Improving Variable Ordering for Saturation by Learning from Small Problem Instances

Johannes Gareis[1], David White[1]
Gerald Lüttgen[1] and Gianfranco Ciardo[2,3]

[1] Software Technologies Research Group, University of Bamberg, 96045 Germany
[2] Computer Science & Engineering Dept., University of California, Riverside, CA 92521 (until Dec 2013)
[3] Department of Computer Science, Iowa State University, Ames, IA 50011-1040 (from Jan 2014)

Abstract. In the domain of symbolic model checking, Saturation is a popular exploration strategy for building a system's state space. A decision diagram is employed to encode the state space; however, its resulting size and thus time taken to build is highly dependent on the ordering of the system's variables. Heuristics for determining good orderings have thus far focused on the static, structural properties of the model. However, a system often exhibits complex dynamic behavior that is difficult to estimate statically.
In this work we propose that variable orderings be learnt from multiple executions of the Saturation algorithm on small problem instances, with the expectation that learnt variable orderings will also perform well when the problem is scaled up. The benefits are twofold: firstly, by learning orderings through executing the algorithm, we expect to capture some of the dynamic behavior, and secondly, as Saturation has proven difficult to parallelize, a machine learning step could prove an easy route to parallelization. To this end we detail two approaches, one for models that are of fixed-size and only scale in terms of the information passed between model components, and one for models with topologies that grow in a regular manner when scaled.

Keywords: Symbolic Model Checking, Machine Learning, Decision Diagrams, Variable Ordering, Saturation Algorithm

1 Introduction

As systems become more complex, it is increasingly important to ensure their correct operation. A common approach to this problem is to first construct a model describing the underlying problem in a formal notation. One such notation are Petri nets [10], which consist of a set of places connected by transitions. The behavior is modeled by conditionally permitting tokens to flow between places in the net. The potential state space of the model can then be explored by a symbolic model checking algorithm to ensure that the model does not display

any undesirable behavior. However, when models are scaled to realistic scenarios they can exhibit prohibitively large state spaces and thus make model checking infeasible. Model scaling in a Petri net can take one of two forms, either through the addition of model components, e.g., adding additional philosophers in the Dinning Philosophers problem [8], or in terms of the information flowing between model components, e.g., the number of parts to be manufactured in the Finite Manufacturing System [8].

One approach to tackle large state spaces is to choose an efficient encoding, such as that offered by decision diagrams. The places of a Petri net, i.e., the variables of the system, are assigned to levels in the decision diagram and then information is propagated by firing events until a fixed point is reached. The time taken to reach a fixed point is highly dependent on the strategy selected to perform the propagation. The Saturation algorithm [8] has been shown to be a good approach, despite being difficult to parallelize [9]. However, it is highly dependent on selecting a good variable ordering, i.e., an ordering of the places of the Petri net; badly chosen variable orderings can result in increased time and memory requirements of multiple orders of magnitude. Unfortunately, finding the best order is an NP-hard problem [8].

Heuristics thus far have focused on determining good orderings statically by looking only at the structure of the net, for example, by grouping places via invariants [7]. These static approaches are limited as Saturation often displays much more complex dynamic behavior at runtime, which is very difficult to predict statically. Thus, in this work, we consider a very different approach to the problem; we wish to *learn good orderings* for nets via observing the runtime performance of the Saturation algorithm in terms of the key user observable measurements, time and space. We expect that good orderings learnt on small problem instances, which have state spaces that can be explored quickly, will also turn out to be good orderings when the problem is scaled up. Essentially we are trying to learn a correlation between variable ordering and the execution time/space requirement with the goal of extrapolating this correlation to larger problem instances. This type of approach has two potential advantages, the first is that by actually solving realistic, but small, versions of the problem we hope to handle some of the dynamic behavior that static approaches find hard to predict. Secondly, as Saturation has proven difficult to parallelize internally, an embarrassingly parallel machine learning approach that produces good variable orderings as a pre-processing step could provide an easy route for Saturation to benefit from modern multicore architectures.

In this paper we describe two possible solutions to learning the desired correlations. The first applies to fixed-size nets, i.e., the problem scales based on the number of tokens present in the net, rather than adding additional places and transitions (Sec. 2). Since the number of tokens does not alter the set of variables, any observations we learn for small problem instances will be directly applicable to large instances. The second approach applies to variable size nets, i.e., the problem scales by adding additional places and transitions to the net (Sec. 3). Therefore, the challenge here is to correlate regularities in good vari-

able orderings with changes in topology for small problem instances. Such learnt regularities can then be extrapolated from in order to directly construct good quality orderings for large scale versions of the problem.

2 Learning Variable Orderings for Fixed Size Nets

As stated above, for fixed-size nets where the problem instance size only varies in the number of tokens N, projecting learnt variable orderings onto more complex nets is trivial as the set of places does not change. Our learning approach is to treat the evaluation of the problem as a black box, which means we may only vary the input and observe differences in the output. For this approach to be successful it will be necessary to evaluate a large number of small problem instances to produce sufficient data for any potential patterns to be learnt. However, we must select a suitably high N such that all behaviors of the net under analysis are exercised. Since this will likely result in a problem instance that is too complex to generate a sufficiently large set of training data, we must find an alternative method for generating the small problem instances.

If we may not reduce N below a certain level then the only other way to reduce the size of the problem is to reduce the size of the net. However, the result of arbitrarily removing places from the net would be one that likely no longer displays any functionality present in the original net, and thus, any information learnt regarding variable orderings would likely be useless.

The solution to reducing the size of a net in a sensible manner lies in its invariants. Net invariants describe a relationship between a set of places in terms of the number of tokens shared between them. Thus they typically describe some small-scale functionality of the net. If we retain the elements of an invariant in the reduced net then it will also maintain the associated functionality. However, due to the tight relationship between the places of an invariant, orderings of just those places from one invariant will likely be of similar quality. Thus if we are to learn information regarding variable ordering from a reduced net, it must consist of at least two invariants. Furthermore, for the reason outlined above, the selected invariants must share at least one place otherwise again the ordering becomes trivial.

2.1 Approach

The goal of our learning algorithm is to determine the preference for two variables of the original net to be placed near to each other in the ordering. This preference data will then be used to find the ordering which maximizes all preferences for variable pair proximities.

Our approach (see Fig. 1) commences with the construction of a large number of reduced nets as described above. Two or more invariants of the original net which share at least one place are randomly selected and are used to form a reduced net. For each reduced net a set of random permutations of the places present in that net is chosen. Given that the places of an invariant interact very

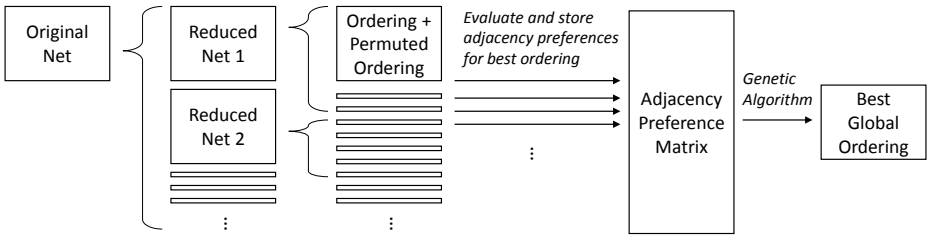

Fig. 1. An overview of the approach proposed in Sec. 2.

tightly with each other, it makes sense to keep them adjacent in the ordering wherever possible. Thus the places of an invariant which are not shared with any other invariant are always kept contiguous. For each random permutation, a second permutation is created by making a minor modification to the first. This modification exchanges the position of two places in the ordering while respecting the preference for the non-shared places of an invariant to be kept contiguous. The reduced net is then evaluated with both orderings and the best ordering is selected based on lowest peak-memory performance. Finally, the preference for the two variables that were exchanged to be adjacent to their neighbors is recorded for the best ordering.

The result of performing the above procedure for all reduced nets and their associated orderings is a symmetric matrix which aggregates the learnt adjacency preferences, i.e., how important it is for two variables to be placed in close proximity in the final ordering. This matrix will likely express many mutually exclusive preferences and thus the optimal is not trivial to find. We solve this problem via a genetic algorithm which evolves a population consisting of individuals (variable orderings) according to a fitness measure (the sum of the adjacency preferences). A mutation operator allows individuals to change by flipping the position of two variables in the ordering (again, subject to keeping the non-shared places of an invariant contiguous).

2.2 Implementation

Our approach is implemented in Python and employs the external tools GreatSPN [2] to compute the invariants and SMART [5] to compute the state-space of a Petri net. Python was chosen due to its suitability for rapid prototyping of a new tool and the ease of integration with pre-existing applications. The evaluation of the reduced nets is an embarrassingly parallel problem and thus we employ multicore parallelism to speed up the process. We avoid any global synchronization by storing each result independently and then aggregating the information in the adjacency preference matrix afterwards. We employ DEAP [1] (Distributed Evolutionary Algorithms in Python) to provide the genetic algorithm framework which also provides built-in support for multicore parallelism.

The Python program takes a Petri net as input in the SMART tool's native format. We extend this format to directly include the invariants computed by

GreatSPN. The net is then disassembled into invariants and the reduced nets are produced. A reduced net is passed to SMART by constructing a suitable file and invoked via the subprocess functionality of Python. The memory results are then read from the standard output and recorded. We found that the internal time measurement of SMART did not accurately reflect the runtime, so we implemented an external timer and used this value instead.

2.3 Evaluation

We evaluated our approach on two fixed size nets: Flexible Manufacturing System (FMS) and Kanban. The success of a learnt ordering is evaluated by applying it to the original complete net. Since we wish to show that the learnt ordering scales, we perform the evaluation for multiple values of N set by the parameter *Net parameters for evaluation*. Due to the random nature of the training data, it is necessary to sample the results over multiple runs of the approach. For each run, the full approach is performed and the best learnt ordering is evaluated (Parameter: *Number of runs*). There are a number of additional parameters involved in the approach which are discussed below:

- *Net parameter for learning:* There is a tradeoff to be considered when selecting the value of N for which learning takes place. As discussed, if N is set too low then the behavior of the system is not realistic and the orderings we learn will not be of use. However, a small value of N is advantageous as these problem instances may be explored quickly and we can generate more data points to learn from in a specified time than for a higher value of N. Interestingly there is also an argument for learning with a large value of N. Since the measurements have exponential tendencies with respect to N, the differences between good and bad orderings become exaggerated which creates richer training data.
- *Number of reduced-net problems:* The number of reduced net problems generated for learning.
- *Max selected invariants:* Limits the number of invariants that may be selected for generating a reduced net. If set too high then it allows the original net to be generated, thus the set of reduced-net problems might include many versions of the original net.
- *Time-out for SMART during learning:* Since the reduced nets and applied orderings are generated randomly, it is quite possible there will exist some that take a very long time to solve. We impose a time-out to prevent such problems unnecessarily increasing the runtime.
- *Time-out for SMART during evaluation:* For the same reason we also impose a time-out on the evaluation of learnt orderings, however, this is significantly longer than the learning time-out.

Flexible Manufacturing System. The Flexible Manufacturing System describes a system where three different types of parts are processed by three

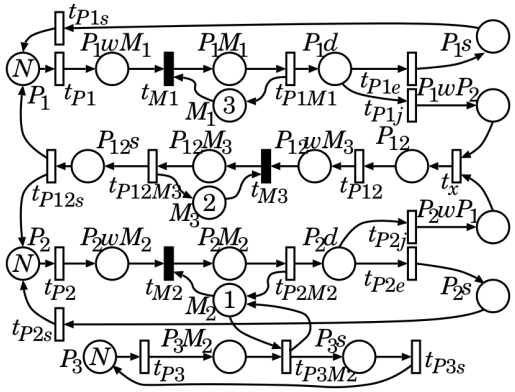

Fig. 2. The Petri net for the Flexible Manufacturing System problem [8]

different machines. The total number of all parts is N which comprises the net scaling parameter. Machine 1 can only work on parts of type 1. Machine 2 machines parts of type 2 and parts of type 3, if there are no parts of type 2 left. After the processing on machine 1 and 2 the parts can be shipped or joined to the parts of machine 3. If the parts are shipped, machine 1 and 2 get new raw parts of their corresponding type. The three machines and the processing of the parts are modeled by a Petri net, which is shown in Fig. 2 [8, 6].

The following parameters were used on this problem:

- *Net parameters for evaluation:* 10, 20, 30, 40, 50
- *Number of runs:* 50
- *Net parameter for learning:* 15
- *Number of reduced-net problems:* 10000
- *Max selected invariants (MSI):* 1, 2 or 3
- *Time-out for SMART during learning:* 400s
- *Time-out for SMART during evaluation:* 4000s

We began the evaluation by setting *Max selected invariants* (MSI) to 2. As discussed earlier, we are interested in learning orderings from reduced nets which display some interaction between the places of two invariants, otherwise we end up with trivial problems. The time and memory results of this experiment can be seen in Fig. 4. The results are encouraging and show when evaluated using memory the learnt orderings are always better than the ordering proposed by the domain expert for all evaluation N's. When evaluated using execution time, both orderings have similar performance.

We wanted to see what effect adding some complete versions of the original net to the set of reduced problems would have, so we increased MSI to 3. The results are shown in Fig. 5. Interestingly, this change significantly reduces the quality of the learnt ordering both in terms of memory and time requirements. This means that the addition of some complete versions of the original net reduces the suitability of the training data.

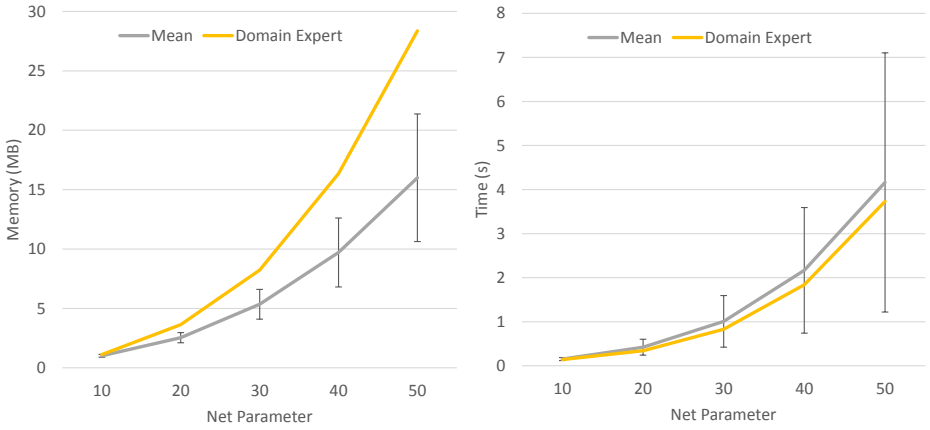

Fig. 3. FMS: peak memory usage (left) and time (right) for MSI = 1.

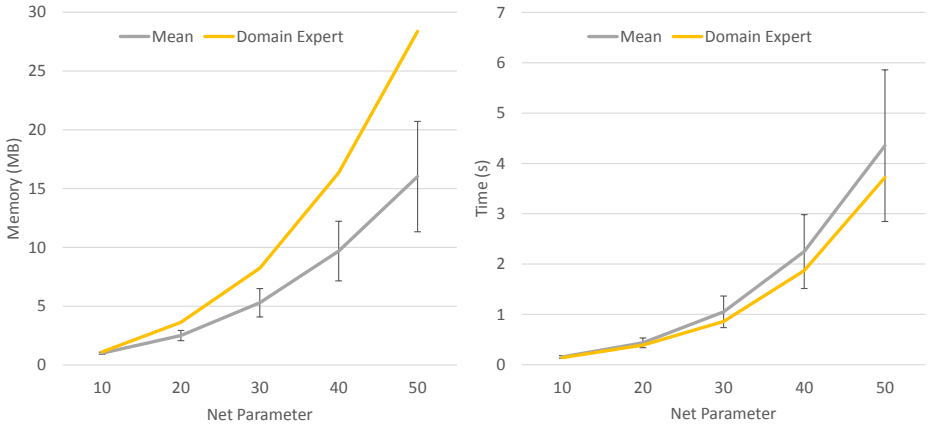

Fig. 4. FMS: peak memory usage (left) and time (right) for MSI = 2.

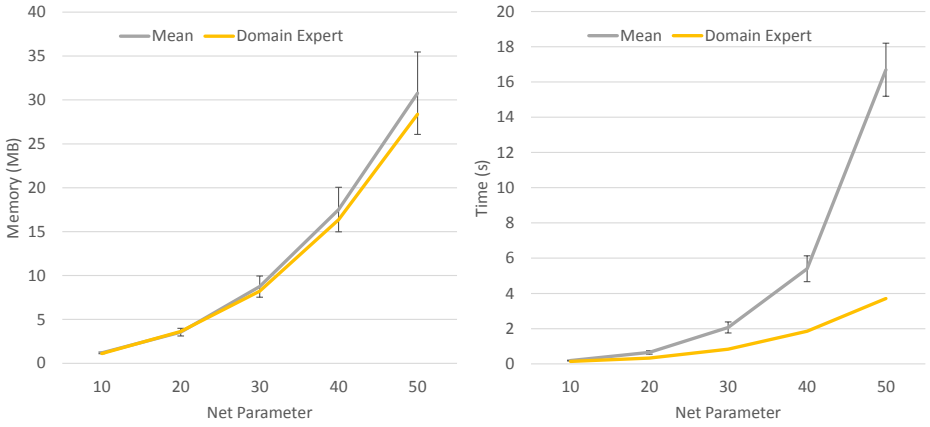

Fig. 5. FMS: peak memory usage (left) and time (right) for MSI = 3.

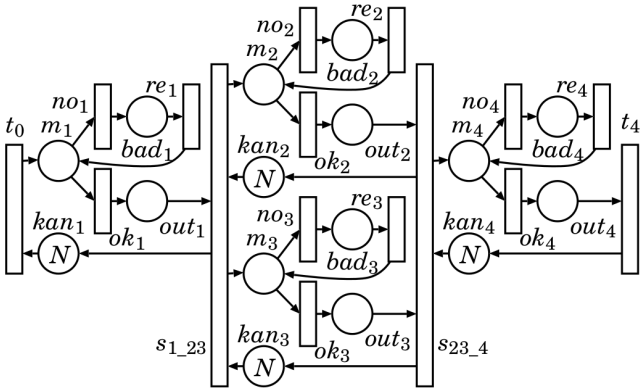

Fig. 6. The Petri net for the Kanban problem [8]

This was a surprising result and led us to consider what information was actually being learnt when MSI=2. Thus we reduced MSI to 1 and reran the evaluation. The results are shown in Fig. 3 and surprisingly are almost identical to those of MSI=2. Since these results stem from reduced nets consisting of the places of only one invariant, we can draw only one unfortunate conclusion: that all our approach really learns is that the places of one invariant must be kept in close proximity. Regarding the set of places shared between invariants, there are two possibilities: a) their order is unimportant or b) our approach is incapable of learning good orderings for the shared places. Lastly, we note that different values for the other parameters were tested, but we observed no difference in the overall data trends.

Kanban. Given the disappointing results for the FMS problem, it is important to see if this trend continues to other problems. Thus we evaluated the approach on the other fixed-size net available to us, Kanban, which models an assembly line consuming tokens, i.e. the kanbans, and consists of 4 sub-machines. After a token is machined at sub-machine 1 it divides into two parts, which go to sub-machine 2 and 3. Before entering sub-machine 4, the two parts are rejoined and machined before finally leaving the whole system. It is important to note that a kanban can only enter a sub-machine if there is another machined kanban that can leave, since it might happen that a machined kanban has to be reprocessed. This means, for example, that a kanban can only leave sub-machine 1 if there exist kanbans at sub-machines 2 and 3 that can also leave. The system is parametrized by the number of kanbans N being available at each sub-machine. Fig. 6 shows the Petri net of the Kanban problem [8, 6].

The parameters are set identically to those of the FMS problem with the exception of the *Net parameters for evaluation*. These ranged from 10 to 100 as the Kanban problem typically requires less time to solve than FMS. Again other parameter values were tested but with no differences in the data trends.

The results of the approach on the Kanban problem are given in Fig. 7 for MSI=1 and Fig. 8 for MSI=5 and display two interesting characteristics. Firstly, the learnt orderings outperform the one chosen by the domain expert by a significant amount, both in terms of memory and time requirements. Secondly, the performance of the learnt orderings is identical (within experimental error) for all values of MSI, which is very different to the FMS problem. This behavior is illustrated clearly in Fig. 9.

To understand these characteristics, we must look at the invariants of the model (see Fig. 6) which are as follows:

1. $\{m_1, re_1, kan_1, out_1\}$
2. $\{m_2, re_2, kan_2, out_2\}$
3. $\{m_3, re_3, kan_3, out_3\}$
4. $\{m_4, re_4, kan_4, out_4\}$
5. $\{m_2, re_2, out_2, kan_3\}$
6. $\{m_3, re_3, out_3, kan_2\}$

Invariants 1 – 4 each correspond to a sub-machine, and invariants 5 and 6 relate sub-machine 2 to sub-machine 3. The first four invariants completely cover the places of the net without intersecting, which we term the *non-intersecting set*.

The ordering proposed by the domain expert groups the places by sub-machine, corresponding to the first four invariants, and these are placed in order in the MDD with the places of sub-machine 1 at the bottom and the places of sub-machine 4 at the top. This appears to be a perfectly logical ordering, however, the ordering of sub-machines in the learnt orderings seems arbitrary. To discover how an arbitrary order could be out-performing an apparently logical one, we enumerated all possible orderings of the non-intersecting set of invariants. Note that this was not possible for the FMS problem as it did not have a non-intersecting set of invariants which completely covered the places of the net.

The data from this experiment is given in Table 1 and shows the peak memory required for increasing values of the net parameter N. The ordering[4] is given in terms of the indexes of the non-intersecting set of invariants as given above. It is clear that the order in which the invariants of the non-intersecting set are placed makes an enormous different to the required memory. The best performance is obtained with a reverse ordering modulo the position of sub-machines 2 and 3. Note that the ordering proposed by the domain expert, marked with (DE), is ranked rather low in the table. With this insight both observations raised earlier for the Kanban problem can now be explained. Since our approach forces the non-shared places of an invariant to remain adjacent in an ordering, the quality of an ordering becomes dominated by the order of the non-intersecting set of invariants. Thus if our approach randomly picks an ordering for this set, then on average it will be better than the domain expert's ordering (see Table 1). We expect that this random-like behavior is occurring when MSI=1. Since we

[4] In Table 1 and the following, the first item in an ordering is always placed at the bottom of the decision diagram and the last item at the top (see Sec. 3.3 of [4]).

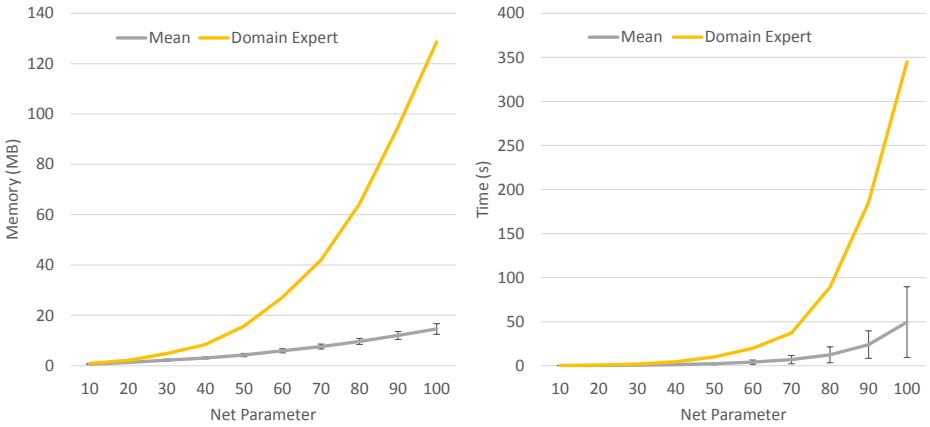

Fig. 7. Kanban: peak memory usage (left) and time (right) for MSI = 1.

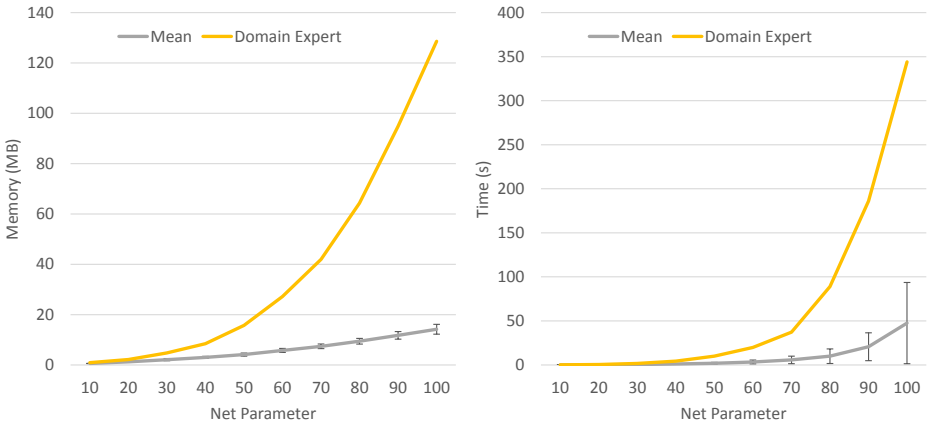

Fig. 8. Kanban: peak memory usage (left) and time (right) for MSI = 5.

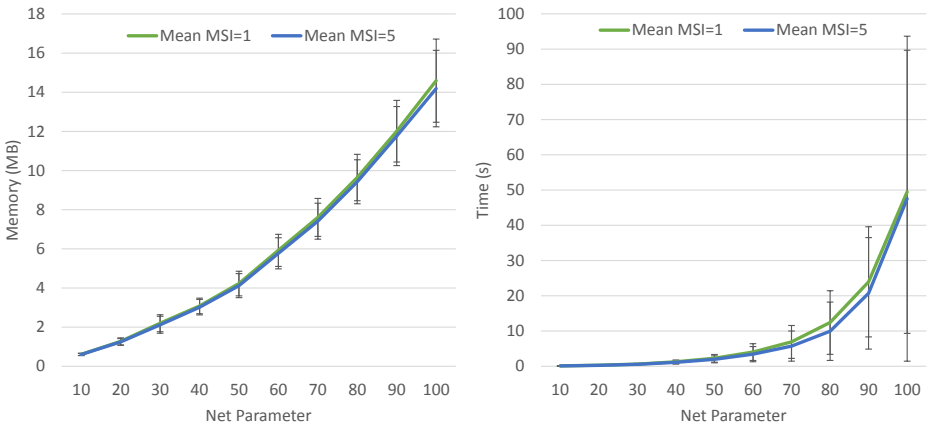

Fig. 9. Kanban: comparison for MSI=1 and MSI=5 for memory usage and time.

Table 1. Peak memory results for all ordering permutations based on the non-intersecting set of invariants. Orders are described using the invariant indexes above. Values are in MBs and TO indicates the timeout of 15s was reached. The columns are labeled by increasing value of net size parameter so scaling performance can be observed. Permutations are sorted for N=40 and the domain expert's ordering is indicated by (DE).

Ordering	10	20	30	40	50	60	70	80	90	100
4, 2, 3, 1	0.28	0.74	1.48	2.08	2.70	3.25	4.00	4.66	5.75	5.64
4, 3, 2, 1	0.28	0.74	1.48	2.08	2.70	3.25	4.00	4.66	5.75	5.64
4, 1, 2, 3	0.33	1.06	1.82	2.46	2.98	3.45	4.16	4.47	5.06	TO
4, 1, 3, 2	0.33	1.06	1.82	2.46	2.98	3.45	4.16	4.47	5.06	TO
4, 2, 1, 3	0.34	1.09	2.05	3.35	4.36	6.42	8.48	11.03	TO	TO
4, 3, 1, 2	0.34	1.09	2.05	3.35	4.36	6.42	8.48	11.03	TO	TO
1, 4, 2, 3	0.44	1.49	2.94	3.69	5.20	7.11	9.07	TO	TO	TO
1, 4, 3, 2	0.44	1.49	2.94	3.69	5.20	7.11	9.07	TO	TO	TO
2, 4, 3, 1	0.48	2.05	2.89	3.93	5.07	6.75	TO	TO	TO	TO
3, 4, 2, 1	0.48	2.05	2.89	3.93	5.07	6.75	TO	TO	TO	TO
2, 4, 1, 3	0.55	2.21	3.23	4.41	6.21	7.97	TO	TO	TO	TO
3, 4, 1, 2	0.55	2.21	3.23	4.41	6.21	7.97	TO	TO	TO	TO
2, 3, 4, 1	0.43	1.60	4.23	5.39	13.52	15.26	23.75	TO	TO	TO
3, 2, 4, 1	0.43	1.60	4.23	5.39	13.52	15.26	23.75	TO	TO	TO
2, 1, 4, 3	1.01	3.59	4.96	6.63	TO	TO	TO	TO	TO	TO
3, 1, 4, 2	1.01	3.59	4.96	6.63	TO	TO	TO	TO	TO	TO
2, 3, 1, 4	0.46	1.94	4.76	7.08	13.93	21.66	TO	TO	TO	TO
3, 2, 1, 4	0.46	1.94	4.76	7.08	13.93	21.66	32.31	TO	TO	TO
1, 2, 3, 4 (DE)	0.48	2.02	4.79	8.47	15.77	27.19	TO	TO	TO	TO
1, 3, 2, 4	0.48	2.02	4.79	8.47	15.77	TO	TO	TO	TO	TO
2, 1, 3, 4	0.96	3.57	6.61	8.68	TO	TO	TO	TO	TO	TO
3, 1, 2, 4	0.96	3.57	6.61	8.68	TO	TO	TO	TO	TO	TO
1, 2, 4, 3	0.54	2.18	4.29	10.50	TO	TO	TO	TO	TO	TO
1, 3, 4, 2	0.54	2.18	4.29	10.50	TO	TO	TO	TO	TO	TO

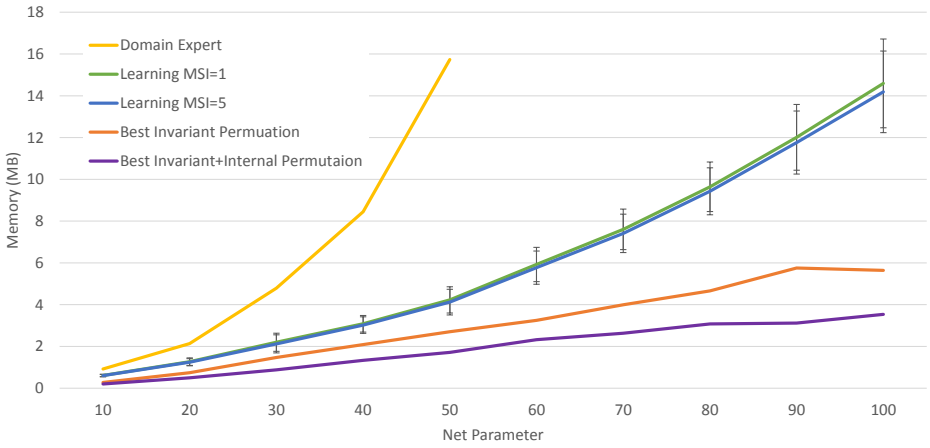

Fig. 10. Comparison of the best ordering from each source: the domain expert, the learning approach for two values of MSI, the enumeration of invariants and the enumeration of internal order within an invariant.

observe the same performance when MSI=5, the learnt data for the places shared between invariants must be poor or obfuscated such that the learnt ordering acts like a random choice as well.

To show how much improvement could still be obtained over our approach, we have plotted the performance of the best ordering from Table 1 in the series *Best Invariant Permutation* of Fig. 10. Given the improvement, it is interesting to consider if this enumeration approach has merit. While it is true that this approach is only feasible for nets where the non-intersecting set is small, by looking at the data it is clear that bad orderings could be quickly eliminated and only promising orderings explored further. It is important to state that the relative ordering of places within an invariant was kept constant at m, re, kan, out, and it might be that opportunities for further optimization exist by exploring permutations of the internal invariant ordering.

In the Kanban domain this process is simple as all invariants from the non-intersecting set have a regular structure. Thus we ran a second enumeration experiment to vary the internal order of places inside a machine and then apply this to each invariant. The best order for invariants from the previous experiment was used. The results are reported in Table 2 and again show improvement is possible, although not as drastic as before. Note that again the internal order suggested by the domain expert is low in the ranking, which gives further evidence to the argument explaining the performance of the learnt orderings. The best internal ordering in combination with the best invariant ordering is shown in Fig. 10.

Table 2. Peak memory results for internal permutation of the places of each invariant. Values are in MBs. The columns are labeled by increasing value of net size parameter so scaling performance can be observed. Permutations are sorted for N=100 and the domain expert's ordering is indicated by (DE).

Ordering	10	20	30	40	50	60	70	80	90	100
out, *re*, *m*, *kan*	0.20	0.50	0.88	1.34	1.71	2.32	2.63	3.08	3.12	3.54
re, *out*, *m*, *kan*	0.21	0.51	0.90	1.35	1.73	2.35	2.64	3.10	3.10	3.56
re, *m*, *out*, *kan*	0.22	0.55	1.03	1.35	1.73	2.03	2.50	2.70	3.20	3.72
out, *re*, *kan*, *m*	0.21	0.52	0.98	1.39	1.79	2.42	2.75	3.15	3.15	3.74
re, *out*, *kan*, *m*	0.21	0.53	1.00	1.40	1.81	2.45	2.76	3.16	3.22	3.77
out, *m*, *re*, *kan*	0.23	0.58	1.08	1.46	1.90	2.40	2.75	2.90	3.35	3.86
re, *m*, *kan*, *out*	0.23	0.57	1.06	1.40	1.80	2.12	2.60	2.86	3.37	3.91
out, *m*, *kan*, *re*	0.24	0.61	1.14	1.56	2.17	2.58	2.96	3.05	3.60	4.17
kan, *out*, *re*, *m*	0.24	0.65	1.28	2.02	2.70	3.28	3.52	3.83	4.93	4.68
kan, *re*, *out*, *m*	0.25	0.66	1.30	2.02	2.71	3.28	3.52	3.83	4.93	4.69
re, *kan*, *out*, *m*	0.23	0.58	1.12	1.57	2.21	2.84	2.94	3.47	4.02	4.72
kan, *re*, *m*, *out*	0.25	0.67	1.31	2.04	2.73	3.50	3.56	3.97	5.21	4.74
kan, *out*, *m*, *re*	0.25	0.68	1.32	2.07	2.77	3.80	3.61	4.01	5.19	4.74
re, *kan*, *m*, *out*	0.23	0.59	1.14	1.58	2.22	2.85	2.98	3.50	4.05	4.77
m, *re*, *out*, *kan*	0.27	0.71	1.39	1.92	2.47	2.92	3.51	4.14	5.13	4.92
m, *out*, *re*, *kan*	0.27	0.72	1.40	1.93	2.48	2.93	3.54	4.15	5.13	4.93
m, *kan*, *re*, *out*	0.27	0.69	1.30	1.80	2.53	3.26	4.12	4.56	5.31	5.55
m, *kan*, *out*, *re*	0.27	0.70	1.31	1.81	2.54	3.27	4.13	4.57	5.32	5.57
m, *re*, *kan*, *out* (DE)	0.28	0.74	1.48	2.08	2.70	3.25	4.00	4.66	5.75	5.64
m, *out*, *kan*, *re*	0.28	0.76	1.51	2.28	2.78	3.34	4.09	4.78	5.92	5.72
out, *kan*, *re*, *m*	0.23	0.62	1.20	1.75	2.49	3.15	3.45	4.35	5.05	5.75
out, *kan*, *m*, *re*	0.24	0.64	1.24	1.92	2.54	3.22	3.53	4.43	5.16	5.88
kan, *m*, *re*, *out*	0.28	0.76	1.48	2.38	3.23	4.45	5.69	6.41	6.98	8.22
kan, *m*, *out*, *re*	0.28	0.76	1.49	2.39	3.24	4.46	5.70	6.42	7.00	8.23

2.4 Conclusion

In this section we proposed an approach for determining good variable orderings based on learning ordering behavior from reduced versions of the original problem. It is necessary to use invariants to reduced the original problem in a sensible way, but unfortunately this seems to result in the approach only being capable of learning that proximity between the places of an invariant is important, which is trivial and not of interest. Regrettably the interesting information of good orders over the places shared between invariants is not learnt. This behavior was shown on two domains, the Flexible Manufacturing System and Kanban. We also wanted to evaluate it on the n-queens problem instantiated with a reasonably high n, e.g. 9, which would result in a net with many invariants and could potentially show the worth of the net reduction strategy. However, unfortunately we were unable to obtain the set of invariants as GreatSPN timed out.

While investigating the results observed from the Kanban problem, we discovered that a exhaustive multi-stage search guided by invariants might be suitable for nets with an appropriate set of invariants, and might even be feasible if that set is not too large and non-promising orderings can be quickly pruned. However, with regard to the original approach proposed in this section it is clear that the preexisting static ordering heuristics would perform just as well and undoubtedly be far quicker to compute.

3 Learning Variable Orderings for Variable Size Nets

In this section we consider domains for which additional places and transitions are added to the Petri net representation as the problem scales. In contrast to the domains considered in the previous section, good quality variable orderings for one value of N (the net scaling parameter) are now no longer trivially applicable to another value of N. The key observation to enable a transfer of information between models for different values of N is that the topology of the model grows with structural regularities. The examples we consider in this section have ring topologies, i.e., the model is scaled from N to $N + 1$ through the addition of a node to the ring, as is the case in the Dinning Philosophers problem. It is important to note that a node will likely contain multiple places and transitions; however, we will abstract from this and only consider problem scaling at the level of nodes.

Therefore, the goal here is to learn regularities in good orderings for small sizes of the ring topology. By extrapolating from these learnt regularities we hope to directly produce high quality orderings for large scale versions of the problem. This approach is much more speculative in nature and, thus, we first wish to discover if such an approach has merit and should be continued in future work.

3.1 Approach

Since we have no intuition as to what good orderings would be for ring networks, and we can produce small instances of the problem easily when compared to

fixed-size nets, then the obvious starting point for this approach is an exhaustive enumeration of all variable orderings. We aim to perform this for as many small instances of the problem as is computationally feasible in order to maximize the number of data points available for learning regularities. To reduce the number of permutations, and thus increase the number of data points, we will keep the places of a single node adjacent in the ordering. Thus, for n nodes, we have $n!$ permutations. Clearly this will quickly grow infeasible; however, by that point we hope to have collected enough data to find the desired regularities in the orderings. With the data from the exhaustive enumeration to hand, we can quickly locate the best quality orderings for each value of N and try to learn the desired regularities between orderings. We will initially learn the regularities and apply them to large scale problem instances by hand. If the approach shows merit, we will then consider learning approaches that are able to discover the regularities automatically.

3.2 Implementation

As before, the approach is implemented in Python and uses SMART [5] to compute the state space of a Petri net. Given the enormous number of problems to be solved in this enumeration approach, parallelization is of key importance. To prevent synchronization issues we attempted to store a description of each problem and its result separately, but it soon became apparent that this was out of scope for Python's lists. Thus the permutations, and hence problem instances, are generated on demand by an in-place algorithm. Initially, we handed out the problems to workers from a thread pool; however, due to Python's global interpreter lock this resulted in poor performance. To overcome this restriction we switched to using Python's multiprocessing tool, and all threads were replaced by processes.

3.3 Proof of Concept

We will apply the proof of concept implementation to two domains, which both have variable size nets and scale by the addition of a node in a ring topology.

Dining Philosophers. We begin by considering the classical concurrency problem of the Dinning Philosophers [8]. The model consists of N philosophers, where each is represented by a sub-model that comprises one node in a ring topology. Philosophers attempt to eat by acquiring two resources, namely the fork to their left and right, one at a time. However, since the model only contains one fork between each philosopher, deadlock can arise. The Petri net for the i-th dining philosopher is shown in Fig. 11. Note that the two places representing forks are shared between philosophers.

We found it feasible to evaluate all permutations of the ordering until 9!. The data gathered from this exploratory experiment is shown in Tables 3 and 4. They

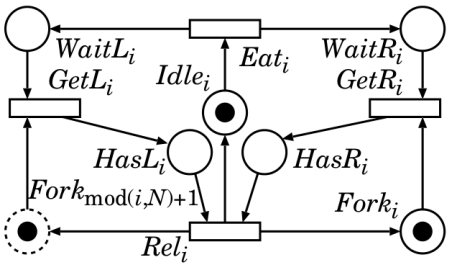

Fig. 11. The Petri net for a single philosopher in the Dinning Philosophers problem [8].

show the top five best performing orderings for $N = 4$ through $N = 9$ when evaluated in terms of the peak memory requirement of SMART. Note that only permutations of the node ordering in the ring were considered; the ordering of places comprising a philosopher were held constant at $\langle Idle_i, WaitL_i, WaitR_i, HasL_i, HasR_i, Fork_i\rangle$. The orderings are given by the ordering of the places for subscript i.

Even for this small range of values of N, clear patterns emerge from the data. The subscript ordering $\langle N-2, N-3, \ldots, 0, N-1\rangle$ (highlighted in blue) performs best for $N = 4$ and $N = 5$; however, from $N = 6$ its ranking drops. Starting at $N = 6$, the best ordering is $\langle N-3, N-4, \ldots, 0, N-1, N-2\rangle$ (highlighted in orange), and this trend continues until $N = 9$. There exist other high quality orderings which repeat in the data of Table 3.

Given this strong preference for permutations where only a cyclic shift takes place, we decided to explore such permutations for larger values of N. It is necessary to generate all cyclic shift permutations of both normal and reverse orderings, and hence the number of permutations to be enumerated reduces to a much more manageable $N * 2$. We will not show the data here for the sake of space; however, we found that the ordering $\langle N-3, N-4, \ldots, 0, N-1, N-2\rangle$ (shown in orange) continues to perform best until at least $N = 100$.

This ordering is slightly surprising and does not seem very logical, which is again exactly the type of interesting information we hoped the approaches of this paper would bring to light. However, the ordering proposed by the domain expert $\langle 0, 1, 2, \ldots, \mathrm{N}\rangle$, which seems more logical, does not perform significantly worse. In fact, we only noted a 22% difference between the best and worse peak memory requirements for all cyclic permutations when $N = 500$. We would postulate that any ordering that is close to a cyclic permutation performs quite well. Before considering how the regularities of the orderings observed in this section can be discovered automatically, we turn to a second domain to see whether regularities are also present.

Slotted Ring Transmission Protocol. This domain considers the problem of arbitrating access to a single shared resource. Nodes are arranged in a ring topology, where a token is passed between them; if a node has the token, then

Table 3. Top five permutations in the Dinning Philosophers problem for small values of N. Permutations are shown in order of peak memory requirement, with the best performing at the top. Colors are used to indicate instances from the same pattern.

N=4	N=5	N=6
2, 1, 0, 3	3, 2, 1, 0, 4	3, 2, 1, 0, 5, 4
1, 0, 3, 2	2, 1, 0, 4, 3	4, 3, 2, 1, 0, 5
3, 2, 1, 0	2, 1, 0, 3, 4	3, 2, 0, 1, 5, 4
0, 1, 3, 2	3, 1, 0, 2, 4	3, 2, 1, 0, 4, 5
1, 0, 2, 3	3, 2, 0, 1, 4	3, 1, 0, 2, 5, 4

N=7	N=8	N=9
4, 3, 2, 1, 0, 6, 5	5, 4, 3, 2, 1, 0, 7, 6	6, 5, 4, 3, 2, 1, 0, 8, 7
5, 4, 3, 2, 1, 0, 6	5, 4, 3, 2, 0, 1, 7, 6	6, 5, 4, 3, 2, 1, 0, 7, 8
4, 3, 2, 1, 0, 5, 6	6, 5, 4, 3, 2, 1, 0, 7	7, 5, 4, 3, 2, 1, 0, 6, 8
5, 3, 2, 1, 0, 4, 6	5, 4, 3, 2, 1, 0, 6, 7	7, 6, 5, 4, 3, 2, 1, 0, 8
4, 3, 2, 0, 1, 6, 5	5, 4, 3, 1, 0, 2, 7, 6	6, 5, 4, 3, 2, 0, 1, 8, 7

Table 4. Peak memory requirements for the Dinning Philosophers permutations of Table 3. Values are in kB.

N=4	N=5	N=6	N=7	N=8	N=9
27.51	33.39	39.21	49.35	55.05	60.99
27.54	33.51	39.58	49.66	55.85	61.56
27.71	34.11	40.01	49.92	55.86	61.73
28.32	34.19	40.05	50.09	55.89	61.74
28.36	34.24	40.13	50.16	56.00	61.80

it may access the shared resource. The internal places of a node are always kept in the following order $\langle A_i, B_i, C_i, D_i, E_i, F_i, G_i, H_i \rangle$. The Petri net for a node in the Slotted Ring Protocol is shown in Fig. 12.

As before, we present the best five permutations for small values of N in Table 5, with the associated memory results being displayed in Table 6. The regularities are even more striking for this domain, where the relative quality of each permutation remains constant as N increases. The best permutation states that the nodes of the ring should be ordered in reverse, and this is the same permutation as that proposed by the domain expert. In contrast to the Dinning Philosophers problem, only the best ranked permutation is cyclic, the others are all reverse orderings with small changes to the order of the first few subscripts. Nevertheless, we exhaustively explored all cyclic permutations until $N = 15$ and found that the simple reverse ordering is always the best permutation.

3.4 Applying Machine Learning to Locate Regularities

We showed in the previous section that there exist regularities in high-quality orderings over small values of N. If the ordering determined from such a set of

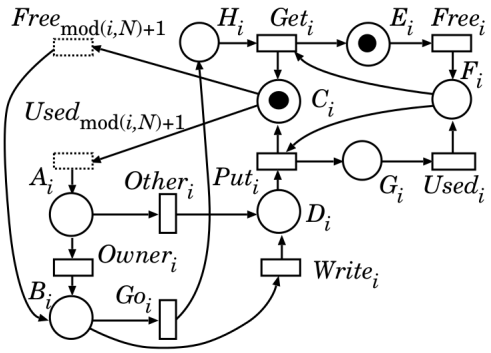

Fig. 12. The Petri net for a single node in the Slotted Ring Protocol [8].

regularities is applied to large scale instances of the problem, then we also observed good performance, at least for the two ring-based topologies investigated. Thus, the key question is how the regularities can be extracted automatically.

The difficulty arises from the fact that, in general, the orderings that correspond to a regularity could be ranked differently over the values of N used for enumeration. We saw this type of behavior in the permutation rankings of the Dinning Philosophers, but not in those of the Slotted Ring Protocol. When all instances of a regularity appear at the same position in the ranking, then the problem becomes significantly easier. One possibility would be to apply approaches from the domain of inductive learning, which synthesize programs from input/output examples. By phrasing the problem as a program that generates the $N+1$th ordering from the Nth ordering, we could provide the appropriate input/output pairs from which to learn. An approach such as Igor [11] might be able to learn the required function if suitable background knowledge of integers were given, and potentially the approach of the Magic Haskeller [3] could learn this with no background knowledge.

Returning to the problem in its complete form, as would be required to learn the patterns observed in the Dinning Philosophers data, the scheme is less clear. One possibility would be to define an edit distance in terms of the cost of transforming an ordering for a problem of size N into one for $N+1$. The only two transformations required would be *move* (from index to index) and *insert* element N. For example, taking the permutations for $N=6$ and $N=7$ from the Dining Philosophers problem, $\langle 3,2,1,0,5,4\rangle$ and $\langle 4,3,2,1,0,6,5\rangle$, the sequence of transformations could be (a) move element at index *end* to the front of the list, then (b) insert N at position $end-1$. The learning problem would then be to search for the edit distance that minimized the following two criteria: firstly, the cost of the edit distance in terms of operations, and secondly, the memory required by the permutation instances described by that edit distance. Unfortunately, such an approach would likely be very expensive, and even more so when combined with the cost of generating the training data through enumeration.

Table 5. Top five permutations in the Slotted Ring Protocol for small values of N. Permutations are shown in order of peak memory requirement, with the best performing at the top. Colors are used to indicate instances from the same pattern.

N=4	N=5	N=6
3, 2, 1, 0	4, 3, 2, 1, 0	5, 4, 3, 2, 1, 0
3, 2, 0, 1	4, 3, 2, 0, 1	5, 4, 3, 2, 0, 1
3, 1, 0, 2	4, 3, 1, 0, 2	5, 4, 3, 1, 0, 2
2, 3, 1, 0	4, 3, 0, 1, 2	5, 4, 3, 0, 1, 2
0, 3, 2, 1	4, 2, 1, 0, 3	5, 4, 2, 1, 0, 3

N=7	N=8
6, 5, 4, 3, 2, 1, 0	7, 6, 5, 4, 3, 2, 1, 0
6, 5, 4, 3, 2, 0, 1	7, 6, 5, 4, 3, 2, 0, 1
6, 5, 4, 3, 1, 0, 2	7, 6, 5, 4, 3, 1, 0, 2
6, 5, 4, 3, 0, 1, 2	7, 6, 5, 4, 3, 0, 1, 2
6, 5, 4, 2, 1, 0, 3	7, 6, 5, 4, 2, 1, 0, 3

Table 6. Peak memory requirements for the Slotted Ring Protocol permutations of Table 5. Values are in kB.

N=4	N=5	N=6	N=7	N=8
75.74	127.53	193.83	313.83	460.86
76.34	128.24	194.55	314.55	461.58
78.17	132.21	196.85	317.07	464.11
78.44	133.58	198.14	318.30	465.34
78.92	133.70	206.45	323.79	467.80

Lastly we note that if domain knowledge were included, such as the fact that the topology was a ring, and valid solutions were limited to only cyclic permutations, then the search for the best permutation becomes trivial and is specified only by the node at which the ring is cut. Thus, for all possible cut points, the set of permutation instances in the training data can be found and their overall performance measured; the cut point with the lowest memory requirement for the set of instances would be selected.

4 Conclusions & Future Work

This paper considered the problem of model scale encountered by symbolic model checking approaches which use decision diagrams for state space encoding. We presented two approaches to learn regularities in good variable orderings from small problem instances, which could then be applied to larger instances. The first approach is applicable for problems in which the underlying model consists of a fixed number of components, i.e., the model grows in complexity only in terms of the information flow between components. We proposed a method of

generating small problem instances by partitioning the model and then solving these to learn patterns. However, this approach appeared to only learn that the places of an invariant should be kept adjacent, which is trivial information and, given the time needed to create the training data, has little advantage over the best static ordering heuristics. Nevertheless, this result serves to strengthen the argument to use static heuristics which employ invariant information such as the one advocated in [7].

A second, more speculative approach was designed to consider problems for which the number of elements in the model scales with the problem's size. Thus, the goal was to learn how the structural changes due to model scale correlate with good orderings. By investigating classic concurrency problems such as Dining Philosophers, we discovered correlations between topology and ordering quality at small scale, which we successfully applied to scaled up versions of the problem. Finally, we proposed various machine learning approaches that could potentially discover these correlations automatically.

Of the two approaches, we feel that only the second would be of interest to pursue in the future. The key factor in determining its worth would be if more problem domains could be found where a good ordering is very unclear, i.e., domains that do not display a ring-based topology. For the two domains considered in this paper, any cyclic permutation has performance in the same order; as discussed earlier, the difference between the best and worst cyclic permutations for the Dining Philosophers was only 22%. A domain such as n-queens which displays a grid topology could be a good starting point.

References

1. DEAP: Distributed Evolutionary Algorithms in Python. `https://code.google.com/p/deap/`. Last accessed 14/12/2014.
2. GreatSPN. `http://www.di.unito.it/~greatspn/index.html`. Last accessed 14/12/2014.
3. MagicHaskeller: An Inductive Functional Programming System for Casual/Beginner Haskell Programmers. `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html`. Last accessed 14/12/2014.
4. SMART User Manual. `https://web.archive.org/web/20100707064932/http://www.cs.ucr.edu/~ciardo/SMART/SMARTman.pdf`. Last accessed 14/12/2014.
5. Ciardo, G., Jones, R., Miner, A., and Siminiceanu, R. Logical and Stochastic Modeling with Smart. In *Computer Performance Evaluation. Modelling Techniques and Tools*, vol. 2794 of *LNCS*, pp. 78–97. Springer, 2003.
6. Ciardo, G., Lüttgen, G., and Miner, A. S. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.
7. Ciardo, G., Lüttgen, G., and Yu, A. J. Improving static variable orders via invariants. In *ICATPN 2007*, vol. 4546 of *LNCS*, pp. 83–103. Springer, 2007.
8. Ciardo, G., Zhao, Y., and Jin, X. Ten Years of Saturation: A Petri Net Perspective. In *Transactions on Petri Nets and Other Models of Concurrency V*, vol. 6900 of *LNCS*, pp. 51–95. Springer, 2012.
9. Ezekiel, J., Lüttgen, G., and Siminiceanu, R. To Parallelize or to Optimize? *Journal of Logic and Computation*, 21(1):85–120, 2011.
10. Ogata, S., Tsuchiya, T., and Kikuno, T. SAT-Based Verification of Safe Petri Nets. In *Automated Technology for Verification and Analysis*, vol. 3299 of *LNCS*, pp. 79–92. Springer, 2004.
11. Schmid, U. and Kitzelmann, E. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3–4):237 – 248, 2011.