

Generic Partition Refinement and Weighted Tree Automata

Hans-Peter Deifel, Stefan Milius^{*}, Lutz Schröder^{*}, and Thorsten Wißmann^{*}

Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

{hans-peter.deifel,stefan.milius,lutz.schroeder,thorsten.wissmann}@fau.de

Abstract. Partition refinement is a method for minimizing automata and transition systems of various types. Recently, we have developed a partition refinement algorithm that is generic in the transition type of the given system and matches the run time of the best known algorithms for many concrete types of systems, e.g. deterministic automata as well as ordinary, weighted, and probabilistic (labelled) transition systems. Genericity is achieved by modelling transition types as functors on sets, and systems as coalgebras. In the present work, we refine the run time analysis of our algorithm to cover additional instances, notably weighted automata and, more generally, weighted tree automata. For weights in a cancellative monoid we match, and for non-cancellative monoids such as (the additive monoid of) the tropical semiring even substantially improve, the asymptotic run time of the best known algorithms. We have implemented our algorithm in a generic tool that is easily instantiated to concrete system types by implementing a simple refinement interface. Moreover, the algorithm and the tool are modular, and partition refiners for new types of systems are obtained easily by composing pre-implemented basic functors. Experiments show that even for complex system types, the tool is able to handle systems with millions of transitions.

1 Introduction

Minimization is a basic verification task on state-based systems, concerned with reducing the number of system states as far as possible while preserving the system behaviour. It is used for equivalence checking of systems and as a preprocessing step in further system analysis tasks, such as model checking.

In general, minimization proceeds in two steps: (1) remove unreachable states, and (2) identify behaviourally equivalent states. Here, we are concerned with the second step, which depends on which notion of equivalence is imposed on states; we work with notions of *bisimilarity* and generalizations thereof. Classically, bisimilarity for labelled transition systems obeys the principle “states x and y are bisimilar if for every transition $x \rightarrow x'$, there exists a transition $y \rightarrow y'$ with x' and y' bisimilar”. It is thus given via a fixpoint definition, to be understood as a *greatest* fixpoint, and can therefore be iteratively approximated from above. This is the principle behind *partition refinement* algorithms: Initially all states are tentatively

^{*} Supported by the DFG project COAX (MI 717/5-2 and SCHR 1118/12-2)

considered equivalent, and then this initial partition is iteratively refined according to observations made on the states until a fixpoint is reached. Unsurprisingly, such procedures run in polynomial time. Its comparative tractability (in contrast, e.g. trace equivalence and language equivalence of non-deterministic systems are PSPACE complete [24]) makes minimization under bisimilarity interesting even in cases where the main equivalence of interest is linear-time, such as word automata.

Kanellakis and Smolka [24] in fact provide a minimization algorithm with run time $\mathcal{O}(m \cdot n)$ for ordinary transition systems with n states and m transitions. However, even faster partition refinement algorithms running in $\mathcal{O}((m+n) \cdot \log n)$ have been developed for various types of systems over the past 50 years. For example, Hopcroft’s algorithm minimizes deterministic automata for a fixed input alphabet A in $\mathcal{O}(n \cdot \log n)$ [22]; it was later generalized to variable input alphabets, with run time $\mathcal{O}(n \cdot |A| \cdot \log n)$ [17,26]. The Paige-Tarjan algorithm minimizes transition systems in time $\mathcal{O}((m+n) \cdot \log n)$ [27], and generalizations to labelled transition systems have the same time complexity [23,13,33]. Minimization of weighted systems is typically called *lumping* in the literature, and Valmari and Franchesini [35] have developed a simple $\mathcal{O}((m+n) \cdot \log n)$ lumping algorithm for systems with rational weights.

In earlier work [14,37] we have developed an efficient *generic* partition refinement algorithm that can be easily instantiated to a wide range of system types, most of the time either matching or improving the previous best run time. The genericity of the algorithm is based on modelling state-based systems as coalgebras following the paradigm of universal coalgebra [30], in which the branching structure of systems is encapsulated in the choice of a functor, the *type functor*. This allows us to cover not only classical relational systems and various forms of weighted systems, but also to combine existing system types in various ways, e.g. nondeterministic and probabilistic branching. Our algorithm uses a functor-specific *refinement interface* that supports a graph-based representation of coalgebras. It allows for a generic complexity analysis, and indeed the generic algorithm has the same asymptotic complexity as the above-mentioned specific algorithms; for Segala systems [32] (systems that combine probabilistic and non-deterministic branching, also known as Markov decision processes), it even improves on the best known run time and matches the run time of a recent algorithm [19] discovered independently and almost at the same time.

The new contributions of the present paper are twofold. On the theoretical side, we show how to instantiate our generic algorithm to weighted systems with weights in a monoid (generalizing the group-weighted case considered previously [14,37]). We then refine the complexity analysis of the algorithm, making the complexity of the implementation of the type functor a parameter $p(n, m)$, where n and m are the numbers of nodes and edges, respectively, in the graph representation of the input coalgebra. In the new setup, the previous analysis becomes the special case where $p(n, m) = 1$. Under the same structural assumptions on the type functor and the refinement interface as previously, our algorithm runs in time $\mathcal{O}(m \cdot \log n \cdot p(n, m))$. Instantiated to the case of weighted systems over non-cancellative monoids (with $p(n, m) = \log(n)$), such as the

additive monoid $(\mathbb{N}, \max, 0)$ of the tropical semiring, the run time of the generic algorithm is $\mathcal{O}(m \cdot \log^2 m)$, thus markedly improving the run time $\mathcal{O}(m \cdot n)$ of previous algorithms for weighted automata [9] and, more generally, (bottom-up) weighted tree automata [20] for this case. In addition, for cancellative monoids, we again essentially match the complexity of the previous algorithms [9,20].

Our second main contribution is a generic and modular implementation of our algorithm, the *Coalgebraic Partition Refiner* (CoPaR). Instantiating CoPaR to coalgebras for a given functor requires only to implement the refinement interface. We provide such implementations for a number of basic type functors, e.g. for non-deterministic, weighted, or probabilistic branching, as well as (ranked) input and output alphabets or output weights. In addition, CoPaR is *modular*: For any type functor obtained by composing basic type functors for which a refinement interface is available, CoPaR automatically derives an implementation of the refinement interface. We explain in detail how this modularity is realized in our implementation and, extending Valmari and Franchescini's ideas [35], we explain how the necessary data structures need to be implemented so as to realize the low theoretical complexity. We thus provide a working efficient partition refiner for all the above mentioned system types. In particular, our tool is, to the best of our knowledge, the only available implementation of partition refinement for many composite system types, notably for weighted (tree) automata over non-cancellative monoids. The tool including source code and evaluation data is available at <https://git8.cs.fau.de/software/copar>.

2 Theoretical Foundations

Our algorithmic framework [14,37] is based on modelling state-based systems abstractly as *coalgebras* for a (set) *functor* that encapsulates the transition type, following the paradigm of *universal coalgebra* [30]. We proceed to recall standard notation for sets and maps, as well as basic notions and examples in coalgebra. We fix a singleton set $1 = \{*\}$; for every set X we have a unique map $! : X \rightarrow 1$. We denote composition of maps by $(-) \cdot (-)$, in applicative order. Given maps $f : X \rightarrow A$, $g : X \rightarrow B$ we define $\langle f, g \rangle : X \rightarrow A \times B$ by $\langle f, g \rangle(x) = (f(x), g(x))$. We model the transition type of state based systems using *functors*. Informally, a functor F assigns to a set X a set FX of structured collections over X , and an F -coalgebra is a map c assigning to each state x in a system a structured collection $c(x) \in FX$ of successors. The most basic example is that of transition systems, where F is powerset, so a coalgebra assigns to each state a set of successors. Formal definitions are as follows.

Definition 2.1. A *functor* $F : \mathbf{Set} \rightarrow \mathbf{Set}$ assigns to each set X a set FX , and to each map $f : X \rightarrow Y$ a map $Ff : FX \rightarrow FY$, preserving identities and composition ($\text{Fid}_X = \text{id}_{FX}$, $F(g \cdot f) = Fg \cdot Ff$). An F -*coalgebra* (C, c) consists of a set C of *states* and a *transition* structure $c : C \rightarrow FC$. A *morphism* $h : (C, c) \rightarrow (D, d)$ of F -coalgebras is a map $h : C \rightarrow D$ that preserves the transition structure, i.e. $Fh \cdot c = d \cdot h$. Two states $x, y \in C$ of a coalgebra $c : C \rightarrow FC$ are *behaviourally equivalent* ($x \sim y$) if there exists a coalgebra morphism h with $h(x) = h(y)$.

Example 2.2. (1) The *finite powerset* functor \mathcal{P}_ω maps a set X to the set $\mathcal{P}_\omega X$ of all *finite* subsets of X , and a map $f: X \rightarrow Y$ to the map $\mathcal{P}_\omega f = f[-]: \mathcal{P}_\omega X \rightarrow \mathcal{P}_\omega Y$ taking direct images. \mathcal{P}_ω -coalgebras are finitely branching (unlabelled) transition systems, and two states are behaviourally equivalent iff they are bisimilar.

(2) For a fixed finite set A , the functor given by $FX = 2 \times X^A$, where $2 = \{0, 1\}$, sends a set X to the set of pairs of boolean values and functions $A \rightarrow X$. An F -coalgebra (C, c) is a deterministic automaton (without an initial state). For each state $x \in C$, the first component of $c(x)$ determines whether x is a final state, and the second component is the successor function $A \rightarrow X$ mapping each input letter $a \in A$ to the successor state of x under input letter a . States $x, y \in C$ are behaviourally equivalent iff they accept the same language in the usual sense.

(3) For a commutative monoid $(M, +, 0)$, the *monoid-valued* functor $M^{(-)}$ sends each set X to the set of maps $f: X \rightarrow M$ that are finitely supported, i.e. $f(x) = 0$ for almost all $x \in X$. An F -coalgebra $c: C \rightarrow M^{(C)}$ is, equivalently, a finitely branching M -weighted transition system: For a state $x \in C$, $c(x)$ maps each state $y \in C$ to the weight $c(x)(y)$ of the transition from x to y . For a map $f: X \rightarrow Y$, $M^{(f)}: M^{(X)} \rightarrow M^{(Y)}$ sends a finitely supported map $v: X \rightarrow M$ to the map $y \mapsto \sum_{x \in X, f(x)=y} v(x)$, corresponding to the standard image measure construction. As the notion of behavioural equivalence of states in $M^{(-)}$ -coalgebras, we obtain weighted bisimilarity (cf. [9,25]), given coinductively by postulating that states $x, y \in C$ are behaviourally equivalent ($x \sim y$) iff

$$\sum_{z' \sim z} c(x)(z') = \sum_{z' \sim z} c(y)(z') \quad \text{for all } z \in C.$$

For the Boolean monoid $(2 = \{0, 1\}, \vee, 0)$, the monoid-valued functor $2^{(-)}$ is (naturally isomorphic to) the finite powerset functor \mathcal{P}_ω . For the monoid of real numbers $(\mathbb{R}, +, 0)$, the monoid-valued functor $\mathbb{R}^{(-)}$ has \mathbb{R} -weighted systems as coalgebras, e.g. Markov chains. In fact, finite Markov chains are precisely finite coalgebras of the *finite distribution functor*, i.e. the subfunctor \mathcal{D}_ω of $\mathbb{R}_{\geq 0}^{(-)}$ (and hence of $\mathbb{R}^{(-)}$) given by $\mathcal{D}_\omega(X) = \{\mu \in \mathbb{R}_{\geq 0}^{(X)} \mid \sum_{x \in X} \mu(x) = 1\}$. For the monoid $(\mathbb{N}, +, 0)$ of natural numbers, the monoid-valued functor is the bag functor \mathcal{B}_ω , which maps a set X to the set of finite multisets over X .

3 Generic Partition Refinement

We recall some key aspects of our generic partition refinement algorithm [14,37], which *minimizes* a given coalgebra, i.e. computes its quotient modulo behavioural equivalence; we center the presentation around the implementation and use of our tool.

The algorithm [37, Algorithm 4.5] is parametrized over a type functor F , represented by implementing a fixed *refinement interface*, which in particular allows for a representation of F -coalgebras in terms of nodes and edges (by no means implying a restriction to relational systems!). Our previous analysis has established that the algorithm minimizes F -coalgebras with n nodes and m edges in time $\mathcal{O}(m \cdot \log n)$, assuming $m \geq n$ and that the operations of the refinement

interface run in linear time. In the present paper, we generalize the analysis, establishing a run time in $\mathcal{O}(m \cdot \log n \cdot p(n, m))$, where $p(n, m)$ is a factor in the time complexity of the operations implementing the refinement interface, and depends on the functor at hand. In many instances, $p(n, m) = 1$, reproducing the previous analysis. In some cases, $p(n, m)$ is not constant, and our new analysis still applies in these cases, either matching or improving the best known run time in most instances, most notably weighted systems over non-cancellative monoids.

We proceed to discuss the design of the implementation, including input formats of our tool **CoPaR** for composite functors built from pre-implemented basic blocks and for systems to be minimized (Section 3.1). The refinement interface and its implementation are described in Section 3.2.

3.1 Generic System Specification

CoPaR accepts as input a file that represents a finite F -coalgebra $c: C \rightarrow FC$, and consists of two parts. The first part is a single line specifying the functor F . Each of the remaining lines describes one state $x \in C$ and its one-step behaviour $c(x)$. Examples of input files are shown in Fig. 1.

Functor specification. Functors are specified as composites of basic building blocks; that is, the functor given in the first line of an input file is an expression determined by the grammar

$$T ::= \mathbf{X} \mid F(T, \dots, T) \quad (F: \text{Set}^k \rightarrow \text{Set}) \in \mathcal{F}, \quad (1)$$

where the character \mathbf{X} is a terminal symbol and \mathcal{F} is a set of predefined symbols called *basic functors*, representing a number of pre-implemented functors of type $F: \text{Set}^k \rightarrow \text{Set}$. Only basic functors need to be implemented explicitly (Section 3.2); for composite functors, the tool derives instances of the algorithm automatically (Section 3.3). Basic functors currently implemented include (finite) powerset \mathcal{P}_ω , the bag functor \mathcal{B}_ω , monoid-valued functors $M^{(-)}$, and polynomial functors for finite many-sorted signatures Σ , based on the description of the respective refinement interfaces given in our previous work [14,37] and, in the case of $M^{(-)}$ for unrestricted commutative monoids M (rather than only groups) the newly developed interface described in Section 5. Since behavioural equivalence is preserved and reflected under converting G -coalgebras into F -coalgebras for a subfunctor G of F [37, Proposition 2.13], we also cover subfunctors, such as the finite distribution functor \mathcal{D}_ω as a subfunctor of $\mathbb{R}^{(-)}$. With the polynomial constructs $+$ and \times written in infix notation as usual, the currently supported

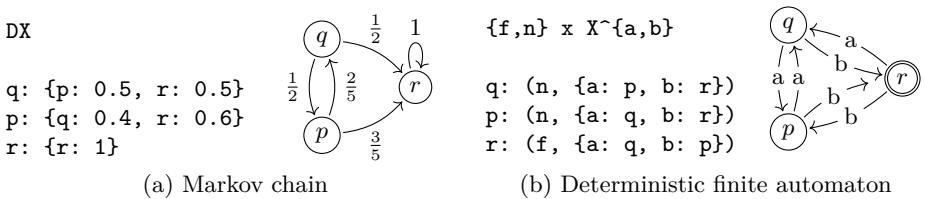


Fig. 1: Examples of input files with encoded coalgebras

grammar is effectively

$$\begin{aligned} T &::= \mathbf{x} \mid \mathcal{P}_\omega T \mid \mathcal{B}_\omega T \mid \mathcal{D}_\omega T \mid M^{(T)} \mid \Sigma \\ \Sigma &::= C \mid T + T \mid T \times T \mid T^A \quad C ::= \mathbb{N} \mid A \quad A ::= \{s_1, \dots, s_n\} \end{aligned} \quad (2)$$

where the s_k are strings subject to the usual conventions for C-style identifiers, exponents F^A are written $F\hat{\sim}A$, and M is one of the monoids $(\mathbb{Z}, +, 0)$, $(\mathbb{R}, +, 0)$, $(\mathbb{C}, +, 0)$, $(\mathcal{P}_\omega(64), \cup, \emptyset)$ (i.e. the monoid of 64-bit words with bitwise or), and $(\mathbb{N}, \max, 0)$ (the additive monoid of the tropical semiring). Note that C effectively ranges over at most countable sets, and A over finite sets. A term T determines a functor $F: \mathbf{Set} \rightarrow \mathbf{Set}$ in the evident way, with \mathbf{x} interpreted as the argument, i.e. $F(\mathbf{x}) = T$. It should be noted that the implementation treats composites of polynomial (sub-)terms as a single functor in order to minimize overhead incurred by excessive decomposition, e.g. $X \mapsto \{0, 1\} + \mathcal{P}_\omega(\mathbb{R}^{(X)}) + X \times X$ is composed from the basic functors \mathcal{P}_ω , $\mathbb{R}^{(-)}$ and the 3-sorted polynomial functor $\Sigma(X, Y, Z) = \{0, 1\} + X + Y \times Z$.

Coalgebra specification. The remaining lines of an input file define a finite F -coalgebra $c: C \rightarrow FC$. Each line of the form $x: \sqcup t$ defines a state $x \in C$, where x is a C-style identifier, and t represents the element $t = c(x) \in FC$. The syntax for t depends on the specified functor F , and follows the structure of the term T defining F ; we write $t \in T$ for a term t describing an element of FC :

- $t \in \mathbf{x}$ iff t is one of the named states specified in the file.
- $t \in T_1 \times \dots \times T_n$ is given by $t ::= (t_1, \dots, t_n)$ where $t_i \in T_i$, $i = 1, \dots, n$.
- $t \in T_1 + \dots + T_n$ is given by $t ::= \text{inj}_{\sqcup} i. t_i$ where $i = 1, \dots, n$ and $t_i \in T_i$.
- $t \in \mathcal{P}_\omega T$ and $t \in \mathcal{B}_\omega T$ are given by $t ::= \{t_1, \dots, t_n\}$ with $t_1, \dots, t_n \in T$.
- $t \in M^{(T)}$ is given by $t ::= \{t_1: \sqcup m_1, \dots, t_n: \sqcup m_n\}$ with $m_1, \dots, m_n \in M$ and $t_1, \dots, t_n \in T$, denoting $\mu \in M^{(TC)}$ with $\mu(t_i) = m_i$ and $\mu(t) = 0$ otherwise.

For example, for the functor F given by the term $T = \mathcal{P}_\omega(\{a, b\} \times \mathbb{R}^{(X)})$, the one-line declaration $\mathbf{x}: \{(\mathbf{a}, \{\mathbf{x}: 2.4\}), (\mathbf{a}, \{\}), (\mathbf{b}, \{\mathbf{x}: -8\})\}$ defines an F -coalgebra with a single state x , having two a -successors and one b -successor, where successors are elements of $\mathbb{R}^{(X)}$. One a -successor is constantly zero, and the other assigns weight 2.4 to x ; the b -successor assigns weight -8 to x . Two more examples are shown in Fig. 1.

Parsing input files. After reading the functor term T , the tool builds a parser for the functor-specific input format and parses an input coalgebra specified in the above syntax into an intermediate format described in Section 3.2. In the case of a composite functor, the parsed coalgebra then undergoes a substantial amount of preprocessing that also affects how transitions are counted; we defer the discussion of this point to Section 3.3, and assume for the time being that $F: \mathbf{Set} \rightarrow \mathbf{Set}$ is a basic functor with only one argument.

3.2 Refinement Interfaces

New functors are added to the framework by implementing a *refinement interface* (Definition 3.2). The interface relates to an abstract encoding of the functor and its coalgebras in terms of nodes and edges:

Definition 3.1. An *encoding* of a functor F consists of a set A of *labels* and a family of maps $\flat: FX \rightarrow \mathcal{B}_\omega(A \times X)$, one for every set X . The *encoding* of an F -coalgebra $c: C \rightarrow FC$ is given by the map $\langle F!, \flat \rangle \cdot c: C \rightarrow F1 \times \mathcal{B}_\omega(A \times C)$ and we say that the coalgebra has $n = |C|$ states and $m = \sum_{x \in C} |\flat(c(x))|$ edges.

An encoding does by no means imply a reduction from F -coalgebras to $\mathcal{B}_\omega(A \times (-))$ -coalgebras, i.e. the notions of behavioural equivalence for $\mathcal{B}_\omega(A \times (-))$ and F , respectively, can be radically different. The encoding just fixes a representation format, and \flat is not assumed to be natural (in fact, it fails to be natural in all encodings we have implemented except the one for polynomial functors). Encodings typically match how one intuitively draws coalgebras of various types as certain labelled graphs. For instance for Markov chains (see Fig. 1), i.e. coalgebras for the distribution functor \mathcal{D}_ω , the set of labels is the set of probabilities $A = [0, 1]$, and $\flat: \mathcal{D}_\omega X \rightarrow \mathcal{B}_\omega([0, 1] \times X)$ assigns to each finite probability distribution $\mu: X \rightarrow [0, 1]$ the bag $\{(\mu(x), x) \mid x \in X, \mu(x) \neq 0\}$.

The implementation of a basic functor contains two ingredients: (1) a parser that transforms the syntactic specification of an input coalgebra (Section 3.1) into the encoded coalgebra, and (2) the implementation of the refinement interface.

To understand the motivation behind the definition of a refinement interface, suppose that the generic partition refinement has already computed some block of states $B \subseteq C$ in its partition and that states in $S \subseteq B$ have different behaviour than those in $B \setminus S$. From this information, the algorithm has to infer whether states $x, y \in C$ that are in the same block and have successors in B exhibit different behaviour and thus have to be separated. For example, in the classical Paige-Tarjan algorithm [27], i.e. for $F = \mathcal{P}_\omega$, x and y can stay in the same block provided that (a) x has a successor in S iff y has one and (b) x has a successor in $B \setminus S$ iff y has one. Equivalently, $\mathcal{P}_\omega \langle \chi_S, \chi_{B \setminus S} \rangle (c(x)) = \mathcal{P}_\omega \langle \chi_S, \chi_{B \setminus S} \rangle (c(y))$, where $\chi_S: C \rightarrow 2$ is the usual characteristic function of the subset $S \subseteq C$. In the example of Markov chains, i.e. $F = \mathcal{D}_\omega$, $x, y \in C$ can stay in the same block if $\sum_{x' \in S} c(x)(x') = \sum_{y' \in S} c(y)(y')$ and $\sum_{x' \in B \setminus S} c(x)(x') = \sum_{y' \in B \setminus S} c(y)(y')$, i.e. if $\mathcal{D}_\omega \langle \chi_S, \chi_{B \setminus S} \rangle (c(x)) = \mathcal{D}_\omega \langle \chi_S, \chi_{B \setminus S} \rangle (c(y))$. Note that the element $(1, 1)$ is not in the image of $\langle \chi_S, \chi_{B \setminus S} \rangle: C \rightarrow 2 \times 2$. Since, moreover, $S \subseteq B$, we can equivalently consider the map

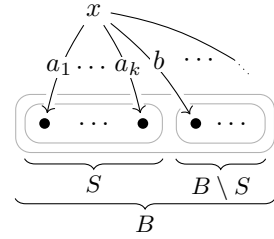


Fig. 2: Splitting a block

$$\chi_S^B: C \rightarrow 3, \quad \chi_S^B(x \in S) = 2, \quad \chi_S^B(x \in B \setminus S) = 1, \quad \chi_S^B(x \in C \setminus B) = 0. \quad (3)$$

That is, two states $x, y \in C$ can stay in the same block in the refinement step provided that $F\chi_S^B(c(x)) = F\chi_S^B(c(y))$. Thus, it is the task of a refinement interface to compute $F\chi_S^B \cdot c$ efficiently and incrementally.

Definition 3.2. Given an encoding (A, \flat) of the set functor F , a *refinement interface* for F consists of a set W of *weights* and functions

$$\text{init}: F1 \times \mathcal{B}_\omega A \rightarrow W \quad \text{and} \quad \text{update}: \mathcal{B}_\omega A \times W \rightarrow W \times F3 \times W$$

satisfying the following coherence condition: There exists a family of *weight maps* $w: \mathcal{P}X \rightarrow (FX \rightarrow W)$ such that for all $t \in FX$ and all sets $S \subseteq B \subseteq X$,

$$\begin{aligned} w(X)(t) &= \text{init}(F!(t), \mathcal{B}_\omega \pi_1(b(t))) \\ (w(S)(t), F\chi_S^B(t), w(B \setminus S)(t)) &= \text{update}(\{a \mid (a, x) \in b(t), x \in S\}, w(B)(t)). \end{aligned}$$

Note that the comprehension in the first argument of **update** is to be read as a *multiset* comprehension. In contrast to **init** and **update**, the function w is not called by the algorithm and thus does not form part of the refinement interface. However, its existence ensures the correctness of our algorithm. Intuitively, X is the set of states of the input coalgebra (C, c) , and for every $x \in C$, $w(B)(c(x)) \in W$ is the overall weight of edges from x to the block $B \subseteq C$ in the coalgebra (C, c) . The axioms in Definition 3.2 assert that **init** receives in its first argument the information which states of C are (non-)terminating, in its second argument the bag of labels of all outgoing edges of a state $x \in C$ in the graph representation of (C, c) , and it returns the total weight of those edges. The operation **update** receives a pair consisting of the bag of labels of all edges from some state $x \in C$ into the set $S \subseteq C$ and the weight of all edges from x to $B \subseteq C$, and from only this information (in particular **update** does not know x , S , and B explicitly) it computes the triple consisting of the weight $w(S)(c(x))$ of edges from x to S , the result of $F\chi_S^B \cdot c(x)$ and the weight $w(B \setminus S)(c(x))$ of edges from x to $B \setminus S$ (e.g. in the Paige-Tarjan algorithm, the number of edges from x to S , the value for the three way split, and the number of edges from x to $B \setminus S$, cf. Fig. 2). Those two computed weights are needed for the next refinement step, and $F\chi_S^B \cdot c(x)$ is used by the algorithm to decide whether or not two states $x, y \in C$ that are contained in the same block and have some successors in B remain in the same block for the next iteration.

For a given functor F , it is usually easy to derive the operations **init** and **update** once an appropriate choice of the set W of weights and weight maps w is made, so we describe only the latter in the following; see [14,37] for full definitions.

Example 3.3. (1) For $F = \mathbb{R}^{(-)}$ we can take $W = \mathbb{R}^2$, and $w(B)(t)$ records the accumulated weight of $X \setminus B$ and B : $w(B)(t) = (\sum_{x \in X \setminus B} t(x), \sum_{x \in B} t(x))$, i.e. $w(B) = F\chi_B: FX \rightarrow F2$.

(2) More generally, let F be one of the functors $G^{(-)}, \mathcal{B}_\omega, \Sigma$ where G is a group and Σ a signature with bounded arity, represented as a polynomial functor. Then we can take $W = F2$ (e.g. $W = \mathbb{R}^2$ for $F = \mathbb{R}^{(-)}$ as above) and $w(B) = F\chi_B: FX \rightarrow F2$.

(3) For $F = \mathcal{P}_\omega$, we need $W = 2 \times \mathbb{N}$, and $w(B)(t) = (|t \setminus B| \geq 1, |t \cap B|)$ records whether there is an edge to $X \setminus B$ and counts the numbers of edges into B .

In order to ensure that iteratively splitting blocks using $F\chi_S^B$ in each iteration correctly computes the minimization of the given coalgebra, we require that the type functor F is *zippable*, i.e. the evident maps $\langle F(X+!), F(!+Y) \rangle: F(X+Y) \longrightarrow F(X+1) \times F(1+Y)$ are injective [37, Definition 5.1]. All functors mentioned in Example 2.2 are zippable, and zippable functors are closed under products,

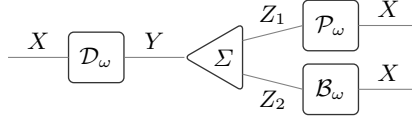


Fig. 3: Visualization of $FX = \mathcal{D}_\omega(\Sigma(\mathcal{P}_\omega X, \mathcal{B}_\omega X))$ for $\Sigma(Z_1, Z_2) = \mathbb{N} \times Z_1 \times Z_1$

coproducts, and subfunctors [37, Lemma 5.4] but not under functor composition; e.g. $\mathcal{P}_\omega \mathcal{P}_\omega$ fails to be zippable [37, Example 5.9].

The main correctness result [37] states that for a zippable functor equipped with a refinement interface, our algorithm correctly minimizes the given coalgebra. The low time complexity of our algorithm hinges on the time complexity of the implementations of `init` and `update`. We have shown previously [37, Theorem 6.22] that if both `init` and `update` run in linear time in the input list (of type $\mathcal{B}_\omega A$) *alone* (independently of n, m), then our generic partition refinement algorithm runs in time $\mathcal{O}((m+n) \cdot \log n)$ on coalgebras with n states and m edges. In order to cover instances where the run time of `init` and `update` depends also on n, m , we now generalize this to the following new result:

Theorem 3.4. *Let F be a zippable functor equipped with a refinement interface. Suppose further that $p(n, m)$ is a function such that in every run of the partition refinement algorithm on F -coalgebras with n states and m edges,*

- (1) *all calls to `init` and `update` on $\ell \in \mathcal{B}_\omega A$ run in time $\mathcal{O}(|\ell| \cdot p(n, m))$;*
- (2) *all comparisons of values of type W run in time $\mathcal{O}(p(n, m))$.*

Then the algorithm runs in overall time $\mathcal{O}((m+n) \cdot \log n \cdot p(n, m))$.

Obviously, for $p(n, m) \in \mathcal{O}(1)$, we obtain the previous complexity. Indeed, for the functors $G^{(-)}$, \mathcal{P}_ω , \mathcal{B}_ω , where G is an abelian group, we can take $p(n, m) = 1$; this follows from our previous work [37, Examples 6.4 and 6.6]. For a *ranked alphabet* Σ , i.e. a signature with arities of operations bounded by, say, r , we can take $p(m, n) = r \in \mathcal{O}(1)$ if Σ (or just r) is fixed. We will discuss in Section 5 how Theorem 3.4 instantiates to weighted systems, i.e. to monoid-valued functors $M^{(-)}$ for unrestricted commutative monoids M .

3.3 Combining Refinement Interfaces

In addition to supporting genericity via direct implementation of the refinement interface for basic functors, our tool is *modular* in the sense that it automatically derives a refinement interface for functors built from the basic ones according to the grammar (1). In other words, for such a functor the user does not need to write a single line of new code. Moreover, when the user implements a refinement interface for a new basic functor, this automatically extends the effective grammar.

For example, our tool can minimize systems of type $FX = \mathcal{D}_\omega(\mathbb{N} \times \mathcal{P}_\omega X \times \mathcal{B}_\omega X)$. To achieve this, a given F -coalgebra is transformed into one for the functor $F'X = \mathcal{D}_\omega X + (\mathbb{N} \times X \times X) + \mathcal{P}_\omega X + \mathcal{B}_\omega X$. This functor is obtained as the sum of all basic functors involved in F , i.e. of all the nodes in the visualization of the functor term F (Fig. 3). Then the components of the refinement interfaces of the four involved functors \mathcal{D}_ω , Σ , \mathcal{P}_ω , and \mathcal{B}_ω are combined by disjoint union $+$. The

transformation of a coalgebra $c: C \rightarrow FC$ into a F' -coalgebra introduces a set of intermediate states for each edge in the visualization of the term F in Fig. 3. E.g. Y contains an intermediate state for every \mathcal{D}_ω -edge, i.e. $Y = \{(x, y) \mid \mu(x)(y) \neq 0\}$. Successors of such intermediate states in Y lie in $\mathbb{N} \times Z_1 \times Z_2$, and successors of intermediate states in Z_1 and Z_2 lie in $\mathcal{P}_\omega X$ and $\mathcal{B}_\omega X$, respectively. Overall, we obtain an F' -coalgebra on $X + Y + Z_1 + Z_2$, whose minimization yields the minimization of the original F -coalgebra. The correctness of this construction is established in full generality in [37, Section 7].

CoPaR moreover implements a further optimization of this procedure that leads to fewer intermediate states in the case of polynomial functors Σ : Instead of putting the refinement interface of Σ side by side with those of its arguments, CoPaR includes a systematic procedure to combine the refinement interfaces of the arguments of Σ into a single refinement interface. For instance, starting from $FX = \mathcal{D}_\omega(\mathbb{N} \times \mathcal{P}_\omega X \times \mathcal{B}_\omega X)$ as above, a given F -coalgebra is thus transformed into a coalgebra for the functor $F''X = \mathcal{D}_\omega X + \mathbb{N} \times \mathcal{P}_\omega X \times \mathcal{B}_\omega X$, effectively inducing intermediate states in Y as above but avoiding Z_1 and Z_2 .

3.4 Implementation Details

Our implementation is geared towards realizing both the level of genericity and the efficiency afforded by the abstract algorithm. Regarding genericity, each basic functor is defined (in its own source file) as a single Haskell data type that implements two type classes: a class that directly corresponds to the refinement interface given in Definition 3.2 with its methods `init` and `update`, and a parser that defines the coalgebra syntax for the functor. This means that new basic functors can be implemented without modifying any of the existing code, except for registering the new type in a list of existing functors (refinement interfaces are in `src/Copar/Functors`).

A key data structure for the efficient implementation of the generic algorithm are refinable partitions, which store the current partition of the set C of states of the input coalgebra during the execution of the algorithm. This data structure has to provide constant time operations for finding the size of a block, marking a state and counting the marked states in a block. Splitting a block in marked and unmarked states must only take linear time in the number of marked states of this block. In CoPaR, we use such a data structure described (for use in Markov chain lumping) by Valmari and Franceschinis [35].

Our abstract algorithm maintains two partitions P, Q of C , where P is one transition step finer than Q ; i.e. P is the partition of C induced by the map $Fq \cdot c: C \rightarrow FQ$, where $q: C \rightarrow Q$ is the canonical quotient map assigning to every state the block which contains it. The key to the low time complexity is to choose in each iteration a *subblock*, i.e. a block S in P whose surrounding *compound block*, i.e. the block B in Q such that $S \subseteq B$, satisfies $2 \cdot |S| \leq |B|$, and then refine Q (and P) as explained in Section 3.2 (see Fig. 2). This idea goes back to Hopcroft [22], and is also used in all other partition refinement algorithms mentioned in the introduction. Our implementation maintains a queue of subblocks S satisfying the above property, and the termination condition $P = Q$ of the main loop then translates to this queue being empty.

One optimization that is new in CoPaR in relation to [35,37] is that weights for blocks of exactly one state are not computed, as those cannot be split any further. This has drastic performance benefits for inputs where the algorithm produces many single-element blocks early on, e.g. for nearly minimal systems or fine grained initial partitions, see [11] for details and measurements.

4 Instances

Many systems are coalgebras for functors composed according to the grammar (2). In Table 1, we list various system types that can be handled by our algorithm, taken from [14,37] except for weighted tree automata, which are new in the present paper. In all cases, m is the number of edges and n is the number of states of the input coalgebra, and we compare the run time of our generic algorithm with that of specifically designed algorithms from the literature. In most instances we match the complexity of the best known algorithm. In the one case where our generic algorithm is asymptotically slower (LTS with unbounded alphabet), this is due to assuming a potentially very large number of alphabet letters – as soon as the number of alphabet letters is assumed to be polynomially bounded in the number n of states, the number m of transitions is also polynomially bounded in n , so $\log m \in \mathcal{O}(\log n)$. This argument also explains why ‘<’ and ‘=’, respectively, hold in the last two rows of Table 1, as we assume Σ to be (fixed and) finite; the case where Σ is infinite and unranked is more complicated. Details on the instantiation to weighted tree automata are discussed in Section 5. We comment briefly on some further instances and initial partitions:

Further system types can be handled by our algorithm and tool by combining functors in various ways. For instance, general Segala systems are coalgebras for the functor $\mathcal{P}_\omega \mathcal{D}_\omega(A \times (-))$, and are minimized by our algorithm in time $\mathcal{O}((m+n) \cdot \log n)$, improving on the best previous algorithm [2]; other type functors for various species of probabilistic systems are listed in [3], including the ones for reactive systems, generative systems, stratified systems, alternating systems, bundle systems, and Pnueli-Zuck systems.

Initial partitions: Note that in the classical Paige-Tarjan algorithm [27], the input includes an initial partition. Initial partitions as input parameters are covered via the genericity of our algorithm: Initial partitions on F -coalgebras are accommodated by moving to the functor $F'X = \mathbb{N} \times FX$, where the first component of a coalgebra assigns to each state the number of its block in the initial partition. Under the optimized treatment of the polynomial functor $\mathbb{N} \times (-)$ (Section 3.3), this transformation does not enlarge the state space and also leaves the complexity parameter $p(n, m)$ unchanged [37]; that is, the asymptotic run time of the algorithm remains unchanged under adding initial partitions.

5 Weighted Tree Automata

We proceed to take a closer look at weighted tree automata as a worked example. In our previous work, we have treated the case where the weight monoid is a group; in the present paper, we extend this treatment to unrestricted monoids. As indicated previously, it is this example that mainly motivates the refinement

Table 1: Asymptotic complexity of the generic algorithm (2017/2019) compared to specific algorithms, for systems with n states and m transitions, respectively $m_{\mathcal{P}_\omega}$ nondeterministic and $m_{\mathcal{D}_\omega}$ probabilistic transitions for Segala systems. For simplicity, we assume that $m \geq n$ and, like [22,20], that A, Σ are finite.

System	Functor	Run-Time		Specific algorithm (Year)	
DFA	$2 \times (-)^A$	$n \cdot \log n$	=	$n \cdot \log n$	1971 [22]
Transition Systems	\mathcal{P}_ω	$m \cdot \log n$	=	$m \cdot \log n$	1987 [27]
Labelled TS	$\mathcal{P}_\omega(\mathbb{N} \times -)$	$m \cdot \log m$	=	$m \cdot \log m$	2004 [15]
			>	$m \cdot \log n$	2009 [33]
Markov Chains	$\mathbb{R}^{(-)}$	$m \cdot \log n$	=	$m \cdot \log n$	2010 [35]
Segala Systems	$\mathcal{P}_\omega(A \times -) \cdot \mathcal{D}$	$m_{\mathcal{D}_\omega} \cdot \log m_{\mathcal{P}_\omega}$	<	$m \cdot \log n$	2000 [2]
			=	$m_{\mathcal{D}_\omega} \cdot \log m_{\mathcal{P}_\omega}$	2018 [19]
Colour Refinement	\mathcal{B}_ω	$m \cdot \log n$	=	$m \cdot \log n$	2017 [5]
Weighted Tree Automata	$M \times M^{(\Sigma(-))}$	$m \cdot \log^2 m$	<	$m \cdot n$	2007 [20]
	$M \times M^{(\Sigma(-))}$ (M cancellative)	$m \cdot \log m$	=	$m \cdot \log n$	2007 [20]

of the run time analysis discussed in Section 3.2, and we will see that in the case of non-cancellative monoids, the generic algorithm improves on the run time of the best known specific algorithms in the literature.

Weighted tree automata simultaneously generalize tree automata and weighted (word) automata. A partition refinement construction for weighted automata (w.r.t. weighted bisimilarity) was first considered by Buchholz [9, Theorem 3.7]. Högberg et al. first provided an efficient partition refinement algorithm for tree automata [21], and subsequently for weighted tree automata [20]. Generally, tree automata differ from word automata in replacing the input alphabet, which may be seen as sets of unary operations, with an algebraic signature Σ :

Definition 5.1. Let $(M, +, 0)$ be a commutative monoid. A (bottom-up) *weighted tree automaton* (WTA) (over M) consists of a finite set X of states, a finite signature Σ , an output map $f: X \rightarrow M$, and for each $k \geq 0$, a transition map $\mu_k: \Sigma_k \rightarrow M^{X^k \times X}$, where Σ_k denotes the set of k -ary input symbols in Σ ; the maximum arity of symbols in Σ is called the *rank*.

A weighted tree automaton is thus equivalently a finite coalgebra for the functor $M \times M^{(\Sigma)}$ (where $M^{(\Sigma)}(X) = M^{(\Sigma X)}$) where $\Sigma: \mathbf{Set} \rightarrow \mathbf{Set}$ is a polynomial functor. Indeed, we can regard the output map as a transition map for a constant symbol, so it suffices to consider just the functor $M^{(\Sigma)}$ (and in fact the output map is ignored in the notion of backward bisimulation used by Högberg et al. [20]). For weighted systems, *forward* and *backward* notions of bisimulation

are considered in the literature [9,20]; we do not repeat the definitions here but focus on backward bisimulation, as it corresponds to behavioural equivalence:

Proposition 5.2. *Backward bisimulation of weighted tree automata coincides with behavioural equivalence of $M^{(\Sigma)}$ -coalgebras.*

Since $M^{(\Sigma)}$ is a composite of $M^{(-)}$ and a polynomial functor Σ , the modularity of our approach implies that it suffices to provide a refinement interface for $M^{(-)}$. For the case where M is a group, a refinement interface with $p(n, m) = 1$ has been given in our previous work. For the general case, we distinguish, like Högberg et al. [20], between cancellative and non-cancellative monoids, because we obtain a better complexity result for the former.

5.1 Cancellative Monoids

Recall that a commutative monoid $(M, +, 0)$ is *cancellative* if $a + b = a + c$ implies $b = c$. It is well-known that every cancellative commutative monoid M embeds into an abelian group G via the Grothendieck construction. Hence, we can convert $M^{(-)}$ -coalgebras into $G^{(-)}$ -coalgebras and use the refinement interface for $G^{(-)}$ from our previous work, obtaining

Theorem 5.3. *On weighted tree automata with n states, k transitions and rank r over a cancellative monoid, our algorithm runs in time $\mathcal{O}((rk + n) \cdot \log(k + n) \cdot r)$.*

Note that rk may be replaced with the number m of edges of the corresponding coalgebra. Thus, for a fixed signature and $m \geq n$, we obtain the bound in Table 1.

5.2 Non-cancellative Monoids

The refinement interface for $G^{(-)}$ for a group G (in which the cancellative monoid M in Section 5.1 is embedded) crucially makes use of inverses for the fast computation of the weights returned by **update**. For a non-cancellative monoid $(M, +, 0)$, we instead need to maintain bags of monoid elements and consider subtraction of bags. For the encoding of $M^{(-)}$, we take labels $A = M_{\neq 0} = M \setminus \{0\}$, and $\mathfrak{b}(f) = \{(f(x), x) \mid x \in X, f(x) \neq 0\}$ for $f \in M^{(-)}$. The refinement interface for $M^{(-)}$ has weights $W = M \times \mathcal{B}(M_{\neq 0})$ and

$$w(B)(f) = \left(\sum_{x \in X \setminus B} f(x), (m \mapsto |\{x \in B \mid f(x) = m\}|) \right) \in M \times \mathcal{B}(M_{\neq 0});$$

that is, $w(B)(f)$ returns the total weight of $X \setminus B$ under f and the bag of non-zero elements of M occurring in f . The interface functions **init**: $M^{(1)} \times \mathcal{B}_\omega M_{\neq 0} \rightarrow W$, **update**: $\mathcal{B}_\omega M_{\neq 0} \times W \rightarrow W \times M^{(3)} \times W$ are

$$\text{init}(f, \ell) = (0, \ell)$$

$$\text{update}(\ell, (r, c)) = ((r + \Sigma(c - \ell), \ell), (r, \Sigma(c - \ell), \Sigma(\ell)), (r + \Sigma(\ell), c - \ell)),$$

where for $a, b \in \mathcal{B}Y$, the bag $a - b$ is defined by $(a - b)(y) = \max(0, a(y) - b(y))$; $\Sigma: \mathcal{B}M \rightarrow M$ is the canonical summation map defined by $\Sigma(b) = \sum_{m \in M} b(m) \cdot m$; and we denote elements of $M^{(3)}$ as triples over M .

We implement the bags $\mathcal{B}(M_{\neq 0})$ used in $W = M \times \mathcal{B}(M_{\neq 0})$ as balanced search trees with keys $M_{\neq 0}$ and values \mathbb{N} , following Adams [1]. In addition, we store in every node the value $\Sigma(b)$, where b is the bag encoded by the subtree rooted at that node. Hence, for every bag b , the value $\Sigma(b)$ is immediately available at the root node of the search tree encoding b . It is not difficult to see that maintaining those values in the nodes only adds a constant overhead into the operations of our data structure for bags and that the size of the search trees is bounded by $\min(|M|, m)$. Thus, we obtain:

Proposition 5.4. *The above function $\text{update}(\ell, (r, c))$ can be computed in $\mathcal{O}(|\ell| \cdot \log \min(|M|, m))$, where m is the number of all edges of the input coalgebra.*

Corollary 5.5. *On a weighted tree automaton with n states, k transitions, and rank r over an (unrestricted) monoid M , our algorithm runs in time $\mathcal{O}((rk + n) \cdot \log(k + n) \cdot (\log k + r))$, respectively $\mathcal{O}((rk + n) \cdot \log(k + n) \cdot r)$ if M is finite.*

More precisely, the analysis using Theorem 3.4 shows that rk can be replaced with the number m of edges of the input coalgebra. Assuming $m \geq n$ we thus obtain the bound given in Table 1. In addition to guaranteeing a good theoretical complexity, our tool immediately yields an efficient implementation. For the case of non-cancellative monoids, this is, to the best of our knowledge, the only available implementation of partition refinement for weighted tree automata.

5.3 Evaluation and Benchmarking

We report on a number of benchmarks that illustrate the practical scalability of our algorithm instantiated for weighted tree automata. Previous studies on the practical performance of partition refinement on large labelled transition systems [34,33] show that memory rather than run time seems to be the limiting factor. Since labelled transition systems are a special case of weighted tree automata, we expect to see similar phenomena. Hence, we evaluate the maximal automata sizes that can be processed on a typical current computer setup: We randomly generate weighted tree automata for various signatures and monoids, looking for the maximal size of WTAs that can be handled with 16 GB of RAM, and we measure the respective run times of our tool, compiled with GHC version 8.4.4 on a Linux system and executed on an Intel® Core™ i5-6500 processor with 3.20GHz clock rate. We fix $|\Sigma| = 4$ and evaluate all combinations of rank r and weight monoid M for r ranging over $\{1, \dots, 5\}$ and M over $2 = (2, \vee, 0)$, $\mathbb{N} = (\mathbb{N}, \max, 0)$ (the additive monoid of the tropical semiring), and $2^{64} = (2, \vee, 0)^{64} \cong (\mathcal{P}_\omega(64), \cup, \emptyset)$. We write n for the number of states, k for the number of transitions, and m for the number of edges in the graphical presentation; in fact, we generate only transitions of the respective maximal rank r , so $m = k(r + 1)$. Table 2 lists the maximal values of n that fit into the mentioned 16 GB of RAM when $k = 50 \cdot n$, and associated run times. For $M = (2, \vee, 0)$, the optimized refinement interface for \mathcal{P}_ω needs less memory, allowing for higher values of n , an effect that decreases with increasing rank r . We restrict to generating at most 50 different elements of M in each automaton, to avoid situations where all states are immediately distinguished in

Table 2: Processing times (in seconds) t_p for parsing and t_a for partition refinement on maximal weighted tree automata with n states and $50 \cdot n$ random transitions fitting into 16 GB of memory. File sizes range from 117 MB to 141 MB, and numbers m of edges from 11 million to 17 million.

$\Sigma X =$	$4 \times X$			$4 \times X^2$			$4 \times X^3$			$4 \times X^4$			$4 \times X^5$		
M	n	t_p	t_a	n	t_p	t_a	n	t_p	t_a	n	t_p	t_a	n	t_p	t_a
2	132177	53	188	98670	46	243	85016	47	187	59596	41	146	49375	38	114
\mathbb{N}	113957	61	141	92434	55	175	69623	49	152	57319	47	140	48962	45	112
2^{64}	114888	58	100	95287	54	138	70660	49	107	62665	48	92	49926	44	72

the first refinement step. In addition, the parameters are chosen so that with high likelihood, the final partition distinguishes all states, so the examples illustrate the worst case. The first refinement step produces in the order of $|\Sigma| \cdot \min(50, |M|)^r$ subblocks (cf. Section 3.4), implying earlier termination for high values of $|M|$ and r and explaining the slightly longer run time for $M = (2, \vee, 0)$ on small r . We note in summary that WTAs with well over 10 million edges are processed in less than five minutes, and in fact the run time of minimization is of the same order of magnitude as that of input parsing. Additional evaluations on DFAs, Segala Systems, and benchmarks for the Prism model checker [28], as well as a comparison with existing specific tools by Antti Valmari [35] and from the mCRL2 toolset [10] are in the full version of this paper [12].

6 Conclusion and Future Work

We have instantiated a generic efficient partition refinement algorithm that we introduced in recent work [14,37] to weighted (tree) automata, and we have refined the generic complexity analysis of the algorithm to cover this case. Moreover, we have described an implementation of the generic algorithm in the form of the tool CoPaR, which supports the modular combination of basic system types without requiring any additional implementation effort, and allows for easy incorporation of new basic system types by implementing a generic refinement interface.

In future work, we will further broaden the range of system types that our algorithm and tool can accommodate, and provide support for base categories beyond sets, e.g. nominal sets, which underlie nominal automata [8,31].

Concerning genericity there is an orthogonal approach by Ranzato and Taparo [29] that is generic over *notions of process equivalence* but fixes the system type to standard labelled transition systems; see also [18]. Similarly, Blom and Orzan [6,7] present *signature refinement*, which covers, e.g. strong and branching bisimulation as well as Markov chain lumping, but requires adapting the algorithm for each instance. These algorithms have also been improved using symbolic techniques (e.g. [36]). Moreover, many of the mentioned approaches and others [4,6,7,16,36] focus on parallelization. We will explore in future work whether symbolic and distributed methods can be lifted to coalgebraic generality. A further important aim is genericity also along the axis of process equivalences.

References

1. S. Adams. Efficient sets - A balancing act. *J. Funct. Program.*, 3(4):553–561, 1993.
2. C. Baier, B. Engelen, and M. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60:187–231, 2000.
3. F. Bartels, A. Sokolova, and E. de Vink. A hierarchy of probabilistic system types. In *Coagebraic Methods in Computer Science, CMCS 2003*, vol. 82 of *ENTCS*, pp. 57 – 75. Elsevier, 2003.
4. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A modular tool for on-the-fly equivalence checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005*, vol. 3440 of *LNCS*, pp. 581–585. Springer, 2005.
5. C. Berkholz, P. S. Bonsma, and M. Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017.
6. S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In *Parallel and Distributed Model Checking, PDMC 2003*, vol. 89 of *ENTCS*, pp. 99–113. Elsevier, 2003.
7. S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *J. Softw. Tools Technol. Transfer*, 7(1):74–86, 2005.
8. M. Bojańczyk, B. Klin, and S. Lasota. Automata theory in nominal sets. *Log. Methods Comput. Sci.*, 10(3), 2014.
9. P. Buchholz. Bisimulation relations for weighted automata. *Theor. Comput. Sci.*, 393:109–123, 2008.
10. O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019*, pp. 21–39, 2019.
11. H.-P. Deifel. Implementation and evaluation of efficient partition refinement algorithms. Master’s thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2019. <https://hpdeifel.de/master-thesis-deifel.pdf>.
12. H.-P. Deifel, S. Milius, L. Schröder, and T. Wißmann. Generic partition refinement and weighted tree automata, 2019. <https://arxiv.org/abs/1811.08850>.
13. S. Derisavi, H. Hermanns, and W. Sanders. Optimal state-space lumping in Markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.
14. U. Dorsch, S. Milius, L. Schröder, and T. Wißmann. Efficient Coalgebraic Partition Refinement. In *Concurrency Theory, CONCUR 2017*, vol. 85 of *LIPIcs*, pp. 32:1–32:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
15. A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.
16. H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using CADP. In *Formal Methods Europe, FME 2002*, vol. 2391 of *LNCS*, pp. 410–429. Springer, 2002.
17. D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.
18. J. Groote, D. Jansen, J. Keiren, and A. Wijs. An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Log.*, 18(2):13:1–13:34, 2017.
19. J. Groote, J. Verduzco, and E. de Vink. An efficient algorithm to determine probabilistic bisimulation. *Algorithms*, 11(9):131, 2018.

20. J. Högberg, A. Maletti, and J. May. Bisimulation minimisation for weighted tree automata. In *Developments in Language Theory, DLT 2007*, vol. 4588 of *LNCs*, pp. 229–241. Springer, 2007.
21. J. Högberg, A. Maletti, and J. May. Backward and forward bisimulation minimization of tree automata. *Theor. Comput. Sci.*, 410:3539–3552, 2009.
22. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pp. 189–196. Academic Press, 1971.
23. D. Huynh and L. Tian. On some equivalence relations for probabilistic processes. *Fund. Inform.*, 17:211–234, 1992.
24. P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.
25. B. Klin and V. Sassone. Structural operational semantics for stochastic and weighted transition systems. *Inf. Comput.*, 227:58–83, 2013.
26. T. Knuutila. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, 250:333–363, 2001.
27. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
28. PRISM. Benchmarks fms and wlan. <http://www.prismmodelchecker.org/casestudies/fms.php> and [wlan.php](http://www.prismmodelchecker.org/casestudies/wlan.php). Accessed: 2018-11-16.
29. F. Ranzato and F. Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206:620–651, 2008.
30. J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.
31. L. Schröder, D. Kozen, S. Milius, and T. Wißmann. Nominal automata with name binding. In *Foundations of Software Science and Computation Structures, FOSSACS 2017*, vol. 10203 of *LNCs*, pp. 124–142, 2017.
32. R. Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995.
33. A. Valmari. Bisimilarity minimization in $\mathcal{O}(m \log n)$ time. In *Applications and Theory of Petri Nets, PETRI NETS 2009*, vol. 5606 of *LNCs*, pp. 123–142. Springer, 2009.
34. A. Valmari. Simple bisimilarity minimization in $\mathcal{O}(m \log n)$ time. *Fund. Inform.*, 105(3):319–339, 2010.
35. A. Valmari and G. Franceschinis. Simple $\mathcal{O}(m \log n)$ time Markov chain lumping. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010*, vol. 6015 of *LNCs*, pp. 38–52. Springer, 2010.
36. T. van Dijk and J. van de Pol. Multi-core symbolic bisimulation minimization. *J. Softw. Tools Technol. Transfer*, 20(2):157–177, 2018.
37. T. Wißmann, U. Dorsch, S. Milius, and L. Schröder. Efficient and modular coalgebraic partition refinement, 2019. <https://arxiv.org/abs/1806.05654>.

Appendix

Table of Contents

Generic Partition Refinement and Weighted Tree Automata	1
<i>Hans-Peter Deifel, Stefan Milius, Lutz Schröder, and Thorsten Wißmann</i>	
1 Introduction	1
2 Theoretical Foundations	3
3 Generic Partition Refinement	4
3.1 Generic System Specification	5
3.2 Refinement Interfaces	6
3.3 Combining Refinement Interfaces	9
3.4 Implementation Details	10
4 Instances	11
5 Weighted Tree Automata	11
5.1 Cancellative Monoids	13
5.2 Non-cancellative Monoids	13
5.3 Evaluation and Benchmarking	14
6 Conclusion and Future Work	15
A Omitted Proofs	18
A.1 Proof of Theorem 3.4	18
A.2 Proof of Proposition 5.2	19
A.3 Details for Section 5.1	19
A.4 Proof of Proposition 5.2	20
A.5 Proof of Theorem 5.3	20
A.6 Details for Section 5.2	21
A.7 Proof of Proposition 5.4	22
A.8 Proof of Corollary 5.5	24
B Further Benchmarks	24
B.1 More details for main benchmark	24
B.2 Maximally dense WTAs	25
B.3 Benchmarks for DFAs and PRISM Models	26

A Omitted Proofs

A.1 Proof of Theorem 3.4

The case where $p(n, m) = 1$ is proved in [37, Theorem 6.22]. We reduce the general case to this one as follows. Observe that the previous complexity analysis counts the number of basic operation performed by the algorithm (e.g. comparisons of values of type W) including those performed by `init` and `update`. In that analysis `init` and `update` were assumed to have run time in $\mathcal{O}(|\ell|)$, and the total number of basic operations of the algorithm is then in $\mathcal{O}((m + n) \cdot \log n)$.

The number of steps taken by our algorithm under the current assumptions (1) and (2) is thus bounded above by the run time of the algorithm under the above assumptions (i.e. assuming $p(n, m) = 1$) but assuming that every basic operation takes $p(n, m)$ steps. Hence, clearly the overall run time lies in $\mathcal{O}((m + n) \cdot \log n \cdot p(n, m))$. \square

A.2 Proof of Proposition 5.2

Given a weighted tree automaton $(X, f, (\mu_k)_{k \in \mathbb{N}})$ as in Definition 5.1 we see that it is, equivalently, a finite coalgebra for the functor $FX = M \times M^{(\Sigma X)}$, where we identify the signature Σ with its corresponding polynomial functor $X \mapsto \coprod_{\sigma/k \in \Sigma} X^k$. Indeed $(\mu_k)_{k \geq 0}$ is equivalently expressed by a map

$$\bar{\mu}: X \rightarrow M^{(\Sigma X)} \quad \text{with} \quad \bar{\mu}(x)(\sigma(x_1, \dots, x_k)) := \mu_k(\sigma)((x_1, \dots, x_k), x). \quad (4)$$

Thus we obtain a coalgebra

$$c: X \rightarrow M \times M^{(\Sigma X)} \quad \text{with} \quad c(x) = (f(x), \bar{\mu}(x)). \quad (5)$$

Note that $\bar{\mu}(x)$ is finitely supported because X and ΣX are finite.

Definition A.1 (Högberg et al. [20, Definition 16]). A *backward bisimulation* on a weighted tree automaton $(X, f, (\mu_k)_{k \in \mathbb{N}})$ is an equivalence relation $R \subseteq X \times X$ such that for every $(p, q) \in R$, $\sigma/k \in \Sigma$, and $L \in \{D_1 \times \dots \times D_k \mid D_1, \dots, D_k \in X/R\}$:

$$\sum_{w \in L} \mu_k(\sigma)(w, p) = \sum_{w \in L} \mu_k(\sigma)(w, q).$$

Remark A.2. Note that $w \in L$ means that $w \in X^k$ such that $e^k(w) = L$, where $e^k: X^k \rightarrow (X/R)^k$ is the k -fold power of the canonical quotient map $e: X \rightarrow X/R$.

A.3 Details for Section 5.1

Every submonoid of a group is cancellative, for example $(\mathbb{N}, +, 0)$ and $(\mathbb{Z}, \cdot, 1)$. Conversely via the *Grothendieck construction*, every cancellative commutative monoid $(M, +, 0)$ can be embedded into the following group:

$$G = (M \times M)/\equiv \quad (a_+, a_-) \equiv (b_+, b_-) \text{ iff } a_+ + b_- = a_- + b_+,$$

where the group structure is given by the usual componentwise addition on the product: $[(a_+, a_-)] + [(b_+, b_-)] = [(a_+ + b_+, a_- + b_-)]$, and $-[(a_+, a_0)] = [(a_-, a_+)]$. The embedding of M is given by $m \mapsto [(m, 0)]$.

The embedding $M \hookrightarrow G$ extends to a monic natural transformation $M^{(-)} \hookrightarrow G^{(-)}$, and therefore, computing behavioural equivalence for $M^{(-)}$ reduces to that of $G^{(-)}$ [37, Proposition 2.13].

A.4 Proof of Proposition 5.2

We need to show that an equivalence relation $R \subseteq X \times X$ is a backward bisimulation iff the canonical quotient map $e: X \rightarrow X/R$ is an $M^{(\Sigma(-))}$ -coalgebra homomorphism (with domain (X, c) as defined in (5)). First, let $x \in X, \sigma/k \in \Sigma$, and $L \in (X/R)^k$ for some equivalence relation R . Then we have the following equalities, where note that $M^{(\Sigma e)}: M^{(\Sigma X)} \rightarrow M^{(\Sigma(X/R))}$ and $\sigma(L) \in \Sigma(X/R)$:

$$\begin{aligned} \sum_{w \in L} \mu_k(\sigma)(w, x) &= \sum_{\substack{w \in X^k \\ e^k(w) = L}} \mu_k(\sigma)(w, x) = \sum_{\substack{\sigma(w) \in \Sigma X \\ \text{ar}(\sigma) = k \\ \Sigma e(\sigma(w)) = \sigma(L)}} \mu_k(\sigma)(w, x) = \sum_{\substack{\sigma(w) \in \Sigma X \\ \Sigma e(\sigma(w)) = \sigma(L)}} \bar{\mu}(x)(\sigma(w)) \\ &= \sum_{\substack{t \in \Sigma X \\ \Sigma e(t) = \sigma(L)}} \bar{\mu}(x)(t) = M^{(\Sigma e)}(\bar{\mu}(x))(\sigma(L)). \end{aligned}$$

Hence, for every equivalence relation $R \subseteq X \times X$ we have the following chain of equivalences:

$$\begin{aligned} &R \text{ is a backward bisimulation} \\ \Leftrightarrow \forall (p, q) \in R, \sigma/k \in \Sigma, L \in (X/R)^k: \sum_{w \in L} \mu_k(\sigma)(w, p) &= \sum_{w \in L} \mu_k(\sigma)(w, q) \\ \Leftrightarrow \forall (p, q) \in R, \sigma/k \in \Sigma, L \in (X/R)^k: \\ &M^{(\Sigma e)}(\bar{\mu}(p))(\sigma(L)) = M^{(\Sigma e)}(\bar{\mu}(q))(\sigma(L)) \\ \Leftrightarrow \forall (p, q) \in R, t \in \Sigma(X/R): M^{(\Sigma e)}(\bar{\mu}(p))(t) &= M^{(\Sigma e)}(\bar{\mu}(q))(t) \\ \Leftrightarrow \forall (p, q) \in R: M^{(\Sigma e)}(\bar{\mu}(p)) &= M^{(\Sigma e)}(\bar{\mu}(q)) \\ \Leftrightarrow \forall (p, q) \in R: (M^{(\Sigma e)} \cdot \bar{\mu})(p) &= (M^{(\Sigma e)} \cdot \bar{\mu})(q). \end{aligned}$$

The last equation holds precisely if e is a coalgebra homomorphism. Indeed, those equations hold precisely if there exists a map $r: X/R \rightarrow M^{(\Sigma(X/R))}$ such that $r(e(x)) = (M^{(\Sigma e)} \cdot \bar{\mu})(x)$, i.e. such that the following square commutes

$$\begin{array}{ccc} X & \xrightarrow{\bar{\mu}} & M^{(\Sigma X)} \\ e \downarrow & & \downarrow M^{(\Sigma e)} \\ X/R & \xrightarrow{\tau} & M^{(\Sigma(X/R))} \end{array} \quad \square$$

A.5 Proof of Theorem 5.3

The functor $FX = M \times M^{(\Sigma(-))}$ is decomposed into $F''X = M \times M^{(X)} + \Sigma X$ according to Section 3.3. Given a coalgebra $\langle o, w \rangle: X \rightarrow M \times M^{(\Sigma X)}$ we introduce the set of intermediate states Y , one for every outgoing transition from every $x \in X$, i.e.

$$Y = \{(x, s) \mid x \in X \text{ and } s \in \Sigma X \text{ with } w(x, s) \neq 0\}.$$

The given coalgebra structure $\langle o, w \rangle$ yields the two evident maps $\xi_1: X \rightarrow M \times M^{(Y)}$ and $\xi_2: Y \rightarrow \Sigma X$ given by

$$\begin{aligned} \xi_1(x) &= (o(x), t_x), \quad \text{where } t_x(x', s) = \begin{cases} w(x, s) & \text{if } x' = x \\ 0 & \text{else,} \end{cases} \\ \xi_2(x, s) &= s. \end{aligned}$$

Partition refinement is now performed on the following T'' -coalgebra:

$$X + Y \xrightarrow{\xi_1 + \xi_2} M \times M^{(Y)} + \Sigma X \xrightarrow{\text{id}_M \times M^{(\text{inr})} + \Sigma \text{inl}} M \times M^{(X+Y)} + \Sigma(X + Y),$$

where $\text{inl}: X \rightarrow X + Y$ and $\text{inr}: Y \rightarrow X + Y$ denote the evident injections into the disjoint union.

Clearly, we have $n' := |X + Y| = n + k$ and the number of edges of the above coalgebra is at most $m' := (r + 1) \cdot k$. Since the refinement interface for T'' is a combination of those of $M \times M^{(-)}$ and Σ its factor $p(n', m')$ is the maximum of the factors $p_M(n', m')$ and $p_\Sigma(n', m')$ of those two refinement interfaces, respectively, since we either call the former or the latter one for a state in $X + Y$; in symbols:

$$p(n', m') = \max(p_M(n', m'), p_\Sigma(n', m')) = \max(1, r),$$

where we use that M is cancellative and $p_\Sigma(n', m') = r$ (see the end of Section 3.2).

By Theorem 3.4 we thus obtain an overall time complexity of

$$\mathcal{O}((m' + n') \cdot \log n' \cdot r) = \mathcal{O}((r \cdot k + n) \cdot \log(n + k) \cdot r). \quad \square$$

A.6 Details for Section 5.2

Proposition A.3. *The refinement interface for $M^{(-)}$ is correct, for every monoid $(M, +, 0)$.*

Proof. We prove that the refinement interface for $M^{(-)}$ defined in Section 5.2 satisfies the two axioms in Definition 3.2. Let $t \in FX$. For the first axiom we compute as follows:

$$\begin{aligned} w(X)(t) &= (\sum_{x \in X \setminus X} t(x), (m \mapsto |\{x \in X \mid t(x) = m\}|)) \\ &= (0, \mathcal{B}_\omega \pi_1(\{(m, x) \in M_{\neq 0} \times X \mid t(x) = m\})) \\ &= (0, \mathcal{B}_\omega \pi_1(\{(t(x), x) \mid x \in X, t(x) \neq 0\})) \\ &= (0, \mathcal{B}_\omega \pi_1(b(t))) \\ &= \text{init}(F!(t), \mathcal{B}_\omega \pi_1(b(t))). \end{aligned}$$

Let us now verify the second axiom concerning **update**. In order to simplify the notation, we define the *restriction* of bags of edges by

$$(t \downarrow B) \in \mathcal{B}(M_{\neq 0}), \quad (t \downarrow B)(m) = |\{x \in B \mid t(x) = m\}| \quad \text{for } B \subseteq X, t \in M^{(X)}$$

and define their sum by

$$\sum_B t := \Sigma(t \downarrow B) = \sum_{x \in B} t(x) \quad \text{for } B \subseteq X.$$

So we have

$$\begin{aligned} w(B)(t) &= (\sum_{x \in X \setminus B} t(x), m \mapsto |\{x \in B \mid t(x) = m\}|) \\ &= (\sum_{X \setminus B} t, (t \downarrow B)) \end{aligned}$$

With the subtraction of bags defined by $(a - b)(y) = \max(0, a(y) - b(y))$ for $a, b \in \mathcal{B}_\omega Y$, we have

$$(t \downarrow B) - (t \downarrow S) = (t \downarrow (B \setminus S)) \quad \text{for } S \subseteq B \subseteq X.$$

Then for $S \subseteq B \subseteq X$ we can compute as follows:

$$\begin{aligned} &\langle w(S), F\chi_S^B, w(B \setminus S) \rangle(t) \\ &= (w(S)(t), F\chi_S^B(t), w(B \setminus S)(t)) \\ &= ((\sum_{X \setminus S} t, (t \downarrow S)), (\sum_{X \setminus B} t, \sum_{B \setminus S} t, \sum_S t), (\sum_{X \setminus (B \setminus S)} t, (t \downarrow (B \setminus S)))) \\ &= ((\sum_{X \setminus B} t + \underbrace{\sum_{B \setminus S} t}_{\sum_{B \setminus S} t}, (t \downarrow S)), (\sum_{X \setminus B} t, \Sigma(t \downarrow (B \setminus S)), \Sigma(t \downarrow S)), \\ &\quad (\sum_{X \setminus B} t + \sum_S t, ((t \downarrow B) - (t \downarrow S)))) \\ &= ((\sum_{X \setminus B} t + \Sigma((t \downarrow B) - (t \downarrow S)), (t \downarrow S)), \\ &\quad (\sum_{X \setminus B} t, \Sigma((t \downarrow B) - (t \downarrow S)), \Sigma(t \downarrow S)), \\ &\quad (\sum_{X \setminus B} t + \Sigma(t \downarrow S), ((t \downarrow B) - (t \downarrow S)))) \\ &\stackrel{(*)}{=} \text{update}((t \downarrow S), (\sum_{X \setminus B} t, (t \downarrow B))) \\ &= \text{update}((m \in M_{\neq 0} \mapsto |\{x \in S \mid t(x) = m\}|), w(B)(t)) \\ &= \text{update}(\{m \in M_{\neq 0} \mid (m, x) \in \flat(t), x \in S\}, w(B)(t)) \\ &= \text{update}(\{a \in A \mid (a, x) \in \flat(t), x \in S\}, w(B)(t)), \end{aligned}$$

where the step labelled $(*)$ uses the definition of **update**:

$$\text{update}(\ell, (r, c)) = ((r + \Sigma(c - \ell), \ell), (r, \Sigma(c - \ell), \Sigma(\ell)), (r + \Sigma(\ell), c - \ell)). \quad \square$$

A.7 Proof of Proposition 5.4

We implement the bags $\mathcal{B}(M_{\neq 0})$ used in $W = M \times \mathcal{B}(M_{\neq 0})$ as balanced search trees with keys $M_{\neq 0}$ and values \mathbb{N} following Adams [1]. In addition, we store in

every node x the value $\Sigma(b)$, where b is the bag encoded by the subtree rooted at x ; in the following we simply write $\Sigma(x)$. Hence, for every bag b , the value $\Sigma(b)$ is immediately available at the root node of the search tree for b .

Note that our search trees cannot have more nodes than the size $|M|$ of their index set, and the number of nodes is also not greater than the number m of all edges. Hence, their size is bounded by $\min(|M|, m)$.

Recall e.g. [?, Section 14], that the key operations **insert**, **delete** and **search** have logarithmic time complexity in the size of a given balanced binary search tree.

We still need to argue that maintaining the values $\Sigma(x)$ in the nodes does not increase this complexity. This is obvious for the **search** operation as it does not change its argument search tree. For **insert** and **delete** recall from *op. cit.* that those operation essentially trace down one path starting at the root to a node (at most a leaf) of the given search tree. In addition, after insertion or deletion of a node, the balanced structure of the search tree has to be fixed. This is done by tracing back the same path to the root and (possibly) performing *rotations* on the nodes occurring on that path. Rotations are local operations changing the structure of a search tree but preserving the inorder key ordering of subtrees (see Fig. 4).

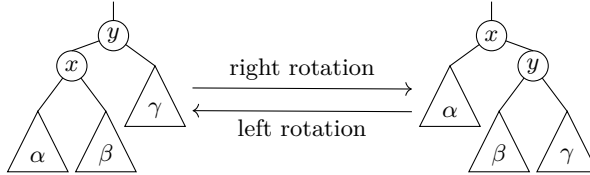


Fig. 4: Rotation operations in binary search trees.

Clearly, in order to maintain the correct summation values in a search tree under a rotation, we only need to adjust those values in the nodes x and y . For example, for right rotation the new values are:

$$\Sigma(y) = \Sigma(\beta) + \Sigma(\gamma); \quad \Sigma(x) = \Sigma(\alpha) + \Sigma(y).$$

In addition, when inserting or deleting a node x we must recompute the $\Sigma(y)$ of all nodes y along the path from the root to x when we trace that path back to the root. This can clearly be performed in constant time for each node y since $\Sigma(y)$ is the sum (in M) of those values stored at the child nodes of y .

In summary, we see that maintaining the desired summation values only requires an additional constant overhead in the backtracing step. Consequently, the operations of our balanced binary search trees have a run time in $\mathcal{O}(\log \min(|M|, m))$, so that we obtain the desired overall time complexity of update. \square

A.8 Proof of Corollary 5.5

We proceed precisely as in the proof of Theorem 5.3 using that $p_M(n', m') = \log \min(|M|, m')$ (in lieu of $p_M(n', m') = 1$). We proceed by case distinction. If M is a finite monoid, then $p_M(n', m')$ is in $\mathcal{O}(1)$. Hence, we obtain the same overall complexity as in Theorem 5.3 as desired.

If M is infinite, we have that $p_M(n', m')$ is in $\mathcal{O}(\log m')$. Thus, by Theorem 3.4 we obtain:

$$\begin{aligned} \mathcal{O}((m' + n') \cdot \log n' \cdot \max(\log m', r)) &= \mathcal{O}((rk + n) \cdot \log(n + k) \cdot (\log(rk) + r)) \\ &= \mathcal{O}((rk + n) \cdot \log(n + k) \cdot (\log k + r)), \end{aligned}$$

where we use that $\log(rk) = \log r + \log k$ in the second equation and, in the first one, that \max can be replaced by $+$ in \mathcal{O} -notation. \square

Remark A.4. Here we provide a more refined comparison of the complexity of the Högberg et al.'s algorithm with the instances of our algorithm for weighted tree automata.

(1) For arbitrary (non-cancellative) monoids they provide a complexity of $\mathcal{O}(r \cdot k \cdot n)$ [20, Theorem 27]. Again, the number m of edges of the input coalgebra satisfies $m \leq rk$. Moreover, for a fixed input signature, m and k are asymptotically equivalent. Assuming further that $m \geq n$, which means that there are no isolated states, we see that the bound in Table 1 indeed improves the complexity of $\mathcal{O}(m \cdot n)$ Högberg et al.'s algorithm. To see this note first that the number m of edges is in $\mathcal{O}(n^{r+1})$ so that we obtain

$$\mathcal{O}(m \cdot \log(m)^2) \subsetneq \mathcal{O}(m \cdot {}^{r+1}\sqrt{m}) \subseteq \mathcal{O}(m \cdot {}^{r+1}\sqrt{n^{r+1}}) = \mathcal{O}(m \cdot n).$$

In the first step, it is used that for every $d \geq 1$ and $0 < c < 1$ we have $\mathcal{O}(\log^d(m)) \subsetneq \mathcal{O}(m^c)$.

(2) For cancellative monoids, the time bound given in *op. cit.* is $\mathcal{O}(r^2 \cdot t \cdot \log n)$ [20, Theorem 29]. Under our standard assumption that $rt \geq n$ our complexity from Theorem 5.3 lies in $\mathcal{O}(r^2 \cdot t \cdot \log(t + n))$, which is only very slightly worse.

B Further Benchmarks

B.1 More details for main benchmark

The benchmarks are designed in such a way that the run time is maximal. This means that:

- (1) In the first partition that is computed, all states are identified.
- (2) In the final partition, all states are distinguished.

We minimize randomly generated coalgebras for

$$FX = M \times M^{(\Sigma X)}$$

with $\Sigma X = 4 \times X^r$, for $r \in \{1, \dots, 5\}$. When generating a coalgebra with n states, we randomly create 50 outgoing transitions per state, leading to $50 \cdot n$

transitions in total. As described in Section 3.3, we need to introduce one intermediate state per transition, leading to an actual number $n' = 51 \cdot n$ of states. Every transition of rank r has one incoming edge and r outgoing edges, hence $m = (r + 1) \cdot k = 50 \cdot (r + 1) \cdot n$. The performance evaluation is listed in Table 3.

Table 3: Extended version of Table 2: We take n states and 50 transitions per state, leading to n' states and m edges in total. Parsing takes t_p seconds; the first partition has P_1 blocks and its computation takes t_i seconds; the final partition has P_f blocks and its computation takes additional t_r seconds.

r	Monoid M	n	n'	m	Size	P_1	P_f	t_p	t_i	t_r
1	$(2, \vee, 0)$	132177	6741027	13217700	117 MB	6	132177	53	32	156
1	$(\mathbb{N}, \max, 0)$	114888	5859288	11488800	122 MB	416	114888	58	34	66
1	$(\mathcal{P}_\omega(64), \cup, \emptyset)$	113957	5811807	11395700	131 MB	54	113957	61	32	109
2	$(2, \vee, 0)$	98670	5032170	14800500	123 MB	6	98670	46	31	212
2	$(\mathbb{N}, \max, 0)$	95287	4859637	14293050	136 MB	404	95287	54	30	108
2	$(\mathcal{P}_\omega(64), \cup, \emptyset)$	92434	4714134	13865100	141 MB	54	92434	55	31	144
3	$(2, \vee, 0)$	85016	4335816	17003200	138 MB	6	85016	47	20	167
3	$(\mathbb{N}, \max, 0)$	70660	3603660	14132000	127 MB	397	70660	49	25	82
3	$(\mathcal{P}_\omega(64), \cup, \emptyset)$	69623	3550773	13924600	132 MB	54	69623	49	25	127
4	$(2, \vee, 0)$	59596	3039396	14899000	119 MB	6	59596	41	25	121
4	$(\mathbb{N}, \max, 0)$	62665	3195915	15666250	136 MB	397	62665	48	26	66
4	$(\mathcal{P}_\omega(64), \cup, \emptyset)$	57319	2923269	14329750	130 MB	54	57319	47	25	115
5	$(2, \vee, 0)$	49375	2518125	14812500	116 MB	6	49375	38	24	90
5	$(\mathbb{N}, \max, 0)$	49926	2546226	14977800	127 MB	376	49926	44	20	52
5	$(\mathcal{P}_\omega(64), \cup, \emptyset)$	48962	2497062	14688600	129 MB	54	48962	45	20	92

B.2 Maximally dense WTAs

As another benchmark, we generated very dense WTAs, showing that also in this degenerated case, $m = 10$ million edges (in the sense of graphical representation) can be handled with 16 GB of RAM within a few minutes, as described in the following.

For a fixed number $n = |C|$ of states, monoid M and signature Σ , we uniformly generated $c(x) \in M^{(\Sigma^C)}$ for every state $x \in C$:¹ for every $y \in \Sigma^C$, we put $c(x)(y) = 0$ with probability 0.7 and otherwise choose $c(x)(y) \in M$ randomly. We then counted the number m of edges, checked whether the minimization stayed within the RAM limit, and if so, recorded the run time. Table 4 lists the sizes of the largest WTAs that can be handled. For each configuration we

¹ We overload notation and write Σ both for a signature and its associated polynomial functors on sets.

Table 4: Dense WTAs: For $(2, \vee, 0)$, the final partition consists of very few blocks (less than 20, depending on the polynomial). For the other cases, all states are distinguished after the initialization phase.

M	ΣX	p	n	$n'/10^6$	$m/10^6$	MiB	t_p	t_i	t_r	t
$(2, \vee, 0)$	$8 \times X$	0.7	1478	5.24	10.48	82	35	28	37	115
	$1 + 4 \times X^2$	0.7	151	4.13	12.39	83	33	21	34	102
	$4 + 3 \times X + 2 \times X^2$	0.7	190	4.15	12.41	83	33	19	73	143
	$3 \times X^5$	0.7	11	1.59	9.57	47	20	11	11	45
$(\mathbb{Z}, \max, 0)$	$8 \times X$	0.7	1450	5.05	10.09	182	43	27	40	127
	$1 + 4 \times X^2$	0.7	150	4.05	12.15	162	41	22	40	121
	$4 + 3 \times X + 2 \times X^2$	0.7	188	4.02	12.02	162	40	15	39	111
	$3 \times X^5$	0.7	11	1.59	9.57	79	23	11	21	60
$(\mathcal{P}_\omega(64), \cup, \emptyset)$	$8 \times X$	0.7	1408	4.76	9.52	164	49	25	37	121
	$1 + 4 \times X^2$	0.7	148	3.89	11.67	151	44	21	37	118
	$4 + 3 \times X + 2 \times X^2$	0.7	186	3.89	11.64	152	44	18	38	115
	$3 \times X^5$	0.7	11	1.60	9.57	77	25	11	21	61

generated five automata, and averaged their values for Table 4; each of the maximal deviations is insignificantly small. Note that with higher rank r , even a small number n of states leads to millions of edges.

We see that the upper limit for the number of edges is roughly 10 million, independent of the choices of the monoid or the signature. In more detail, only the choice of the signature contributes to n and nuances of m . This is hardly surprising because the signature determines the branching degree of an automaton and also determines that there are $0.3 \cdot |\Sigma C|$ many expected transitions. With roughly 10 million edges, the file sizes vary slightly, depending on the representation of the coefficients from the different monoids (coefficients vanish for $(2, \vee, 0)$).

In Table 4 we show the benchmarks of very dense WTAs.

B.3 Benchmarks for DFAs and PRISM Models

Another benchmark suite consists of randomly generated DFAs for a fixed size n and alphabet A , which were generated by uniformly choosing a successor for each state and letter of the alphabet and for each state whether it is final or not. All of the resulting automata were already minimal, which means that our algorithm has to refine the initial partition $X/Q = \{X\}$ until all blocks are singletons, which requires a maximal number of iterations. The results in Table 5a and 5b show that the implementation can handle coalgebras with 10 million edges, and that parsing the input takes more time than the actual refinement for these particular systems.

In addition, we translated the benchmark suite of the model checker PRISM [?] to coalgebras for the functors $\mathbb{N} \times \mathbb{R}^{(X)}$ for continuous time markov chains (CTMC) and $\mathbb{N} \times \mathcal{P}_\omega(\mathbb{N} \times (\mathcal{D}_\omega X))$ for Markov decision processes (MDP). In contrast to

Input	Time (in s) to				Input	Time (in s) to			
n	Parse	Init	Refine		n	Parse	Init	Refine	
1000	2.40	0.76	0.36		600	44.75	1.82	2.88	
2000	4.96	1.58	0.74		700	50.93	4.29	3.18	
3000	7.39	2.11	1.40		800	60.78	2.54	4.16	
4000	10.20	3.20	1.67		900	68.34	2.76	4.60	
5000	13.06	4.05	2.10		1000	75.79	3.05	5.21	

(a) DFAs for $|A| = 10^3$ (b) DFAs for $|A| = 10^4$

Table 5: Performance on randomly generated DFAs

PRISM Model	Input		Time (s) to			Time (s) of		
	States	Edges	Parse	Init	Refine	Valmari	mCRL2	
fms (n=4)	35910	237120	0.48	0.12	0.16	0.21	–	
fms (n=5)	152712	1111482	2.46	0.68	1.10	1.21	–	
fms (n=6)	537768	4205670	9.94	2.91	5.56	5.84	–	
wlan2_collide (COL=2, TRANS_TIME_MAX=10)	65718	94452	0.51	0.29	0.59	0.14	0.42	
wlan0_time_bounded (TRANS_TIME_MAX=10, DEADLINE=100)	582327	771088	5.26	3.07	5.52	0.92	3.18	
wlan1_time_bounded (TRANS_TIME_MAX=10, DEADLINE=100)	1408676	1963522	13.42	6.17	16.13	2.52	8.58	

Table 6: Performance on PRISM benchmarks

DFAs, these functors are compositions of several basic functors and thus require the construction described in Section 3.3. Two of those benchmarks [28] are shown in Table 6 with different parameters, resulting in three differently sized coalgebras each.² The *fms* family of systems model a flexible manufacturing system as CTMC, while the *wlan* benchmarks model various aspects of the IEEE 802.11 Wireless LAN protocol as MDP.

Table 6 also includes the total run time of two additional partition refinement tools: A C++ implementation³ of the algorithm described in [35] by Antti Valmari which can minimize MDPs as well as CTMCs, and the tool `ltspbisim` from the mCRL2 toolset [10] version 201808.0 which implements a recently discovered refinement algorithm for MDPs [19] (but does not support CTMCs, hence there is no data in the first three lines).

The results in Table 6 show that refinement for the *fms* benchmarks is faster than for the respective *wlan* ones, even though the first group has more edges. This is due to (a) the fact that the functor for MDPs is more complex and thus

² The full set of benchmarks and their results can be found at <https://git8.cs.fau.de/software/copar-benchmarks>

³ Available at <https://git8.cs.fau.de/hpd/mdpmin-valmari>

introduces more indirection into our algorithms, as explained in Section 3.3, and (b) that our optimization for one-element blocks fires much more often for *fms*.

It is also apparent that CoPaR is slower than both of the other tools in our comparison, by a factor of up to 15 for the presented examples. This performance difference can be partly attributed to the fact that our implementation is written in Haskell and the other tools are written in C++. In addition, CoPaR's genericity and modularity take a toll on performance.