Compositional Reachability in Petri Nets

Julian Rathke, Paweł Sobociński, and Owen Stephens

ECS, University of Southampton, UK

Abstract. We introduce a divide-and-conquer algorithm for a modified version of the reachability/coverability problem in 1-bounded Petri nets that relies on the compositional algebra of nets with boundaries: we consider the algebraic decomposition of the net of interest as part of the input. We formally prove the correctness of the technique and contrast the performance of our implementation with state-of-the-art tools that exploit partial order reduction techniques on the global net.

Introduction

For finite-state Petri nets, the reachability problem—i.e. whether some target marking is reachable from the initial marking— is PSPACE-complete [4]. While compositional approaches to model checking were identified by the founders of the discipline [5] as a way of combating state-explosion, the large majority of model checkers work with the *global* statespace – which, in the case of Petri nets, means computing the *state graph*: a transition system where the states are markings and transitions reflect the firing of net-transitions. Of course, state graphs of large nets are prohibitively large to build naively; much of the research effort to date has focussed on taming the state explosion problem by exploiting symmetries and partial order reduction techniques [9, 12, 16, 18, 25].

Most real-life concurrent systems, however, are regular in their structure: they are naturally specified as a composition of relatively simple, often repeated, components. We contend that by allowing model checkers access to this high level information, we can exploit divide and conquer techniques to improve performance. The tool `Penrose`, described in this paper, exploits the high-level structure of a net, which is provided as input, to perform reachability checking. `Penrose` is written in Haskell and has not been optimised; despite this, it outperforms mature, state-of-the-art tools in a number of well-known examples.

Let us consider how a divide and conquer approach can help in checking reachability; consider a net, N, composed of two subnets, N_1 and N_2, with disjoint places. Any reachability question on N, stated as a desired marking, can be restated as a pair of desired markings on N_1 and N_2. Checking this pair of reachability questions independently is more efficient than directly checking reachability in N, since state graph size is exponential in the number of places of the corresponding net. However, such a naive approach is unsound: N_2's behaviour is constrained by its interactions with N_1, and vice versa. What is required is a representation of the behaviour of the subnet N_2, say, in which its dependency

and effect upon its environment (the rest of the system) is accounted for. The notion of a *Petri Net with Boundaries* provides such a representation.

Roughly speaking, a Petri Net with Boundaries [2] (PNB) represents a subnet that is to be placed within a larger environment. The key feature of this model is a representation of how a net's transitions may connect with its environment, via "boundary ports". The state graph of a PNB is an automaton in which transition labels record interactions on these boundary ports. Using PNBs, reachability checking of a composed net, N, formed of N_1 and N_2, can be achieved by independently checking the pair of reachability problems on N_1 and N_2 using their labelled state graphs.

Once the state graph of a component PNB has been built, what remains important, in terms of checking reachability of the larger system is only its boundary interactions. This means that state graphs may be minimised with respect to behaviour that does not interact on a boundary. Moreover, these minimised graphs may be further minimised after composition. Our technique exploits this fact to keep the size of state graphs as small as possible. This may appear counter-intuitive as a means of obtaining efficiency, as minimisation is known to be expensive. Judicious use of memoisation comes to the fore here: we target our technique at a class of regular systems that feature many repeated component nets. As such, we expect many repeated reachability checks on the component nets and, crucially, many repeated compositions of such components.

Structure of the paper. In §1 we present the necessary background on Petri Nets with Boundaries, followed in §2 with definitions of the automata encoding reachability in PNBs. The details of our algorithm are given in §3 and in §4 we describe its implementation and detail our experimental results. Finally, we present a proof of correctness of the technique §5 and conclude in §6.

1 Background

In this paper, all Petri nets are assumed to be 1-bounded (a.k.a. *elementary net systems*): there is at most one token at each place. They are closely related to *1-safe nets*, indeed, any 1-safe net is 1-bounded. However, 1-bounded nets are not necessarily 1-safe: the semantics of 1-bounded nets simply prohibits the firing of a transition that would violate the restriction during any execution.

The compositional algebra of Petri nets with boundaries (PNB) is the theoretical workhorse that enables our approach. Here we only give a cursory overview: for formal details, the reader is referred to [2, 21].

A PNB is a Petri net with extra structure: two finite ordinals of *boundary ports*, to which net transitions can connect. Intuitively, transitions connected to a boundary port are not yet completely specified. The two sets of ports are drawn, from top to bottom, on the left and right hand sides of an enclosing box. An example is on the left in Fig. 1: here both boundaries consist of one port. We write $P : (1, 1)$ to mean that P is a PNB with both boundaries of size 1. Differently to [2, 21] we consider "vanilla" PNBs to be unmarked; in §2 we introduce a marked variant that contains both an initial and a target marking.

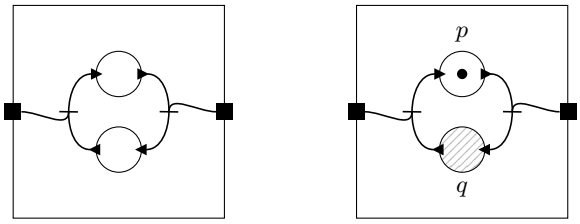

Fig. 1: An example PNB and marked PNB $(1, 1)$

There are two operations for composing PNBs: synchronisation on a common boundary (;) and a non-commutative, parallel composition that we call tensor ($\otimes$). The most interesting operation is synchronisation: we refer to [2] for the formalities, but the graphical intuition shown in Fig. 2 suffices for most examples. Note that the size of the right boundary of P agrees with the size of the left boundary of Q—nets can be composed iff they agree on the size of their intermediate boundary. Given $X : (k, l)$ and $Y : (l, m)$, the composition is written $X \,;\, Y : (k, m)$. Transitions of the composed net—called *minimal synchronisations*—are, in general, sets of transitions of the two components. In Fig. 2, the transition $\{t, a\}$ results from synchronising t and a. Transition t can synchronise both with a and b; indeed, both choices are taken into account (b also synchronises with u). Transition c has no complementary transition to synchronise with and thus no composite transition results. Finally, v does not connect to any places, only to the fourth boundary port, and is thus synchronised with d.

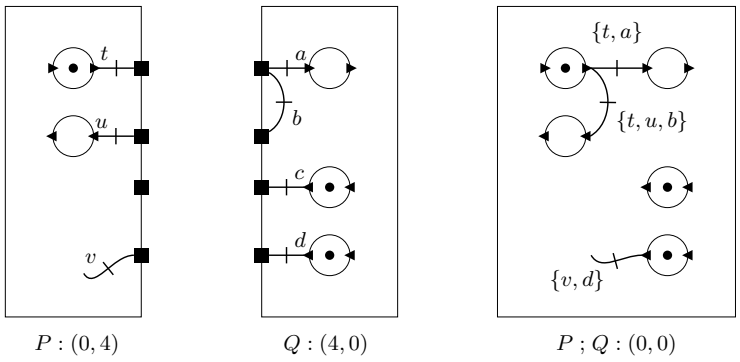

Fig. 2: Example synchronisation

The second PNB composition operation, tensor, is graphically represented by "stacking" one net on another; intuitively, it is a non-communicating parallel composition. Differently from synchronisation, any two nets can be tensored: given nets $X : (k, l)$ and $Y : (m, n)$, we have $X \otimes Y : (k + m, l + n)$. Both ';' and '$\otimes$' are associative up-to-isomorphism of PNBs, but neither is commutative.

2 From Marked Nets to Automata

A *marked* PNB consists of a PNB together with two subsets of net-places: the *initial* and *target marking*. Graphically, the places belonging to an initial marking are decorated with a token, whilst those belonging to the target marking are shaded. A place can be in both, only one, or neither the initial and a target marking. An example of a marked net is illustrated in the right part of Fig. 1.

Ordinary PNBs have a labelled transition system (LTS) semantics that captures the *step semantics* of the underlying net. The labels record the interactions on the boundaries, as we explain below with the aid of an example. Consider the LTS in the left part of Fig. 3 that corresponds to the left net in Fig. 1.

Let $\mathbb{B} = \{0, 1\}$ and consider a PNB $P : (k, l)$. The states of its LTS correspond to *markings* of P, the transitions to the firings of sets of independent, enabled transitions. Transition labels come from the set $\mathbb{B}^k \times \mathbb{B}^l$. Throughout the paper we write α/β for $(\alpha, \beta) \in \mathbb{B}^k \times \mathbb{B}^l$. A transition labelled with α/β indicates the firing of a set of transitions that is connected to ports on the left as indicated by the 1s in α, and on the right by the 1s in β. For example, in the LTS of Fig. 3, the rightmost transition firing in the PNB of Fig. 1 is represented by the transition labelled 0/1 in the LTS.

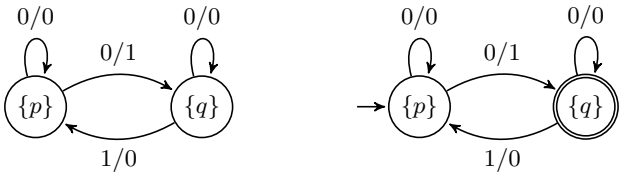

Fig. 3: LTS/NFA semantics of the PNB and marked PNB of Fig. 1

Just as a PNB gives rise to an LTS, a marked PNB gives rise to a nondeterministic finite automaton (NFA). The states and transitions are as described above; the initial state is the state representing the initial marking, while the final state is the state representing the target marking.

The NFAs that arise from PNBs can be composed using operations corresponding to PNB compositions; a specialised nomenclature is therefore useful:

Definition 1. *A non-deterministic finite automaton with boundaries (NFAB) $A : (k, l)$ is a non-deterministic finite automaton A with alphabet $\mathbb{B}^k \times \mathbb{B}^l$. Let $\mathcal{L}(A) \subseteq (\mathbb{B}^k \times \mathbb{B}^l)^*$ denote the language of A.*

Given a PNB $P : (k, l)$, we denote the resulting NFAB $[\![P]\!] : (k, l)$. Note that any ordinary marked net, N, can be regarded as a PNB, $N : (0, 0)$ with no boundaries. The resulting NFAB $[\![N]\!]$ has the alphabet $\mathbb{B}^0 \times \mathbb{B}^0$, which is the singleton—this is precisely the state graph of N w.r.t. step semantics. The following observation, which is central to our approach, is immediate.

Proposition 2. *Supposed that N is a marked net. Then the final marking is reachable from the initial marking iff $\mathcal{L}([\![N]\!]) \neq \varnothing$* □

Finally, we need to explain how NFABs are composed. If $A : (k, l)$, $B : (l, m)$ and $C : (n, o)$ are NFABs then both $A \,;\, B : (k, m)$ and $A \otimes C : (k+n, l+o)$ have as states pairs (a, b) where a is a state of A and b is a state of B. Initial and final states are simply the product of the initial and final states of the component NFABs. The only difference is how the transition relations are defined:

$$\frac{a \xrightarrow{\alpha/\gamma} a' \quad b \xrightarrow{\gamma/\beta} b'}{(a,b) \xrightarrow{\alpha/\beta} (a',b')}\,(;) \qquad \frac{a \xrightarrow{\alpha/\beta} a' \quad b \xrightarrow{\gamma/\delta} b'}{(a,b) \xrightarrow{\alpha\gamma/\beta\delta} (a',b')}\,(\otimes)$$

The following is straightforward to show and builds on known compositionality properties of PNBs [2].

Proposition 3 (Compositionality). *Given PNBs P, Q, we have $[\![P \,;\, Q]\!] \cong [\![P]\!] \,;\, [\![Q]\!]$ and $[\![P \otimes Q]\!] \cong [\![P]\!] \otimes [\![Q]\!]$, where $\cong$ is isomorphism of automata, defined in the obvious way as bijective mappings on states and transitions.*

Of particular interest to our approach are NFA transitions witnessing the firing of net transitions that are not connected to any boundary ports. Thus we let $\tau_{k,l} \stackrel{\text{def}}{=} 0^k/0^l$. We will refer to $\tau_{k,l}$ as a τ-move or *silent move*.

3 Compositional Reachability

In this section we explain our technique for compositional reachability checking and present our algorithm. We will use the following running example, which features a net that is particularly suitable for our approach.

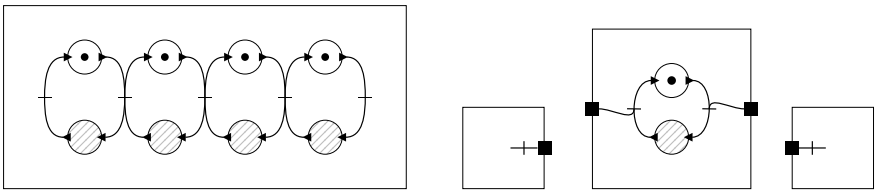

Fig. 4: The net B_4 as a composition of nets $\top$, b_1 and $\bot$.

Example 4. Consider the marked PNB, B_4, shown in Fig. 4 that models a 4 place buffer [9]. It enjoys a simple, regular structure, yet the number of transitions that need to be fired in order to reach the target marking is quadratic in the size of the buffer[1]. Using marked PNBs, we can express B_4 as:

$$\top \,;\, (b_1 \,;\, (b_1 \,;\, (b_1 \,;\, (b_1 \,;\, \bot)))) \tag{1}$$

[1] Precisely, the length of the minimal firing sequence of B_i is the i^{th} triangle number.

Our procedure takes a *decomposition* of a net as an input: roughly an expression akin to (1), expressing a net as a composition of simple components. We represent decompositions using *wiring expressions*; a wiring expression is the abstract syntax tree, t, of a PNB expression, where internal nodes are labelled with either ; or $\otimes$, and leaves are (possibly repeated) variables.

$$T \ ::= \ x \mid T \ ; T \mid T \otimes T$$

Now, a wiring expression together with an assignment map, $\mathcal{V}$, taking variables to marked PNBs, can be evaluated recursively to obtain a marked PNB, $[\![t]\!]_{\mathcal{V}}$:

$$[\![x]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} \mathcal{V}(x) \qquad [\![t_1 \ ; t_2]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} [\![t_1]\!]_{\mathcal{V}} \ ; [\![t_2]\!]_{\mathcal{V}} \qquad [\![t_1 \otimes t_2]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} [\![t_1]\!]_{\mathcal{V}} \otimes [\![t_2]\!]_{\mathcal{V}}$$

We assume that variable assignments are compatible with t, in the sense that only nets with compatible boundaries are composed; in recent work, we used a simple type system to ensure this [24]. Given a net $N : (k, l)$, we say that $(t, \mathcal{V})$ is a *wiring decomposition* of N if $[\![t]\!]_{\mathcal{V}} \cong N$.

Example 5. The following are the wiring expression and variable assignment that correspond to (1):

$$t = x_1 \ ; (x_2 \ ; (x_2 \ ; (x_2 \ ; (x_2 \ ; x_3)))) \qquad \mathcal{V} = \{x_1 \mapsto \top, x_2 \mapsto b_1, x_3 \mapsto \bot\} \quad (2)$$

observe that $[\![t]\!]_{\mathcal{V}}$ is B_4, shown in the left side of Fig. 4; $(t, \mathcal{V})$ is therefore a wiring decomposition of B_4.

Consider any net N, together with corresponding initial and target markings. Given any ordinary PNB expression for N, notice that we can extend it into a wiring decomposition: first by translating the expression into a wiring expression, second by translating the initial and target markings of N component-wise into marked PNBs, that we bind to the variables. The *specification* of the reachability problem is thus naturally compositional.

The core idea of our algorithm is to convert a wiring decomposition $(t, \mathcal{V})$ of a net N to an—ideally small—NFAB that represents the "protocol" that N must adhere to w.r.t. its context (i.e. the nets connected to its boundaries), in order to reach its local target marking. The key property exploited by the algorithm is that *weak language equivalence is a congruence*[2] (Proposition 8): any weak language-preserving modifications can be made to a PNB's NFAB whilst ensuring a faithful representation of all interactions the PNB must perform to reach its target marking. Showing that weak language equivalence is a congruence is thus the key technical ingredient needed to show the correctness of our technique; this is the topic of §5.

Concretely, given an NFAB we perform *(i)* τ-closure, ignoring internal moves, and *(ii)* NFA minimisation to prune the statespace, preserving language equivalence. We leverage the structure of wiring decompositions by using memoisation to prevent repeated computation. Our algorithm maintains two maps:

[2] The adjective 'weak' refers to the forgetting of the τ-moves. See §5 for the formal definition.

1. `knownNetNFAs`, from component nets to their corresponding reduced NFABs,
2. `knownNFAComps`, from two NFABs and composition type, to reduced NFABs.

The second memoisation map is checked for membership up-to language-equivalence: $(n_1, n_2, Op) \in$ `knownNFAComps` is true if `knownNFAComps` contains an entry (n_1', n_2', Op) such that $n_1 \sim n_1'$ and $n_2 \sim n_2'$, where $\sim$ is language equivalence. The essence of this optimisation is that if we perform a (potentially expensive) composition and reduction on a pair of NFABs, we never repeat this computation for *any* pair of NFABs that are pairwise language equivalent.

The core algorithm is given in Fig. 5, and we briefly outline its steps here. The input wiring decomposition is traversed; each unique leaf (component net) is converted to its NFAB, which is then τ-closed and reduced, with memoisation (Line 3) preventing repeated performance of this conversion for equal components. On composition nodes, the procedure recurses on both child branches, to obtain a (reduced) NFAB for each; then, if the pair of NFABs is unique (up-to language-equivalence) they are composed using the appropriate operation on NFABs, before being τ-closed and reduced. Again, memoisation (Line 13) prevents unnecessary repeated computation.

Require: `knownNetNFAs`, `knownNFAComps` initially empty
1: **procedure** WDTONFA(t)
2: **if** t is a PNB **then**
3: **if** $t \in$ `knownNetNFAs` **then**
4: **return** `knownNetNFAs`$[t]$
5: **else**
6: $n \leftarrow$ REDUCE(τ-CLOSE(NETTONFA(t)))
7: `knownNetNFAs`$[t] := n$
8: **return** n
9: **end if**
10: **else** ▷ t is $(t_1, t_2,$ `OP`$)$
11: $n_1 \leftarrow$ WDTONFA(t_1)
12: $n_2 \leftarrow$ WDTONFA(t_2)
13: **if** $(n_1, n_2,$ `OP`$) \in$ `knownNFAComps` **then** ▷ Up-to language equivalence
14: **return** `knownNFAComps`$[(n_1, n_2,$ `OP`$)]$
15: **else**
16: $n \leftarrow$ REDUCE(τ-CLOSE(n_1 `OP` n_2))
17: `knownNFAComps`$[(n_1, n_2,$ `OP`$)] := n$
18: **return** n
19: **end if**
20: **end if**
21: **end procedure**

Fig. 5: Core Algorithm

Now since any ordinary net N can be considered as a PNB with no boundaries, running our algorithm on any wiring decomposition of N as input will

construct an NFAB with the singleton alphabet $\{\tau_{0,0}\}$. Up to weak language equivalence there are only two such NFABs, both with one state that is either accepting (indicating a *reachable* desired marking) or non-accepting (indicating an *unreachable* desired marking). Therefore, running our core algorithm on a wiring decomposition of N decides the classical reachability problem for N.

It should be highlighted that our algorithm decides a modified version of the reachability problem: we take a wiring decomposition as input. When run on the trivial decomposition, the performance would typically be unsatisfactory since, for example, no partial order reduction techniques are employed when generating the state graph. The scalability of our algorithm thus depends on finding efficient decompositions. In our experience, finding suitable candidate decompositions is not difficult: concurrent and distributed systems are typically designed, and described, in a component-wise, rather than monolithic manner.

4 Implementation and Results

The core algorithm described in the previous section has been implemented in Haskell, as part of the `Penrose` tool. In the implementation we use current state-of-the-art algorithms for both: language-equivalence checking via bisimulation up-to techniques due to Bonchi and Pous [1], and NFA minimisation using forward and backwards variants of simulation of Clemente and Mayr [15].

Example 6. The running time of our implementation on the buffer nets of our running example is *linear* w.r.t. the size of the input net. Indeed, each additional component simply leads to another (successful) memoisation-map lookup, with constant cost. Contrast this with the fact that the minimum firing sequence is quadratic w.r.t. the size of the buffer, as described in Exm. 4. Checking reachability for buffer nets thus asymptotically outperforms approaches based on firing transitions in the global net: see the first five rows in the results Table 1.

As we have explained in §3, `Penrose` takes as input a wiring decomposition; Since problem descriptions in the literature are naturally described in terms of their constituent components, it is little work to arrive at high-level descriptions using the DSL recently introduced by the 2nd and 3rd authors [24]. The DSL programs evaluate to a wiring expression, that we have used as inputs to `Penrose`; alternatively, they can be evaluated to monolithic nets, which we have used as input to other tools for performance comparisons. Indeed, using `Penrose` it is possible to generate arbitrary instances of commonly used benchmarks.

The relative performance of `Penrose` was evaluated [3] by comparing it with the current state-of-the-art tools, all of which use unfolding-based approaches:

[3] All experiments were run on an Ubuntu Linux virtual machine (4-core 32-bit CPU, 8GB RAM) hosted on an Intel i7-2600 3.40GHz CPU, 16GB of RAM, running 64-bit Ubuntu Linux[4]. Tool performance was recorded using the standard Unix `time` command, measuring total (wall-clock) time and peak memory usage.

[4] Some tools required a 32-bit platform, hence the virtual machine.

1. LOLA [20], an established tool and winner of the reachability category of 2013 Petri Net model checking competition,
2. The PUNF[5] unfolder, which uses parallelisation techniques from [10] and CLP[6] checker, which uses linear-programming techniques from [14],
3. CUNF [19] unfolder, a recently-introduced tool[7], using the CNA checker.

In Table 1, T indicates a time-out (5 minutes), M denotes memory-exhaustion (8GB) and / marks an incorrect result; the best performance of each problem instance is highlighted.

As mentioned, `Penrose` directly computes using the particular wiring decomposition, $(t, \mathcal{V})$, that specifies each problem; all other tools were provided input that was generated by first computing $[\![t]\!]_{\mathcal{V}}$, and then converting into suitable format[8]. The time taken for this conversion was not included in the performance benchmarking—only the processing time of the individual tools was recorded.

The majority of problems in Table 1 are taken from Corbett's [6] benchmarks, except those marked with a $*$, which we briefly discuss here: *counter* is taken from [24], and models a distributed n-bit counter. It is an unsafe net, leading to incorrect results from CLP/CNA. *replicator* is taken from [22], modelling a sequence of components that can output an unbounded number of tokens at their right boundary after receiving a single token on their left boundary. Again, it is an unsafe net. Taken from [12], *iter-choice* models a sequence of transition choices. A single component is illusrated in Fig. 6, the transition choice is between t_1 and t_2. Due to an exponentially-sized unfolding, the results show that moderately-sized instances[9] of iter-choice cannot be handled by the tested tools. `Penrose`, on the other hand, is able to handle very large instances quickly. Merged Processes [12] were designed to avoid such exponential unfoldings.

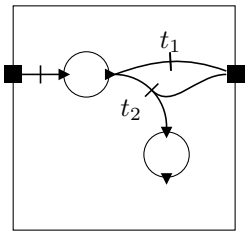

Fig. 6: iter-choice net

The time-vs-problem size results for `Penrose` are plotted in Fig. 7. There is a clear distinction between the scalable examples, and those that are (much) less scalable. The causes of the poor performance include increase in (language inequivalent) NFA sizes, as the wiring decomposition is traversed: each new NFA grows the memoisation map, and larger NFAs take longer to check for language (in)equivalence. The examples with good performance (e.g. Exm. 4) quickly reach fixed-points w.r.t. composition, in that no new NFAs are ever generated after a certain point; indeed, the buffer reaches a fixed point at $n = 1$.

Statespace growth is unavoidable in systems such as the counter; it is an inherent problem with the compositional approach. In this case, avoiding the

[5] `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/`
[6] `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/clp/`
[7] `http://code.google.com/p/cunf/`
[8] Either LL_NET format or LOLA's input format.
[9] Checking for an alternating taken/not-taken marking

Table 1: Time and Memory results

Problem		Time (s)				Max Resident (MB)			
name	size	LOLA	CLP	CNA	Penrose	LOLA	CLP	CNA	Penrose
buffer	8	**0.001**	0.003	0.017	**0.001**	**7.51**	33.30	38.45	14.36
buffer	32	**0.001**	0.013	0.824	**0.001**	**7.51**	34.49	48.09	14.35
buffer	512	0.058	*T*	*M*	**0.001**	83.44	*T*	*M*	**14.40**
buffer	4096	*T*	*T*	*M*	**0.002**	*T*	*T*	*M*	**14.70**
buffer	32768	*T*	*T*	*M*	**0.005**	*T*	*T*	*M*	**16.07**
over	8	31.039	0.008	1.071	**0.003**	3812.00	37.63	141.85	**16.53**
over	32	*M*	*T*	*M*	**0.003**	*M*	*T*	*M*	**16.52**
over	512	*M*	*T*	*M*	**0.003**	*M*	*T*	*M*	**16.52**
over	4096	*M*	*T*	*M*	**0.003**	*M*	*T*	*M*	**16.53**
over	32768	*M*	*T*	*M*	**0.004**	*M*	*T*	*M*	**17.85**
dac	8	**0.001**	0.003	0.017	**0.001**	**7.51**	33.28	38.85	15.37
dac	32	**0.001**	0.005	0.028	**0.001**	**7.50**	34.50	49.45	15.34
dac	512	0.005	*T*	255.847	**0.001**	20.62	*T*	6012.00	**15.36**
dac	4096	2.462	*T*	*M*	**0.002**	166.07	*T*	*M*	**15.62**
dac	32768	*T*	*T*	*M*	**0.009**	*T*	*T*	*M*	**16.91**
philo	8	**0.002**	0.003	0.016	0.004	**8.86**	33.22	38.54	16.98
philo	32	*M*	**0.003**	0.017	0.004	*M*	33.53	40.87	**16.97**
philo	512	*M*	0.020	0.086	**0.004**	*M*	41.69	290.77	**16.90**
philo	4096	*M*	7.853	*M*	**0.004**	*M*	172.76	*M*	**17.13**
philo	32768	*M*	*T*	*M*	**0.005**	*M*	*T*	*M*	**18.32**
hartstone	8	**0.000**	0.002	/	0.002	**7.51**	33.05	/	15.48
hartstone	32	**0.001**	0.002	/	0.002	**7.52**	33.22	/	15.48
hartstone	512	0.002	0.005	/	**0.001**	17.82	36.38	/	**15.49**
hartstone	4096	0.044	0.029	/	**0.002**	96.27	58.15	/	**15.74**
hartstone	32768	56.050	3.008	*M*	**0.003**	727.63	278.23	*M*	**17.10**
iter-choice∗	8	0.006	5.025	19.062	**0.002**	36.37	465.17	1570.64	**15.34**
iter-choice∗	32	*M*	*T*	*T*	**0.002**	*M*	*T*	*T*	**15.34**
iter-choice∗	512	*M*	*T*	*T*	**0.002**	*M*	*T*	*T*	**15.38**
iter-choice∗	4096	*M*	*T*	*T*	**0.002**	*M*	*T*	*T*	**15.56**
iter-choice∗	32768	*M*	*T*	*T*	**0.003**	*M*	*T*	*T*	**16.95**
replicator∗	8	**0.001**	/	0.016	**0.001**	**7.51**	/	38.15	15.39
replicator∗	32	**0.001**	/	0.017	**0.001**	**7.51**	/	39.41	15.40
replicator∗	512	0.002	/	1.023	**0.001**	**14.72**	/	77.87	15.41
replicator∗	4096	0.062	/	64.046	**0.002**	86.85	/	3256.00	**15.56**
replicator∗	32768	91.646	/	*M*	**0.006**	1524.50	/	*M*	**16.97**
counter∗	8	**0.001**	/	/	0.050	**7.51**	/	/	20.61
counter∗	16	**0.000**	/	/	4.056	**7.51**	/	/	22.52
counter∗	32	**0.001**	/	/	46.027	**7.51**	/	/	39.66
counter∗	64	**0.001**	/	/	*T*	**8.60**	/	/	*T*
token-ring	8	**0.001**	0.007	0.071	1.024	**7.51**	39.96	89.81	20.74
token-ring	16	**1.824**	*T*	*T*	12.034	318.08	*T*	*T*	**24.77**
token-ring	32	*M*	*T*	*T*	**133.636**	*M*	*T*	*T*	**39.65**
token-ring	64	*M*	*T*	*T*	*T*	*M*	*T*	*T*	*T*

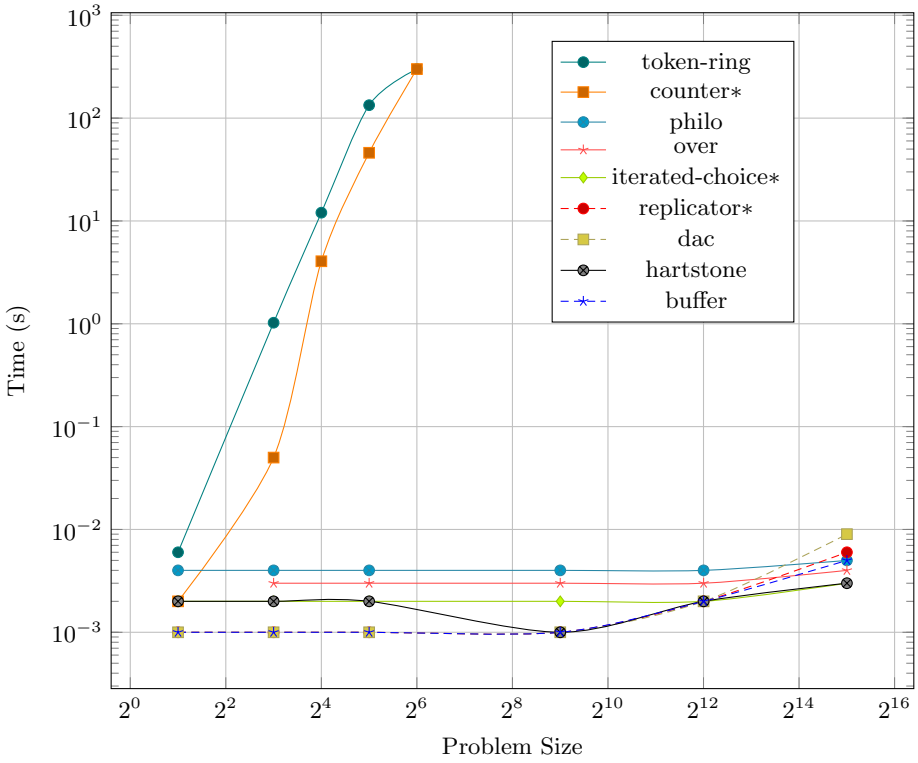

Fig. 7: Time vs Problem size for `Penrose`

slowdown could be achieved by observing a monotone increase in state size and abandoning the component-wise approach, falling back to other techniques.

It could be argued that the playing field is unfair: `Penrose` uses a formal description of the decomposition of a problem at hand into smaller components while other tools take a global, monolithic net as input. This is, however, precisely our point: there is no reason for model checkers not to take advantage of compositional descriptions–it is how *real* systems are designed and described.

5 Proof of Correctness

In this section we outline a proof that the algorithm presented in Fig. 5 is correct. Given NFABs $A, B : (k, l)$, A and B are said to be *(strong) language equivalent*, written $A \sim B$, if $\mathcal{L}(A) = \mathcal{L}(B)$. The following result is simple to show, using the definitions of ; and $\otimes$ on NFABs.

Proposition 7 (Strong language equivalence is a congruence). *Suppose that A and A' are NFABs and $A \sim A'$. Then the following hold, where in*

each point below, B ranges over those NFAB where the composition is defined.

$$(i)\ A\ ;\ B \sim A'\ ;\ B \qquad (ii)\ B\ ;\ A \sim B\ ;\ A'$$
$$(iii)\ A \otimes B \sim A' \otimes B \qquad (iv)\ B \otimes A \sim B \otimes A' \quad \square$$

Now let $\widehat{\ } : (\mathbb{B}^k \times \mathbb{B}^l)^* \to (\mathbb{B}^k \times \mathbb{B}^l - \{\tau_{k,l}\})^*$ be the unique monoid homomorphism where, on elements x of $\mathbb{B}^k \times \mathbb{B}^l$, $\widehat{x} = \epsilon$ if $x = \tau_{k,l}$ and x otherwise. Intuitively, $\widehat{\mathbf{x}}$ results from stripping the silent moves from $\mathbf{x}$. Given a NFAB $A : (k, l)$ we define $\mathcal{L}^\tau(A) \stackrel{\text{def}}{=} \{\widehat{\mathbf{x}} \mid \mathbf{x} \in \mathcal{L}(A)\}$. NFABs $A, B : (k, l)$ are said to be *weak language equivalent*, written $A \approx B$, when $\mathcal{L}^\tau(A) = \mathcal{L}^\tau(B)$.

An NFAB $A : (k, l)$ is said to be *reflexive* if for all states $a \in A$ we have a loop transition $a \xrightarrow{\tau_{k,l}} a$. When we restrict our attention to reflexive NFABs, also weak language equivalence is a congruence.

Proposition 8. *Suppose that A and A' are reflexive NFABs and $A \approx A'$. Then the following hold, where at each point below B ranges those over reflexive NFABs where the composition is defined.*

$$(i)\ A\ ;\ B \approx A'\ ;\ B \qquad (ii)\ B\ ;\ A \approx B\ ;\ A'$$
$$(iii)\ A \otimes B \approx A' \otimes B \qquad (iv)\ B \otimes A \approx B \otimes A' \quad \square$$

Any NFAB that results from a marked PNB is reflexive, since the empty set of transitions can fire at any marking, yielding a τ-move in the underlying NFAB. The reductions performed in Fig. 5 replace reflexive NFABs with smaller, weak language equivalent automata. Correctness is a straightforward consequence.

Theorem 9. *The algorithm in Fig. 5 is correct: the computed NFAB is weak language equivalent to the semantics of the corresponding global net.* $\square$

6 Related Work and Discussion

We introduced a technique for checking reachability that takes a decomposition of a net as input and relies on the use of weak language equivalence to discard local state. The compositional approach was briefly discussed in [22] and in the technical report [23], where further examples are described in detail. Initial efforts were based on determinisation, which was considerably more expensive than our current use of NFA minimisation [15] and language equivalence checking [1].

The algebra of automata with boundaries used in this paper is an instance of the algebra of Span(Graph) [11]. The goal of the more recent work [2, 3, 21] was to lift this algebra to the level of nets in a compositional way and explore connections with process algebra: our approach ignores local state and focusses only on external interactions: here we were inspired by the ideas of Milner [17].

The tools that we have used in order to compare our performance are based on the unfolding approach pioneered by McMillan [16]. The algorithm to compute finite complete prefix was improved in [9,13]. Unfoldings carry more information about the computations of nets than merely reachability, for instance, allowing LTL model checking [7]. For an overview of the extensive field see [8].

References

1. F. Bonchi and D. Pous. Checking NFA Equivalence with Bisimulations up to Congruence. In *PoPL '13*.
2. R. Bruni, H. Melgratti, U. Montanari, and P. Sobociński. Connector algebras for C/E and P/T nets' Interactions. *Logical Methods in Computer Science*, 9(3), 2013.
3. R. Bruni, H. C. Melgratti, and U. Montanari. A connector algebra for P/T nets interactions. In *CONCUR*, 2011.
4. A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. In *FSTTCS*, volume 761 of *LNCS*, pages 326–337, 1993.
5. E. M. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LiCS'89*, pages 352–362, 1989.
6. J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.
7. J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In *SPIN*, volume 2057 of *LNCS*, pages 37–56, 2001.
8. J. Esparza and K. Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer, 2008.
9. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Form Method Syst Des*, 30(3):285–210, 2002.
10. K. Heljanko, V. Khomenko, and M. Koutny. Parallelisation of the Petri net Unfolding Algorithm. In *TACAS '02*, pages 371–385, 2002.
11. P. Katis, N. Sabadini, and R. F. C. Walters. Span(Graph): an algebra of transition systems. In *AMAST '97*.
12. V. Khomenko, A. Kondratyev, M. Koutny, and W. Vogler. Merged Processes — A New Condensed Representation of Petri Net Behaviour. In *CONCUR '05*, 2005.
13. V. Khomenko, M. Koutny, and W. Vogler. Canonical prefixes of Petri net unfoldings. *Acta Inform.*, 40(2):95–118, 2003.
14. M. Koutny and V. Khomenko. Linear Programming Deadlock Checking Using Partial Order Dependencies. Technical report, Newcastle University, 2000.
15. R. Mayr and L. Clemente. Advanced Automata Minimization. In *POPL '13*.
16. K. McMillan. A technique of a state space search based on unfolding. *Form Method Syst Des*, 6(1):45–65, 1995.
17. R. Milner. *A Calculus of Communicating Systems*. Prentice Hall, 1989.
18. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science*, 13(1):85–108, Jan. 1981.
19. C. Rodríguez and S. Schwoon. Cunf: A Tool for Unfolding and Verifying Petri Nets with Read Arcs. In *ATVA '13*, pages 492–495, 2013.
20. K. Schmidt. LoLA: A Low Level Analyser. In *ICATPN '00*, June 2000.
21. P. Sobociński. Representations of Petri net interactions. In *CONCUR '10*, 2010.
22. P. Sobociński and O. Stephens. Penrose: Putting Compositionality to Work for Petri Net Reachability. In *CALCO Tools '13*, pages 346–352. Springer, 2013.
23. P. Sobociński and O. Stephens. Reachability via compositionality in Petri nets. arXiv:1303.1399v1, 2013.
24. P. Sobociński and O. Stephens. A Programming Language for Spatial Distribution of Net Systems. In *ICATPN '14*, pages 150–169. Springer, 2014.
25. P. Starke. Reachability analysis of Petri nets using symmetries. *Systems Analysis Modelling Simulation*, 4/5:292–303, 1991.