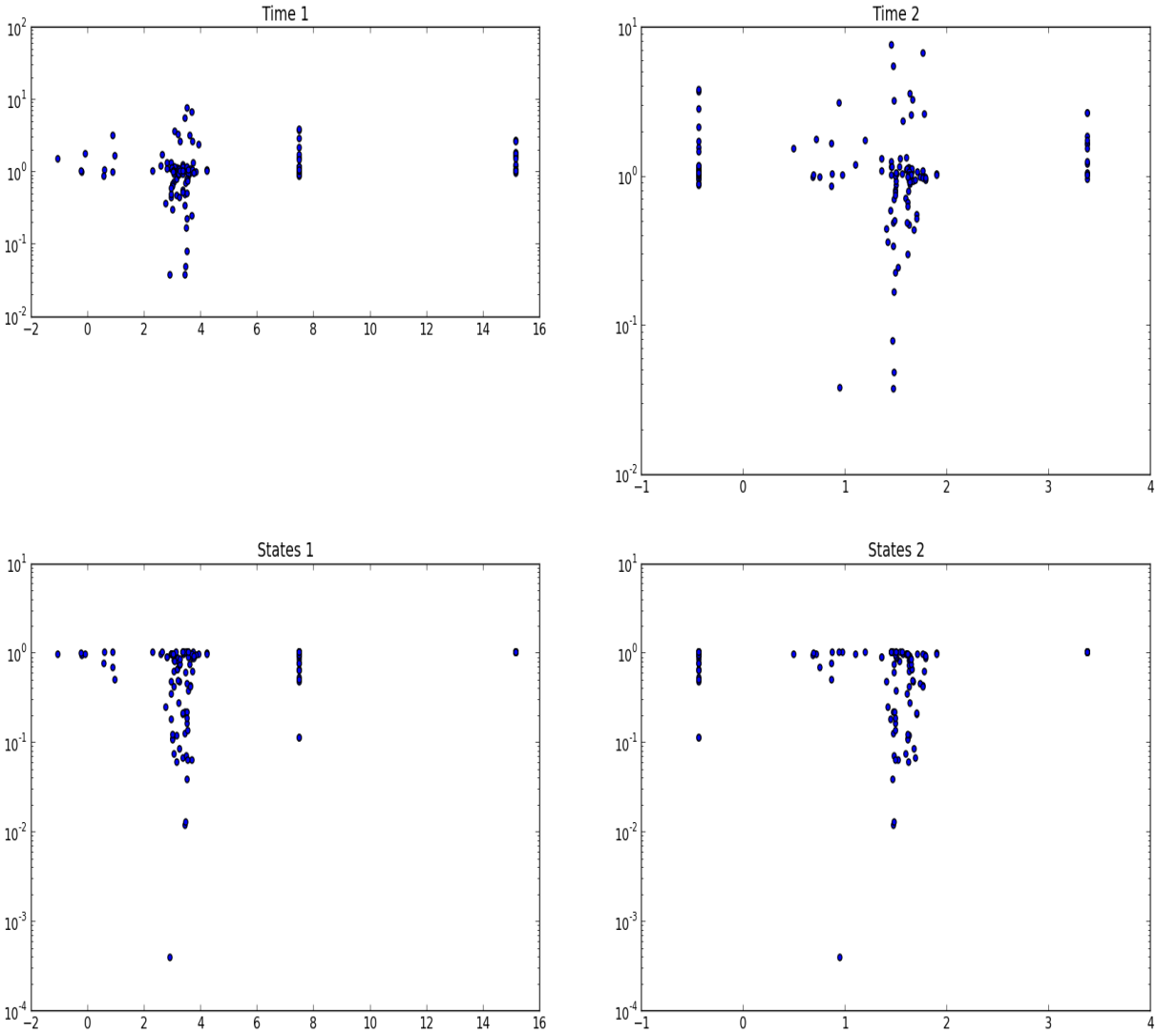


Figure 5: **POR Effectiveness vs. ANN Output** Like in the previous figures, the vertical axes in these plots represent the state space/time requirements of the reduced model relative to the original. The outputs of the ANN are given on the horizontal axes. "Time 1" plots the execution time against the first output, and so on. A color-coded version is available on this project's GitHub page (https://github.com/DennisdeW/PredictPor), which more clearly shows the relation between the advice and the actual effectiveness of POR. *Note the logarithmic scales.*

values, we find that our tool gives good advice in 69% of the test cases, which we believe to be an acceptable result.

4.3 Cross Validation

In order to test the accuracy of the neural network, the goal was to employ K-fold cross validation. As explained in Section 3.2, this means that we first split our dataset into ten (approximately) equal slices. Then, for each slice, we retrain the neural network for all other slices. We then use that network to evaluate the elements in the current slice, and examine the results. Once this process has been completed for every slice, we collect the results and score them according to Equation 1. The time required to train the network is not taken into consideration when calculating *AlgTime*, since this is not a part of the tool's normal operation.

Unfortunately, due to the fact that we had to train a new network for each slice and that these networks are black boxes, the scale of the outputs changed in every network. The 'main' network, the outputs of which are plotted in Figure 5, has relatively low values on the second output, for example, but a new network might have all values above 20. Again, these numbers do not actually represent anything concrete; they only have meaning in relation to themselves. This does mean, however, that the range in which to advice MAYBE instead of NO has to be specified manually each time. Automated verification is made very difficult because of this, and unfortunately it was impossible during this research due to time constraints. Manual inspection of these models in the same way as the main network did show similar results: almost all 'good' models' outputs lie in a rather narrow range of values, but there are also many 'poor' models in those same ranges. In summary, automated verification was infeasible given the time constraint, but manual verification showed results similar to the initial results.

The full table of scores as defined in Section 3.3 is too large to show here, so instead we will summarise the results. All numbers in this paragraph are rounded to the nearest integer. Scores range between -2557 and 2585, with an average of 40. The minimum and maximum values indicate that there are models for which the tool performs very well, and others for which it does very poorly. The positive average score shows that, in general, the tool gives sound advice. In Section 3.3, we defined the final score as being the sum of all individual scores. This sum was found to be 5513, indicating that the positive values strongly outweigh the negative ones.

4.4 New Models

Testing the network on new models, the generated advice is very similar to that for the models used in training. The results were slightly worse than those reported in Section 4.2, but due to the small amount of new models evaluated no conclusions can be drawn from this measurement.

4.5 Gain

The reduction in time and space requirements for models depend largely on the specific model. The scores presented in Section 4.2 use relative numbers, but the absolute values are also highly relevant. For example, some models take under a second to instantiate, with or without POR. These might reduce strongly using POR, but the difference would be hardly noticeable. Conversely, in a model which takes over an hour to instantiate, a slight relative reduction might cause a significant difference.

Assuming that researchers would normally choose to use POR or not at random, which is probably not the case, they would benefit from using this tool. This is because the total score reported in Section 4.3 is positive. The average score of around 40 points indicates that the relative gain may not be very large, so the absolute gain depends very much on the models. On complex ones, the correct choice of whether or not to apply POR could save hours of calculations.

5. CONCLUSIONS

We have seen that despite a clear intuitive connection, little actual correlation appears to exist between the various explored factors and the effectiveness of POR. Nevertheless, the developed tool often produces reasonable advice and may be found useful by researchers working with large models. The absence of a possibility for strong positive advice, however, is somewhat disappointing, as Figures 3 and 5 especially show that for some models POR results in a much shorter run time. To explicitly answer the research questions posed in the introduction:

1. Taking the stricter definition given at the end of Section 4.2, we find a prediction accuracy of 77%. Furthermore, the tool achieves an average score of 40 as defined by Equation 1.
    a. Results on new models appear similar to the original results, but there were too few new models to base any hard claims on.
    b. The gains depend very much on the models in question. For large models, an accurate prediction may save hours. For very small models, the difference may hardly be noticeable.
2. Very little correlation has been found among the explored properties. These were the Do-Not-Accord, Necessary Enabling, Necessary Disabling and May-Be-Coenabled matrices, as well as the global variable count. It is still possible that other properties are more effective, but these could not be investigated due to time constraints.

Future research may include trying to find additional metrics to evaluate the models on, and fine-tuning the neural network structure and training parameters. Specifically, researchers may be interested to find out why the Cascade algorithm described in Section 4.2 is so effective and then use that information to design a better network. A more comprehensive analysis of global variables (for example using dependency graphs) in the models may also help, as we maintain that these should be a helpful clue.

In conclusion, the tool which has been developed over the course of this research gives decent results, even though surprisingly little correlation was found in the analysis of the models. It is our hope that this work will provide direction for future efforts, as improving the process for POR and adapting it for other settings may prove greatly beneficial to people involved in model checking.

6. REFERENCES

[1] R Alur et al. "Partial-order reduction in symbolic state space exploration". In: *Computer Aided Verification, Lecture Notes in Computer Science* 1254/1997 (1997), pp. 340–351. ISSN: 16113349. DOI: 10.1007/3-540-63166-6_34.

[2] J Barnat et al. "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs". In: *Computer Aided Verification (CAV 2013)*. Vol. 8044. LNCS. Springer, 2013, pp. 863–868.

[3] C Flanagan and P Godefroid. "Dynamic partial-order reduction for model checking software". In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 110–121. ISSN: 03621340. DOI: 10.1145/1047659.1040315.

[4] P Godefroid et al. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Vol. 1032. Springer Heidelberg, 1996.

[5] G Kant et al. "Ltsmin: High-performance language-independent model checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 692–707.

[6] DH Maister. *The psychology of waiting lines*. Ed. by Suprenant C. Czepiel J.A. Solomon M.R. Vol. 27. The Service Encounter. Lexington Books, 1985.

[7] TM Mitchell. *Machine Learning*. McGraw-Hill international editions - computer science series. McGraw-Hill Education, 1997. ISBN: 9780070428072.

[8] S Nissen. *FANN Cascade Training*. Accessed 11-06-2016. URL: http://libfann.github.io/fann/docs/files/fann_cascade-h.html.

[9] S Nissen. "Implementation of a fast artificial neural network library (fann)". In: (2003).

[10] R Pelánek. "BEEM: Benchmarks for explicit model checkers". In: *Model Checking Software*. Springer, 2007, pp. 263–267.

[11] R Pelánek. "Properties of state spaces and their applications". In: *International Journal on Software Tools for Technology Transfer* 10.5 (2008), pp. 443–454. ISSN: 14332779. DOI: 10.1007/s10009-008-0070-5. URL: http://link.springer.com/article/10.1007/s10009-008-0070-5.

[12] R Pelánek and P Šimecek. "Estimating state space parameters". In: *Proceedings of the 7th international Workshop on Parallel and Distributed Methods in Verification*. 2008. URL: http://www.fi.muni.cz/reports/files/2008/FIMU-RS-2008-01.pdf.

[13] R Pelánek et al. "Enhancing random walk state space exploration". In: *Proceedings of the 10th international workshop on Formal methods for industrial critical systems - FMICS '05* (2005), pp. 98–105. DOI: 10.1145/1081180.1081193. URL: http://dl.acm.org/citation.cfm?id=1081180.1081193.

[14] D Peled. "All from one, one for all: on model checking using representatives". In: *Computer Aided Verification - 5th International Conference*. Vol. 697. 1993, pp. 409–423. ISBN: 978-3-540-56922-0. DOI: 10.1007/3-540-56922-7_34.

[15] LTSmin Team. *LTSmin Documentation*. Accessed 13-05-2016. URL: http://fmt.cs.utwente.nl/tools/ltsmin/doc/.

[16] NK Treadgold and TD Gedeon. "The Sarprop Algorithm, A Simulated Annealing Enhancement To Resilient Back Propagation". In: *Proceedings International Panel Conference on Soft and Intelligent Computing*. 1996, pp. 293–298.

[17] A Valmari. "Stubborn sets for reduced state space generation". In: *Advances in Petri Nets 1990*. Springer, 1989, pp. 491–515.