Multi-Core Model Checking of Petri Nets with Precompiled Successor Generation

Jakob Dyhr, Mads Johannsen, Isabella Kaufmann, and Søren Moss Nielsen
{jdyhr12,mjohan12,ikaufm12,smni12}@student.aau.dk

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark

Abstract. Model checking of concurrent systems modelled as Petri nets is often a time consuming task and parallelization of this process may reduce the verification time. We present a tool for verification of Petri nets utilizing a multi-core architecture. Our tool extends the VerifyPN engine of TAPAAL to generate a model specific successor generator with the language-independent model checker LTSmin that offers a parallel reachablity analysis algorithm. We show through experiments, that our tool is not faster than VerifyPN when using only a single core, however the use of multiple cores have demonstrated a reduction in verification time from hours to just a few minutes with speedups of up to 43 times on a 64-core machine.

1 Introduction

Model checking [1] is a technique of great success, used to verify finite-state concurrent systems. It suffers from combinatorial explosion also known as state-space explosion, which affects both the time and memory required to explore all the configurations of a system. There have been many attempts to try to overcome this problem effectively [2]. The purpose of model checking is to verify that a system satisfies certain properties, expressed e.g. in temporal logic such as CTL [3] and LTL [4].

Petri net [5] is a modeling language often used to describe concurrent systems, where resources are modelled as tokens. Petri net analysis problems are in general decidable [6] but EXPSPACE-hard [7]. However, we extend the general Petri net with inhibitor arcs making it Turing complete [8] and thereby having all interesting verification problems undecidable [7]. This is the case for unbounded Petri nets, however as bounded nets have finite state spaces, they remain decidable even with inhibitor arcs, and they are usually PSPACE-complete [9].

As the number of cores in modern processors are increasing, the desire to take advantage of multi-core processing rather than single-core processing increases. To the best of our knowledge, parallel model checking is only little explored, as this is the very first year the Model Checking Contest (MCC) [10] has introduced

a multi-core tool category. MCC benchmarks on a database of 404 Petri net models [11]. These models are designed either within the context of academic or industrial projects. The benchmarks are recorded when verifying groups of 16 properties within several verification categories. The competition properties are referred to as queries.

We present a tool that utilizes multi-core architecture and precompiled model generation in an attempt to speed up the model checking. The goal of this work is to gain a linear speedup in verification time compared to the number of cores used. We extend two existing tools: **VerifyPN**, the untimed Petri net engine of TAPAAL [12, 13], and **LTSmin** [14], a language-independent multi-core model checker for labeled transition systems.

Contributions: We describe a multi-core model specific successor generator tool, doing parallel processing using LTSmin. We design a tool for bounded Petri nets with inhibitor arcs, and it can be run as a semi-decision procedure on unbounded nets. We prepared the engine for this years MCC and define a formal syntax and semantics for the informal property language [10] provided by the contest. We verify multiple queries of a model in a single exploration and demonstrate the applicability of reachability analysis and state space exploration on models systematically selected from the MCC database. The degree of which verification speed scales with cores is greatly improved as the state space grows, showing speedups of up to 43 times on a 64-core machine.

Related work: We adopt the technique for verification of timed automata used by the **opaal** tool [15], which generates a model specific successor generator for LTSmin. This has shown good scalability in their field. In contrast, our successor generator is specific for untimed Petri nets with inhibitor arcs.

We build upon VerifyPN [16] by providing a code generation module for successor generation. VerifyPN applies structural reductions [17] on Petri nets when possible. We reuse this technique, and modify it to handle multiple queries at a time. VerifyPN also uses approximative model checking, which is an alternative approach to model checking that does not explore the state space, by solving state-equations [18].

As we engage in this years MCC, we look at the competitors from last year with good results in the categories we participate in. The MCC2014[19] winner, LoLA [20], verifies properties by exhaustive and explicit exploration of a state space. It applies a number of reduction techniques to reduce the explored state space, and supports CTL, LTL, reachability and deadlock analysis, but has no support for parallel model checking.

The winner of the MCC'14 state space generation category is Marcie [21], a tool for qualitative and quantitative analysis of Generalized Stochastic Petri nets. Marcie consist of four engines, all based on interval decision diagrams (IDD). It takes a more symbolic approach than the explicit methods we propose, and makes use of techniques like symbolic state space generation. Marcie does have some features which can utilize more cores like multi-threaded trace generation, but

it does not support parallel verification. The last tool we mention is GreatSPN [22] which came third in the Reachability category. As GreatSPN focuses on providing a friendly and complete framework, they have an extensive list of features as well as a highly developed graphical editor. Some of the techniques involved are reachability graph analysis, markovian solvers, linear programming and simulation. GreatSPN supports verification for a larger number of different types of Petri nets, including Generalized Stochastic Petri nets and their colored extensions. It does not have any support for running multi-core verification.

Outline In the next section we will present the preliminaries for this paper, regarding Petri nets and marking properties. In section 3 will give a short overview of the architecture, and in section 3.2 we present the reasoning behind several implementation choices. In section 4 we evaluate our tool by performing a series of experiments, which are concluded upon in section 5.

2 Preliminaries

A labelled transition system (LTS) is a triple $(S, Act, \rightarrow)$ where S is a set of states, Act is a set of actions and $\rightarrow \subseteq S \times Act \times S$ is a transition relation. We use the notation $s \xrightarrow{\alpha} s'$ iff $(s, s') \in \xrightarrow{\alpha}$.

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^0 = \mathbb{N} \cup \{0\}$ be the set of natural numbers including 0.

Definition 1. *A Petri net is a 5-tuple, $N = (P, T, F, I, W)$ where*
- *P is a finite set of places,*
- *T is a finite set of transitions, where $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,*
- *$I \subseteq (P \times T)$ is a set of inhibitor arcs, and*
- *$W : (F \cup I) \rightarrow \mathbb{N}$ is a weight function.*

A marking on N is a function $M : P \rightarrow \mathbb{N}^0$. All markings on N are denoted $M(N)$.

Fig. 1 (a) illustrates a Petri net. The net consists of four places drawn as circles and two transitions drawn as squares. The arrows are arcs, and can have weights like the one from P2 to T2. Arcs have a weight of 1 if nothing else is specified. The arc with a circle at the end is an inhibitor arc. The dots denote the marking in each place. For example, the current marking is 0, 1, 1 and 0 for P1, P2, P3 and P4 respectively.

The semantics of a Petri net can be described by an LTS. Given $N = (P, T, F, I, W)$ we define (S, Act, $\rightarrow$) as follows: $S = M(N)$ is the set of states, $Act = T$ is the set of labels derived from each transition, and $M \xrightarrow{t} M'$ if and only if :

for all (p,t) $\in F$ we have $M(p) \geq W((p,t))$,
for all (p,t) $\in I$ we have $M(p) < W((p,t))$, and

$$M'(p) = \begin{cases} M(p) & if (p,t) \notin F \wedge (t,p) \notin F \\ M(p) - W((p,t)) & if (p,t) \in F \wedge (t,p) \notin F \\ M(p) + W((t,p)) & if (p,t) \notin F \wedge (t,p) \in F \\ M(p) - W((p,t)) + W((t,p)) & if (p,t) \in F \wedge (t,p) \in F. \end{cases}$$

Fig. 1 (b) illustrates the corresponding LTS of the Petri net in Fig. 1 (a). The LTS has has eight states drawn as circles. Each state consists of a vector with four elements, representing the four places. This vector describes the marking of all places in a given state.

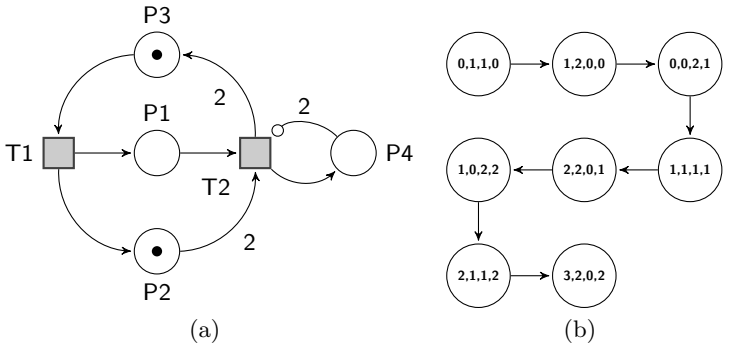

Fig. 1. (a) A Petri net illustrating tokens, places, transitions and inhibitor arcs and (b) the corresponding LTS describing the Petri net.

2.1 Marking Property

We describe a reachability logic based on the informal semantics in the Model Checking Contest (MCC) Property Language [10], and we give a formal syntax and semantics for the language. We give two syntactic categories for boolean and arithmetic expressions:

$$\varphi ::= \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid e_1 \bowtie e_2 \mid \neg\varphi \mid \mathit{deadlock} \mid t \mid \mathit{true} \mid \mathit{false}$$
$$e ::= c \mid p \mid e_1 \oplus e_2 \mid \mathit{bounds}(X)$$

where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $t \in T$, $c \in \mathbb{N}$, $p \in P$, $\oplus \in \{+, -, *\}$ and $X \subseteq P$. The semantics of φ is defined for a marking M as follows:

- $M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$,
- $M \models \varphi_1 \vee \varphi_2$ iff $M \models \varphi_1$ or $M \models \varphi_2$,
- $M \models e_1 \bowtie e_2$ iff $\mathit{eval}_M(e_1) \bowtie \mathit{eval}_M(e_2)$,
- $M \models \neg\varphi$ iff $M \not\models \varphi$,
- $M \models$ deadlock iff $M \xrightarrow{t}$ for all $t \in T$,
- $M \models t$ iff $M \xrightarrow{t} M'$ for some M'.

The semantics of e in a marking M is

- $\mathit{eval}_M(c) = c$,
- $\mathit{eval}_M(p) = M(p)$,
- $\mathit{eval}_M(e_1 \oplus e_2) = \mathit{eval}_M(e_1) \oplus \mathit{eval}_M(e_2)$,
- $\mathit{eval}_M(\mathit{bounds}(X)) = max\{\sum_{p \in X} M'(p) | M \rightarrow^* M'\}$.

We support reachability queries EF and AG defined as follows.

Definition 2. *Given a Petri net N, a marking M and a proposition φ, $(N, M) \models EF\varphi$ iff $M \rightarrow^* M'$ s.t. $M' \models \varphi$. Also $(N, M) \models AG\varphi$ iff $(N, M) \models \neg EF\neg\varphi$.*

We describe the MCC context problems, and how they are expressed in our syntax. Our syntax is more general than the one in MCC Property Language [10], but we will focus on solving these particular problems.

Given a Petri net N and an initial marking M, in ReachabilityDeadlock we ask $(N, M) \models$ EF *deadlock*. In ReachabilityFireabilitySimple we ask $(N, M) \models$ EF φ or $(N, M) \models$ AG φ, where $\varphi ::= \varphi_1 \wedge \varphi_2 \mid t$. In ReachabilityFireability we ask $(N, M) \models$ EF φ or $(N, M) \models$ AG φ, where $\varphi ::= \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid t$. In ReachabilityCardinality we ask $(N, M) \models$ EF φ or $(N, M) \models$ AG φ, where $\varphi ::= \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi$ and $e ::= c \mid e_1 \leq e_2 \mid bounds(X)$. In ReachabilityBounds we ask $(N, M) \models$ EF φ or $(N, M) \models$ AG φ, where $\varphi ::= \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi$ and arithmetic expression $e ::= c \mid e_1 \leq e_2 \mid bounds(X)$. In ReachabilityComputeBounds we ask for the evaluation of $eval_M(bounds(X))$.

The MCC context problems are found on the contests own website [11] along with the models of the competition. For each model, each problem has an XML file containing 16 queries to that specific context (except deadlock).

3 Tool Archictecture

We reuse existing software, one of which is the language-independent model checker LTSmin. LTSmin is a tool set that uses several analysis algorithms to deal with different verification problems. One of the backend algorithms utilizes multi-core model checking, namely the *pins2lts-mc* tool, which we use for both state space exploration and reachability analysis. LTSmin also offers different search strategies for exploring the state space, such as breadth-first-search (BFS) and depth-first-search (DFS). The coupling between language modules such as our tool to the algorithmic backend, is done by implementing the Partitioned Next-State Interface (PINS) [14]. The PINS interface describes a set of functions, for which some are mandatory and others optional. Optional functions can enable high-performance algorithms [14], by considering dependency relations. We are using the dlopen API. as an LTSmin language module to dynamically load the library containing the PINS function definitions. The PINS interface resides between the language modules and the algorithmic backend, as shown in Fig 2.

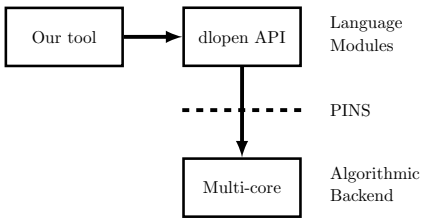

Fig. 2. The PINS interface

The PINS interface has 3 mandatory functions which we define based on the previously defined LTS notation: INITIAL_STATE: () → S. NEXT_STATE: S → 2^S. STATE_LABEL: S → $\{true, false\}$ which is a boolean function answering whether the current state satisfies the GOAL_LABEL, where GOAL_LABEL is the set of conditions for the goal state, depending on the query.

The multi-core tool *pins2lts-mc* spawns a number of threads (workers) [23], whose job is to explore the state space while synchronizing progress information to the other workers. LTSmin stores the visited states in a shared hash table, which is updated whenever a new state is found by a worker. Each worker has its own local work set, containing those states that it is currently examining. Whenever a worker finds a successor of a state in the work set, it checks in the hash table whether the state has been visited by another worker, and adds it to the table and its own work set if the state was not yet visited. The worker steals from other workers if its own work set is empty, and this ensures that the work load is being balanced between each worker.

3.1 Overview

As queries are verified in bunches of 16 in the MCC, our tool supports verification of multiple queries simultaneously, this is referred to as multi-query processing. The possibility of verifying just one query is of course implemented as well. In addition to the Reachability queries, our tool support state space exploration, and is therefore also able to run without a query at all. Verification of a query can be done using a single core or several cores. When using more than one core, we refer to it as multi-core processing. The general flow of our tool is illustrated in Fig.3, where the use of LTSmin as the primary technique for verifying queries becomes is illustrated. **Parser.** As our tool builds on the existing TAPAAL

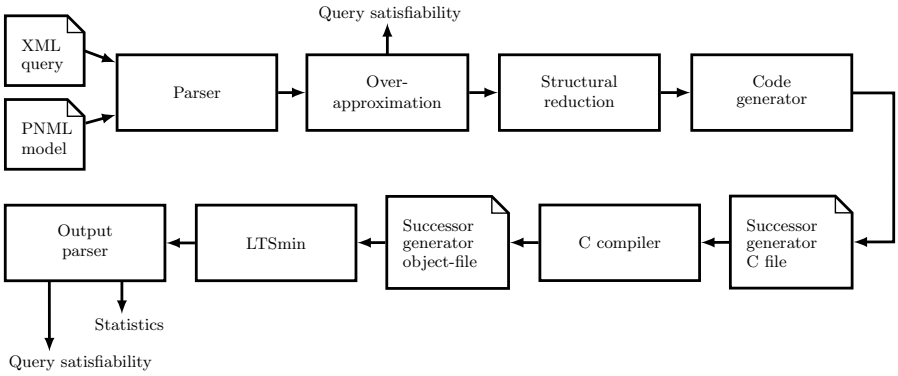

Fig. 3. Architectural overview of our tool

project, more specifically the VerifyPN engine, the code concerned with parsing the net and query are reused, making the intermediate representation of the net an incidence matrix and a string for the query. This representation is used for linear over-approximation, reductions, and as input for the code generator. This shared representation also allows for an expansion of our tool with respect to utilizing the TAPAAL GUI.

Over Approximation. The first step when verifying a query, is the use of the linear over-approximation strategy. Linear over-approximation evaluates whether it can be determined that the state described by a query is unreachable from the initial state by solving the relevant state-equations [18]. For some queries, this will provide the final answer, otherwise the reductions are initiated.

Reductions. In order to make the size of a net smaller, reductions are used to remove unnecessary or redundant places or transitions. Applying one type of reduction modifies the net, such that it might be possible to apply other reductions. The reductions are therefore being applied repeatedly until the net cannot be reduced any further. Obviously, no places with tokens in the initial marking, and no places related to a query can be reduced. These places are referred to as the set of irreducible places. When reducing a model, four general cases are considered [17, 24], illustrated in Fig.4.

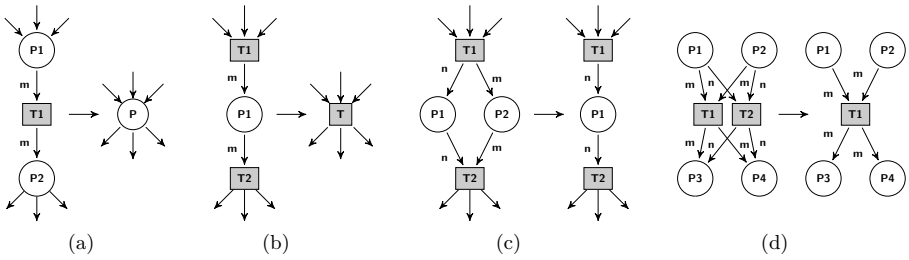

Fig. 4. Reduction applied to different structures of a Petri net.

Not all nets can be reduced and the process takes time. For nets which are unable to be significantly reduced, it can make the verification process slower. The implementation of the linear over-approximation and reduction are largely based on the source code from VerifyPN, and only modified slightly to accommodate multi-query processing and the code generation.

Code Generation. The main contribution of our tool is the generation and use of a successor generator which is specific to each input. This successor generator is implemented in the form of a c-file named autogenerated.c. To work with LTSmin we use the language front-end POSIX/UNIX dlopen API, which connects to the multi-core backend through the PINS interface. It is implemented in the dlopen-impl.c file which compiles with the autogenerated.c file, to create a shared object file, resulting in the model specific successor generator, see Fig. 3. The dlopen-impl.c file is static and it implements the functions not specific to the individual case, and set function pointers to those who do change according to input. The autogenerated.c file then implements the functions pointed to by dlopen-impl.c, the most interesting of them being the *next_state*, and the *state_label* function.

To demonstrate the implementation of these, the relevant parts of the autogenerated successor generator for the Petri net illustrated in Fig.1 is shown in

Listing 1.1 and 1.2. A successor is defined as a resulting state after firing an enabled transition. The *next_state* function takes a state vector, in the form of a 32 bit integer array, and outputs a number of successors. Each successor is reported back through a *do_callback* function. As *next_state* is initiated it instantiate an array called *cpy* to 1. *cpy* keeps track of changes to the current state, updating the value to 'changed' (0) each time a place marking is updated in a successive state. After the *do_callback* call, the array is reset to 1, ready to track changes to a new successor.

The code in Listing 1.1 shows, how the two transitions from Fig. 1 are implemented with separate if statements, deciding fireability, followed by a number of tokens taken and given to the input and output places. A state can have more than one successor, in that case, the call to *next_state* returns all of them, and LTSmin handles the rest.

Listing 1.1. next_state function for the Petri net example illustrated in Fig. 1

```
1  int next_state(void* model, int group, int *src, TransitionCB callback, void *arg) {
2      int cpy[4]; int i; int successors = 0; int placemarkings[4];
3      for(i=0; i < sizeof(cpy) / sizeof(int); i++ ){ cpy[i] = 1; }
4      if (src[2] >= 1) {
5          successors++;
6          memcpy(placemarkings, src, sizeof(int) * 4);
7          placemarkings[2] = placemarkings[2] - 1; cpy[2] = 0;
8          placemarkings[0] = placemarkings[0] + 1; cpy[0] = 0;
9          placemarkings[1] = placemarkings[1] + 1; cpy[1] = 0;
10         do_callback(cpy, placemarkings, callback, arg);
11         for(i=0; i < sizeof(cpy) / sizeof(int); i++ ){ cpy[i] = 1; }
12     }
13     if (src[0] >= 1) {
14         if (src[1] >= 2) {
15             if (src[3] < 2) {
16             successors++;
17             memcpy(placemarkings, src, sizeof(int) * 4);
18             placemarkings[0] = placemarkings[0] - 1; cpy[0] = 0;
19             placemarkings[1] = placemarkings[1] - 2; cpy[1] = 0;
20             placemarkings[2] = placemarkings[2] + 2; cpy[2] = 0;
21             placemarkings[3] = placemarkings[3] + 1; cpy[3] = 0;
22             do_callback(cpy, placemarkings, callback, arg);
23             for(i=0; i < sizeof(cpy) / sizeof(int); i++ ){ cpy[i] = 1; }
24     }}}
25     return successors;
26 }
```

The *state_label* function is used to process the query or queries, and takes the state vector as its only input. Before code generation, queries are parsed from their intermediate string representation, into a C expression. An example could be a *state_label* function for a fireability query asking if transition T2 in Fig. 1 (a) can be fired, which would simply do the same comparison on the relevant fields on the state vector. A more interesting use of the *state_label* function becomes apparent when evaluating ComputeBound and Bounds queries as shown in Listing 1.2. These queries all require that a set of local variables are declared in the autogenerated file, and are harder to evaluate properly using multi-core algorithms. To make up for the inconsistencies introduced when running multiple threads on different parts of the state space, an additional method, *cleanup*, is declared which makes it possible for each thread to declare its result separately.

Listing 1.2. state_label and cleanup for two ReachabilityComputeBound queries

```
1  int state_label(void* model, int label, int* src) {
2      if (MAX[0] < ( 0 + src[1])) { MAX[0] = (0 + src[1]);}
3      if (MAX[1] < ( 0 + src[2])) { MAX[1] = (0 + src[2]);}
4      return label == LABEL_GOAL && 0;
5  }
6
7  void cleanup(){
8      fprintf(stderr, "Query 0 max tokens are '%d'.\n",MAX[0]);
9      fprintf(stderr, "Query 1 max tokens are '%d'.\n",MAX[1]);
10 }
```

3.2 Implementation Choices

As part of the development several implementation choices were made. To be able to argue about our choices different implementations have been tested during the development. All of these tests were conducted on a laptop with 4 cores (intel Core i7-4710HQ CPU @ 2.50GHz 8) and 16 GB memory.

Intermediate representation. The first consideration was which data structure to use for the intermediate representation of the Petri net. While an incidence matrix was already implemented, we considered the possibility that an adjacency list would be a more memory efficient and not noticeably slower structure. To test this, we implemented the adjacency list and ran speed tests on parsing of the Petri net and on the code generation. We also determined how much memory the models needed in each representation. All of the results are presented in Table 1.

Table 1. Comparison of the linked list and incidence matrix structure. We use a single core, and create a successor generator which performs a state space search. The test is done without reductions or linear over-approximation.

Model	Linked list			Incidence matrix		
	Parsing (sec)	Code generation(sec)	Size(KB)	Parsing (sec)	Code generation(sec)	Size(KB)
ARMCacheCoherence	154.94	328.58	222616	43.47	179.6	232636
Diffusion2D	7.08	0.42	81380	3.16	1.18	453220
CircularTrains	0.09	0.25	7460	0.06	1.29	16548
Dekker	170.68	348.53	308468	37.05	189.91	601552
HouseConstruction	0	0.01	1480	0	0.02	1840
ParamProductionCell	0.01	0.04	2972	0.01	0.07	3284
ResAllocation	0.02	0.09	6144	0.05	0.16	6820

The tests showed, that the adjacency list is indeed a bit more memory efficient, but also remarkably slower than expected. We also see that code generation is considerable faster on larger models with the original incidence matrix. Based on these results, we chose to keep the incidence matrix as the data structure for the intermediate representation.

Reductions. The structural reductions have been implemented as they can greatly reduce the state space. This is indeed desirable in a EXPSPACE-hard

problem [7]. The magnitude with which the reductions affect the state space vary largely between models. As shown in the Appendix in Table 5 the Kanban-problem is reduced to almost 5% of the original state space size, whereas the TokenRing-problem is unaffected at any level. As the reductions are applied with respect to each place in the query, we concatenate the queries, when doing multi-query processing. This is done to ensure that the reductions are done with respect to all places in all queries. Introducing queries gives a bound as to how much a model can be reduced, and during multi-query processing, a larger percentage of the reducible places may become irreducible. To counter the effect of this as much as possible, the structural reductions are implemented to run after the linear over-approximations.

Successor generator. An efficient implementation of the *next_state* procedure can provide great speedup in the verification time and two options were considered. The procedural design focuses on running through the transitions in a loop, checking for fireability and providing each successor within the loop. The second design has an if-statement for each transition hardcoded into the next_state function. All preconditions in the statement are checked individually, allowing for short-circuiting when a single precondition is not met. We tested both designs and the results can be seen in the Appendix in Table 6. The fastest of the two is the second design, and as the state space grows, the effect of the implementation increases noticeable. When the autogenerated.c file is created, compilation is the next task, and the options for compiler optimization in gcc and their effect are illustrated in the Appendix in Table 7. These results argue for the use of optimization.

When choosing how to run LTSmin, we consider the flags available. First of all we dynamically set the amount of cores available to LTSmin. This is given as an input to our tool upon execution. DFS is chosen as the default, but there is still the risk that a strategy can use all the memory too quickly. When this happens LTSmin aborts without providing an answer. A call is made back to the main function triggering a restart of LTSmin this time using the BFS strategy, without recompiling. This implementation is weighted against not providing an answer at all and is, despite the extra overhead, considered a viable design. The source code is available at [25].

4 Experiments

To measure the performance of our tool, we compare it to VerifyPN on state space exploration and reachability analysis. Furthermore we test the scalability in regards to the number of cores, when doing state space exploration and reachability analysis. We benchmark on a cluster consisting of 9 machines each utilizing 64 cores (AMD Opteron 6376) and 1 Tb of memory. The cluster uses *slurm* (Simple Linux Utility for Resource Management) to allocate resources and schedule jobs.

Prior to doing the experiments, we have conducted single core state space exploration with and without reductions on all models from the MCC compe-

tition and recorded the time it takes. For every experiment, models are chosen based on their state-space exploration time. We define the size of the net, as the running time of this single-core computation. We pick a set of times and systematically choose a number of models closest to each of these times. This is done because of our three hypothesises.

Hypothesis 1. We expect that LTSmin is faster than VerifyPN on models with more complex state spaces, due to time taken for code generation and pre-compilation. **Hypothesis 2.** We expect to see a linear speedup in verification time compared to the number of cores used. **Hypothesis 3.** We expect to see that a difference in verification time when performing BFS and DFS on ReachabilityCardinality.

4.1 Single core state space comparison

We compare our tool with VerifyPN on single core state space exploration. This is done on two models that take 10, 50, 100, 500, 700, 2000, 2500 and 3600 seconds, resulting in 16 models total. We have chosen these intervals to be able to precisely determine which size of models can be faster verified with our tool.

The results in Table 8 show that VerifyPN is overall faster when running with a single core and the difference range between VerifyPN being twice as fast and several hundred times faster. There are two instances in the table, where our engine is the faster. These are larger models requiring several minutes to explore, likely because the compilation overhead is less influential in comparison to the entire runtime on these models. We believe, that the slow results on state space exploration is caused by not implementing the optional PINS functions. As we do not use any of the optimizations available, results can be greatly improved on. An example of this could be implementing a *WRITEMATRIX*, supported by our chosen language front-end, to reduce the number of hash computations [14].

4.2 Multi core state space

We test the scalability of our tool in regards to the following number of cores: 1, 8, 16, 24, 32, 40, 48, 56, 64. This is done on three models that takes 60, 600, 1800, 2400 and 3600 seconds, resulting in 15 models total. The specifics of these models such as number of places, transition, state space size etc. can be found in the Appendix in Table 9. These models have been chosen so we can see whether models in the same time interval experiences equal speedup, and to see how much the speedup differs between the time intervals. This experiment ran with state space exploration, and the results are illustrated in Table 2, which shows the speedup in verification time, compared to the single core result.

In all the models, except HouseConstruction-PT-005, we see a speed-up i verification time, making the verification time up to 8 times as fast when the number or cores increase from 1 to 8. When utilizing 64 cores we can achieve a speed-up of 43. As shown in the table, time consuming models benefit the most from additional cores, especially when we are above 24 cores. In general,

Table 2. Speedup of verification time measured in times faster. The verification is run with a state space exploration on multiple cores. Neither reductions nor linear over-approximation are enabled.

Size	Model	Single-core verification time (sec)	Number of cores								State space size
			8	16	24	32	40	48	56	64	
60	HouseConstruction-PT-005	39.05	4	6	7	3	3	3	3	2	1188000
60	ParamProductionCell-PT-1	40.75	8	12	14	16	14	12	10	9	25632
60	SimpleLoadBal-PT-05	63.94	7	12	14	16	16	14	13	12	116176
600	ParamProductionCell-PT-2	509.98	8	14	19	24	28	30	32	33	349874
600	TokenRing-PT-010	668.42	8	14	18	23	26	30	31	33	58905
600	MAPK-PT-008	671.42	7	12	15	17	16	14	14	12	6110600
1800	ResAllocation-PT-R020C002	1878.90	7	13	17	21	24	26	28	28	11534000
1800	GlobalResAllocation-PT-03	1981.21	7	11	14	15	18	19	20	20	6320
1800	ParamProductionCell-PT-3	2178.02	8	14	19	25	30	34	38	42	1465206
2400	LamportFastMutEx-PT-4	2400.31	8	15	20	25	30	34	37	40	1914800
2400	Railroad-PT-010	2417.53	8	14	20	25	30	34	37	40	2038000
2400	ParamProductionCell-PT-5	2463.13	8	14	20	25	30	34	38	41	1657242
3600	ParamProductionCell-PT-0	3526.67	8	15	19	25	30	34	38	41	2776936
3600	QuasiCertifProtocol-PT-06	3567.98	8	14	19	25	30	34	38	42	2272000
3600	ParamProductionCell-PT-4	3594.06	8	14	20	25	30	35	39	43	2409739

the smaller models which took about 1 to 10 minutes had limited speedup after this point. Occasionally the models which took about 1 minute, even extended their verification time when more cores where added. We attribute this to the overhead of handling a large amount of threads.

It is clear that the effect of running the verification with multiple cores is not based on state space size alone. We can see that ResAllocation-PT-R020C002 have the largest state space of the chosen models, but that it only gain a 28 speedup. The model is also verified in half the time it takes for the models with the largest size. We suspect that the structure of the state space is just as important as state space size for both verification time and scalability.

4.3 Reachability Verification Analysis

In this test we use the same models as in Section 4.2, each with the 16 ReachabilityCardinality queries proposed for MCC. The models have been compared across search strategy, with 1, 16, 32 and 64 cores. Instead of exploring the entire state space we make a reachability analysis with ReachabilityCardinality queries one query at the time. First we present the results from the single query verifications to evaluate the effect of different search strategies. We then present the runtime results of the multi query processing utility against the results of VerifyPN's heuristic search. We compare these results to the results from VerifyPN as a means to assess our tool as an alternative to the VerifyPN engine. All results can be found at our online repository [25].

Search Strategy. As per the observations made prior in the paper the DFS is providing the fastest run-times in most cases. This conclusion is based on

Table 3. We show the difference between the BFS and DFS search strategies on single query, for early termination queries. The table contains a subset of the test data from a verification of 16 ReachabilityCardinality queries with 16 cores run one by one. The chosen results have a strictly greater difference than 5 seconds. Verification ran without reductions, and with linear over-approximation. We give the result in estimated DFS speedup measured in times.

Query	Verification time (sec)		DFS speedup
	BFS	DFS	
GlobalResAllocation-PT-03	24.46	18.06	1.35
GlobalResAllocation-PT-03	23.98	18.16	1.32
LamportFastMutEx-PT-4	36.25	1.32	27.52
LamportFastMutEx-PT-4	4.83	108.50	0.04
MAPK-PT-008	3.18	12.86	0.25
ParamProductionCell-PT-0	87.83	5.31	16.53
ParamProductionCell-PT-0	10.09	0.81	12.54
ParamProductionCell-PT-3	6.66	1.09	6.09
ParamProductionCell-PT-3	6.08	0.78	7.85
ParamProductionCell-PT-4	28.36	0.76	37.21
ParamProductionCell-PT-4	12.16	0.69	17.67
ParamProductionCell-PT-4	12.15	0.75	16.20
ParamProductionCell-PT-4	7.57	0.63	12.12
ParamProductionCell-PT-5	12.04	0.78	15.46
ParamProductionCell-PT-5	13.10	0.75	17.42
ParamProductionCell-PT-5	1.55	10.33	0.15
QuasiCertifProtocol-PT-06	26.54	7.43	3.57
QuasiCertifProtocol-PT-06	21.64	16.36	1.32
QuasiCertifProtocol-PT-06	18.82	7.63	2.46
QuasiCertifProtocol-PT-06	23.04	15.10	1.53
QuasiCertifProtocol-PT-06	14.23	4.04	3.52
Railroad-PT-010	8.94	0.69	13.05

the overall runtimes for the models presented in Table 9. In Table 3 we see the speedup factor for the DFS strategy. The results presented in the table is only that of queries with an early termination, where the difference between the BFS - and DFS strategy was greater than 5 seconds. As these are only 22 out of 118 early terminated queries, we observe that it is only on a small fraction of queries, the difference is noticeable. It is from this small set we conclude to use the DFS strategy as default for the MCC. To support this decision we have also calculated the overall DFS speedup, based on all early terminated runtimes for BFS and DFS. Giving a total DFS speedup for all queries at 1.57.

Tool Comparison. In Section 4.2 we compare our tool to VerifyPN on a single core architecture and is shows the VerifyPN engine is the faster in Table 8. This changes when our tool can utilize multi-query and multi-core processing, where we are often able to verify the collection of queries faster than the VerifyPN engine. This is presented in Table 4, that illustrates, how long it takes for each model to have all their queries verified. The VerifyPN time is the sum of each query resolved individually, and the results for our tool are based on multi-query processing.

Table 4. Comparison of our tool and VerifyPN on 16 ReachabilityCardinality queries. Our tool is doing multi-query and multi-core processing on 64 cores, VerifyPN runs the queries one by one on a single core. The total verification time is given in seconds. Reductions are enabled for VerifyPN, disabled for our tool and both uses linear over-approximation.

All Queries				
Size	Model	Our tool (64 cores)		VerifyPN (single-core)
		DFS	BFS	Heuristics
60	HouseConstruction-PT-005	20.619	19.657	0.255
60	ParamProductionCell-PT-1	4.294	4.214	5.155
60	SimpleLoadBal-PT-05	5.258	5.096	93.934
600	ParamProductionCell-PT-2	14.946	14.931	60.18
600	TokenRing-PT-010	18.8	18.63	165.449
600	MAPK-PT-008	87.239	54.952	83.869
1800	ResAllocation-PT-R020C002	77.553	64.374	Timeout
1800	GlobalResAllocation-PT-03	82.276	83.152	111.564
1800	ParamProductionCell-PT-3	52.156	51.064	486.233
2400	LamportFastMutEx-PT-4	57.778	56.906	Timeout
2400	Railroad-PT-010	59.358	58.864	417.457
2400	ParamProductionCell-PT-5	58.829	57.547	1365.224
3600	ParamProductionCell-PT-0	85.101	82.946	188.015
3600	QuasiCertifProtocol-PT-06	84.295	82.988	4712.158
3600	ParamProductionCell-PT-4	83.515	82.275	827.704

5 Conclusion

We have constructed a verification tool for bounded Petri nets with inhibitor arcs. The tool pre-compiles a model specific successor generator and utilizes LTSmins multi-core backend to do exhaustive exploration on a reduced state space. We reuse components from VerifyPN such as structural reductions and over-approximation, and extend them to support multiple queries. Based on the property language from MMC, we defined a generalized syntax and semantics for reachability context problems for Petri nets. We sought out to confirm three hypothesis described in section 4, and have performed experiments accordingly.

Hypothesis 1. Experiments demonstrate that LTSmin performed surprisingly slow in state space exploration on a single core compared to VerifyPN, but we believe that optimizing the *next_state* function will resolve this. **Hypothesis 2.** We confirm an increase in performance on several models, when using multi-core processing and we see a speedup effect on 64 cores, that is up to 43 times. Additionally the results show better verification times for our multi-core processing engine than VerifyPN, when state space size reaches a certain limit depending on the architecture and amount of cores. These results lead to the conclusion that with more cores, larger models can be solved within reasonable time. **Hypothesis 3.** Results show that DFS is generally faster than BFS when doing ReachabilityCardinality, having a total speedup of 1.57 times over BFS. The experiments demonstrate that our tool is generally faster than VerifyPN, when performing ReachabilityCardinality analysis on a 64-core machine.

VerifyPN has an advantage on some models and in order to determine the exact cause, further experiments are required. We suspect the heuristic search of VerifyPN may impact this, and after confirming this suspicion, a possible future implementation could be adding support for heuristic searches in LTSmin. In regards to the contest, an observation was that some nets are one-safe [26], and an alternative implementation of the data structure for bounded nets could effect memory usage positively.

References

[1] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. "Model checking: algorithmic verification and debugging". In: *Communications of the ACM* 52.11 (2009), pp. 74–84.

[2] *Petri Nets Tools Database Quick Overview*. URL: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html`.

[3] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. 1981, pp. 52–71.

[4] Amir Pnueli. "The temporal logic of programs". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.

[5] CA Petri. "Kommunikation mit Automaten. Technische Hochschule, Darmstadt". PhD thesis. Ph. D. Thesis, 1962.

[6] Ernst W Mayr. "An algorithm for the general Petri net reachability problem". In: *SIAM Journal on computing* 13.3 (1984), pp. 441–460.

[7] Catherine Dufourd and Alain Finkel. "Polynomial-time many-one reductions for Petri nets". In: *Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 312–326.

[8] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[9] Javier Esparza. "Decidability and complexity of Petri net problems - an introduction". In: *Lectures on Petri Nets I: Basic Models*. Springer, 1998, pp. 374–428.

[10] *The Property Language Manual for Model Checking Contest @ Petri Nets 2015*. URL: `http://mcc.lip6.fr`.

[11] *Model Checking Contest @ Petri Nets 2015 benchmark models*. URL: `http://mcc.lip6.fr/models.php`.

[12] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jirí Srba. "TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 492–497.

[13] Joakim Byg, Kenneth Yrke Jørgensen, and Jirí Srba. "TAPAAL: Editor, Simulator and Verifier of Timed-Arc Petri Nets". In: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. 2009, pp. 84–89.

[14] Gijs Kant, Alfons Laarman, Jeroen Mijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance, Language-Independent Model Checking". In: (2015).

[15] Andreas Engelbredt Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen, and Jaco van de Pol. "Multi-core Reachability for Timed Automata". In: *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*. 2012, pp. 91–106.

[16] Jonas Finnemann Jensen, Thomas Søndersø Nielsen, Jirí Srba, and Lars Kærlund Østergaard. URL: `https://code.launchpad.net/~verifypn-maintainers/verifypn/u1.1`.

[17] Gérard Berthelot and Gerard Roucairol. "Reduction of Petri-nets". In: (1976), pp. 202–209.

[18] T. Murata. "State equation, controllability, and maximal matchings of petri nets". In: *Automatic Control, IEEE Transactions on* 22.3 (June 1977), pp. 412–416.

[19] *Model Checking Contest @ Petri Nets 2014*. URL: `http://mcc.lip6.fr/2014`.

[20] Karsten Wolf. "Generating Petri Net State Spaces". English. In: Lecture Notes in Computer Science 4546 (2007). Ed. by Jetty Kleijn and Alex Yakovlev, pp. 29–42.

[21] Monika Heiner, Christian Rohr, and Martin Schwarick. "MARCIE - Model Checking and Reachability Analysis Done Efficiently". In: *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*. 2013, pp. 389–399.

[22] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. "GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets". In: *Perform. Eval.* 24.1-2 (Nov. 1995), pp. 47–68.

[23] Alfons Laarman, Jaco van de Pol, and Michael Weber. "Multi-core LTSmin: Marrying modularity and scalability". In: *NASA Formal Methods*. Springer, 2011, pp. 506–511.

[24] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580.

[25] Jakob Dyrh, Mads Johannsen, Isabella Kaufmann, and Søren Moss Nielsen. URL: `https://code.launchpad.net/~tapaal-dist/verifypn/verifypnLTSmin`.

[26] Allan Cheng, Javier Esparza, and Jens Palsberg. "Complexity results for 1-safe nets". In: *Foundations of software technology and theoretical computer science*. Springer, 1993, pp. 326–337.

6 Appendix

Table 5. Test on how much a model can be reduced. Each model have been reduced as much as possible and the size of the state space is shown both before and after. Additionally the percentage of the remaining state space size is given

Model	State Space Size		Percent of remaining state space
	Original	Reduced	
Angiogenesis-PT-05	6527668	6426463	98
HouseConstruction-PT-005	1187984	6	0.0005
CircularTrains-PT-012	195	195	100
DatabaseWithMutex-PT-02	153	41	27
CSRepetitions-PT-02	7424	525	7
MAPK-PT-080	25759365	12500253	49
ERK-PT-000001	13	13	100
Eratosthenes-PT-010	32	32	100
CircadianClock-PT-000001	128	128	100
Vasy2003-PT-none	60210	60210	100
PermAdmissibility-PT-01	52537	52537	100
SharedMemory-PT-000005	1863	243	13
MAPK-PT-040	25644962	12451934	49
MAPK-PT-020	25743054	12609230	49
Peterson-PT-2	20754	1254	6
ERK-PT-000100	64358827	31973415	50
FMS-PT-005	2895018	812	0.03
PhilosophersDyn-PT-03	325	325	100
RwMutex-PT-r0010w0010	1034	1	0.1
Railroad-PT-005	1838	487	26
ERK-PT-000010	47047	47047	100
TokenRing-PT-005	166	166	100
MAPK-PT-008	6110643	6110643	100
Angiogenesis-PT-10	10223735	4885558	48
SimpleLoadBal-PT-02	832	832	100
LamportFastMutEx-PT-2	380	380	100
ResAllocation-PT-R002C002	8	4	50
TokenRing-PT-010	58905	58905	100
Kanban-PT-0020	28166085	1460151	5
HouseConstruction-PT-002	1501	3	0.2
Kanban-PT-0005	2546432	3276	0.1

In this table we compare different design for the successor generators *next_state* function. In the *Procedural* design we use a loop to run through all transitions. The loop consists of a few calls to helper functions which determine fireability, adds and removes tokens. In the *if* design, we simply write every possible transition directly, in the form of a long if-chain. To test these designs, we made a small Petri net which would be easy to create a handmade successor generator for and would have an states space which grew exponentially in relation to the initial amount of tokens. This amounted in a Petri net which modeled three identical separate concurrent processes that ran independently. This model is initially named TestModel, and is futher named after the amount of initial tokens. Example: TestModel-PT-5, if their are 5 tokens in the initial marking.

Table 6. Comparison of the procedural design and the if-statement design in relation to the successor generator structure. It show the overall verification in seconds. The test is based on a handmade model and run with using 1 core with state space exploration.

Model	Verification time (sec)		State space size
	Procedural	If	
TestModel-PT-5	0.2	0.15	9261
TestModel-PT-10	7.49	4.96	287496
TestModel-PT-15	70.01	48.93	2515456
TestModel-PT-20	359.33	286.35	12326391

Table 7. Compile time, verification time and state space size on a 4-core run for state space exploration with different compiler optimizations for gcc. No reductions are used in the verification

Model	State Space Size
TokenRing-PT-005	166
Railroad-PT-005	1838
CircularTrains-PT-024	86515
Kanban-PT-0005	2546432
SimpleLoadBal-PT-10	116176
MAPK-PT-008	6110600
FMS-PT-010	25010000000

Model	-O 1		
	Compile Time (sec)	Verification Time (sec)	Total
TokenRing-PT-005	0.52	0.01	0.53
Railroad-PT-005	0.21	0.04	0.25
CircularTrains-PT-024	0.14	0.60	0.75
Kanban-PT-0005	0.07	11.24	11.31
SimpleLoadBal-PT-10	3.41	185.61	189.02
MAPK-PT-008	0.11	66.34	66.45
FMS-PT-010	0.13	33.82	33.95

Model	-O 2		
	Compile Time (sec)	Verification Time (sec)	Total
TokenRing-PT-005	0.65	0.01	0.66
Railroad-PT-005	0.31	0.033	0.34
CircularTrains-PT-024	0.13	0.6	0.73
Kanban-PT-0005	0.12	11.06	11.18
SimpleLoadBal-PT-10	5.85	182.75	188.59
MAPK-PT-008	0.17	56.20	56.38
FMS-PT-010	0.11	32.52	32.631

Model	-O 3		
	Compile Time (sec)	Verification Time (sec)	Total
TokenRing-PT-005	0.76	0.01	0.77
Railroad-PT-005	0.39	0.026	0.42
CircularTrains-PT-024	0.15	0.49	0.65
Kanban-PT-0005	0.11	10.3	10.41
SimpleLoadBal-PT-10	6.12	139.83	145.96
MAPK-PT-008	0.24	69.63	69.87
FMS-PT-010	0.13	26.63	26.761

Table 8. Comparison of our tool and VerifyPN on state space exploration. The test is single-core, and neither reductions nor linear over-approximation are enabled on any of the tools

Size	Model	Verification time (sec)	
		Our tool	VerifyPN
10	LamportFastMutEx-PT-3	13.46	0.4
10	Dekker-PT-010	12.52	0.2
50	HouseConstruction-PT-005	40.43	5.1
50	ParamProductionCell-PT-1	43.82	1.5
100	Kanban-PT-0005	76.33	21.09
100	RwMutex-PT-r0010w0500	98.3	2.35
500	RwMutex-PT-r0010w1000	438.72	3.7
500	ParamProductionCell-PT-2	513.89	56.44
700	MAPK-PT-008	595.16	50.45
700	RwMutex-PT-r0020w0010	775.15	1212.1
2000	ResAllocation-PT-R020C002	1844.96	Timeout
2000	GlobalResAllocation-PT-03	1973.86	2.93
2500	ParamProductionCell-PT-5	2450.87	796.01
2500	Dekker-PT-015	2604.22	534.09
3600	QuasiCertifProtocol-PT-06	3561.68	558.07
3600	ParamProductionCell-PT-4	3605.39	1730.02

Table 9. Overveiw of the models chosen for experiment 2 and 3

Size	Model	Files Size(kB)	Places	Transitions	Arcs	State space size
60	HouseConstruction-PT-005	13.1	26	18	51	1188000
60	ParamProductionCell-PT-1	170.3	231	202	846	25632
60	SimpleLoadBal-PT-05	155.8	59	180	1158	116176
600	ParamProductionCell-PT-2	170.3	231	202	846	349874
600	TokenRing-PT-010	820.3	1	2	4	58905
600	MAPK-PT-008	25.6	22	30	90	6110600
1800	ResAllocation-PT-R020C002	82.9	80	42	200	11534000
1800	GlobalResAllocation-PT-03	6000	33	4791	38652	6320
1800	ParamProductionCell-PT-3	170.3	231	202	846	1465206
2400	LamportFastMutEx-PT-4	111.9	135	230	990	1914800
2400	Railroad-PT-010	79.8	118	156	898	2038000
2400	ParamProductionCell-PT-5	170.3	231	202	846	1657242
3600	ParamProductionCell-PT-0	146.6	198	176	730	2776936
3600	QuasiCertifProtocol-PT-06	156.3	270	116	659	2272000
3600	ParamProductionCell-PT-4	170.3	231	202	846	2409739

Table 10. Comparison on the BFS, DFS and heuristic search strategy on early termination queries. The table contains a subset of the data from a verification of 16 ReachabilityCardinality queries on 15 models. Our tool ran single query on 16 cores with linear over-approximation and without reductions, while VerifyPN ran single query on a single core with both linear over-approximation and reductions. We show the average verification time pr. query in seconds.

Early Termination Queries			
Queries	Our tool (64 cores)		VerifyPN(Single core)
	BFS	DFS	Heuristics
GlobalResAllocation-PT-03	28.95	20.30	6.74
HouseConstruction-PT-005	2.15	1.48	0.01
LamportFastMutEx-PT-4	2.65	4.48	0.49
MAPK-PT-008	5.52	8.20	0.13
ParamProductionCell-PT-0	8.13	3.10	2.86
ParamProductionCell-PT-1	2.95	2.92	0.09
ParamProductionCell-PT-2	2.98	3.27	0.16
ParamProductionCell-PT-3	3.75	2.89	0.09
ParamProductionCell-PT-4	3.74	2.71	0.22
ParamProductionCell-PT-5	3.45	3.60	4.64
QuasiCertifProtocol-PT-06	7.07	4.99	11.54
Railroad-PT-010	3.42	2.42	0.06
ResAllocation-PT-R020C002	2.28	19.95	2400.01
SimpleLoadBal-PT-05	1.57	2.65	0.04

Table 11. Comparison on the BFS, DFS and heuristic search strategy on queries without early termination. The table contains a subset of the data from a verification of 16 ReachabilityCardinality queries on 15 models. Our tool ran single query on 16 cores with linear over-approximation and without reductions, while VerifyPN ran single query on a single core with both linear over-approximation and reductions. We show the average verification time pr. query in seconds.

No Early Termination Queries			
Queries	Our tool (64 cores)		VerifyPN(single core)
	BFS	DFS	Heuristics
GlobalResAllocation-PT-03	85.23	76.65	7.03
HouseConstruction-PT-005	19.24	19.24	19.24
LamportFastMutEx-PT-4	56.98	57.89	3174.64
MAPK-PT-008	56.85	56.85	56.85
ParamProductionCell-PT-0	82.99	84.17	26.57
ParamProductionCell-PT-1	4.22	4.26	0.56
ParamProductionCell-PT-2	14.84	14.86	4.98
ParamProductionCell-PT-3	51.95	52.27	60.69
ParamProductionCell-PT-4	82.51	84.72	206.26
ParamProductionCell-PT-5	58.04	58.79	189.07
QuasiCertifProtocol-PT-06	83.78	83.72	917.04
Railroad-PT-010	58.32	59.71	34.77
ResAllocation-PT-R020C002	64.88	76.43	830.78
SimpleLoadBal-PT-05	4.99	5.10	9.37
TokenRing-PT-010	18.60	18.97	10.34