Fast Coalgebraic Bisimilarity Minimization

JULES JACOBS, Radboud University, The Netherlands
THORSTEN WISSMANN, Radboud University, The Netherlands

Coalgebraic bisimilarity minimization generalizes classical automaton minimization to a large class of automata whose transition structure is specified by a functor, subsuming strong, weighted, and probabilistic bisimilarity. This offers the enticing possibility of turning bisimilarity minimization into an off-the-shelf technology, without having to develop a new algorithm for each new type of automaton. Unfortunately, there is no existing algorithm that is fully general, efficient, and able to handle large systems.

We present a generic algorithm that minimizes coalgebras over an arbitrary functor in the category of sets as long as the action on morphisms is sufficiently computable. The functor makes at most $O(m \log n)$ calls to the functor-specific action, where n is the number of states and m is the number of transitions in the coalgebra.

While more specialized algorithms can be asymptotically faster than our algorithm (usually by a factor of $O(\frac{m}{n})$), our algorithm is especially well suited to efficient implementation, and our tool *Boa* often uses much less time and memory on existing benchmarks, and can handle larger automata, despite being more generic.

CCS Concepts: • **Theory of computation**;

Additional Key Words and Phrases: Coalgebra, Partition Refinement, Monotone Neighbourhoods

ACM Reference Format:
Jules Jacobs and Thorsten Wißmann. 2023. Fast Coalgebraic Bisimilarity Minimization. *Proc. ACM Program. Lang.* 7, POPL, Article 52 (January 2023), 28 pages. https://doi.org/10.1145/3571245

1 INTRODUCTION

State-based systems arise in various shapes throughout computer science: as automata for regular expressions, as control-flow graphs of programs, Markov decision processes, (labelled) transition systems, or as the small-step semantics of programming languages. If the programming language of interest involves concurrency, bisimulation can capture whether two systems exhibit the same behaviour [Milner 1980; Winskel 1993]. In model checking, a state-based system is derived from the implementation and then checked against its specification.

It is often beneficial to reduce the size of a state-based system by merging all equivalent states. Moore's algorithm [Moore 1956] and Hopcroft's $O(n \log n)$ algorithm [Hopcroft 1971] do this for the deterministic finite automata that arise from regular expressions, and produce the equivalent automaton with minimal number of states. In model checking, state-space reduction can be effective as a preprocessing step [Baier and Katoen 2008]. For instance, in probabilistic model checking, the time saved in model checking due to the smaller system exceeds the time needed to minimize the system [Katoen et al. 2007].

Subsequent to Hopcroft [1971], a variety of algorithms were developed for minimizing different types of automata. Examples are algorithms for

Authors' addresses: Jules Jacobs, Radboud University, Nijmegen, The Netherlands; Thorsten Wißmann, Radboud University, Nijmegen, The Netherlands.

2475-1421/2023/1-ART52
https://doi.org/10.1145/3571245

- transition systems (without action labels) [Kanellakis and Smolka 1983, 1990], labelled transition systems [Valmari 2009], which arise from the verification concurrent systems,
- weighted bisimilarity [Valmari and Franceschinis 2010] for Markov chains and probabilistic settings (such as probabilistic model checking [Katoen et al. 2007]),
- Markov decision processes [Baier et al. 2000; Groote et al. 2018] that combine concurrency with probabilistic branching,
- weighted tree automata [Björklund et al. 2007, 2009] that arise in natural language processing [May and Knight 2006].

Recently, those algorithms and system equivalences were subsumed by a coalgebraic generalization [Deifel et al. 2019; Dorsch et al. 2017; Wißmann et al. 2021]. This generic algorithm is parametrized by a (Set-)functor that describes the concrete system type of interest. Functors are a standard notion in category theory and a key notion in the Haskell programming language. In coalgebraic automaton minimization, the functor is used to attach transition data to each state of the automaton. For instance, the powerset functor models non-deterministic branching in transition systems, and the probability distribution functor models probabilistic branching in Markov chains.

The users of a coalgebraic minimization algorithm may create their own system type by composing the provided basic functors, allowing them to freely combine deterministic, non-deterministic, and probabilistic behaviour. For instance, the functor to model Markov decision processes is the composition of the functors of transition systems and the functor for probability distributions. This generalization points to the enticing possibility of turning automata minimization for different types of automata into an off-the-shelf technology.

Unfortunately, there are two problems that currently block this vision. Firstly, although the generic algorithm has excellent $O(m \log n)$ asymptotic complexity, where n is the number of states and m is the number of edges, it is slow in practice, and the data structures required for partition refinement suffer from hungry memory usage. A machine with 16GB of RAM required several minutes to minimize tree automata with 150 thousand states and ran out of memory when minimizing tree automata larger than 160 thousand states [Deifel et al. 2019; Wißmann et al. 2021]. This problem has also been observed for algorithms for specific automata types, e.g., transition systems [Valmari 2010]. In order to increase the total memory available, a distributed partition refinement algorithm has been developed [Birkmann et al. 2022], (and previously also for specific automata types, e.g., labelled transition systems [Blom and Orzan 2005]), but this algorithm runs in $O(n^2)$ and requires expensive distributed hardware.

Secondly, the generic algorithm does not work for all Set-functors, because it places certain restrictions on the functor type necessary for the tight run time complexity. For instance, the algorithm is not capable of minimizing frames for the monotone neighbourhood logic [Hansen and Kupke 2004a,b], arising in game theory [Parikh 1985; Pauly 2001; Peleg 1987].

We present a new algorithm that works for *all* system types given by computable Set-functors, requiring only an implementation of the functor's action on morphisms, which is then used to compute so-called *signatures of states*, a notion originally introduced for labelled transition systems [Blom and Orzan 2005]. The algorithm makes at most $O(m \log n)$ calls to the functor implementation, where n and m are the number of states and edges in the automaton, respectively. In almost all instances, one such call takes $O(k)$ time, where k is the maximum out-degree of a state, so the overall run time is in $O(km \log n)$. We compensate for this extra factor because our algorithm has been designed to be efficient in practice and does not need large data structures: we only need the automaton with predecessors and a refinable partition data structure.

We provide an implementation of our algorithm in our tool called *Boa*. The user of the tool can either encode their system type as a composition of the functors natively supported by *Boa*, or

extend *Boa* with a new functor by providing a small amount of Rust code that implements the functor's action on morphisms.

Empirical evaluation of our implementation shows that the memory usage is much reduced, in certain cases by more than 100x compared to the distributed algorithm [Birkmann et al. 2022], such that the benchmarks that were used to illustrate its scalability can now be solved on a single computer. Running time is also much reduced, in certain cases by more than 3000x, even though we run on a single core rather than a distributed cluster. We believe that this is a major step towards coalgebraic partition refinement as an off-the-shelf technology for automaton minimization.

The rest of the paper is structured as follows.

Section 2: Coalgebraic bisimilarity minimization and our algorithm in a nutshell.
Section 3: The formal statement of behavioural equivalence of states, and examples for how this reduces to known notions of equivalence for particular instantiations.
Section 4: Detailed description of our coalgebraic minimization algorithm for any computable set functor, and time complexity analysis showing that the algorithm makes at most $O(m \log n)$ calls to the functor operation.
Section 5: Instantiations of the algorithm showing its genericity.
Section 6: Benchmark results showing our algorithm outperforms earlier work.
Section 7: Conclusion and future work.

2 FAST COALGEBRAIC BISIMILARITY MINIMIZATION IN A NUTSHELL

This section presents the key ideas of our fast coalgebraic minimization algorithm. We start with an introduction to coalgebra, and how the language of category theory provides an elegant unifying framework for different types of automata. No knowledge of category theory is assumed; we will go from the concrete to the abstract, and category theoretic notions have been erased from the presentation as much as possible.

Let us thus start by looking at three examples of automata: deterministic finite automata on the alphabet $\{a, b\}$, transition systems, and Markov chains. The usual way of visualizing is depicted in the first row of Figure 1. For instance, a deterministic finite automaton on state set C is usually described via a transition function $\delta\colon C \times \{a, b\} \to C$ and a set of accepting states $F \subseteq C$ (the initial state is not relevant for the task of computing equivalent states). In order to generalize various types of automata, however, we take a *state-centric* point of view, where we consider all the data as being *attached to a particular state*:

- In a finite automaton on the alphabet $\{a, b\}$ each state has two successors: one for the input letter a and one for the input letter b. Each state also carries a boolean that determines whether the state is accepting (double border), or not (single border). For instance, state **3** in the deterministic automaton in the left column of Figure 1 is not accepting, but after transitioning via a it goes to state **5**, which is accepting. We can specify any deterministic automaton entirely via a map

$$c\colon C \to \{\mathsf{F}, \mathsf{T}\} \times C \times C$$

  This map sends every state $q \in C$ to $(x, q_a, q_b) := c(q)$, where $x \in \{\mathsf{F}, \mathsf{T}\}$ specifies if q is accepting, and $q_a, q_b \in C$ are the target states for in input a and b, respectively.
- A transition system consists of a (finite) set of locations C, plus a (finite) set of transitions "$\to$" $\subseteq C \times C$. For instance, state **3** in the figure can transition to state **4** or **5** or to itself, whereas **5** cannot transition anywhere. A transition system is specified by a map

$$c\colon C \to \mathcal{P}_{\mathrm{f}}(C)$$

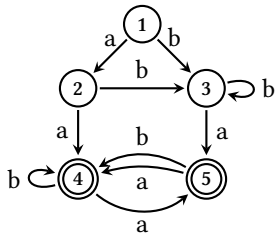
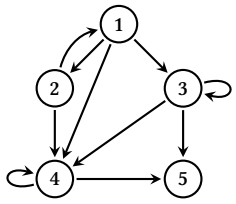
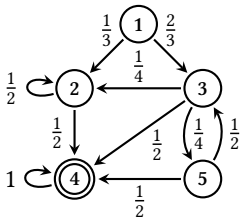
	DFA	Transition system	Markov chain
			
Functor	$F(X) = \{\mathsf{F},\mathsf{T}\} \times X \times X$	$F(X) = \mathcal{P}_\mathrm{f}(X)$	$F(X) = \{\mathsf{F},\mathsf{T}\} \times \mathcal{D}(X)$
Coalgebra $c\colon C \to F(C)$	$\mathbf{1} \mapsto (\mathsf{F},\mathbf{2},\mathbf{3})$ $\mathbf{2} \mapsto (\mathsf{F},\mathbf{4},\mathbf{3})$ $\mathbf{3} \mapsto (\mathsf{F},\mathbf{5},\mathbf{3})$ $\mathbf{4} \mapsto (\mathsf{T},\mathbf{5},\mathbf{4})$ $\mathbf{5} \mapsto (\mathsf{T},\mathbf{4},\mathbf{4})$	$\mathbf{1} \mapsto \{\mathbf{2},\mathbf{3},\mathbf{4}\}$ $\mathbf{2} \mapsto \{\mathbf{1},\mathbf{4}\}$ $\mathbf{3} \mapsto \{\mathbf{3},\mathbf{4},\mathbf{5}\}$ $\mathbf{4} \mapsto \{\mathbf{4},\mathbf{5}\}$ $\mathbf{5} \mapsto \{\,\}$	$\mathbf{1} \mapsto (\mathsf{F},\{\mathbf{2}:\frac{1}{3},\mathbf{3}:\frac{2}{3}\})$ $\mathbf{2} \mapsto (\mathsf{F},\{\mathbf{2}:\frac{1}{2},\mathbf{4}:\frac{1}{2}\})$ $\mathbf{3} \mapsto (\mathsf{F},\{\mathbf{2}:\frac{1}{4},\mathbf{4}:\frac{1}{2},\mathbf{5}:\frac{1}{4}\})$ $\mathbf{4} \mapsto (\mathsf{T},\{\mathbf{4}:1\})$ $\mathbf{5} \mapsto (\mathsf{F},\{\mathbf{3}:\frac{1}{2},\mathbf{4}:\frac{1}{2}\})$
Equivalence	$\mathbf{2} \equiv \mathbf{3}, \mathbf{4} \equiv \mathbf{5}$	$\mathbf{1} \equiv \mathbf{2}, \mathbf{3} \equiv \mathbf{4}$	$\mathbf{2} \equiv \mathbf{3} \equiv \mathbf{5}$
Minimized $c'\colon C' \to F(C')$	$\mathbf{1} \mapsto (\mathsf{F},\mathbf{2},\mathbf{2})$ $\mathbf{2} \mapsto (\mathsf{F},\mathbf{4},\mathbf{2})$ $\mathbf{4} \mapsto (\mathsf{T},\mathbf{4},\mathbf{4})$	$\mathbf{1} \mapsto \{\mathbf{1},\mathbf{3}\}$ $\mathbf{3} \mapsto \{\mathbf{3},\mathbf{5}\}$ $\mathbf{5} \mapsto \{\,\}$	$\mathbf{1} \mapsto (\mathsf{F},\{\mathbf{2}:1\})$ $\mathbf{2} \mapsto (\mathsf{F},\{\mathbf{2}:\frac{1}{2},\mathbf{4}:\frac{1}{2}\})$ $\mathbf{4} \mapsto (\mathsf{T},\{\mathbf{4}:1\})$

Fig. 1. Examples of different system types and their encoding as coalgebras for the state set $C = \{\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5}\}$.

where $\mathcal{P}_\mathrm{f}(C)$ is the set of finite subsets of C. This maps sends every location q to the set of locations $c(q) \subseteq C$ to which a transition exists.

- A Markov chain consists of a set of states, and for each state a probability distribution over all states describes the transition behaviour. That is, for each pair of states $q, q' \in C$, the probability $p_{q,q'} \in [0,1]$ denoting the probability to transition from q to q'. We also attach a boolean label to each state (again, indicated by double border). For instance, state $\mathbf{1}$ in the figure steps to state $\mathbf{2}$ with probability $\frac{1}{3}$ and to state $\mathbf{3}$ with probability $\frac{2}{3}$. Such a Markov chain is specified by a map

$$c\colon C \to \{\mathsf{F},\mathsf{T}\} \times \mathcal{D}(C)$$

where $\mathcal{D}(C)$ is the set of finite probability distributions over C.

We call the data $c(q)$ attached to a state q the **successor structure** of the state q.

By generalizing the pattern above, different types of automata can be treated in a uniform way: In all these examples, we have a set of states C (where $C = \{\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5}\}$ in the figure), and then a map $c\colon C \to F(C)$ for the successor structures, for some construction F turning the set of states C into another set $F(C)$. Such a mapping $F\colon \mathrm{Set} \to \mathrm{Set}$ (in programming terms one should think of F as a type constructor) is called a functor, and describes the automaton type. This point of view allows us to easily consider variations, such as labelled transition systems, given by $F(X) = \mathcal{P}_\mathrm{f}(\{a,b\} \times X)$, and Markov chains where the states are not labelled but the transitions are labelled, given by $F(X) = \mathcal{D}(\{a,b\} \times X)$. Other examples, such as monoid weighted systems, Markov Decision processes, and tree automata, are given in Section 3. Representing an automaton of type F by attaching a successor structure of type $F(C)$ to each state $q \in C$ brings us to the following definition:

Definition 2.1. An automaton of type F, or finite F-coalgebra, is a pair (C, c) of a finite set of states C, and a function $c\colon C \to F(C)$ that attaches the successor structure of type $F(C)$ to each state in C.

Since C is a finite set of states, we can give such a map c by listing what each state in C maps to. For the concrete automata in Figure 1, the representation using such a mapping $c\colon C \to F(C)$ is given in the "Coalgebra" row.

2.1 Behavioural Equivalence of States in F-automata, Generically

We now know how to uniformly *represent* an automaton of type F, but **we need a uniform way to state what it means for states to be equivalent**. Intuitively, we would like to say that two states are equivalent if the successor structures attached to the two states by the map $c\colon C \to F(C)$ are equivalent. The difficulty is that the successor structure may itself contain other states, so equivalence of states requires equivalence of successor structures and vice versa.

A way to cut this knot is to consider a *proposed* equivalence of states, and then define what it means for this equivalence to be valid, namely: an equivalence of states is *valid* if proposed to be equivalent states have equivalent successor structures, where equivalence of the successor structures is considered up to the *proposed* equivalence of states. In short, the proposed equivalence should be compatible with the transition structure specified by the successor structures.

Rather than representing a proposed equivalence as an equivalence relation $R \subseteq C \times C$ on the state space C, it is better to use a surjective map $r\colon C \to C'$ that assigns to each state a canonical representative in C' identifying its equivalence class (also called *block*). That is, two states q, q' are equivalent according to r, if $r(q) = r(q')$. Intuitively, r partitions the states into blocks or equivalence classes $\{q \in C \mid r(q) = y\} \subseteq C$ for each canonical representative $y \in C'$. Not only does this representation of the equivalence avoid quadratic overhead in the implementation, but it is also more suitable to state the stability condition:

An equivalence $r\colon C \to C'$ is *stable*, if for every two equivalent states q_1, q_2 (i.e., with $r(q_1) = r(q_2)$), the successor structures $c(q_1)$ and $c(q_2)$ attached to the states become *equal* after replacing states q inside the successor structures with their canonical representative $r(q)$.

This guarantees that we can build a minimized automaton with the canonical representatives $r(q) \in C'$ as state space. If we do this replacement for both the source and the target of all transitions, we obtain a potentially smaller automaton $c'\colon C' \to F(C')$.

In order to gain intuition about this, let us investigate our three examples in Figure 1:

- In the finite automaton, the states $\mathbf{4} \equiv \mathbf{5}$ and $\mathbf{2} \equiv \mathbf{3}$ can be shown to be equivalent, so we have $C' = \{\mathbf{1}, \mathbf{2}, \mathbf{4}\}$ and $r\colon C \to C'$ with $\mathbf{3} \mapsto \mathbf{2}$ and $\mathbf{5} \mapsto \mathbf{4}$ (and also $\mathbf{1} \mapsto \mathbf{1}$, $\mathbf{2} \mapsto \mathbf{2}$, $\mathbf{4} \mapsto \mathbf{4}$, which we will use implicitly in future examples). We can check that this equivalence is compatible with c by verifying that the successor structures of supposedly equivalent states become *equal* after substituting $\mathbf{5} \mapsto \mathbf{4}$ and $\mathbf{3} \mapsto \mathbf{2}$. After substituting $\mathbf{5} \mapsto \mathbf{4}$ we indeed have that $c(\mathbf{2}) = (\mathsf{F}, \mathbf{4}, \mathbf{3})$ and $c(\mathbf{3}) = (\mathsf{F}, \mathbf{5}, \mathbf{3})$ become equal, and that $c(\mathbf{4}) = (\mathsf{T}, \mathbf{5}, \mathbf{4})$ and $c(\mathbf{5}) = (\mathsf{T}, \mathbf{4}, \mathbf{4})$ become equal. So this equivalence is stable.
- For the transition system, the states $\mathbf{3} \equiv \mathbf{4}$ are equivalent, and $\mathbf{1} \equiv \mathbf{2}$ are equivalent. We can verify, for instance, that states $c(\mathbf{1}) = \{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$ and $f(\mathbf{2}) = \{\mathbf{1}, \mathbf{4}\}$ are equivalent, because after substituting $\mathbf{4} \mapsto \mathbf{3}$ and $\mathbf{2} \mapsto \mathbf{1}$, we indeed have $\{\mathbf{1}, \mathbf{3}, \mathbf{3}\} = \{\mathbf{1}, \mathbf{3}\}$, because duplicates can be removed from sets. Note that it is important that the data for transition systems are sets rather than lists or multisets. Multisets also give a valid type of automaton, but they do not give the same notion of equivalence.
- For the Markov chain, we can verify $\mathbf{2} \equiv \mathbf{3} \equiv \mathbf{5}$. Consider that all three of these states step to state $\mathbf{4}$ with probability $\frac{1}{2}$. With the remaining probability $\frac{1}{2}$ these states step to one of the states

$\mathbf{2} \equiv \mathbf{3} \equiv \mathbf{5}$, i.e. they stay in this block. State $\mathbf{3}$ steps to either state $\mathbf{2}$ or $\mathbf{5}$ with probability of $\frac{1}{4}$ each. If we however assume that state $\mathbf{5}$ behaves equivalent to $\mathbf{2}$, then the branching of state $\mathbf{3}$ is the same as going to state $\mathbf{2}$ with probability $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ directly. Thus, when substituting $\mathbf{5} \mapsto \mathbf{2}$ and $\mathbf{3} \mapsto \mathbf{2}$ the distribution $c(\mathbf{3}) = (\mathsf{F}, \{\mathbf{2}\colon \frac{1}{4}, \mathbf{4}\colon \frac{1}{2}, \mathbf{5}\colon \frac{1}{4}\})$, collapses to $(\mathsf{F}, \{\mathbf{2}\colon \frac{1}{2}, \mathbf{4}\colon \frac{1}{2}\})$. In other words, edges to equivalent states get merged by summing up their probability.

Here we assumed that we were given an equivalence, which we check to be stable. Our next task is to determine how to find the maximal stable equivalence. We shall see that this only requires a minor modification to checking that a given equivalence is stable: if we discover that an equivalence is *not* stable, we can use that information to iteratively refine the equivalence until it is stable.

2.2 Minimizing F-automata, Generically: The Naive Algorithm

In this section we describe a **naive but generic method for minimizing F-automata** [König and Küpper 2014]. The method is based on the observation that we can start by optimistically assuming that *all* states are equivalent, and then use the stability check described in the preceding section to determine how to split up into finer blocks. By iterating this procedure we will arrive at the minimal automaton.

Let us thus see what happens if we blindly assume *all* states to be equivalent, and perform the substitution where we change every state to state $\mathbf{1}$. For the finite automaton in Figure 1, we get

$$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1}) \qquad \mathbf{2} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1}) \qquad \mathbf{3} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1}) \qquad \mathbf{4} \mapsto (\mathsf{T}, \mathbf{1}, \mathbf{1}) \qquad \mathbf{5} \mapsto (\mathsf{T}, \mathbf{1}, \mathbf{1})$$

Clearly, even though we assumed all states to be equivalent, the states $\mathbf{1}, \mathbf{2}, \mathbf{3}$ are still distinct from $\mathbf{4}, \mathbf{5}$ because the former three are not accepting whereas the latter two are. Therefore, even if we initially assumed all states to be equivalent, we discover inequivalent states. Let us thus try the equivalence $\mathbf{1} \equiv \mathbf{2} \equiv \mathbf{3}$ and $\mathbf{4} \equiv \mathbf{5}$, and apply substitution where we send $\mathbf{2} \mapsto \mathbf{1}$, $\mathbf{3} \mapsto \mathbf{1}$ and $\mathbf{5} \mapsto \mathbf{4}$:

$$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1}) \qquad \mathbf{2} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{1}) \qquad \mathbf{3} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{1}) \qquad \mathbf{4} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4}) \qquad \mathbf{5} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$$

We have now discovered *three* distinct blocks of states: state $\mathbf{1}$, states $\mathbf{2} \equiv \mathbf{3}$ and states $\mathbf{4} \equiv \mathbf{5}$. If we apply a substitution for *that* equivalence, we get:

$$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{2}, \mathbf{2}) \qquad \mathbf{2} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{2}) \qquad \mathbf{3} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{2}) \qquad \mathbf{4} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4}) \qquad \mathbf{5} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$$

We did not discover new blocks; we still have three distinct blocks of states: $\mathbf{1}$, states $\mathbf{2} \equiv \mathbf{3}$ and states $\mathbf{4} \equiv \mathbf{5}$. Hence, there is no need to change the substitution map sending each state to a representative in the $\equiv$-class, and so we reached a fixed point. We can now read off the minimized automaton by deleting states $\mathbf{3}$ and $\mathbf{5}$ from the last automaton above.

The reader may observe that the process sketched above is quite general, and can be used to minimize a large class of automata. The sketch translates into the pseudocode in Algorithm 1.

Algorithm 1 Sketch of the naive partition refinement algorithm

```
procedure NaiveAlgorithm(automaton)        ▷ Finds equivalent states of automaton
    Put all states in one block (i.e., assume that all states are equivalent)
    while number of blocks grows do
        Substitute current block numbers in the successor structures
        Split up blocks according to the successor structures
```

The execution trace of this naive algorithm for our three example automata of Figure 1 can be found in Figure 2. What the algorithm only needs is the ability to obtain a canonicalized successor structure after applying a substitution to the successor states. In general this may involve some amount of computation. For instance, for transition systems, a purely textual substitution would

$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1})$	$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1})$	$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{2}, \mathbf{2})$
$\mathbf{2} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1})$	$\mathbf{2} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{1})$	$\mathbf{2} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{2})$
$\mathbf{3} \mapsto (\mathsf{F}, \mathbf{1}, \mathbf{1})$	$\mathbf{3} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{1})$	$\mathbf{3} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{2})$
$\mathbf{4} \mapsto (\mathsf{T}, \mathbf{1}, \mathbf{1})$	$\mathbf{4} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$	$\mathbf{4} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$
$\mathbf{5} \mapsto (\mathsf{T}, \mathbf{1}, \mathbf{1})$	$\mathbf{5} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$	$\mathbf{5} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$
$\mathbf{1} \mapsto \{\mathbf{1}\}$	$\mathbf{1} \mapsto \{\mathbf{1}\}$	$\mathbf{1} \mapsto \{\mathbf{1}, \mathbf{3}\}$
$\mathbf{2} \mapsto \{\mathbf{1}\}$	$\mathbf{2} \mapsto \{\mathbf{1}\}$	$\mathbf{2} \mapsto \{\mathbf{1}, \mathbf{3}\}$
$\mathbf{3} \mapsto \{\mathbf{1}\}$	$\mathbf{3} \mapsto \{\mathbf{1}, \mathbf{5}\}$	$\mathbf{3} \mapsto \{\mathbf{3}, \mathbf{5}\}$
$\mathbf{4} \mapsto \{\mathbf{1}\}$	$\mathbf{4} \mapsto \{\mathbf{1}, \mathbf{5}\}$	$\mathbf{4} \mapsto \{\mathbf{3}, \mathbf{5}\}$
$\mathbf{5} \mapsto \{\,\}$	$\mathbf{5} \mapsto \{\,\}$	$\mathbf{5} \mapsto \{\,\}$
$\mathbf{1} \mapsto (\mathsf{F}, \{\mathbf{1} : 1\})$	$\mathbf{1} \mapsto (\mathsf{F}, \{\mathbf{1} : 1\})$	$\mathbf{1} \mapsto (\mathsf{F}, \{\mathbf{2} : 1\})$
$\mathbf{2} \mapsto (\mathsf{F}, \{\mathbf{1} : 1\})$	$\mathbf{2} \mapsto (\mathsf{F}, \{\mathbf{1} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$	$\mathbf{2} \mapsto (\mathsf{F}, \{\mathbf{2} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$
$\mathbf{3} \mapsto (\mathsf{F}, \{\mathbf{1} : 1\})$	$\mathbf{3} \mapsto (\mathsf{F}, \{\mathbf{1} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$	$\mathbf{3} \mapsto (\mathsf{F}, \{\mathbf{2} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$
$\mathbf{4} \mapsto (\mathsf{T}, \{\mathbf{1} : 1\})$	$\mathbf{4} \mapsto (\mathsf{T}, \{\mathbf{4} : 1\})$	$\mathbf{4} \mapsto (\mathsf{T}, \{\mathbf{4} : 1\})$
$\mathbf{5} \mapsto (\mathsf{F}, \{\mathbf{1} : 1\})$	$\mathbf{5} \mapsto (\mathsf{F}, \{\mathbf{1} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$	$\mathbf{5} \mapsto (\mathsf{F}, \{\mathbf{2} : \frac{1}{2}, \mathbf{4} : \frac{1}{2}\})$

Fig. 2. Execution of the naive algorithm for the three automata of Figure 1.

lead to $\{\mathbf{1}, \mathbf{1}, \mathbf{1}\}$ assuming all states are conjectured equivalent in the first step, and the canonical form of this set is $\{\mathbf{1}\}$. Note that the states $\mathbf{1} - \mathbf{4}$ all have successor structure $\{\mathbf{1}\}$ in the first step of the algorithm, but they get distinguished from state $\mathbf{5}$, which has successor structure $\{\,\}$.

We see that in order to talk about equivalence of states, and in order to perform minimization, we need a notion of substitution and canonicalization. As it turns out, this corresponds exactly to the standard definition of functor in category theory (for Set):

Definition 2.2. $F\colon \mathsf{Set} \to \mathsf{Set}$ is a functor, if given an $p\colon A \to B$ (i.e., a "substitution"), we have a mapping $F[p]\colon F(A) \to F(B)$. Furthermore, this operation must satisfy $F[id] = id$ and $F[p \circ g] = F[p] \circ F[g]$.

We thus require all automata types to be given by functors in the sense of Definition 2.2. We can then talk about equivalence of states, and minimize automata by repeatedly applying this operation $F[p]$ as sketched above. A more formal naive algorithm will be discussed in Section 4.2.

2.3 The Challenge: A Generic *and* Efficient Algorithm

The problem with the naive algorithm sketched in Section 2.2 is that it processes all transitions in every iteration of the main loop. In certain cases, partition refinement (in general) may take $\Theta(n)$ iterations to converge, where n is the number of states. This can happen, for instance, if the automaton has a long chain of transitions, so in each iteration, only one state is moved to a different block. Figure 3 contains three example automata for which the naive algorithm takes $\Theta(n)$ iterations (provided one generalizes the examples to have n nodes).

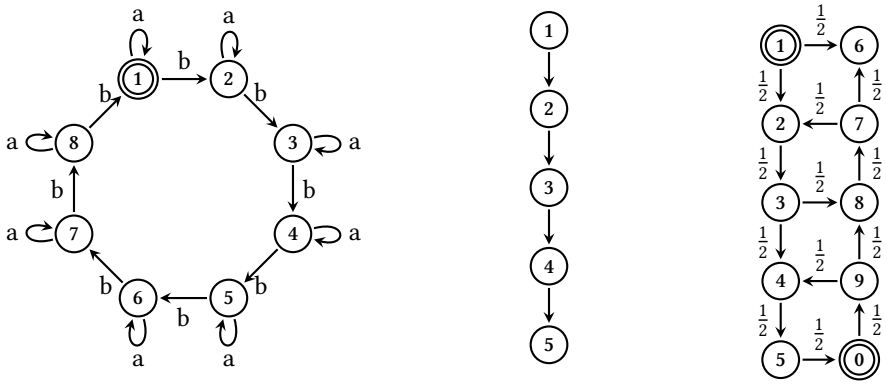


Fig. 3. Examples of shapes of automata on which the naive algorithm runs in $\Theta(n^2)$.

Since naive algorithm computes new successor structures for all states in each iteration, the functor operation is applied $O(n^2)$ times in total. Thus, the challenge we set out to solve is the following:

Can we find an asymptotically and practically efficient algorithm for automaton minimization that uses only the successor structure recomputation operation $F[p]$?

By using only $F[p]$, we do not impose further conditions on the functor F beside $F[p]$ being computable Since the algorithm does not inspect F any further, the only condition imposed on the functor is that $F[p]$ is computable for all substitutions p on the state space.

2.4 Hopcroft's Trick: The Key to Efficient Automaton Minimization

A key part of the solution is a principle often called "Hopcroft's trick" or "half the size" trick, which underlies all known asymptotically efficient automata minimization algorithms. To understand the trick, consider the following game:

(1) We start with a set of objects, *e.g.,* $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
(2) We chop the set into two parts arbitrarily, *e.g.,* $\{1, 3, 5, 7, 9\}, \{2, 4, 6, 8\}$.
(3) We select one of the sets, and chop it up arbitrarily again, *e.g.,* $\{1, 3\}, \{5, 7, 9\}, \{2, 4, 6, 8\}$.
(4) We continue the game iteratively (possibly until all sets are singletons).

Once the game is complete, we trace back the history of one particular element, say 3, and count how many times it was in the smaller part of a split:

The number of times an element was part of the smaller half of a split is $O(\log n)$.

One can prove this bound by considering the evolution of the size of the set containing the element. Initially, this size is n. Each time the element was part of the smaller part of the split, the size of the surrounding set gets cut in at least half, which can happen at most $O(\log n)$ times before we reach a singleton.

This indicates that for efficient algorithms, we should make sure that the running time of the algorithm is only proportional to the smaller halves of the splits. In other words, when we split a block, we have to make sure that we do not loop over the larger half of the split.

A slightly more general bound results from considering a game where we can split each set into an arbitrary number of parts, rather than 2:

The number of times an element was part of a smaller part of the split is $O(\log n)$.

In this case, "a smaller part of the split" is to be understood as any part of the split except the largest part. Thus, if we split $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ into $\{1, 3\}$, $\{5, 7, 9\}$, $\{2, 4, 6, 8\}$, then $\{1, 3\}$, $\{5, 7, 9\}$ are both considered "smaller parts", whereas $\{2, 4, 6, 8\}$ is the larger part.

In terms of algorithm design, our goal shall thus be that when we do a k-way split of a block, we may do operations proportional to all the $k - 1$ smaller parts of the split, but never an operation proportional to the largest part of the split.

2.5 A Sketch of our Generic and Efficient Algorithm

We design our algorithm based on the naive algorithm and Hopcroft's trick. The main problem with the naive algorithm is that it recomputes the successor structures of *all* states at each step. The reader may already have noticed that many of the successor structures in fact stay the same, and are unnecessarily recomputed. The successor structure of a state only changes if the block number of one of its successors changes. *The key to a more efficient algorithm is to minimize the number of times a block number changes, so that successor structure recomputation is avoided as much as possible.*

In the naive algorithm, we see that when we split a block of states into smaller blocks, we have freedom about which numbers to assign to each new sub-block. We therefore choose to *keep the old number for the largest sub-block*. Hopcroft's trick will then ensure that a state's number changes at most $O(\log n)$ times.

In order to reduce recomputation of successor structures, our algorithm tracks for each block of states (*i.e.,* states with the same block number), which of the states are *dirty*, meaning that at least one of their successors' number changed. The remaining states in the block are *clean*, meaning that the successors did not change.

Importantly, all clean states of a block *have the same successor structure*, because (A) their successors did not change (B) if their successor structure was different in the last iteration, they would have been placed in different blocks. Therefore, in order to recompute the successor structures of a block, it suffices to recompute the dirty states and *one* of the clean states, because we know that all the clean states have the same successor structure.

This sketch translates into the pseudocode of Algorithm 2.

Algorithm 2 Sketch of the optimized partition refinement algorithm

```
procedure PartRefSetFun(automaton)          ▷ Finds equivalent states of automaton
    Put all states in one block (i.e., assume that all states are equivalent)
    Mark all states dirty
    while number of blocks grows do
        Pick a block with dirty states
        Compute the successor structures of the dirty states and one clean state
        Mark all states in the block clean
        Split up the block, keeping the old block number for the largest sub-block
        Mark all predecessors of changed states dirty
```

Let us investigate the complexity of this algorithm in terms of the number of successor structure recomputations. By Hopcroft's trick, a state's number can now change at most $O(\log n)$ times, since we do not change the block number of the largest sub-block. Whenever we change a state's number, all the predecessors of that state will need to be marked dirty, and be recomputed. If we take a more global view, we can see that a recomputation may be triggered for every edge in the automaton, for each time the number of the destination state of the edge changes. Therefore, if

there are m edges, there will be at most $O(m \log n)$ successor structure recomputations, *i.e.,* at most $O(m \log n)$ calls to the functor operation.

In order to make the algorithm asymptotically efficient in terms of the total number of primitive computation steps, we must make sure to never do any operation that is proportional to the number of clean states in a block. Importantly, we must be able to split a block into k sub-blocks without iterating over the clean states. To do this, we have to devise efficient data structures to keep track of the blocks and their dirty states (Section 4.3).

We implement our algorithm (Section 4.4) with these data structures and efficient methods for computing the functor operation in our tool, *Boa*. When using *Boa*, the user can either encode their automata using a composition of the built-in functors, or implement their own functor operation and instantiate the algorithm with that.

Practical efficiency of the algorithm. Previous work on algorithms that apply to classes of functors that support more specialized operations in addition to just the functor operation can give better asymptotic complexity when one considers more fine-grained accounting than just the number of calls to the functor operation [Deifel et al. 2019; Dorsch et al. 2017; Wißmann et al. 2021; Wißmann et al. 2020]. Perhaps surprisingly, even though our algorithm is very generic and doesn't have access to these specialized operations, our algorithm is much faster than the more specialized algorithm in practice (Section 6).

However, the limiting factor in practice is not necessarily time but space. The aforementioned algorithm requires on the order of 16GB of RAM for minimizing automata with 150 thousand states [Deifel et al. 2019; Wißmann et al. 2021]. In order to be able to access more memory, distributed algortithms have been developed [Birkmann et al. 2022; Blom and Orzan 2005]. Using a cluster with 265GB of memory, the distributed algorithm was able to minimize an automaton with 1.3 million states and 260 million edges. By contrast, *Boa* is able to minimize the same automaton using only 1.7GB of memory.

The reason is that we do not need any large auxiliary data structures; most of the 1.7GB is used for storing the automaton itself. Furthermore, because we only need to compute the functor operation for states in the automaton, we are able to store the automaton in an efficient immutable binary format.

In the rest of the paper we will first give a more formal definition of bisimilarity in coalgebras (Section 3), we describe how we represent our automata, and which basic operations we need (Section 4.1), we describe the auxiliary data structures required by our algorithm (Section 4.3), we describe our algorithm and provide complexity bounds (Section 4.4), we show a variety of functor instances that our algorithm can minimize (Section 5), we compare the practical performance to earlier work (Section 6), and we conclude the paper (Section 7).

3 COALGEBRA AND BISIMILARITY, FORMALLY

In this section we define formally what it means for two states in a coalgebra to be behaviourally equivalent, and we give examples to show that behavioural equivalence in coalgebras reduces to known notions of bisimilarity for specific functors.

Recall that we model state-based systems as coalgebras for set functors (Definition 2.2):

Definition 3.1. An *F-coalgebra* consists of a carrier set C and a structure map $c\colon C \to FC$.

Intuitively, the carrier C of a coalgebra (C, c) is the set of states of the system, and for each state $x \in C$, the map provides $c(x) \in FC$ that is the structured collection of successor states of x. If $F = \mathcal{P}_\mathrm{f}$, then $c(x)$ is simply a finite set of successor states. The functor determines a canonical notion of behavioural equivalence.

Definition 3.2. A *homomorphism* between coalgebras $h\colon (C,c) \to (D,d)$ is a map $h\colon C \to D$ with $F[h](c(x)) = d(h(x))$ for all $x \in C$. States x, y in a coalgebra (C,c) are *behaviourally equivalent* if there is some other coalgebra (D,d) and a homomorphism $h\colon (C,c) \to (D,d)$ such that $h(x) = h(y)$.

Example 3.3. We consider coalgebras for the following functors (see also Table 1):

(1) Coalgebras for $\mathcal{P}_\mathrm{f}$ are finitely-branching transition systems and states x, y are behaviourally equivalent iff they are bisimilar.

(2) An (algebraic) signature is a set Σ together with a map $\mathrm{ar}\colon \Sigma \to \mathbb{N}$. The elements of $\sigma \in \Sigma$ are called *operation symbols* and $\mathrm{ar}(\sigma)$ is the arity. Every signature induces a functor defined by
$$\tilde{\Sigma}X = \{(\sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)}) \mid \sigma \in \Sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)} \in X\}$$
on sets and for maps $f\colon X \to Y$ defined by
$$\tilde{\Sigma}[f](\sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)}) = (\sigma, f(x_1), \ldots, f(x_{\mathrm{ar}(\sigma)})).$$
A state in a $\tilde{\Sigma}$-coalgebra describes a possibly infinite Σ-tree, with nodes labelled by $\sigma \in \Sigma$ with $\mathrm{ar}(\sigma)$ many children. Two states are behaviourally equivalent iff they describe the same Σ-tree.

(3) Deterministic finite automata on alphabet A are coalgebras for the signature Σ with 2 operation symbols of arity $|A|$. States are behaviourally equivalent iff they accept the same language.

(4) For a commutative monoid $(M, +, 0)$, the *monoid-valued* functor $M^{(X)}$ [Gumm and Schröder 2001, Def. 5.1] can be thought of as M-valued distributions over X:
$$M^{(X)} := \{\mu\colon X \to M \mid \mu(x) \neq 0 \text{ for only finitely many } x \in X\}$$
The map $f\colon X \to Y$ is sent by $M^{(-)}$ to
$$M^{(f)}\colon M^{(X)} \to M^{(Y)} \qquad M^{(f)}(\mu) = \big(y \mapsto \sum_{x \in X, f(x)=y} \mu(x)\big)$$
Coalgebras for $M^{(-)}$ are weighted systems whose weights come from M.
A coalgebra $c\colon C \to M^{(C)}$, sends a state $x \in C$ and another state $y \in C$ to a weight $m := c(x)(y) \in M$ which is understood as the weight of the transition $x \xrightarrow{m} y$, where $c(x)(y) = 0$ is understood as no transition. The coalgebraic behavioural equivalence captures weighted bisimilarity [Klin 2009]. Concretely, a weighted bisimulation is an equivalence relation $R \subseteq C \times C$ such that for all $x\,R\,y$ and $z \in C$:
$$\sum_{z\,R\,z'} c(x)(z') = \sum_{z\,R\,z'} c(y)(z')$$

(5) Taking $M = (\mathbb{Q}, +, 0)$, we get that $M^{(X)}$ are linear combinations over X. If we restrict to the subfunctor $\mathcal{D}(X) = \{f \in \mathbb{Q}_{\geq 0}^{(X)} \mid \sum_{x \in X} f(x) = 1\}$ where the weights are nonnegative and sum to 1, we get (rational finite support) probability distributions over X.[1]

(6) For two functors F and G, we can consider the coalgebra over their composition $F \circ G$. Taking $F = \mathcal{P}_\mathrm{f}$ and $G = A \times (-)$, coalgebras over $F \circ G$ are labelled transition systems with strong bisimilarity. Taking $F = \mathcal{P}_\mathrm{f}$ and $G = \mathcal{D}$, coalgebras over $F \circ G$ are Markov decision processes with probabilistic bisimilarity [Larsen and Arne Skou 1991, Def. 6.3], [Bartels et al. 2003, Thm. 4.2]. For $F = M^{(-)}$ and $G = \Sigma$ for some signature functor, FG-coalgebras are weighted tree automata and coalgebraic behavioural equivalence is backward bisimilarity [Björklund et al. 2009; Deifel et al. 2019].

Sometimes, we need to reason about successors and predecessors of a general F-coalgebra:

[1] In models of computation where addition of rational numbers isn't linear time, one can restrict to fixed-precision rationals $Q_q = \{\frac{p}{q} \mid p \in \mathbb{Z}\}$ for some fixed $q \in \mathbb{N}_{>0}$ to obtain our time complexity bound.

Table 1. List of functors, their coalgebras, and the accompanying notion of behavioural equivalence. The first five is given in Example 3.3, the last introduced later in Section 5.

Functor $F(X)$	Coalgebras $c\colon C \to FC$	Coalgebraic behavioural equivalence
$\mathcal{P}_\mathrm{f}(X)$	Transition Systems	(Strong) Bisimilarity
$\mathcal{P}_\mathrm{f}(A \times X)$	Labelled Transition Systems	(Strong) Bisimilarity
$M^{(X)}$	Weighted Systems (for a monoid M)	Weighted Bisimilarity
$\mathcal{P}_\mathrm{f}(\mathcal{D}(X))$	Markov Decision Processes	Probabilistic Bisimilarity
$M^{(\check{\Sigma}X)}$	Weighted Tree Automata	Backwards Bisimilarity
$\mathcal{N}(X)$	Monotone Neighbourhood Frames	Monotone Bisimilarity

Definition 3.4. Given a coalgebra $c\colon C \to FC$ and a state $x \in C$, we say that $y \in C$ is a *successor of* x if $c(x)$ is not in the image of $Fi_y\colon F(C \setminus \{y\}) \to FC$, where $i_y\colon C \setminus \{y\} \rightarrowtail C$ is the canonical inclusion. Likewise, x is a *predecessor* of y, and the *outdegree* of x is the number of successors of x.

Intuitively, y is a successor of x if y appears somewhere in the term that defines $c(x) \in F(C)$, like we did in the "coalgebra" row in Figure 1. We will access the predecessors in the minimization algorithm, and moreover, the total and maximum number of successors will be used in the run time complexity analysis.

4 COALGEBRAIC PARTITION REFINEMENT

In this section we will describe how the coalgebraic notions of the preceding section can be used for automata minimization.

4.1 Representing Abstract Data

When writing an abstract algorithm, it is crucial for the complexity analysis, how the abstract data is actually represented in memory. We understand finite sets like the carrier of the input coalgebra as finite cardinals $C \cong \{0, \dots, |C| - 1\} \subseteq \mathbb{N}$, and a map $f\colon C \to D$ for finite C is represented by an array of length $|C|$.

Coalgebra implementation. The coalgebra $c\colon C \to FC$ that we wish to minimize is given to the algorithm as a black-box, because it only needs to interact with the coalgebra via a specific interface. Whenever the algorithm comes up with a partition $p\colon C \to C'$, two states $x, y \in C$ need to be moved to different blocks if $F[p](c(x)) \neq F[p](c(y))$. Hence, the algorithm needs to derive $F[p](c(x))$ for states of interest $x \in C$. Since all partitions are finite, we can assume $C' \subseteq \mathbb{N}$, and so for simplicity, we consider partitions as maps $p\colon C \to \mathbb{N}$ with the image $\mathcal{I}m(p) = \{0, \dots, |C'| - 1\}$ and so $F[p](c(x))$ is an element of the set $F\mathbb{N}$.

For the case of labelled transition systems, i.e. $F(X) = \mathcal{P}_\mathrm{f}(A \times X)$, the binary representation of $F[p](c(x))$ is called the *signature of* $x \in C$ *with respect to* p [Blom and Orzan 2005]. This straightforwardly generalizes to arbitrary functors F [Birkmann et al. 2022; Wißmann et al. 2020], so we reuse the terminology *signature* for the binary encoding of the successor structure of $x \in C$ with respect to the blocks the partition p of the previous iteration.

Beside the signatures, the optimized minimization algorithm needs to be able to determine the predecessors of a state, in order to determine which states to mark dirty. Formally, we require:

Definition 4.1. The *implementation* of an F-coalgebra $c\colon C \to FC$ is the data $(n, \mathrm{sig}, \mathrm{pred})$ where:

(1) $n \in \mathbb{N}$ is a natural number such that $C \cong \{0, \dots, n-1\}$
(2) $\mathsf{sig}\colon C \times (C \to \mathbb{N}) \to 2^*$ is a function that given a state and a partition, computes the successor structure of the state (represented a binary data), satisfying for all partitions $p\colon C \to \mathbb{N}$ (encoded as an array of size $|C|$) that

$$\forall x, y \in C\colon \qquad \mathsf{sig}(x, p) = \mathsf{sig}(y, p) \quad \Leftrightarrow \quad F[p](c(x)) = F[p](c(y)) \tag{1}$$

(3) $\mathsf{pred}\colon C \to \mathcal{P}_\mathrm{f} C$ is a function such that $\mathsf{pred}(x)$ contains the predecessors of x.

Passing such a general interface makes the algorithm usable as a library, because the coalgebra can be represented in an arbitrary fashion in memory, as long as the above functions can be implemented.

The equivalence involving sig (1) specifies that the binary data of type 2^* returned by sig is some normalized representation of $F[p](c(x)) \in F\mathbb{N}$. For example, in the implementation for $F = \mathcal{P}_\mathrm{f}$, an element of $F\mathbb{N} = \mathcal{P}_\mathrm{f}\mathbb{N}$ is a set of natural numbers. Since e.g. $\{2, 0\}$ and $\{0, 2, 2\} \in \mathcal{P}_\mathrm{f}\mathbb{N}$ are the same set, the sig function essentially needs to sort the arising sets and remove duplicates:

Example 4.2. We can represent $\mathcal{P}_\mathrm{f}$-coalgebras $c\colon C \to \mathcal{P}_\mathrm{f} C$ by keeping for every state $x \in C$ an array of its successors $c(x) \subseteq C$ in memory. As a pre-processing step, we directly compute the predecessors for each state $x \in C$ and keep them as an array $\mathsf{pred}(x) \subseteq C$ for every state x in memory as well (computing the predecessors of all states can be done in linear time, and thus does not affect the complexity of the algorithm). With $n := |C|$, the remaining function sig is implemented as follows:

(1) Given $p\colon C \to \mathbb{N}$ and $x \in C$, create a new array t of integers of size $|c(x)|$. For each successor $y \in c(x)$, add $p(y) \in \mathbb{N}$ to t; this runs linearly in the length of t because we assume that the map p is represented as an array with $O(1)$ access.
(2) Sort t via radix sort and then remove all duplicates, with both steps taking linear time.
(3) Return the binary data blob of the integer array t.

For $\mathcal{P}_\mathrm{f}$, the computation of the signature of a state $x \in C$ thus takes $O(|c(x)|)$ time.

We discuss further instances in Section 5 later.

Renumber. By encoding everything as binary data in a normalized way, we are able to make heavy use of radix sort, and thus achieve linear bounds on sorting tasks. This trick is also used in the complexity analysis of Kanellakis and Smolka, who refer to it as lexicographic sorting method by Aho, Hopcroft, and Ullman [Aho et al. 1974]. We use this trick in order to turn arrays of binary data $p\colon B \to 2^*$ into their corresponding partitions $p'\colon B \to \{0, \dots, |\mathcal{Im}(p)| - 1\}$ satisfying $p(x) = p(y) \iff p'(x) = p'(y)$ for all $x, y \in B$. The pseudocode is listed in Algorithm 3: first, a permutation $r\colon B \to B$ is computed such that $p \circ r\colon B \to 2^*$ is sorted. This radix sort runs in $O(\sharp(p))$, where $\sharp(p) = \sum_{x \in B} |p(x)|$ is the total size of the entire array p. Since identical entries in p are now adjacent, a simple for-loop iterates over r and readily assigns block numbers.

Lemma 4.3. *Algorithm 3 runs in time $O(\sharp(p))$ for the parameter $p\colon B \to 2^*$ and returns a map $p'\colon B \twoheadrightarrow b$ for some $b \in \mathbb{N}$ such that for all $x, y \in B$ we have $p(x) = p(y) \Leftrightarrow p'(x) = p'(y)$.*

In the actual implementation, we use hash maps to implement Renumber. This is faster in practice but due to the resolving of hash-collisions, the theoretical worst-case complexity of the implementation has an additional log factor.

The renumbering can be understood as the compression of a map $p\colon B \to 2^*$ to an integer array $p'\colon B \to \mathbb{N}$. In the algorithm, the array elements of type 2^* are encoded signatures of states.

Algorithm 3 Renumbering an array using radix sort

procedure RENUMBER($p : B \to 2^*$)
  Create a new array r of size $|B|$ containing numbers $0..|B|$
  Sort r by the key $p : B \to 2^*$ using radix-sort
  Create a new array $p' : B \to \mathbb{N}$
  $j := 0$
  for $i \in 0..|B|$ **do**
    if $i > 0$ and $p[r[i-1]] \neq p[r[i]]$ **then** $j := j + 1$
    $p'[r[i]] := j$
  return p'

4.2 The Naive Method Coalgebraically

To illustrate the use of the encoding and notions defined above, let us restate the naive method (Algorithm 1, [Kanellakis and Smolka 1983; König and Küpper 2014]) in Algorithm 4. Recall that the basic idea is that it computes a sequence of partitions $p_i : C \to P_i$ ($i \in \mathbb{N}$) for a given input coalgebra $c : C \to FC$. Initially this partition identifies all states $p_0 : C \to 1$. In the first iteration, the map $p' : (C \xrightarrow{c} FC \xrightarrow{F[p]} F\mathbb{N})$ sends each state to its *output behaviour* (this distinguishes final from non-final states in DFAs and deadlock from live states in transition systems). Then this partition is refined successively under consideration of the transition structure: x, y are identified by $p_{i+1} : C \to P_{i+1}$ iff they are identified by the composed map

$$C \xrightarrow{c} FC \xrightarrow{F[p_i]} FP_i.$$

The algorithm terminates as soon as $p_i = p_{i+1}$, which then identifies precisely the behaviourally equivalent states in the input coalgebra (C, c).

Algorithm 4 The naive algorithm, also called *final chain partitioning*

procedure NAIVEALGORITHM'($c : C \to FC$)
  Create a new array $p : C \to \mathbb{N} := (x \mapsto 0)$ ▷ *i.e.* $p[x] = 0$ *for all* $x \in C$
  while $|\mathcal{I}m(p)|$ changes **do**
    compute $p' : C \to 2^* := x \mapsto \mathsf{sig}(x, p)$ ▷ $p'[x] \in 2^*$ *is the encoding of* $F[p](c(x)) \in F\mathbb{N}$
    $p : C \to \mathbb{N} := $ RENUMBER(p')

Recently, Birkmann et al. [Birkmann et al. 2022] have adapted this algorithm to a distributed setting, with a run time in $O(m \cdot n)$.

4.3 The Refinable Partition Data Structure

For the naive method it sufficed to represent the quotient on the state space $p : C \to \mathbb{N}$ by a simple array. For more efficient algorithms like our Algorithm 2, it is crucial to quickly perform certain operations on the partition, for which we have built upon a refinable partition data structure [Valmari 2009; Valmari and Lehtinen 2008]. The data structure keeps track of the partition of the states into blocks. A key requirement for our algorithm is the ability to split a block into k sub-blocks, where k is arbitrary. The refinable partition also tracks for each state whether it is *clean* or *dirty*, and a *worklist* of blocks with at least one dirty state.

Let us define the exposed functionality of the refinable partition data structure:

(1) Given (the natural number identifying) a block B, return its dirty states B_{di} in $O(|B_{\mathsf{di}}|)$.

(2) Given a block B, return one arbitrary clean state in $O(1)$ if there is any. We denote this by the set B_{cl_1} of cardinality at most 1. B_{cl_1} contains a clean state of B or is empty if all states of B are dirty.
(3) Return an arbitrary block with a dirty state and remove it from the worklist, in $O(1)$.
(4) MarkDirty(s): mark state s dirty, and put its block on the worklist, in $O(1)$.
(5) Split(B, A): split a block B into many sub-blocks according to an array $A\colon B_{\mathrm{di}} \to \mathbb{N}$. The array A indicates that the i-th dirty state is placed in the sub-block $A[i]$, meaning that two states s_1, s_2 stay together iff $A[s_1] = A[s_2]$. The clean states are placed in the 0-th sub-block, with those states satisfying $A[s] = 0$.
The block identifier of B gets re-used as the identifier for largest sub-block, and all states of B are marked clean. Split returns the list of all newly allocated sub-blocks, i.e. those except the re-used one.
For the time complexity of our algorithm, it is important that Split(B, A) runs in time $O(|B_{\mathrm{di}}|)$, regardless of the number of clean states.

In order to implement these operations with the desired run time complexity, we maintain the following data structures:

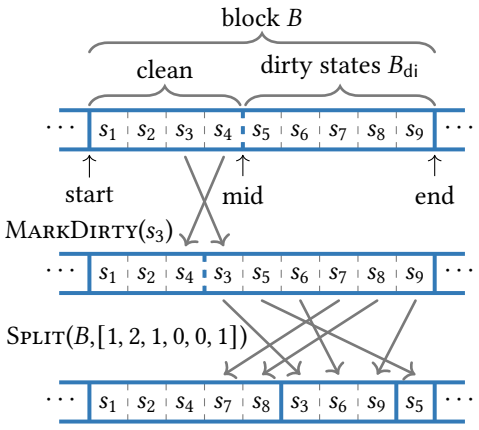
- `loc2state` is an array of size $|C|$ containing all states of C. Every block is a section of this array, and the other stuctures are used to quickly find and update the entries in the `loc2state` array. A visualization of an extract of this array is shown in Algorithm 5; for example lowermost row shows three blocks of size 5, 3, and 1, respectively.
- The array `state2loc` is inverse to `loc2state`; `state2loc`$[s]$ provides the index ("location") of state s in `loc2state`.
- `blocks` is an array of tuples $(start, mid, end)$ and specifies the blocks of the partition. A block identifier B is simply an index in this array and `blocks`$[B] = (start, mid, end)$ means that block B starts at `loc2state`$[start]$ and ends before `loc2state`$[end]$, as indicated in the visualization in Algorithm 5. The range $start..mid$ contains the clean states of B and $mid..end$ the dirty states. E.g. $mid = end$ iff the block has no dirty states.
- The array `block_of` of size $|C|$ that maps every state $s \in C$ to the ID $B =$ `block_of`$[s]$ of its surrounding block.
- `worklist` is a list of block identifiers and mentions those blocks with at least one dirty state.

With this data, we can implement the above-mentioned interface:

(1) For a block B, its dirty states B_{di} are the states `loc2state`$[mid..end]$ where `blocks`$[B] = (start, mid, end)$.
(2) One arbitrary clean state B_{cl_1} of a given block B is determined in a similar fashion: for `blocks`$[B] = (start, mid, end)$, if $start = mid$, then there is one clean state $B_{\mathrm{cl}_1} = \{\}$, and otherwise we chose $B_{\mathrm{cl}_1} =$ {`loc2state`$[start]$}.
(3) Returning an arbitrary block containing a dirty state is just a matter of extracting one element from `worklist`.
(4) The pseudocode of MarkDirty is listed in Algorithm 5: when marking a state $s \in C$ dirty, we first find the boundaries $(start, mid, end) =$ `blocks`$[B]$ of the surrounding block $B =$ `block_of`$[s]$. By the index `state2loc`$[s]$, we can check in $O(1)$ whether s is in the first ("clean") or second ("dirty") part of the block. Only if s wasn't dirty already, we need to do something: if B did not contain dirty states yet ($start = mid$), B now needs to be added to the `worklist`. Then, we change the location of s in the main array such that it becomes the last clean state, and then we make it dirty by moving the decrementing the index mid.
In the example in Algorithm 5, the content of `loc2state` is visualized. The bold dashed line visualizes the mid position, so states on the left of it are clean, states on the right are dirty. The

Algorithm 5 Refinable partition data structure with n-way split

```
procedure MarkDirty(s)
    ▷ Determine the block data ◁
    B := block_of[s]
    j := state2loc[s]
    (start, mid, end) := blocks[B]
    ▷ Do nothing if already dirty ◁
    if mid ≤ j then return
    ▷ Add to worklist if first dirty state ◁
    if mid = end then worklist.add(B)
    ▷ Swap s with the last clean state ◁
    s' := loc2state[mid − 1]
    state2loc[s'] := j
    state2loc[s] := mid
    loc2state[j] := s'
    loc2state[mid] := s
    ▷ Move marker to make s dirty ◁
    blocks[B].mid −= 1
```




```
procedure Split(B, A: B_di → ℕ)
    ▷ Cumulative counts of sub-block sizes ◁
    (start, mid, end) := blocks[B]
    D[0.. max_i A[i] + 1] := 0
    D[0] := mid − start
    for j ∈ B_di do
        D[A[j]] += 1
    i_max = argmax_i D[i]
    for i ∈ 1..|D| do
        D[i] += D[i − 1]
    ▷ Re-order the states by A-value ◁
    dirty := copy(loc2state[mid..end])
    for i ∈ reverse(0..|A|) do
        D[A[i]] −= 1
        j := start + D[A[i]]
        loc2state[j] := dirty[i]
        state2loc[loc2state[i]] := j
    D[0] −= mid − start
    ▷ Create blocks and assign IDs ◁
    D.add(end − start)
    old_block_count := |blocks|
    for i ∈ 0..|D| − 1 do
        j_0 := start + D[i]
        j_1 := start + D[i + 1]
        if i = i_max then
            blocks[B] = (j_0, j_1, j_1)
        else
            blocks.add(j_0, j_1, j_1)
            idx := |blocks| − 1
            block_of[loc2state[j_0..j_1]] := idx
    return old_block_count..|blocks|
```

call to MarkDirty(s_3) transforms the first row into the second row: it does so by moving s_3 from the clean states of B to the dirty ones, while s_4 stays clean.

(5) The pseudocode of Split is listed in Algorithm 5: for a block B, the caller provides us with an array $A\colon B_{di} \to \mathbb{N}$ that specifies which of the states stay together and which are moved to separate blocks. In the visualized example, $A = [1, 2, 1, 0, 0, 1]$ represents the map

$$s_3 \mapsto 1, \quad s_5 \mapsto 2, \quad s_6 \mapsto 1, \quad s_7 \mapsto 0, \quad s_8 \mapsto 0, \quad s_9 \mapsto 1$$

So Split(B, A) needs to create new blocks s_3, s_6, s_9 and s_5, while s_7, s_8 stay with the clean states. In any case, the clean states stay in the same block, so we can understand A as an efficient representation of the map

$$\bar{A}\colon B \to \mathbb{N} \qquad \bar{A}(s) = \begin{cases} A(s) & \text{if } s \in B_{di}, \\ 0 & \text{otherwise.} \end{cases}$$

Then, two states $s, s' \in B$ stay in the same block iff $\bar{A}[s] = \bar{A}[s']$. In the implementation, we first create an auxiliary array D which has different meanings. Before the definition of i_{max}, it counts the sizes of the resulting blocks:

$$D[i] = \{j \in B \mid \bar{A}[j] = i\}.$$

We compute D by initializing $D[0]$ with the number of clean states ($mid - start$) and iterating over A. The index of the largest block remembered in i_{max}, and then we change the meaning of D such that it now holds partial sums $D[i] := \sum_{0 \le j < i} D[j]$. For every new block i, this sum $D[i]$ denotes the end of the block, relative to the start of the old block B.

We use the sums to re-order the states such that states belonging to the same sub block come next to each other. The for-loop moves every state $i \in B$ to the end of the new block $A[i]$ and decrements $D[A[i]]$ such that the next state belonging to $A[i]$ is inserted before that. Finally, we do not need to move the clean states to sub-block 0, so we simply decrement $D[0]$ by the number of clean states. Since we have inserted all the elements at the end of their future subblocks and have decremented the entry of D during each insertion, the entries of D now point to the *first element* of each future subblock.

Having the states in the right position within B, we can now create the subblocks with the right boundaries. For convenience, we add the (relative) end of B to D, because then, every sub block i ranges from $D[i]$ to $D[i+1]$. We had saved the index of the largest subblock i_{max}, which will inherit the block identifier of B and the entry `blocks`$[B]$. For all other subblocks, we add a new block to `blocks`. All new blocks have no dirty states, so $mid = end$ for the new entries. If we have added a new block, then we need to update `block_of`$[s]$ for every state s in the subblock.

4.4 Optimized Algorithm

With the refinable partition data structure at hand, we can improve on the naive algorithm without restricting the choice of F. Our efficient algorithm is given in Algorithm 6. We start by creating a refinable partition data structure with a single block for all the states. We then iterate while there is still a block with dirty states, i.e. with states whose signatures should be recomputed. We split the block into sub-blocks in a refinement step that is similar to the naive algorithm, and re-use the old block for the largest sub-block.

To achieve our complexity bound, this splitting must happen in time $|B_{\mathrm{di}}|$, regardless of the number of clean states. Fortunately, this is possible because the clean states all have the same signature, because all their successors remained unchanged. Hence, it suffices to compute the signature for one arbitrary clean state, denoted by B_{cl_1}. Depending on the functor, it might happen that there are dirty states $d \in B_{\mathrm{di}}$ that have the same signature as the clean states. Having marked a state as "dirty" just means that the signature might have changed compared to the previous run, so it might be that the signature of a dirty state turns out to be identical to the clean states in the block B.

The wrapper Renumber′ then first compresses $p\colon B_{\mathrm{di}} \to 2^*$ to $A\colon B_{\mathrm{di}} \to \mathbb{N}$. Then, Renumber′ ensures that those dirty states $d \in B_{\mathrm{di}}$ with the same signature as the clean states satisfy $A(d) = 0$. This is used in Split: in the splitting operation, two dirty states $d, d' \in B_{\mathrm{di}}$ stay in the same block iff $A(d) = A(d')$ and the clean states end up in the same block as the dirty states d with $A(d) = 0$.

After the block B is split, we need to mark all states $x \in B$ as dirty whose signature might have possibly changed due to the updated partition. If the successor y of $x \in B$ was moved to a new block, i.e. if $p(y)$ changed, this might affect the signature of x. Conversely, if no successor of x changed block, then the signature of x remains unchanged:

Lemma 4.4. *If for a finite coalgebra $c\colon C \to FC$, two partitions $p_1, p_2\colon C \to \mathbb{N}$ satisfy $p_1(y) = p_2(y)$ for all successors y of $x \in C$, then $F[p_1](c(x)) = F[p_2](c(x))$.*

Algorithm 6 Optimized Partition Refinement for all Set functors

procedure PartRefSetFun(C, sig, pred) ▷ *i.e. for the implementation of* $c\colon C \to FC$
  Create a new refinable partition structure $p\colon C \to \mathbb{N}$
  Init p to have one block of all states, and all states marked dirty.
  while there is a block B with a dirty state **do**
    Compute the arrays
      $(sigs_{\mathsf{di}}\colon B_{\mathsf{di}} \to 2^*) := (x \mapsto \mathrm{sig}(x, p))$
      $(sigs_{\mathsf{cl}}\colon B_{\mathsf{cl}_1} \to 2^*) := (x \mapsto \mathrm{sig}(x, p))$
    } Compute signatures, in total $O(m \log n)$ calls to the coalgebra
    $A\colon B_{\mathsf{di}} \to \mathbb{N} := \textsc{Renumber}'(sigs_{\mathsf{di}}, sigs_{\mathsf{cl}})$
    $\vec{B}_{new} := \textsc{Split}(B, A)$
    } Split B according to signatures in $O(|B_{\mathsf{di}}|)$
    for every $B' \in \vec{B}_{new}$ and $s \in B'$ **do**
      for every $s' \in \mathrm{pred}(s)$ **do**
        MarkDirty(s')
    } Mark dirty all states with a successor in a new block in total time $O(m \log n)$
  return the partition p
procedure Renumber'($p\colon B_{\mathsf{di}} \to 2^*, q : B_{\mathsf{cl}_1} \to 2^*$)
  $(A\colon B_{\mathsf{di}} \to \mathbb{N}) := \textsc{Renumber}(p)$
  if there are $d \in B_{\mathsf{di}}, c \in B_{\mathsf{cl}_1}$ with $p(d) = q(c)$ **then**
    Swap the values 0 and $A(d)$ in the array A.
  } Ensure that $A(d) = 0$ for all dirty states d that have the same signature as the clean states.
  return A

We can now prove correctness of the partition refinement for coalgebras:

Theorem 4.5. *For a given coalgebra* $c\colon C \to FC$*, Algorithm 6 computes behavioural equivalence.*

4.5 Complexity Analysis

We structure the complexity analysis as a series of lemmas phrased in terms of the number of states $n = |C|$ and the total number of transitions m defined by

$$m := \sum_{x \in C} |\mathrm{pred}(x)|$$

As a first observation, we exploit that Split re-uses the block index for the largest resulting block. Thus, whenever x is moved to a block with a different index, the new block has at most half the size of the old block, leading to the logarithmic factor, by Hopcroft's trick:

Lemma 4.6. *A state is moved into a new block at most* $O(\log n)$ *times, that is, for every* $x \in C$*, the value of* $p(x)$ *in Algorithm 6 changes at most* $\lceil \log_2 |C| \rceil$ *many times.*

When a state is moved to a different block, all its predecessors are marked dirty. If there are m transitions in the system, and each state is moved to different block at most $\log n$ times, then:

Lemma 4.7. *MarkDirty is called at most* $m \cdot \lceil \log n \rceil + n$ *many times (including initialization).*

In the actual implementation, we arrange the pointers in the initial partition directly such that all states are marked dirty when the main loop is entered for the first time. The overall run time is dominated by the complexity of sig and pred. Here, we assume that sig always takes at least the time needed to write its return value. On the other hand, we allow that pred returns a pre-computed array by reference, taking only $O(1)$ time. The pre-computation of pred can be done at the beginning of the algorithm by iterating over the entire coalgebra once, e.g. it can be done along

with input parsing. This runs linear in the overall size of the coalgebra, and thus is dominated by the complexity of the algorithm:

Proposition 4.8. *The run time complexity of Algorithm 6 amounts to the time spent in* sig *and in* pred *plus* $O(m \cdot \log n + n)$.

Thus, it remains to count how often the algorithm calls sig. Roughly, sig is called for every state that becomes dirty, so we can show:

Theorem 4.9. *The number of invocations of* sig *in Algorithm 6 is bounded by* $O(m \cdot \log n + n)$.

Corollary 4.10. *If* sig *takes* f *time, if* pred *runs in* $O(1)$ *(returning a reference) and* $m \geq n$*, then Algorithm 6 computes behavioural equivalence in the input coalgebra in* $O(f \cdot m \cdot \log n)$ *time.*

Example 4.11. For $\mathcal{P}_\mathsf{f}$-coalgebras, sig takes $O(k)$ time, if every state has at most k successors. Then Algorithm 6 minimizes $\mathcal{P}_\mathsf{f}$-coalgebras in time $O(k \cdot m \cdot \log n)$. Note that $m \leq k \cdot n$, so the complexity is also bounded by $O(k^2 \cdot n \log n)$.

4.6 Comparison to Related Work on the Algorithmic Level

We can classify partition refinement algorithms by their time complexity, and by the classes of functors they are applicable to. For concrete system types, there are more algorithms than we can recall, so instead, we focus on early representatives and on generic algorithms.

The Hopcroft line of work. One line of work originates in Hopcroft's 1971 work on DFA minimization [Hopcroft 1971], and continues with Kanellakis and Smolka's [Kanellakis and Smolka 1983, 1990] work on partition refinement for transition systems running in $O(k^2 n \log n)$ where k is the maximum out-degree. It was a major achievement by Paige and Tarjan [Paige and Tarjan 1987] to reduce the run time to $O(kn \log n)$ by counting transitions and storing these transition counters in a clever way, which subsequently lead to a fruitful line of research on transition system minimization [Garavel and Lang 2022]. This was generalized to coalgebras in Deifel, Dorsch, Milius, Schröder and Wißmann's work on *CoPaR*, which is applicable to a large class of functors satisfying their *zippability* condition. These algorithms keep track of a worklist of blocks with respect to which *other* blocks still have to be split. Our algorithm, by contrast, keeps track of a worklist of blocks that *themselves* still potentially have to be split. Although similar at first sight, they are fundamentally different: in the former, one is given a block, and must determine how to split all the predecessor blocks, whereas in our case one is given a block, which is then split based on its successors.

The advantage of the former class of algorithms is that they have optimal time complexity $O(kn \log n)$, provided one can implement the special splitting procedure for the functor. The additional memory needed for the transition counters is linear in kn.

Our algorithm, by contrast, has an extra factor of k, but is applicable to all computable set-functors. By investing this extra time-factor k, we reduce the memory consumption because we do not need to maintain transition counters or intermediate states like *CoPaR*.

A practical advantage of our algorithm is that *one* recomputation of a block split can take into account the changes to *all* the other blocks that happened since the recomputation. The Hopcroft-*CoPaR* line of work, on the other hand, has to consider each change of the other blocks separately. This advantage is of no help in the asymptotic complexity, because in the worst case only one other split happened each time, and then our algorithm does in $O(k)$ what *CoPaR* can do in $O(1)$. However, as we shall see in the benchmarks of Section 6, in practice our algorithm outperforms *CoPaR* and *mCRL2*, even though our algorithm is applicable to a more general class of functors.

The Moore line of work. Another line of work originates in Moore's 1956 work on DFA minimization [Moore 1956], which in retrospect is essentially the naive algorithm specialized to DFAs. In this class, the most relevant for us is the algorithm by König and Küpper [König and Küpper 2014] for coalgebras, and the distributed algorithm of Birkmann, Deifel, and Milius [Birkmann et al. 2022]. Like our algorithm, algorithms in this class split a block based on its successors, and can be applied to general functors. Unlike the Hopcroft-*CoPaR* line of work and our algorithm, the running time of these algorithms is $O(kn^2)$.

Another relevant algorithm in this class is the algorithm of Blom and Orzan [Blom and Orzan 2005] for transition systems. Their main algorithm runs in time $O(kn^2)$, but in a side note they mention a variation of their algorithm that runs in $O(n \log n)$ *iterations*. They do not further analyse the time complexity or describe how to implement an iteration, because the main focus of their paper is a distributed implementation of the $O(kn^2)$ algorithm, and the $O(n \log n)$ variation precludes distributed implementation. Out of all algorithms, Blom and Orzan's $O(n \log n)$ variation is the most similar to our algorithm, in particular because their algorithm is in the Moore line of work, yet also re-uses the old block for the largest sub-block (which is a feature that usually appears in the Hopcroft-*CoPaR* line of work). However, their block splitting is different from ours and is only correct for labelled transition systems but can not be easily applied to general functors F.

5 INSTANCES

We give a list of examples of instances that can be supported by our algorithm. We start with the instances that were already previously supported by *CoPaR*, and then give examples of instances that were not previously supported by $n \log n$ algorithms.

5.1 Instances also Supported by *CoPaR*

Products and coproducts. The simplest instances are those built using the product $F \times G$ and disjoint union $F + G$, or in general, signature functors for countable signatures Σ. The binary encoding of an element of signature functor $(\sigma, x_1, \ldots, x_k) \in \tilde{\Sigma} X$ starts with a specification of σ, followed by the concatenation of encodings of the parameters $x_1, \ldots, x_k$. The functor implementation can simply apply the substitution recursively to these elements $x_1, \ldots, x_k$, without any further need for normalization.

Powerset. The finite powerset functor $\mathcal{P}_\mathrm{f}$ can be used to model transition systems as coalgebras. In conjunction with products and coproducts, we can model nondeterministic (tree) automata and labelled transition systems. The binary encoding of an element $\{x_1, \ldots, x_k\}$ of the powerset functor, is stored as a list of elements prefixed by its length. The functor implementation can recursively apply the substitution to the elements of a set $\{x_1, \ldots, x_k\}$, and subsequently normalize by sorting the resulting elements and removing adjacent duplicates.

Monoid-valued functors. The binary encoding of $\mu \in M^{(X)}$ (for a countable monoid M) is an array of pairs $(x_i, \mu(x_i))$. The binary encoding stores a list of these pairs prefixed by the length of the list. The functor implementation recursively applies the substitution to the x_i, and then sorts the pairs by the x_i value, and removes adjacent duplicate x_i by summing up their associated monoid values $\mu(x_i)$.

5.2 Instances not Supported by *CoPaR*

Composition of functors without intermediate states. The requirement of zippability in the $m \log n$ algorithm [Deifel et al. 2019] is not closed under the composition of functors $F \circ G$. As a workaround, one can introduce explicit intermediate states between F- and G-transitions. This introduces potentially many more states into the coalgebra, which leads to increased memory usage. Our

algorithm can use the composed functor directly without splitting states, because it works for any computable functor. This is important for practical efficiency.

Monotone Modal Logics and Monotone Bisimulation. When reasoning about game-theoretic settings [Parikh 1985; Pauly 2001; Peleg 1987], the arising modal logics have modal operators that talk about the ability of agents to enforce properties in the future. This leads to *monotone modal logics* whose domain of reasoning are *monotone neighbourhood frames* and the canonical notion of equivalence is *monotone bisimulation*. It was shown by Hansen and Kupke [Hansen and Kupke 2004a,b] that these are an instance of coalgebras and coalgebraic behavioural equivalence for the monotone neighbourhood functor. Instead of the original definition, it suffices for our purposes to work with the following equivalent characterization:

Definition 5.1 [Hansen and Kupke 2004a, Lem 3.3]. The monotone neighbourhood functor $\mathcal{N}\colon \mathsf{Set} \to \mathsf{Set}$ is given by

$$\mathcal{N}X = \{N \in \mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}X \mid N \text{ upwards closed}\} \text{ and } \mathcal{N}(f\colon X \to Y)(N) = {\uparrow}\{f[S] \mid S \in N\}.$$

where $\uparrow$ denotes upwards closure.

Hence, in a coalgebra $c\colon C \to \mathcal{N}C$, the successor structure of a state $x \in C$ is an upwards closed family of neighbourhoods $c(x)$.

To avoid redundancy, we do not keep the full neighbourhoods in memory, but only the least elements in this family: given a family $N \in \mathcal{N}X$ for finite X, we define the map

$$\mathsf{atom}_X\colon \mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}X \to \mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}X \qquad \mathsf{atom}_X(N) = \{S \in N \mid \nexists S' \in N : S' \subsetneq S\}$$

which transforms a monotone family into an antichain by taking the minimal elements in the monotone family.

Definition 5.2. We can implement $\mathcal{N}$-coalgebras as follows: For a coalgebra $c\colon C \to \mathcal{N}C$, keep for every state $x \in C$ an array of arrays representing $\mathsf{atom}_C(c(x)) \in \mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}X$. The predecessors of a state y needs to be computed in advance and is given by

$$\mathsf{pred}(y) = \{x \in C \mid y \in A \text{ for some } A \in \mathsf{atom}_C(c(x))\}.$$

For the complexity analysis, we specify the out-degree as

$$k := \max_{x \in C} \sum_{S \in \mathsf{atom}_C(c(x))} |S|.$$

For the signature $\mathsf{sig}(x, p)$ of a state x w.r.t. $p\colon C \to \mathbb{N}$, do the following:

(1) Compute $\mathcal{P}_\mathrm{f}[\mathcal{P}_\mathrm{f}[p]](t)$ for $t := \mathsf{atom}_C(c(x))$ by using the sig-implementation of $\mathcal{P}_\mathrm{f}$ first for each nested set and then on the outer set. This results in a new set of sets $t' := \mathcal{P}_\mathrm{f}[\mathcal{P}_\mathrm{f}[p]](t)$.
(2) For the normalization, iterate over all pairs $S, T \in t'$ and remove T if $S \subsetneq T$. This step is not linear in the size of t' but takes $O(k^2)$ time.

For such a monotone neighbourhood frame $c\colon C \to \mathcal{N}C$, note that for states $x \in C$, another state $y \in C$ might be contained in multiple sets $S \in c(x)$. Still, the definition of m in the complexity analysis is agnostic of this.

Proposition 5.3. *For a monotone neighbourhood frame $c\colon C \to \mathcal{N}C$, let $k \in \mathbb{N}$ be such that $|\mathsf{atom}_C(c(x))| \leq k$ for all $x \in C$. Algorithm 6 computes monotone bisimilarity in $O(k^2 \cdot m \log n)$ time.*

6 BENCHMARKS

To evaluate the practical performance and memory usage of our algorithm, we have implemented it in our tool *Boa* [Jules Jacobs 2022], written in Rust. The user of *Boa* can either use a composition of the built-in functors to describe their automaton type, or implement their own automaton type by implementing the interface of Section 4.1 in Rust. The user may then input the data of their automaton using either a textual format akin to the representation in the "Coalgebra" row of Figure 1, or use *Boa*'s more efficient and compact binary input format.

We test *Boa* on the benchmark suite of Birkmann, Deifel and Milius [Birkmann et al. 2022], consisting of real-world benchmarks (fms & wlan – from the benchmark suite of the PRISM model checker [Kwiatkowska et al. 2011]), and randomly generated benchmarks (wta – weighted tree automata). For the wta benchmarks, the size of the first 5 was chosen such that *CoPaR* [Deifel et al. 2019] uses 16GB of memory, and the size of the 6th benchmark was chosen by Birkmann, Deifel and Milius to demonstrate the scalability of their distributed algorithm.

The benchmark results are given in Table 2. The first columns list the type of benchmark and the size of the input coalgebra. For the size, the column n denotes the number of states and m is the number of edges as defined in Section 4.5. In the wlan benchmarks for CoPaR [Deifel et al. 2019; Wißmann et al. 2021], the reported number of states and eges also include intermediate states introduced by CoPaR in order to cope with functor composition, a preprocessing step which we do not need in *Boa*, and thus are different from the numbers in Table 2 here.

The three subsequent columns list the running time of *CoPaR*, *DCPR*, and *Boa*. The last two columns list the memory usage of *DCPR* and *Boa*. The benchmark results for *DCPR* and *CoPaR* are those reported by Birkmann, Deifel and Milius [Birkmann et al. 2022], and were run on their high performance computing cluster with 32 workers on 8 nodes with two Xeon 2660v2 chips (10 cores per chip + SMT) and 64GB RAM. The memory usage of *DCPR* is *per worker*, indicated by the ×32.

Execution times of *CoPaR* were taken using one node of the cluster. Some entries for *CoPaR* are missing, indicating that it ran out of its 16GB of memory. The benchmark results for our algorithm were obtained on one core of a 2.3GHz MacBook Pro 2019.

A point to note is that compared to *CoPaR*, the distributed algorithm does best on the randomly generated benchmarks. The distributed algorithm beats *CoPaR* in execution time by taking advantage of the large parallel compute power of the HPC cluster. This comes at the cost of $O(n^2)$ worst case complexity, but randomly generated benchmarks are more or less the *best case* for the distributed algorithm, and require only a very small constant number of iterations, so that the effective complexity is $O(n)$. The real world benchmarks on the other hand, and especially the wlan benchmarks, need more iterations, which results in sequential *CoPaR* outperforming *DCPR*. In general, benchmarks with transition systems with long shortest path lengths will truly trigger the worst case of the $O(n^2)$ algorithm, and can make its execution time infeasably long. In summary, the benchmarks here are not chosen to be favourable to *CoPaR* and our algorithm, as they do not trigger the time complexity advantage to the full extent.

Nevertheless, our algorithm outperforms both *CoPaR* and *DCPR* by a large margin. On the synthetic benchmarks (wta), roughly speaking, when *CoPaR* takes 10 minutes, *DCPR* takes one minute, and our algorithm takes a second. On the real-world wlan benchmark, the difference with *DCPR* is greatest, with the largest benchmark requiring almost an hour on the HPC cluster for *DCPR*, whereas our algorithm completes the benchmark in less than a second on a single thread.

Sequential *CoPaR* is unable to run the largest wta benchmarks, because it requires more memory than the 16GB limit. The distributed algorithm is able to spread the required memory usage among 32 workers, thus staying under the 16GB limit per worker. Our algorithm uses sufficiently less

Table 2. Time and memory usage comparison on the benchmarks of Birkmann, Deifel and Milius [Birkmann et al. 2022]. The columns n, %red, m give the number of states, the percentage of redundant states, and the number of edges, respectively. The results for *Boa* are an average of 10 runs. The results for *CoPaR* and *DCPR* are those reported in Birkmann, Deifel and Milius [Birkmann et al. 2022]. The memory usage of *DCPR* is per worker, indicated by ×32 (for the 32 workers on the HPC cluster)
The functors associated with the benchmarks are as follows: **fms:** $F(X) = \mathbb{Q}^{(X)}$, **wlan:** $F(X) = \mathbb{N} \times \mathcal{P}_{\mathrm{f}}(\mathbb{N} \times \mathcal{D}(X))$, **wta$_r$(M):** $F(X) = M \times M^{(4 \times X^r)}$ where r indicates the branching factor of the tree automaton, and $M = W$ is the monoid of 64-bit words with bitwise-or, $M = Z$ is the monoid of integers with addition, and $M = 2$ is the monoid of booleans with logical-or.

benchmark				time (s)			memory (MB)	
type	**n**	**% red**	**m**	***CoPaR***	***DCPR***	***Boa***	***DCPR***	***Boa***
fms	35910	0%	237120	4	2	0.02	13×32	6
fms	152712	0%	1111482	17	8	0.10	62×32	20
fms	537768	0%	4205670	68	26	0.40	163×32	72
fms	1639440	0%	13552968	232	84	1.29	514×32	199
fms	4459455	0%	38533968	–	406	4.60	1690×32	557
wlan	248503	56%	437264	39	297	0.11	90×32	15
wlan	607727	59%	1162573	105	855	0.30	147×32	38
wlan	1632799	78%	3331976	–	2960	0.81	379×32	92
wta$_5$(2)	86852	0%	21713000	537	71	0.85	701×32	179
wta$_4$(2)	92491	0%	18498200	723	67	0.96	728×32	154
wta$_3$(2)	134207	0%	20131050	689	113	1.34	825×32	175
wta$_2$(2)	138000	0%	13800000	467	129	0.98	715×32	126
wta$_1$(2)	154863	0%	7743150	449	160	0.74	621×32	80
wta$_3$(2)	1300000	0%	195000000	–	1377	22.58	7092×32	1647
wta$_5$(W)	83431	0%	16686200	642	52	1.01	663×32	142
wta$_4$(W)	92615	0%	23153750	511	61	1.21	849×32	193
wta$_3$(W)	94425	0%	14163750	528	59	0.76	639×32	124
wta$_2$(W)	134082	0%	13408200	471	76	0.96	675×32	124
wta$_1$(W)	152107	0%	7605350	566	79	0.76	642×32	82
wta$_3$(W)	944250	0%	141637500	–	675	15.18	6786×32	1231
wta$_5$(Z)	92879	0%	18575800	463	56	0.67	754×32	161
wta$_4$(Z)	94451	0%	23612750	445	61	0.81	871×32	199
wta$_3$(Z)	100799	0%	15119850	391	64	0.62	628×32	135
wta$_2$(Z)	118084	0%	11808400	403	74	0.66	633×32	113
wta$_1$(Z)	156913	0%	7845650	438	82	0.68	677×32	93
wta$_3$(Z)	1007990	0%	151198500	–	645	19.55	5644×32	1325

memory to be able to run all benchmarks on a single machine. In fact, it uses significantly less memory than *DCPR* uses *per worker*. There are several reasons for this:

- Our algorithm does not require large hash tables.
- Our algorithm uses an binary representation with simple in-memory dictionary compression.
- We operate directly on the composed functor instead of splitting states into pieces.

Even the largest benchmarks stay far away from the 16GB memory limit. We are thus able to minimize large coalgebraic transition systems on cheap, consumer grade hardware.

To assess the cost of genericity, we also compare with *mCRL2*, a full toolset for the verification of concurrent systems. Among many other tasks, *mCRL2* also supports minimization of transition systems by strong bisimilarity as part of the `ltsconvert` command[2] and even implements multiple algorithms for that, out of which the algorithm by Jansen et al. [Jansen et al. 2020] turned out to be the fastest. For benchmarking, we ran its implementation in *mCRL2* and compared the fastest with the run time of *Boa*. As input files, we used the *very large transition systems* (VLTS) benchmark suite[3]. Unfortunately, the benchmark suite is not available online in an open format, so the files were converted with the CADP tool to the plain text `.aut` format, supported by *mCRL2* and our tool. The results are shown in Table 3. The benchmark consists of two series of input files, *cwi* and *vasy*, whose file sizes ranged from a few KB to hundreds of MB (biggest vasy ws 145MB in zipped format and biggest cwi was 630MB zipped). Surprisingly, *Boa* is significantly faster than the bisimilarity minimization implemented in *mCRL2*. On all input files, *mCRL2* and *Boa* agreed on the size of the resulting partition, giving confidence in the correctness of the computed partition. It should be noted that *mCRL2* supports a wide range of bisimilarity notions (e.g. branching bisimilarity), which our algorithm can not cover.

7 CONCLUSIONS AND FUTURE WORK

The coalgebraic approach enables generic tools for automata minimization, applying to different types of input automata. With our coalgebraic partition refinement algorithm, implemented in our tool *Boa*, we reduce the time and memory use compared to previous work. This comes at the cost of an extra factor of k (the outdegree of a state) in the time-complexity compared to asymptotically optimal algorithms. Though our asymptotic complexity is not as good as the asymptotically fastest but less generic algorithms, the evaluation shows the efficiency of our algorithm.

We wish to expand the supported system equivalence notions. So far, our algorithm is applicable to functors on Set. More advanced equivalence and bisimilarity notions such as trace equivalence [Hasuo et al. 2007; Silva and Sokolova 2011], branching bisimulations, and others from the linear-time-branching spectrum [van Glabbeek 2001], can be understood coalgebraically using graded monads [Dorsch et al. 2019; Milius et al. 2015], corresponding to changing the base category of the functor from Set to, for example, the Eilenberg-Moore [Silva et al. 2013] or Kleisli [Hasuo et al. 2007] category of a monad. For branching bisimulation, efficient algorithms exist [Groote and Vaandrager 1990; Jansen et al. 2020], whose ideas might embed into our framework. We conjecture that it is possible to adapt the algorithm to nominal sets, in order to minimize (orbit-)finite coalgebras there [Kozen et al. 2015; Milius et al. 2016; Schröder et al. 2017; Wißmann 2023].

Up-to techniques provide another successful line of research for deciding bisimilarity. Bonchi and Pous [Bonchi and Pous 2013] provide a construction for deciding bismilarity of two particular states of interest, where the transition structure is unfolded lazily while the reasoning evolves. By computing the partitions in a similarly lazy way, performance of our minimization algorithm can hopefully be improved even further.

[2] https://www.mcrl2.org/web/user_manual/tools/release/ltsconvert.html
[3] https://cadp.inria.fr/resources/vlts/

Table 3. Time and memory usage comparison on the VLTS benchmark suite (for space reasons, we have excluded the very short running benchmarks). The columns n, %red, m give the number of states, the percentage of redundant states, and the number of edges, respectively. The results are an average of 10 runs. For *mCRL2*, the default `bisim` option was used, which runs the JGKW algorithm [Jansen et al. 2020].

benchmark				time (s)		memory (MB)	
type	**n**	**% red**	**m**	***mCRL2***	***Boa***	***mCRL2***	***Boa***
cwi	142472	97%	925429	0.85	0.08	99	15
cwi	214202	63%	684419	0.63	0.15	111	16
cwi	371804	90%	641565	0.38	0.11	95	22
cwi	566640	97%	3984157	6.19	0.44	414	60
cwi	2165446	98%	8723465	10.72	1.52	978	166
cwi	2416632	96%	17605592	14.87	1.56	1780	247
cwi	7838608	87%	59101007	231.08	17.43	5777	816
cwi	33949609	99%	165318222	312.11	35.41	16698	2809
vasy	52268	84%	318126	0.31	0.04	48	7
vasy	65537	0%	2621480	6.62	0.14	553	28
vasy	66929	0%	1302664	2.56	0.08	275	18
vasy	69754	0%	520633	0.93	0.04	128	11
vasy	83436	0%	325584	0.38	0.04	86	10
vasy	116456	0%	368569	0.47	0.06	105	15
vasy	164865	99%	1619204	1.92	0.23	162	22
vasy	166464	49%	651168	0.81	0.08	116	16
vasy	386496	99%	1171872	0.67	0.08	133	28
vasy	574057	99%	13561040	18.84	2.41	1277	141
vasy	720247	99%	390999	0.38	0.05	88	31
vasy	1112490	99%	5290860	8.86	0.78	579	93
vasy	2581374	0%	11442382	31.95	2.30	2691	285
vasy	4220790	67%	13944372	31.82	2.87	2293	311
vasy	4338672	40%	15666588	34.89	3.12	3160	372
vasy	6020550	99%	19353474	34.91	4.11	2124	534
vasy	6120718	99%	11031292	15.56	2.37	1297	325
vasy	8082905	99%	42933110	72.45	3.79	4313	719
vasy	11026932	91%	24660513	60.57	6.26	2768	661
vasy	12323703	91%	27667803	63.49	8.16	3103	740

ACKNOWLEDGMENTS

We thank Hans-Peter Deifel, Stefan Milius, Jurriaan Rot, Hubert Garavel, Sebastian Junges, Marck van der Vegt, Joost-Pieter Katoen, and Frits Vaandrager for helpful discussions and the anonymous referees for their valuable feedback for improving the paper. Thorsten Wißmann was supported by the NWO TOP project 612.001.852.

REFERENCES

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. 2000. Deciding Bisimilarity and Similarity for Probabilistic Processes. *J. Comput. Syst. Sci.* 60 (2000), 187–231.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

Falk Bartels, Ana Sokolova, and Erik de Vink. 2003. A hierarchy of probabilistic system types. In *Coagebraic Methods in Computer Science, CMCS 2003 (ENTCS, Vol. 82)*. Elsevier, 57 – 75.

Fabian Birkmann, Hans-Peter Deifel, and Stefan Milius. 2022. Distributed Coalgebraic Partition Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Proceedings, Part II (LNCS, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 159–177. https://doi.org/10.1007/978-3-030-99527-0_9

Johanna (Högberg) Björklund, Andreas Maletti, and Jonathan May. 2007. Bisimulation Minimisation for Weighted Tree Automata. In *Developments in Language Theory, DLT 2007 (LNCS, Vol. 4588)*. Springer, 229–241.

Johanna (Högberg) Björklund, Andreas Maletti, and Jonathan May. 2009. Backward and forward bisimulation minimization of tree automata. *Theor. Comput. Sci.* 410 (2009), 3539–3552.

Stefan Blom and Simona Orzan. 2005. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer* 7, 3 (June 2005), 280–291. https://doi.org/10.1007/s10009-004-0185-2

Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 457–468. https://doi.org/10.1145/2429069.2429124

Hans-Peter Deifel, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2019. Generic Partition Refinement and Weighted Tree Automata. In *Formal Methods – The Next 30 Years, Proc. 3rd World Congress on Formal Methods (FM 2019) (LNCS, Vol. 11800)*. Springer, 280–297.

Ulrich Dorsch, Stefan Milius, and Lutz Schröder. 2019. Graded Monads and Graded Logics for the Linear Time - Branching Time Spectrum. In *30th International Conference on Concurrency Theory, CONCUR 2019 (LIPIcs, Vol. 140)*, Wan J. Fokkink and Rob van Glabbeek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:16. https://doi.org/10.4230/LIPIcs.CONCUR.2019.36

Ulrich Dorsch, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2017. Efficient Coalgebraic Partition Refinement. In *Proc. 28th International Conference on Concurrency Theory (CONCUR 2017) (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Hubert Garavel and Frédéric Lang. 2022. Equivalence Checking 40 Years After: A Review of Bisimulation Tools *(Lecture Notes in Computer Science)*. https://doi.org/10.1007/978-3-031-15629-8_13

Jan Friso Groote and Frits W. Vaandrager. 1990. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 443)*, Mike Paterson (Ed.). Springer, 626–638. https://doi.org/10.1007/BFb0032063

Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. 2018. An Efficient Algorithm to Determine Probabilistic Bisimulation. *Algorithms* 11, 9 (2018), 131.

H. Peter Gumm and Tobias Schröder. 2001. Monoid-labeled transition systems. In *Coalgebraic Methods in Computer Science, CMCS 2001 (ENTCS, Vol. 44(1))*. Elsevier, 185–204.

Helle Hvid Hansen and Clemens Kupke. 2004a. A Coalgebraic Perspective on Monotone Modal Logic. *Electron. Notes Theor. Comput. Sci.* 106 (December 2004), 121–143. https://doi.org/10.1016/j.entcs.2004.02.028

Helle Hvid Hansen and Clemens Kupke. 2004b. A Coalgebraic Perspective on Monotone Modal Logic. In *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS (Electronic Notes in Theoretical Computer Science, Vol. 106)*, Jirí Adámek and Stefan Milius (Eds.). Elsevier, 121–143. https://doi.org/10.1016/j.entcs.2004.02.028

Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. 2007. Generic Trace Semantics via Coinduction. *Log. Methods Comput. Sci.* 3, 4 (2007). https://doi.org/10.2168/LMCS-3(4:11)2007

John Hopcroft. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*. Academic Press, 189–196.

David N. Jansen, Jan Friso Groote, Jeroen J. A. Keiren, and Anton Wijs. 2020. An O(m log n) algorithm for branching bisimilarity on labelled transition systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 3–20. https://doi.org/10.1007/978-3-030-45237-7_1

Thorsten Wissmann Jules Jacobs. 2022. Boa: binary coalgebraic partition refinement. https://doi.org/10.5281/zenodo.7150706 The most recent version is at https://github.com/julesjacobs/boa..

Paris C. Kanellakis and Scott A. Smolka. 1983. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. ACM, 228–240.

Paris C. Kanellakis and Scott A. Smolka. 1990. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Inf. Comput.* 86, 1 (1990), 43–68.

Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. 2007. Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007 (LNCS, Vol. 4424)*. Springer, 87–101.

Bartek Klin. 2009. Structural Operational Semantics for Weighted Transition Systems. In *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday (LNCS, Vol. 5700)*, Jens Palsberg (Ed.). Springer, 121–139.

Barbara König and Sebastian Küpper. 2014. Generic Partition Refinement Algorithms for Coalgebras and an Instantiation to Weighted Automata. In *Theoretical Computer Science, IFIP TCS 2014 (LNCS, Vol. 8705)*. Springer, 311–325.

Dexter Kozen, Konstantinos Mamouras, Daniela Petrisan, and Alexandra Silva. 2015. Nominal Kleene Coalgebra. In *Automata, Languages, and Programming, ICALP 2015 (lncs, Vol. 9135)*. springer, 286–298. https://doi.org/10.1007/978-3-662-47666-6

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 585–591.

Kim Guldstrand Larsen and Arne Arne Skou. 1991. Bisimulation through Probabilistic Testing. *Inform. Comput.* 94, 1 (1991), 1–28.

Jonathan May and Kevin Knight. 2006. Tiburon: A Weighted Tree Automata Toolkit. In *Implementation and Application of Automata*, Oscar H. Ibarra and Hsu-Chun Yen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–113.

Stefan Milius, Dirk Pattinson, and Lutz Schröder. 2015. Generic Trace Semantics and Graded Monads. In *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015 (LIPIcs, Vol. 35)*, Lawrence S. Moss and Pawel Sobocinski (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 253–269. https://doi.org/10.4230/LIPIcs.CALCO.2015.253

Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2016. Regular Behaviours with Names. *Applied Categorical Structures* 24, 5 (2016), 663–701. https://doi.org/10.1007/s10485-016-9457-8

Robin Milner. 1980. *A Calculus of Communicating Systems*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-10235-3

Edward F. Moore. 1956. *Gedanken-Experiments on Sequential Machines*. Princeton University Press, 129–154. https://doi.org/doi:10.1515/9781400882618-006

Robert Paige and Robert E. Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989.

Rohit Parikh. 1985. The Logic of Games and its Applications. In *Topics in the Theory of Computation, Selected Papers of the International Conference on 'Foundations of Computation Theory', FCT '83*. Elsevier, 111–139. https://doi.org/10.1016/s0304-0208(08)73078-0

Marc Pauly. 2001. *Logic for Social Software*. Ph. D. Dissertation. https://dare.uva.nl/search?identifier=9ad66ec5-063d-4673-8563-91369d0af7aa

David Peleg. 1987. Concurrent Dynamic Logic. *J. ACM* 34, 2 (apr 1987), 450–479. https://doi.org/10.1145/23005.23008

Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. 2017. Nominal Automata with Name Binding. In *FoSSaCS 2017 (LNCS, Vol. 10203)*, Javier Esparza and Andrzej Murawski (Eds.). Springer, 124–142. https://doi.org/10.1007/978-3-662-54458-7_8

Alexandra Silva, Filippo Bonchi, Marcello M. Bonsangue, and Jan J. M. M. Rutten. 2013. Generalizing determinization from automata to coalgebras. *Log. Methods Comput. Sci.* 9, 1 (2013). https://doi.org/10.2168/LMCS-9(1:9)2013

Alexandra Silva and Ana Sokolova. 2011. Sound and Complete Axiomatization of Trace Semantics for Probabilistic Systems. *Electronic Notes in Theoretical Computer Science* 276 (2011), 291–311. https://doi.org/10.1016/j.entcs.2011.09.027 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

Antti Valmari. 2009. Bisimilarity Minimization in $O(m \log n)$ Time. In *Applications and Theory of Petri Nets, PETRI NETS 2009 (LNCS, Vol. 5606)*. Springer, 123–142.

Antti Valmari. 2010. Simple Bisimilarity Minimization in O(m log n) Time. *Fundam. Informaticae* 105, 3 (2010), 319–339. https://doi.org/10.3233/FI-2010-369

Antti Valmari and Giuliana Franceschinis. 2010. Simple $O(m \log n)$ Time Markov Chain Lumping. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010 (LNCS, Vol. 6015)*. Springer, 38–52.

Antti Valmari and Petri Lehtinen. 2008. Efficient Minimization of DFAs with Partial Transition. In *Theoretical Aspects of Computer Science, STACS 2008 (LIPIcs, Vol. 1)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 645–656.

Rob J. van Glabbeek. 2001. The Linear Time - Branching Time Spectrum I. In *Handbook of Process Algebra*, Jan A. Bergstra, Alban Ponse, and Scott A. Smolka (Eds.). North-Holland / Elsevier, 3–99. https://doi.org/10.1016/b978-044482830-9/50019-9

Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction*. MIT Press.

Thorsten Wißmann, Hans-Peter Deifel, Stefan Milius, and Lutz Schröder. 2021. From generic partition refinement to weighted tree automata minimization. *Formal Aspects of Computing* (March 2021), 1–33. https://doi.org/10.1007/s00165-020-00526-z

Thorsten Wißmann. 2023. Supported Sets – A New Foundation For Nominal Sets And Automata. In *Computer Science Logic (CSL'23) (LIPIcs)*. http://arxiv.org/abs/2201.09825 to appear.

Thorsten Wißmann, Ulrich Dorsch, Stefan Milius, and Lutz Schröder. 2020. Efficient and Modular Coalgebraic Partition Refinement. *Logical Methods in Computer Science* 16:1 (January 2020), 8:1–8:63. https://doi.org/10.23638/LMCS-16(1:8)2020

Received 2022-07-07; accepted 2022-11-07