Coalgebra Encoding for Efficient Minimization

Hans-Peter Deifel
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Stefan Milius
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Thorsten Wißmann
Radboud University Nijmegen, The Netherlands

Abstract

Recently, we have developed an efficient generic partition refinement algorithm, which computes behavioural equivalence on a state-based system given as an encoded coalgebra, and implemented it in the tool CoPaR. Here we extend this to a fully fledged minimization algorithm and tool by integrating two new aspects: (1) the computation of the transition structure on the minimized state set, and (2) the computation of the reachable part of the given system. In our generic coalgebraic setting these two aspects turn out to be surprisingly non-trivial requiring us to extend the previous theory. In particular, we identify a sufficient condition on encodings of coalgebras, and we show how to augment the existing interface, which encapsulates computations that are specific for the coalgebraic type functor, to make the above extensions possible. Both extensions have linear run time.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Logic and verification

Keywords and phrases Coalgebra, Partition refinement, Transition systems, Minimization

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.28

Related Version *Full Version*: https://arxiv.org/abs/2102.12842

Funding *Hans-Peter Deifel*: Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computing” (393541319/GRK2475/1-2019).
Stefan Milius: Supported by Deutsche Forschungsgemeinschaft (DFG) under project MI 717/5-2.
Thorsten Wißmann: Supported by NWO TOP project 612.001.852.

Acknowledgements We would like to thank the anonymous referees for their comments, which helped us to improve the presentation.

1 Introduction

The task of minimizing a given finite state-based system has arisen in different contexts throughout computer science and for various types of systems, such as standard deterministic automata, tree automata, transition systems, Markov chains, probabilistic or other weighted systems. In addition to the obvious goal of reducing the mere memory consumption of the state space, minimization often appears as a subtask of a more complex problem. For instance, probabilistic model checkers benefit from minimizing the input system before performing the actual model checking algorithm, as e.g. demonstrated in benchmarking by Katoen et al. [32].

Another example is the graph isomorphism problem. A considerable portion of input instances can already be decided correctly by performing a step called colour refinement [9], which amounts to the minimization of a weighted transition system wrt. weighted bisimilarity.

6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).
Editor: Naoki Kobayashi; Article No. 28; pp. 28:1–28:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Minimization algorithms typically perform two steps: first a reachable subset of the state set of the given system is computed by a standard graph search, and second, in the resulting reachable system all behaviourally equivalent states are identified. For the latter step one uses *partition refinement* or *lumping* algorithms that start by identifying all states and then iteratively refine the resulting partition of the state set by looking one step into the transition structure of the given system. There has been a lot of research on efficient partition refinement procedures, and the most efficient algorithms for various concrete system types have a run time in $\mathcal{O}(m \log n)$, for a system with n states and m transitions, e.g. Hopcroft's algorithm for deterministic automata [30] and the algorithm by Paige and Tarjan [36] for transition systems, even if the number of action labels is not fixed [43]. Partition refinement of probabilistic systems also underwent a dynamic development [18, 52], and the best algorithms for Markov chain lumping now match the complexity of the relational Paige-Tarjan algorithm [22, 31, 44]. For the minimization of more complex system types such as Segala systems [6, 26] (combining probabilities and non-determinism) or weighted tree automata [29], partition refinement algorithms with a similar quasilinear run time have been designed over the years.

Recently, we have developed a generic partition refinement algorithm [23, 48] and implemented it in the tool CoPaR [19, 51]. This generic algorithm computes the partition of the state set modulo behavioural equivalence for a wide variety of stated-based system types, including all the above. This genericity in the system type is achieved by working with *coalgebras* for a functor which encapsulates the specific types of transitions of the input system. More precisely, the algorithm takes as input a syntactic description of a set functor and an *encoding* of a coalgebra for that functor and then computes the simple quotient, i.e. the quotient of the state set modulo behavioural equivalence. The algorithm works correctly for every zippable set functor (Definition 2.8). It matches, and in some cases even improves on, the run-time complexity of the best known partition refinement algorithms for many concrete system types [51, Table 1].

The reasons why this run-time complexity can be stated and proven generically are: first, the encoding allows us to talk about the number of states and, in particular, the number of transitions of an input coalgebra. But more importantly, every iterative step of partition refinement requires only very few system-type specific computations. These computations are encapsulated in the *refinement interface* [48], which is then used by the generic algorithm.

An important feature of our coalgebraic algorithm is its modularity: in the tool the user can freely combine functors with already implemented refinement interfaces by products, coproducts and functor composition. A refinement interface for the combined functor is then automatically derived. In this way more structured systems types such as (simple and general) Segala systems and weighted tree automata can be handled.

In the present paper, we extend our algorithm to a fully fledged minimizer. In previous work [3] it has been shown that for set functors preserving intersections, every coalgebra equipped with a point, modelling initial states, has a minimization called the *well-pointed modification*. Well-pointedness means that the coalgebra does not have any proper quotients (i.e. it is *simple*) nor proper pointed subcoalgebras (i.e. it is *reachable*), in analogy to minimal deterministic automata being reachable and observable (see e.g. [5, p. 256]). The well-pointed modification is obtained by taking the reachable part of the simple quotient of a given pointed coalgebra [3] (and the more usual reversed order, simple quotient of the reachable part, is correct for functors preserving inverse images [50, Sec. 7.2]). Our previous work on coalgebraic minimization algorithms has focused on computing the simple quotient. Here we extend our algorithm by two missing aspects of minimization and provide their correctness proofs: the computation of (1) the transition structure of the minimized system, and (2) the reachable states of an input coalgebra.

One may wonder why (1) is a step worth mentioning at all because for many concrete system types this is trivial, e.g. for deterministic automata where the transitions between equivalence classes are simply defined by choosing representatives and copying their transitions from the input automaton. However, for other system types this step is not that obvious, e.g. for weighted automata where transition weights need to be summed up and transitions might actually disappear in the minimized system because weights cancel out. We found that in the generic coalgebraic setting enabling the computation of the (encoding of) the transition structure of the minimized coalgebra is surprisingly non-trivial, requiring us to extend the theory behind our algorithm.

In order to be able to perform this computation generically we work with *uniform encodings*, which are encodings that satisfy a coherence property (Definition 3.10). We prove that all encodings used in our previous work are uniform, and that the constructions enabling modularity of our algorithm preserve uniformity (Prop. 3.12). We also prove that uniform encodings are subnatural transformations, but the converse does not hold in general. In addition, we introduce the *minimization interface* containing the new function `merge` (to be implemented together with the refinement interface for each new system type) which takes care of transitions that change as a result of minimization. We provide `merge` operations for all functors with explicitly implemented refinement interfaces (Example 4.4), and show that for combined system types minimization interfaces can be automatically derived (Prop. 4.11); similarly as for refinement interfaces. Our main result is that the (encoded) transition structure of the minimized coalgebra can be correctly computed in linear time (Thm. 4.9).

Concerning extension (2), the computation of reachable states, it is well-known that every pointed coalgebra has a reachable part (being the smallest subcoalgebra) [3, 49]. Moreover, for a set functor preserving intersections it coincides with the reachable part of the canonical graph of the coalgebra [3, Lem. 3.16]. Recently, it was shown that the reachable part of a pointed coalgebra can be constructed iteratively [49, Thm. 5.20] and that this corresponds to performing a standard breadth-first search on the canonical graph. The missing ingredient to turn our previous partition refinement algorithm into a minimizer is to relate the canonical graph with the encoding of the input coalgebra. We prove that for a functor with a subnatural encoding, the encoding (considered as a graph) of every coalgebra coincides with its canonical graph (Theorem 5.6).

Putting everything together, we obtain an algorithm that computes the well-pointed modification of a given pointed coalgebra. Both additions can be implemented with linear run time in the size of the input coalgebra and hence do not add to the run-time complexity of the previous partition refinement algorithm. We have provided such an implementation with the new version of our tool CoPaR.

All proofs and additional details can be found in the full version [21].

Reachability in Coalgebraic Minimization. There are several works on coalgebraic minimization, ranging from abstract constructions to concrete and implemented algorithms [1, 34, 35, 48, 51], that compute the simple quotient [27] of a given coalgebra. These are not concerned with reachability since coalgebras are not equipped with initial states in general.

In Brzozowski's automata minimization algorithm [16], reachability is one of the main ingredients. This is due to the duality of reachability and observability described by Arbib and Manes [4], and this duality is used twice in the algorithm. Consequently, reachability also appears as a subtask in the categorical generalizations of Brzozowski's algorithm [10, 14, 15, 35, 38]. These generalizations concern automata processing input words and so do not cover minimization of (weighted) tree automata. Segala systems are not treated either. Due to the dualization, Brzozowki's classical algorithm for deterministic automata has doubly exponential time complexity in the worst case (although it performs well on certain types of non-deterministic automata, compared to determinization followed by minimization [41]).

2 Background

Our algorithmic framework [48] is defined on the level of coalgebras for set functors, following the paradigm of *universal coalgebra* [39]. Coalgebras can model a wide variety of systems.

In the following we recall standard notation for sets and functions as well as basic notions from the theory of coalgebras. We fix a singleton set $1 = \{*\}$; for each set X, we have a unique map $!\colon X \to 1$. We denote the disjoint union (coproduct) of sets A, B by $A + B$ and use $\mathsf{inl}, \mathsf{inr}$ for the canonical injections into the coproduct, as well as $\mathsf{pr}_1, \mathsf{pr}_2$ for the projections out of the product. We use the notation $\langle \cdots \rangle$, respectively $[\cdots]$, for the unique map induced by the universal property of a product, respectively coproduct. We also fix two sets $2 = \{0, 1\}$ and $3 = \{0, 1, 2\}$ and use the former as a set of boolean values with 0 and 1 denoting *false* and *true*, respectively. For each subset S of a set X, the *characteristic function* $\chi_S\colon X \to 2$ assigns 1 to elements of S and 0 to elements of $X \setminus S$. We denote by Set the category of all sets and maps. We shall indicate injective and surjective maps by $\rightarrowtail$ and $\twoheadrightarrow$, respectively.

Recall that an endofunctor $F\colon \mathsf{Set} \to \mathsf{Set}$ assigns to each set X a set FX, and to each map $f\colon X \to Y$ a map $Ff\colon FX \to FY$, preserving identities and composition, that is we have $F\,\mathsf{id}_X = \mathsf{id}_{FX}$ and $F(g \cdot f) = Fg \cdot Ff$. We denote the composition of maps by $\cdot$ written infix, as usual. An *F-coalgebra* is a pair (X, c) that consists of a set X of *states* and a map $c\colon X \to FX$ called *(transition) structure*. A *morphism* $h\colon (X, c) \to (Y, d)$ of F-coalgebras is a map $h\colon X \to Y$ preserving the transition structure, i.e. $Fh \cdot c = d \cdot h$. Two states $x, y \in X$ of a coalgebra (X, c) are *behaviourally equivalent* if there exists a coalgebra morphism h with $h(x) = h(y)$.

▶ **Example 2.1.** Coalgebras and the generic notion for behavioural equivalence instantiate to a variety of well-known system types and their equivalences:

1. The *finite powerset functor* $\mathcal{P}_\mathsf{f}$ maps a set to the set of all its finite subsets and functions $f\colon X \to Y$ to $\mathcal{P}_\mathsf{f} f = f[-]\colon \mathcal{P}_\mathsf{f} X \to \mathcal{P}_\mathsf{f} Y$ taking direct images. Its coalgebras are finitely branching (unlabelled) transition systems and coalgebraic behavioural equivalence coincides with Milner and Park's (strong) bisimilarity.
2. Given a commutative monoid $(M, +, 0)$, the *monoid-valued functor* $M^{(-)}$ maps a set X to the set of finitely supported functions from X to M. These are the maps $f\colon X \to M$, such that $f(x) = 0$ for all except finitely many $x \in X$. Given a map $h\colon X \to Y$ and a finitely supported function $f\colon X \to M$, $M^{(h)}(f)\colon M^{(X)} \to M^{(Y)}$ is defined as $M^{(h)}(f)(y) = \sum_{x \in X, h(x) = y} f(x)$. Coalgebras for $M^{(-)}$ correspond to finitely branching weighted transition systems with weights from M. If a coalgebra morphism $h\colon (X, c) \to (Y, d)$ merges two states s_1, s_2, then for all transitions $x \xrightarrow{m_1} s_1$, $x \xrightarrow{m_2} s_2$ in (X, c) there must be a transition $h(x) \xrightarrow{m_1 + m_2} h(s_1) = h(s_2)$ in (Y, d) and similarly if more than two states are merged. Coalgebraic behavioural equivalence captures weighted bisimilarity [33, Prop. 2].
   Note that the monoid may have inverses: if $s_2 = -s_1$, then the transitions in the above example cancel each other out, leading to a transition $h(x) \xrightarrow{0} h(s_1)$ with weight 0, which in fact represents the absence of a transition. This happens for example for the monoid $(\mathbb{R}, +, 0)$ of real numbers. A simple minimization algorithm for real weighted transition

(i.e. $\mathbb{R}^{(-)}$-coalgebras) systems is given by Valmari and Franceschinis [44]. These systems subsume Markov chains which are precisely the coalgebras for the finite probability distribution functor $\mathcal{D}$, a subfunctor of $\mathbb{R}^{(-)}$.

3. Given a signature Σ consisting of operation symbols σ, each with a prescribed natural number, its *arity* $\mathsf{ar}(\sigma)$, the *polynomial functor* F_Σ sends each set X to the set of (shallow) terms over X, specifically to the set

$$\{\sigma(x_1, \dots, x_n) \mid \sigma \in \Sigma, \mathsf{ar}(\sigma) = n, (x_1, \dots, x_n) \in X^n\}.$$

The action of F on a function $f\colon X \to Y$ is given by

$$F_\Sigma f(\sigma(x_1, \dots, x_n)) = \sigma(f(x_1), \dots, f(x_n)).$$

A coalgebra structure $c\colon X \to F_\Sigma X$ assigns to a state $x \in X$ an expression $\sigma(x_1, \dots, x_n)$, where σ is an output symbol and x_1 to x_n are the successor states. Two states are behaviourally equivalent if their tree-unfoldings, obtained by repeatedly applying the coalgebra structure c, yields the same (infinite) Σ-tree.

4. For a fixed alphabet A, the functor given by $FX = 2 \times X^A$ is a special case of a polynomial functor over a signature with two symbols of arity $|A|$. An F-coalgebra $c\colon X \to 2 \times X^A$ is the same as a deterministic automaton without an initial state: the structure c assigns a pair (b, t) to each $x \in X$, where the boolean value $b \in 2$ determines its finality, and the function $t\colon A \to X$ assigns to each input letter from $a \in A$ the successor state of x under a. Here, behavioural equivalence coincides with language equivalence in the usual automata theoretic sense.

5. The *bag functor* $\mathcal{B}$ sends a set X to the set of finite multisets over X and functions $f\colon X \to Y$ to $\mathcal{B}f\colon \mathcal{B}X \to \mathcal{B}Y$ given by $\mathcal{B}f(\{\!\!\{ x_1, \dots, x_2 \}\!\!\}) = \{\!\!\{ f(x_1), \dots, f(x_2) \}\!\!\}$, where we use the multiset braces $\{\!\!\{$ and $\}\!\!\}$ to differentiate from standard set notation; in particular $\{\!\!\{ x, x \}\!\!\} \neq \{\!\!\{ x \}\!\!\}$. Coalgebras for $\mathcal{B}$ are finitely branching transition systems where multiple transitions between any two states are allowed, or equivalently, weighted transition systems with positive integers as weights. This follows from the fact that the bag functor is (naturally isomorphic to) the monoid-valued functor for the monoid $(\mathbb{N}, +, 0)$. Hence, behavioural equivalence coincides with weighted bisimilarity again.
Note that every undirected graph may be considered as a $\mathcal{B}$-coalgebra by turning every edge into two directed edges with weight 1. Then two states are behaviourally equivalent iff they are identified by *colour refinement*, also called the 1-dimensional Weisfeiler-Lehman algorithm (see e.g. [9, 17, 46]).

▶ **Example 2.2** (Modularity). New system types can be constructed from existing ones by functor composition. For example, labelled transition systems (LTSs) are coalgebras for the functor $FX = \mathcal{P}_\mathsf{f}(A \times X)$, which is the composite of $\mathcal{P}_\mathsf{f}$ and $A \times -$ for a label alphabet A, and precisely the bisimilar states in an F-coalgebra are behaviourally equivalent. Composing further, *Segala systems* (or *probabilistic LTSs* [26]) are coalgebras for $FX = \mathcal{P}_\mathsf{f}(A \times \mathcal{D}X)$, for which coalgebraic behavioural equivalence instantiates to probabilistic bisimilarity [7]. Another example are *weighted tree automata* [29] with weights in a commutative monoid M and input signature Σ; they are coalgebras for the composed functor $FX = M^{(\Sigma X)}$, for which behavioural equivalence coincides with *backwards bisimilarity* [20].

Simple, Reachable, and Well-Pointed Coalgebras. Minimizing a given pointed coalgebra means to compute its well-pointed modification. We now briefly recall the corresponding coalgebraic concepts. For a more detailed and well-motivated discussion with examples, see e.g. [2, Sec. 9].

First, a *quotient coalgebra* of an F-coalgebra (X, c) is represented by a surjective F-coalgebra morphism, for which we write $q\colon (X, c) \twoheadrightarrow (Y, d)$, and a *subcoalgebra* of (X, c) is represented by an injective F-coalgebra morphism $m\colon (S, s) \rightarrowtail (X, c)$.

A coalgebra (X, c) is called *simple* if it does not have any proper quotient coalgebras [27]. That is, every quotient $q\colon (X, c) \twoheadrightarrow (Y, d)$ is an isomorphism. Equivalently, distinct states $x, y \in X$ are never behaviourally equivalent. Every coalgebra has an (up to isomorphism) unique simple quotient (see e.g. [2, Prop. 9.1.5]).

▶ **Example 2.3.**
1. A deterministic automaton regarded as a coalgebra for $FX = 2 \times X^A$ is simple iff it is observable [5, p. 256], that is, no distinct states accept the same formal language.
2. A finitely branching transition system considered as a $\mathcal{P}_\mathsf{f}$-coalgebra is simple, if it has no pairs of strongly bisimilar but distinct states; in other words if two states x, y are strongly bisimilar, then $x = y$.
3. A similar characterization holds for monoid-valued functors (such as the bag functor) wrt. weighted bisimilarity.

A *pointed* coalgebra is a coalgebra (X, c) equipped with a point $i\colon 1 \to X$, equivalently a distinguished element $i \in X$, modelling an initial state. Morphisms of pointed coalgebras are the point-preserving coalgebra morphisms, i.e. morphisms $h\colon (X, c, i) \to (Y, d, j)$ satisfying $h \cdot i = j$. Quotients and subcoalgebras of pointed coalgebras are defined wrt. these morphisms. A pointed coalgebra (X, c, i) is called *reachable* if it has no proper subcoalgebra, that is, every subcoalgebra $m\colon (S, s, j) \rightarrowtail (X, c, i)$ is an isomorphism. Every pointed coalgebra has a unique reachable subcoalgebra (see e.g. [2, Prop. 9.2.6]). The notion of reachable coalgebras corresponds well with graph theoretic reachability in concrete examples. We elaborate on this a bit more in Section 5.

▶ **Example 2.4.**
1. A deterministic automaton considered as a pointed coalgebra for $FX = 2 \times X^A$ (with the point given by the initial state) is reachable if all of its states are reachable from the initial state.
2. A pointed $\mathcal{P}_\mathsf{f}$-coalgebra is a finitely branching directed graph with a root node. It is reachable precisely when every node is reachable from the root node.
3. Similarly, for monoid-valued functors such as the bag functor, reachability is precisely graph theoretic reachability, where a transition weight of 0 means "no edge".

Finally, a pointed coalgebra (X, c, i) is *well-pointed* if it is reachable and simple. Every pointed coalgebra has a *well-pointed modification*, which is obtained by taking the reachable part of its simple quotient (see [2, Not. 9.3.4]).

▶ Remark 2.5. For a functor preserving inverse images, one may reverse the two constructions: the well-pointed modification is the simple quotient of the reachable part of a given pointed coalgebra [50, Sec. 7.2]. This is the usual order in which minimization of systems is performed algorithmically. However, for a functor that does not preserve inverse images, quotients of reachable coalgebras need not be reachable again [50, Ex. 5.3.27], possibly rendering the usual order incorrect.

Our present paper is concerned with the *minimization problem* for coalgebras, i.e. the problem to compute the well-pointed modification of a given pointed coalgebra in terms of its encoding.

▶ Remark 2.6. Recall that a (sub)natural transformation σ from a functor F to a functor G is a set-indexed family of maps $\sigma_X\colon FX \to GX$ such that for every (injective) function $m\colon X \to Y$ the square below commutes; we also say that σ is *(sub)natural in X*.

From previous results (see [48, Prop. 2.13] and [49, Thm. 4.6]) one obtains the following sufficient condition for reductions of reachability and simplicity. Given a family of maps $\sigma_X\colon FX \to GX$, then every F-coalgebra (X, c) yields a G-coalgebra $(X, \sigma_X \cdot c)$ and we can reduce minimization tasks from F-coalgebras to G-coalgebras as follows:

1. Suppose that $\sigma\colon F \to G$ is *sub-cartesian*, that is the squares below are pullbacks for every injective map $m\colon X \rightarrowtail Y$. Then the reachable part of a pointed F-coalgebra (X, c, i) is obtained from the reachable part of the G-coalgebra $(X, \sigma_X \cdot c, i)$.

$$\begin{tikzcd} FX \arrow[r, "\sigma_X"] \arrow[d, "Fm"'] & GX \arrow[d, "Gm"] \\ FY \arrow[r, "\sigma_Y"] & GY \end{tikzcd}$$

2. Suppose that F is a subfunctor of G, i.e. we have a natural transformation σ with injective components $\sigma_X\colon FX \rightarrowtail GX$. Then the problem of computing the simple quotient for F-coalgebras reduces to that for G-coalgebras: the simple quotient of $(X, \sigma_X \cdot c)$ yields that of (X, c).

Consequently, if F is a subfunctor of G via a subcartesian σ, the minimization problem for F-coalgebras reduces to that for G-coalgebras. For example, the distribution functor $\mathcal{D}$ is a subcartesian subfunctor of $\mathbb{R}^{(-)}$. (For details see the full version [21].)

Preliminaries on Bags. The bag functor defined in Example 2.1 plays an important role in our minimization algorithm, not only as one of many possible system types, but bags are also used as a data structure. To this end, we use a couple of additional properties of this functor.

▶ Remark 2.7.

1. Since $\mathcal{B}$ can also be regarded as a monoid-valued functor for $(\mathbb{N}, +, 0)$, every bag $b = \{\!\{x_1, \dots, x_n\}\!\} \in \mathcal{B}X$ may be identified with a finitely supported function $X \to \mathbb{N}$, assigning to each $x \in X$ its multiplicity in b. We shall often make use of this fact and represent bags as functions.
2. The set $\mathcal{B}X$ itself is a commutative monoid with bag-union as the operation and the empty bag $\{\!\{\}\!\}$ as the identity element. In fact, this is the free commutative monoid over X. It therefore makes sense to consider the monoid-valued functor $(\mathcal{B}X)^{(-)}$ for a monoid of bags. Note that for every pair of sets A, X, the set $(\mathcal{B}A)^{(X)}$ of finitely supported functions from X to $\mathcal{B}A$ is isomorphic to $\mathcal{B}(A \times X)$ as witnessed by the following isomorphism (where swap, curry and $\mathsf{uncurry}$ are the evident canonical bijections):

$$\mathsf{group} = \big(\mathcal{B}(A \times X) \xrightarrow{\mathcal{B}(\mathsf{swap})} \mathcal{B}(X \times A) \xrightarrow{\mathsf{curry}} (\mathcal{B}A)^{(X)}\big), \text{ and}$$
$$\mathsf{ungroup} = \big((\mathcal{B}A)^{(X)} \xrightarrow{\mathsf{uncurry}} \mathcal{B}(X \times A) \xrightarrow{\mathcal{B}(\mathsf{swap})} \mathcal{B}(A \times X)\big).$$

Note that since swap is self-inverse and curry, $\mathsf{uncurry}$ are mutually inverse, group and $\mathsf{ungroup}$ are mutually inverse, too. In symbols:

$$\mathsf{group} \cdot \mathsf{ungroup} = \mathsf{id}_{(\mathcal{B}A)^{(X)}}, \quad \mathsf{ungroup} \cdot \mathsf{group} = \mathsf{id}_{\mathcal{B}(A\times X)}. \tag{1}$$

We often need to filter a bag of tuples $\mathcal{B}(A \times X)$ by a subset $S \subseteq X$. To this end we define the maps $\mathsf{fil}_S\colon \mathcal{B}(A \times X) \to \mathcal{B}(A)$ for sets $S \subseteq X$ and A by

$$\mathsf{fil}_S(f) = \Big(a \mapsto \sum_{x\in S} f(a, x)\Big) = \{\!\{a \mid (a, x) \in f, x \in S\}\!\},$$

where the multiset comprehension is given for intuition.

Zippable Functors. One crucial ingredient for the efficiency of the generic partition refinement algorithm [48] is that the coalgebraic type functor is zippable:

▶ **Definition 2.8** [48, Def. 5.1]**.** A set functor F is called *zippable* if the following maps are injective for every pair A, B of sets:

$$F(A+B) \xrightarrow{\langle F(A+!), F(!+B)\rangle} F(A+1) \times F(1+B).$$

Zippability of a functor allows that partitions are refined incrementally by the algorithm [48, Prop. 5.18], which in turn is the key for allowing a low run time complexity of the implementation. For additional visual explanations of zippability, see [48, Fig. 2]. We shall need this notion in the proof of Proposition 3.3, and later proofs use this result.

It was shown in [48] that all functors in Example 2.1 are zippable. In addition, zippable functors are closed under products, coproducts and subfunctors. However, they are not closed under functor composition, e.g. $\mathcal{P}_\mathsf{f}\mathcal{P}_\mathsf{f}$ is not zippable [48, Ex. 5.10].

The Trnková Hull. For purposes of universal coalgebra, we may assume without loss of generality that set functors preserve injections. Indeed, every set functor preserves nonempty injections (being the split monomorphisms in Set). As shown by Trnková [42, Prop. II.4 and III.5], for every set functor F there exists an essentially unique set functor $\bar{F}$ which coincides with F on nonempty sets and functions, and preserves finite intersections (whence injections). The functor $\bar{F}$ is called the *Trnková hull* of F. Since F and $\bar{F}$ coincide on nonempty sets and maps, the categories of coalgebras for F and $\bar{F}$ are isomorphic.

3 Coalgebra Encodings

In order to make abstract coalgebras tractable for computers and to have a notion of the size of a coalgebra structure in terms of nodes and edges as for standard transition systems, our algorithmic framework encodes coalgebras using a graph-like data structure. To this end, we require functors to be equipped with an encoding as follows.

▶ **Definition 3.1.** An *encoding* of a set functor F consists of a set A of *labels* and a family of maps $\flat_X \colon FX \to \mathcal{B}(A \times X)$, one for every set X, such that the following map is injective:

$$FX \xrightarrow{\langle F!, \flat_X\rangle} F1 \times \mathcal{B}(A \times X).$$

An *encoding* of a coalgebra $c\colon X \to FX$ is given by $\langle F!, \flat_X\rangle \cdot c\colon X \to F1 \times \mathcal{B}(A \times X)$.

Intuitively, the encoding $\flat_X$ of a functor F specifies how an F-coalgebra should be represented as a directed graph, and the required injectivity models that different coalgebras have different representations.

▶ Remark 3.2. Previously [48, Def. 6.1], the map $\langle F!, \flat_X\rangle$ was not explicitly required to be injective. Instead, a family of maps $\flat_X\colon FX \to \mathcal{B}(A \times X)$ and a *refinement interface* for F was assumed. The definition of a refinement interface for F is tailored towards the computation of behaviourally equivalent states and its details are therefore not relevant for the present work. All we need here is that the existence of a refinement interface implies the injectivity condition of Definition 3.1 and consequently, we inherit all examples of encodings from the previous work:

▶ **Proposition 3.3.** *For every zippable set functor F with a family of maps $\flat_X\colon FX \to \mathcal{B}(A \times X)$ and a refinement interface, the family $\flat_X$ is an encoding for F.*

▶ **Example 3.4.** We recall a number of encodings from [48]; the injectivity is clear, and in fact implied by Proposition 3.3:

1. Our encoding for the finite powerset functor $\mathcal{P}_\mathsf{f}$ resembles unlabelled transition systems by taking the singleton set $A = 1$ as labels. The map $\flat_X : \mathcal{P}_\mathsf{f}(X) \to \mathcal{B}(1 \times X) \cong \mathcal{B}(X)$ is the obvious inclusion, i.e. $\flat_X(t)(*, x) = 1$ if $x \in t$ and 0 otherwise.
2. The monoid-valued functor $M^{(-)}$ has labels from $A = M$ and $\flat_X : M^{(X)} \to \mathcal{B}(M \times X)$ is given by $\flat_X(t)(m, x) = 1$ if $t(x) = m \neq 0$ and 0 otherwise.
3. For a polynomial functor F_Σ, we use $A = \mathbb{N}$ as the label set and define the maps $\flat_X : F_\Sigma X \to \mathcal{B}(\mathbb{N} \times X)$ by $\flat_X(\sigma(x_1, \dots, x_n)) = \{\!\!\{ (1, x_1), \dots, (n, x_n) \}\!\!\}$.
   Note that $\flat_X$ itself is not injective if Σ has at least two operation symbols with the same arity. E.g. for DFAs ($F_\Sigma X = 2 \times X^A$), $\flat_X$ only retrieves information about successor states but disregards the "finality" of states. However, pairing $\flat_X$ with $F! : FX \to F1$ yields an injective map.
4. The bag functor $\mathcal{B}$ itself also has $A = \mathbb{N}$ as labels and $\flat_X(t)(n, x) = 1$ if $t(x) = n$ and 0 otherwise. This is just the special case of the encoding for a monoid-valued functor for the monoid $(\mathbb{N}, +, 0)$.

The encoding does by no means imply a reduction of the problem of minimizing F-coalgebras to that of coalgebras for $\mathcal{B}(A \times -)$ (cf. Remark 2.6). In fact, the notions of behavioural equivalence for F-coalgebras and coalgebras for $\mathcal{B}(A \times -)$, can be radically different. If $\flat_X$ is natural in X, then behavioural equivalence wrt. F implies that for $\mathcal{B}(A \times -)$, but not necessarily conversely. However, we do not assume naturality of $\flat_X$, and in fact it fails in all of our examples except one:

▶ **Proposition 3.5.** *The encoding $\flat_X : F_\Sigma X \to \mathcal{B}(A \times X)$ for the polynomial functor F_Σ is a natural transformation.*

▶ **Example 3.6.** The encoding $\flat_X : \mathcal{P}_\mathsf{f}(X) \to \mathcal{B}(1 \times X) \cong \mathcal{B}(X)$ in Example 3.4 item 1 is not natural. Indeed, consider the map $! : 2 \to 1$, for which we have

$$\mathcal{B}(!) \cdot \flat_2(\{0, 1\}) = \mathcal{B}(!)\{\!\!\{ 0, 1 \}\!\!\} = \{\!\!\{ *, * \}\!\!\} \neq \{\!\!\{ * \}\!\!\} = \flat_1(\{*\}) = \flat_1 \cdot \mathcal{P}_\mathsf{f}(!)(\{0, 1\}).$$

Similar examples show that the encodings in Example 3.4 item 2 (for all non-trivial monoids) and item 4 are not natural.

An important feature of our algorithm and tool is that all implemented functors can be combined by products, coproducts and functor composition. That is, the functors from Example 3.4 are implemented directly, but the algorithm also automatically handles coalgebras for more complicated combined functors, like those in Example 2.2, e.g. $\mathcal{P}_\mathsf{f}(A \times -)$. The mechanism that underpins this feature is detailed in previous work [20, 48] and depends crucially on the ability to form coproducts and products of encodings:

▶ **Construction 3.7** [20, 48]**.** Given a family of functors $(F_i)_{i \in I}$ with encodings $(\flat_{X,i})_{i \in I}$ and $(A_i)_{i \in I}$, we obtain the following encodings with labels $A = \coprod_{i \in I} A_i$:

1. for the coproduct functor $F = \coprod_{i \in I} F_i$ we take
$$\flat_X : \coprod_{i \in I} F_i X \xrightarrow{\coprod_{i \in I} \flat_{X,i}} \coprod_{i \in I} \mathcal{B}(A_i \times X) \xrightarrow{[\mathcal{B}(\mathsf{in}_i \times X)]_{i \in I}} \mathcal{B}\big(\coprod_{i \in I} A_i \times X\big).$$
2. for the product functor $F = \prod_{i \in I} F_i$ we take
$$\flat_X : \prod_{i \in I} F_i X \to \mathcal{B}\big(\coprod_{i \in I} A_i \times X\big) \qquad \flat_X(t)(\mathsf{in}_i(a), x) = \flat_i(\mathsf{pr}_i(t))(a, x),$$
   where $\mathsf{in}_i : A_i \to \coprod_j A_j$ and $\mathsf{pr}_i : \prod_j F_j X \to F_i X$ denote the canonical coproduct injections and product projections, respectively.

▶ **Proposition 3.8.** *The families $\flat_X$ defined in Construction 3.7 yield encodings for the functors $\prod_{i\in I} F_i$ and $\coprod_{i\in I} F_i$, respectively.*

▶ Remark 3.9. Since zippable functors are not closed under composition, modularity cannot be achieved by simply providing a construction of an encoding for a composed functor (at least not without giving up on the efficient run-time complexity). Functor composition is reduced to coproducts making a detour via many-sorted sets. Here is a rough explanation of how this works. Suppose that F is a *finitary* set functor, which means that for every $x \in FX$ there exists a finite subset $Y \subseteq X$ and $x' \in FY$ such that $x = Fm(x')$ for the inclusion map $m\colon Y \hookrightarrow X$. Given a finite coalgebra $c\colon X \to FGX$, it can be turned into a 2-sorted coalgebra $(c', d')\colon (X, Y) \to (FY, GX)$ as follows: since F is finitary one picks a finite subset Y of GX such that there exists a map $c'\colon X \to FY$ with $c = Fd' \cdot c'$, where $d'\colon Y \hookrightarrow GX$ is the inclusion map. Then c' and d' are combined into one coalgebra on the disjoint union $X + Y$ as shown below:

$$X + Y \xrightarrow{c'+d'} FY + GX \xrightarrow{[F\,\mathsf{inr}, G\,\mathsf{inl}]} (F + G)(X + Y)$$

for the coproduct of the functors F and G, where $\mathsf{inl}\colon X \to X + Y$ and $\mathsf{inr}\colon Y \to X + Y$ are the two coproduct injections. Full details may be found in [48, Sec. 8].

For the sake of computing the coalgebra structure of the minimized coalgebra, we require that, intuitively, the labels used for encoding FX are independent of the cardinality of X:

▶ **Definition 3.10.** An encoding $\flat_X$ for a set functor F is called *uniform* if it fulfils the following property for every $x \in X$:

$$\begin{tikzcd} FX \arrow[r, "\flat_X"] \arrow[d, "F\chi_{\{x\}}"'] & \mathcal{B}(A \times X) \arrow[dr, "\mathsf{fil}_{\{x\}}"] & \\ & & \mathcal{B}(A) \\ F2 \arrow[r, "\flat_2"] & \mathcal{B}(A \times 2) \arrow[ur, "\mathsf{fil}_{\{1\}}"'] & \end{tikzcd} \tag{2}$$

Intuitively, the condition in Definition 3.10 expresses that in an encoded coalgebra, the edges (and their labels) to a state x do not change if other states $y, z \in X \setminus \{x\}$ are identified by a possible partition on the state space. Diagram (2) expresses the extreme case of such a partition, particularly the one where *all* elements of X except for x are identified in a block, with x being in a separate singleton block.

Fortunately, requiring uniformity does not exclude any of the existing encodings that we recalled above.

▶ **Proposition 3.11.** *All encodings from Example 3.4 are uniform.*

Uniform encodings interact nicely with the modularity constructions:

▶ **Proposition 3.12.** *Uniform encodings are closed under product and coproduct.*

That is, given functors $(F_i)_{i\in I}$ with uniform encodings $(\flat_i)_{i\in I}$, then the encodings for the functors $\coprod_{i\in I} F_i$ and $\prod_{i\in I} F_i$, as defined in Construction 3.7, are uniform.

Admittedly, the condition in Definition 3.10 is slightly technical. However, we will now prove that it sits strictly between two standard properties, naturality and subnaturality.

▶ **Proposition 3.13.**

1. *Every natural encoding is uniform.*
2. *Every uniform encoding is a subnatural transformation.*

The converses of both of the above implications fail in general. For the converse of 1 we saw a counterexample in Example 3.6, and for the converse of 2 we have the following counterexample.

▶ **Example 3.14.** Consider the following encoding for the functor $FX = X \times X \times X$ given by $A = 3 + 3$ and

$$\flat_X : FX \to \mathcal{B}(A \times X)$$
$$\flat_X(x, y, z) = \begin{cases} \{(\mathsf{inl}\,0, x), (\mathsf{inl}\,1, y), (\mathsf{inl}\,2, z)\} & \text{if } y = z, \\ \{(\mathsf{inr}\,0, x), (\mathsf{inr}\,1, y), (\mathsf{inr}\,2, z)\} & \text{if } y \neq z. \end{cases}$$

This encoding is subnatural, since the value of $y = z$ is preserved by injections under F. But it is not uniform, for if $x \neq y \neq z$, then we have

$$\mathsf{fil}_{\{1\}}(\flat(F\chi_{\{x\}}(x, y, z))) = \mathsf{fil}_{\{1\}}(\flat(1, 0, 0)) = \{\mathsf{inl}\,0\} \neq \{\mathsf{inr}\,0\} = \mathsf{fil}_{\{x\}}(\flat(x, y, z)).$$

4 Computing the Simple Quotient

The previous coalgebraic partition refinement algorithm and its tool implementation in CoPaR compute for a given encoding of a coalgebra (X, c) the state set of its simple quotient $q: (X, c) \twoheadrightarrow (Y, d)$, that is the partition Y of the set X corresponding to behavioural equivalence. But the algorithm does not compute the coalgebra structure d of the simple quotient (and note that it is not given the structure c explicitly, to begin with). Here we will fill this gap. We are interested in computing the encoding $Y \xrightarrow{d} FY \xrightarrow{\flat_Y} \mathcal{B}(A \times Y)$ given the encoding $X \xrightarrow{c} FX \xrightarrow{\flat_X} \mathcal{B}(A \times X)$ of the input coalgebra and the quotient map $q: X \twoheadrightarrow Y$.

The edge labels in the encoding of the quotient coalgebra relate to the labels in the encoded input coalgebra in a functor specific way. For example, for weighted transition systems, the labels are the transition weights, which are added whenever states are identified. In contrast, for deterministic automata (or when F is a polynomial functor), the labels (i.e. input symbols) on the transitions remain the same even when states are identified.

Thus, when computing the encoding of the simple quotient, the modification of edge labels is functor specific. Algorithmically, this is reflected by specifying a new interface containing one function `merge`, which is intended to be implemented together with the refinement interface (Section 3) for every functor of interest. The abstract function `merge` is then used in the generic Construction 4.8 in order to compute the encoding of the simple quotient.

▶ **Definition 4.1.** A *minimization interface* for a set functor F equipped with a functor encoding $\flat_X : FX \to \mathcal{B}(A \times X)$ is a function `merge`: $\mathcal{B}(A) \to \mathcal{B}(A)$ such that the following diagram commutes for all $S \subseteq X$:

$$\begin{tikzcd} FX \arrow[r, "\flat_X"] \arrow[d, "F\chi_S"'] & \mathcal{B}(A \times X) \arrow[r, "\mathsf{fil}_S"] & \mathcal{B}(A) \arrow[d, "\mathtt{merge}"] \\ F2 \arrow[r, "\flat_2"] & \mathcal{B}(A \times 2) \arrow[r, "\mathsf{fil}_{\{1\}}"] & \mathcal{B}(A) \end{tikzcd} \tag{3}$$

Intuitively, `merge` expresses what happens on the labels of edges from one state to one block. It receives the bag of all labels of edges from a particular source state x to a *set of* states S that the minimization procedure identified as equivalent. It then computes the edge labels from x to the merged state S of the minimized coalgebra in a functor specific

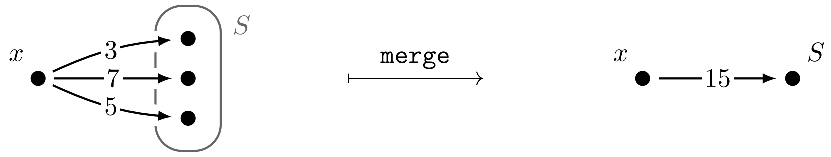

Figure 1 Example application of merge for the monoid-valued functor.

way. Figure 1 depicts this process for a monoid-valued functor (cf. Example 2.1, item 2). In this example, `merge` sums up the labels (which are monoid elements), resulting in a correct transition label to the new merged state.

Before we give formal definitions of `merge` for the functors of interest, let us show that there is a close connection between properties of `merge` and the encoding; this will simplify the definition of `merge` later (Example 4.4).

First, if `merge` receives the bag of labels from a source state to a *single* target state, then there is nothing to be merged and thus `merge` should simply return its input bag. Moreover, we can even characterize uniform encodings by this property:

▶ **Lemma 4.2.** *Given a minimization interface, the following are equivalent:*
1. $\mathtt{merge}(\mathsf{fil}_{\{x\}}(\flat_X(t))) = \mathsf{fil}_{\{x\}}(\flat_X(t))$ *for all* $t \in FX$.
2. $\flat_X$ *is uniform.*

Similarly, the property that `merge` is *always* the identity characterizes *natural* encodings:

▶ **Lemma 4.3.** *For every encoding* $\flat_X : FX \to \mathcal{B}(A \times X)$*, the following are equivalent:*
1. *The identity on* $\mathcal{B}A$ *is a minimization interface.*
2. $\flat_X$ *is a natural transformation.*

▶ **Example 4.4.**
1. For the finite powerset functor $\mathcal{P}_{\mathsf{f}}(-)$, with labels $A = 1$, we define $\mathtt{merge}: \mathcal{B}1 \to \mathcal{B}1$ by $\mathtt{merge}(\ell)(*) = \min(1, \ell(*))$.
2. For monoid-valued functors $M^{(-)}$ with $A = M$, `merge` is defined as
$$\mathtt{merge}(\ell) = \begin{cases} \{\!\!\{ \Sigma\ell \}\!\!\} & \Sigma\ell \neq 0 \\ \{\!\!\{\}\!\!\} & \text{otherwise,} \end{cases}$$
where $\Sigma: \mathcal{B}(M) \to M$ is defined by $\Sigma\{\!\!\{ m_1, \dots, m_n \}\!\!\} = m_1 + \cdots + m_n$.
3. The encoding for the polynomial functor F_Σ for a signature Σ is a natural transformation and hence its minimization interface is given by $\mathtt{merge} = \mathsf{id}$ (see Lemma 4.3).

▶ **Proposition 4.5.** *All* `merge` *maps in Example 4.4 are minimization interfaces and run in linear time in the size of their input bag.*

Having `merge` defined for the functors of interest, we can now use it to compute the encoding of the simple quotient.

▶ **Assumption 4.6.** For the remainder of this section we assume that $F1 \neq \emptyset$.

This is w.l.o.g. since $F1 = \emptyset$ if and only if $FX = \emptyset$ for all sets X, for which there is only one coalgebra (which is therefore its own simple quotient already).

▶ **Proposition 4.7.** *Suppose that the set functor* F *is equipped with a uniform encoding* $\flat_X : FX \to \mathcal{B}(A \times X)$ *and a minimization interface* `merge`*. Then the diagram below commutes for every map* $q: X \to Y$,

$$\begin{tikzcd} FX \arrow[r, "\flat_X"] \arrow[d, "Fq"'] & \mathcal{B}(A \times X) \arrow[r, "\mathcal{B}(A \times q)"] & \mathcal{B}(A \times Y) \arrow[r, "\mathsf{group}"] \arrow[d, dashed] & \mathcal{B}(A)^{(Y)} \arrow[d, "\mathtt{merge}^{(Y)}"] \\ FY \arrow[rr, "\flat_Y"] & & \mathcal{B}(A \times Y) & \mathcal{B}(A)^{(Y)} \arrow[l, "\mathsf{ungroup}"'] \end{tikzcd} \tag{4}$$

Note that the dashed arrow is not simply the identity map because $\flat_X$ fails to be natural for most functors of interest (Example 3.6).

Proof (Sketch). One first proves that $\mathtt{merge}$ preserves empty bags: $\mathtt{merge}(\{\!\!\{\}\!\!\}) = \{\!\!\{\}\!\!\}$. The commutativity of the desired diagram (4) is proven by extending it by every evaluation map $\mathsf{ev}(y)\colon \mathcal{B}(A)^{(Y)} \to \mathcal{B}(A)$, $y \in Y$, which form a jointly injective family. The extended diagram for $y \in Y$ is then proven commutative using (2) for y, (3) for $S = q^{-1}[y]$, which is also used in the form $\chi_{\{y\}} \cdot q = \chi_S$ in addition to two easy properties of ev and fil: $\mathsf{fil}_{\{y\}} = \mathsf{ev}(y) \cdot \mathsf{group}$ and $\mathsf{fil}_{\{y\}} \cdot \mathcal{B}(A \times q) = \mathsf{fil}_S$. ◀

▶ **Construction 4.8.** Given the encoded F-coalgebra $(X, \flat_X \cdot c)$, the quotient $q\colon X \twoheadrightarrow Y$, and a minimization interface for F, we define the map $e\colon Y \to \mathcal{B}(A \times Y)$ as follows: given an element $y \in Y$, choose any $x \in X$ with $q(x) = y$ and put

$$e(y) := (\mathsf{ungroup} \cdot \mathtt{merge}^{(Y)} \cdot \mathsf{group} \cdot \mathcal{B}(A \times q) \cdot \flat_X \cdot c)(x),$$

where the involved types are as follows:

$$\begin{tikzcd} X \arrow[r, "c"] \arrow[d, two heads, "q"'] & FX \arrow[r, "\flat_X"] & \mathcal{B}(A \times X) \arrow[r, "\mathcal{B}(A\times q)"] & \mathcal{B}(A \times Y) \arrow[r, "\mathsf{group}"] & \mathcal{B}(A)^{(Y)} \arrow[d, "\mathtt{merge}^{(Y)}"] \\ Y \arrow[rr, "e"] & & \mathcal{B}(A \times Y) & & \mathcal{B}(A)^{(Y)} \arrow[ll, "\mathsf{ungroup}"'] \end{tikzcd} \tag{5}$$

For the well-definedness and the correctness of Construction 4.8, we need to prove that (5) commutes. Moreover, observe that c is not directly given as input, and that the structure $d\colon Y \to FY$ of the simple quotient is not computed; only their encodings $\flat_X \cdot c$ and $e = \flat_Y \cdot d$ are.

▶ **Theorem 4.9.** *Suppose that $q\colon (X, c) \twoheadrightarrow (Y, d)$ represents a quotient coalgebra. Then Construction 4.8 correctly yields the encoding $e = \flat_Y \cdot d$ given the encoding $\flat_X \cdot c$ and the partition of X associated to q.*

If $\mathtt{merge}$ runs in linear time (in its parameter), then Construction 4.8 can be implemented with linear run time (in the size of the input coalgebra $\flat_X \cdot c$).

In the run time analysis, a bit of care is needed so that the implementation of group has linear run time; see the full version [21] for details. From Proposition 4.5 we see that for every functor from Example 2.1, Construction 4.8 can be implemented with linear run time.

4.1 Modularity of Minimization Interfaces

Modularity in the system type is gained by reducing functor composition to products and coproducts (Remark 3.9). Since we want the construction of the minimized coalgebra structure to benefit from the same modularity, we need to verify closure under product and coproduct for the notions required in Proposition 4.7. We have already done so for uniform encodings (Proposition 3.12); hence it remains to show that minimization interfaces can also be combined by product and coproduct:

▶ **Construction 4.10.** Given a family of functors $(F_i)_{i\in I}$ together with uniform encodings $\flat_i\colon F_iX \to \mathcal{B}(A_i \times X)$ and minimization interfaces $\mathtt{merge}_i\colon \mathcal{B}(A_i) \to \mathcal{B}(A_i)$, we define $\mathtt{merge}$ for the (co)product functors $\prod_{i\in I} F_i$ and $\coprod_{i\in I} F_i$ as follows:

$$\mathtt{merge}\colon \mathcal{B}(\textstyle\coprod_{i\in I} A_i) \to \mathcal{B}(\coprod_{i\in I} A_i) \qquad \mathtt{merge}(t)(\mathsf{in}_i\, a) = \mathtt{merge}_i(\mathsf{filter}_i(t))(a),$$

where $\mathsf{filter}_i\colon \mathcal{B}(\coprod_{j\in I} A_j) \to \mathcal{B}(A_i)$ is given by $\mathsf{filter}_i(f)(a) = f(\mathsf{in}_i(a))$.

Curiously, the definition of `merge` is the same for products and coproducts, e.g. because the label sets are the same (see Construction 3.7). However, the correctness proofs turns out to be quite different. Note that for coproducts, all labels in the image of $\mathsf{fil}_S \cdot \flat_X$ are in the same coproduct component. Thus, filter_i never removes elements and acts as a mere type-cast when the above `merge` is used in accordance with its specification.

▶ **Proposition 4.11.** *The* `merge` *function defined in Construction 4.10 yields a minimization interface for the functors $\prod_{i\in I} F_i$ and $\coprod_{i\in I} F_i$. It can be implemented with linear run-time if each* $\mathtt{merge}_i$ *is linear in its input.*

▶ **Corollary 4.12.** *The class of set functors having a minimization interface contains all polynomial and all monoid-valued functors and is closed under product and coproduct.*

Consequently, Construction 4.8 correctly yields encoded quotient coalgebras for those functors. Note that all functors from Example 4.4 are contained in this class. Furthermore, functor composition can be dealt with by using coproducts as explained in Remark 3.9.

5 Reachability

Having quotiented an encoded coalgebra by behavioural equivalence, the remaining task is to restrict the coalgebra to the states that are actually reachable from a distinguished initial state. For an intersection preserving set functor, the reachable part of a pointed coalgebra can be constructed iteratively, and this reduces to standard graph search on the canonical graph of the coalgebra [49, Cor. 5.26f], which we now recall. Throughout, $\mathcal{P}$ denotes the (full) powerset functor. The following is inspired by Gumm [28, Def. 7.2]:

▶ **Definition 5.1.** Given a functor $F\colon \mathsf{Set} \to \mathsf{Set}$, we define a family of maps $\tau^F_X\colon FX \to \mathcal{P}X$ by $\tau^F_X(t) = \{x \in X \mid 1 \xrightarrow{t} FX \text{ does not factorize through } F(X\setminus\{x\}) \xrightarrow{Fi} FX\}$, where $i\colon X\setminus\{x\} \hookrightarrow X$ denotes the inclusion map.

The *canonical graph* of a coalgebra $c\colon X \to FX$ is the directed graph $X \xrightarrow{c} FX \xrightarrow{\tau^F_X} \mathcal{P}X$. The nodes are the states of (X, c) and one has an edge from x to y whenever $y \in \tau^F_X(c(x))$.

Note that for a pointed coalgebra (X, c, i) its canonical graph is equipped with the same point $i\colon 1 \to X$, that is, the canonical graph is equipped with a root node $i(*) \in X$. As we pointed out in Section 2, reachability of the pointed $\mathcal{P}$-coalgebra $(X, \tau^F_X \cdot c, i)$ precisely means that every $x \in X$ is reachable from the root node in the canonical graph.

▶ **Example 5.2.**
1. For a deterministic automaton considered as a coalgebra for $FX = 2 \times X^A$ the canonical graph is precisely its usual underlying state transition graph.
2. For the finite powerset functor $\mathcal{P}_\mathsf{f}$, it is easy to see that $\tau^{\mathcal{P}_\mathsf{f}}_X\colon \mathcal{P}_\mathsf{f}X \hookrightarrow \mathcal{P}X$ is the inclusion map. Thus, the canonical graph of a $\mathcal{P}_\mathsf{f}$-coalgebra (a finitely branching graph) is itself.
3. For the functor $\mathcal{B}(A \times -)$ the maps $\tau^{\mathcal{B}(A\times -)}_X\colon \mathcal{B}(A\times X) \to \mathcal{P}X$ act as follows
$$\{\!\!\{\, (a_1, x_1), \ldots, (a_n, x_n) \,\}\!\!\} \mapsto \{x_1, \ldots, x_n\}.$$
Hence, if we view a coalgebra $X \to \mathcal{B}(A \times X)$ as a finitely-branching graph whose edges are labelled by pairs of elements of A and $\mathbb{N}$, then the canonical graph is that same graph but without the edge labels. This holds similarly also for other monoid-valued functors.

To perform reachability analysis on encoded coalgebras, we would like that the canonical graph of a coalgebra and its encoding coincide. This clearly follows when, given a set functor F with encoding $\flat_X\colon FX \to \mathcal{B}(A \times X)$, the following equation holds for every set X:

$$\tau_X^F = \big(FX \xrightarrow{\flat_X} \mathcal{B}(A \times X) \xrightarrow{\tau_X^{\mathcal{B}(A\times -)}} \mathcal{P}X\big). \tag{6}$$

▶ **Assumption 5.3.** For the rest of this section we assume that F is an intersection preserving set functor equipped with a subnatural encoding $\flat_X\colon FX \to \mathcal{B}(A \times X)$.

▶ Remark 5.4. That F preserves intersections is an extremely mild condition for set functors. All the functors in Example 3.4 preserve intersections. Furthermore, the collection of intersection preserving set functors is closed under products, coproducts, and functor composition. A subfunctor $\sigma\colon F \rightarrowtail G$ of an intersection preserving functor G preserves intersections if σ is a cartesian natural transformation, that is all naturality squares are pullbacks (cf. Remark 2.6).

Let us note that for every finitary set functor (cf. Remark 3.9) the Trnková hull $\bar{F}$ (see p. 8) preserves intersections [2, Cor. 8.1.17].

We are now ready to show the desired equality (6) by point-wise inclusion in either direction. Under the running Assumption 5.3 it follows that the encoding of a coalgebra can only mention states that are in the coalgebra's canonical graph:

▶ **Proposition 5.5.** *For every $t \in FX$ we have that $\tau_X^{\mathcal{B}(A\times -)}(\flat_X(t)) \subseteq \tau_X^F(t)$.*

Proof (Sketch). This is shown by contraposition. If x is not in $\tau_X^F(t)$, then we know that the map $t\colon 1 \to FX$ factorizes through $F(X \setminus \{x\}) \xrightarrow{Fi} FX$ (cf. Definition 5.1). Using the subnaturality square of $\flat$ for the map i then yields $x \notin \tau_X^{\mathcal{B}(A\times -)}(\flat_X(t))$. ◀

For the converse inclusion, we additionally require that F meets the assumptions of the partition refinement algorithm:

▶ **Theorem 5.6.** *The canonical graph of a finite coalgebra coincides with that of its encoding.*

For every finite set X one proves the equation (6): $\tau_X^F = \tau_X^{\mathcal{B}(A\times -)} \cdot \flat_X$. It suffices to prove the reverse of the inclusion in Proposition 5.5 – again by contraposition. This time the argument is more involved using that the map $\langle F!, \flat_X\rangle$ is injective (Definition 3.1), and that F preserves intersections. (For details see the full version [21].)

As a consequence of Theorem 5.6, the states in the reachable part of a pointed coalgebra (X, c, i) are precisely the states reachable from the node $i(*) \in X$ in the (underlying graph of the) encoding $\flat_X \cdot c\colon X \to \mathcal{B}(A \times X)$, cf. Example 5.23. Thus, given (the encoding of) a pointed coalgebra (X, c, i), its reachable part can be computed in linear time by a standard breadth-first search on the encoding viewed as a graph (ignoring the labels).

This holds for all the functors in Example 3.4 and every functor obtained from them by forming products, coproducts and functor composition.

6 Conclusions and Future Work

We have shown how to extend a generic coalgebraic partition refinement algorithm to a fully fledged minimization algorithm. Conceptually, this is the step from computing the simple quotient of a coalgebra to computing the well-pointed modification of a pointed coalgebra. To achieve this, our extension includes two new aspects: (1) the computation of the transition structure of the simple quotient given an encoding of the input coalgebra and the partition of its state space modulo behavioural equivalence, and (2) the computation of the encoding of the reachable part from the encoding of a given pointed coalgebra. Both of these new steps have also been implemented in the Coalgebraic Partition Refiner CoPaR, together with a new pretty-printing module that prints out the resulting encoded coalgebra in a functor-specific human-readable syntax.

There are a number of questions for further work. This mainly concerns broadening the scope of generic coalgebraic partition refinement algorithms. First, we will further broaden the range of system types that our algorithm and tool can accommodate, and provide support for base categories beside the sets as studied in the present work, e.g. nominal sets, which underlie nominal automata [13, 40].

Concerning genericity, there is an orthogonal approach by Ranzato and Tapparo [37], which is variable in the choice of the *notion of process equivalence* – however within the realm of standard labelled transition systems (see also [25]). Similarly, Blom and Orzan [11, 12] use a technique called *signature refinement*, which handles strong and branching bisimulation as well as Markov chain lumping (see also [45]).

To overcome the bottleneck on memory consumption that is inherent in partition refinement [43, 44], symbolic and distributed methods have been employed for many concrete system types [8, 11, 12, 24, 45, 47]. We will explore in future work whether these methods, possibly generic in the equivalence notion, can be extended to the coalgebraic generality.

References

1 Jiří Adámek, Filippo Bonchi, Barbara König, Mathias Hülsbusch, Stefan Milius, and Alexandra Silva. A coalgebraic perspective on minimization and determinization. In Lars Birkedal, editor, *Proc. Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 7213 of *Lecture Notes Comput. Sci.*, pages 58–73. Springer, 2012.

2 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Initial algebras, terminal coalgebras, and the theory of fixed points of functors. draft book, July 2020. URL: `https://www8.cs.fau.de/ext/milius/publications/files/CoalgebraBook.pdf`.

3 JiříAdámek, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. Well-pointed coalgebras. *Log. Methods Comput. Sci.*, 9(2):1–51, 2014.

4 Michael A. Arbib and Ernest G. Manes. Adjoint machines, state-behaviour machines, and duality. *J. Pure Appl. Algebra*, 6:313–344, 1975.

5 Michael A. Arbib and Ernest G. Manes. *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science. Springer, 1986.

6 Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60:187–231, 2000. `doi:10.1006/jcss.1999.1683`.

7 Falk Bartels, Ana Sokolova, and Erik de Vink. A hierarchy of probabilistic system types. *Theoretical Computer Science*, 327:3–22, 2004.

8 Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR: A modular tool for on-the-fly equivalence checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005*, volume 3440 of *Lecture Notes in Comput. Sci.*, pages 581–585. Springer, 2005. `doi:10.1007/b107194`.

9 Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. `doi:10.1007/s00224-016-9686-0`.

10 Nick Bezhanishvili, Marcello Bonsangue, Helle Hvid Hansen, Dexter Kozen, Clemens Kupke, Prakash Panangaden, and Alexandra Silva. Minimisation in logical form. Technical report, Cornell University, May 2020. available at `arXiv:2005.11551`.

11 Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. In *Parallel and Distributed Model Checking, PDMC 2003*, volume 89 of *Electron. Notes Theor. Comput. Sci.*, pages 99–113. Elsevier, 2003.

12 Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005. `doi:10.1007/s10009-004-0159-4`.

13 Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Log. Methods Comput. Sci.*, 10(3), 2014. `doi:10.2168/LMCS-10(3:4)2014`.

14 Filippo Bonchi, Marcello Bonsangue, Helle Hvid Hansen, Prakash Panangaden, Jan Rutten, and Alexandra Silva. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.*, 15(1):3:1–3:29, 2014.

15 Filippo Bonchi, Marcello Bonsangue, Jan Rutten, and Alexandra Silva. Brzozowski's algorithm (co)algebraically. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics, Kozen Festschrift*, volume 7230 of *Lecture Notes in Comput. Sci.*, pages 12–23. Springer, 2012.

16 Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In J. Fox, editor, *Mathematical Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561. Polytechnic Institute of Brooklyn, Polytechnic Press, 1962.

17 Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, December 1992. `doi:10.1007/bf01305232`.

18 Stefano Cattani and Roberto Segala. Decision algorithms for probabilistic bisimulation. In *Concurrency Theory, CONCUR 2002*, volume 2421 of *Lecture Notes in Comput. Sci.*, pages 371–385. Springer, 2002. `doi:10.1007/3-540-45694-5`.

19 CoPaR: The Coalgebraic Partion Refiner, February 2021. Available at `https://git8.cs.fau.de/software/copar`.

20 Hans-Peter Deifel, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Generic partition refinement and weighted tree automata. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 280–297, Cham, October 2019. Springer International Publishing. `doi:10.1007/978-3-030-30942-8_18`.

21 Hans-Peter Deifel, Stefan Milius, and Thorsten Wißmann. Coalgebra encoding for efficient minimization. full version with appendix. `arXiv:2102.12842`.

22 Salem Derisavi, Holger Hermanns, and William Sanders. Optimal state-space lumping in markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003. `doi:10.1016/S0020-0190(03)00343-0`.

23 Ulrich Dorsch, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Efficient coalgebraic partition refinement. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl, 2017.

24 Hubert Garavel and Holger Hermanns. On combining functional verification and performance evaluation using CADP. In *Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Comput. Sci.*, pages 410–429. Springer, 2002. `doi:10.1007/3-540-45614-7`.

25 Jan Groote, David Jansen, Jeroen Keiren, and Anton 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. `doi:10.1145/3060140`.

26 Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. An efficient algorithm to determine probabilistic bisimulation. *Algorithms*, 11(9):131, 2018. `doi:10.3390/a11090131`.

27 H. Peter Gumm. *Thomas Ihringer: Algemeine Algebra. Mit einem Anhang über Universelle Coalgebra von H. P. Gumm*, volume 10 of *Berliner Studienreihe zur Mathematik*. Heldermann Verlag, 2003.

28 H. Peter Gumm. From T-coalgebras to filter structures and transition systems. In José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan Rutten, editors, *Algebra and Coalgebra in Computer Science*, volume 3629 of *Lecture Notes in Comput. Sci.*, pages 194–212. Springer Berlin Heidelberg, 2005. `doi:10.1007/11548133_13`.

29 Johanna Högberg, Andreas Maletti, and Jonathan May. Backward and forward bisimulation minimization of tree automata. *Theoret. Comput. Sci.*, 410:3539–3552, 2009.

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

31 Dung Huynh and Lu Tian. On some equivalence relations for probabilistic processes. *Fund. Inform.*, 17:211–234, 1992.

32 Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Comput. Sci.*, pages 87–101. Springer, 2007. doi:10.1007/978-3-540-71209-1.

33 Bartek Klin. Structural operational semantics for weighted transition systems. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, volume 5700 of *Lecture Notes in Comput. Sci.*, pages 121–139. Springer, 2009.

34 Barbara König and Sebastian Küppers. A generalized partition refinement algorithm, instantiated to language equivalence checking for weighted automata. *Soft Comput.*, 22:1103–1120, 2018.

35 Nick Nick Bezhanishvili, Clemens Kupke, and Prakash Panangaden. Minimization via duality. In Luke Ong and R. de Queiroz, editors, *Proc. WoLLIC*, volume 7456 of *Lecture Notes in Comput. Sci.* Springer, 2012.

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

37 Francesco Ranzato and Francesco Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206:620–651, 2008. doi:10.1016/j.ic.2008.01.001.

38 Jurriaan Rot. Coalgebraic minimization of automata by initiality and finality. In Lars Birkedal, editor, *Proc. MFPS*, volume 325 of *Electron. Notes Theor. Comput. Sci.*, pages 253–276. Elsevier, 2016.

39 J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

40 Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. Nominal automata with name binding. In *Foundations of Software Science and Computation Structures, FOSSACS 2017*, volume 10203 of *Lecture Notes in Comput. Sci.*, pages 124–142, 2017. doi:10.1007/978-3-662-54458-7.

41 Deian Tabakov and Moshe Vardi. Experimental evaluation of classical automata constructions. In G. Sutcliffe and A. Voronkov, editors, *Proc. LPAR*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 396–411. Springer, 2005.

42 Věra Trnková. On a descriptive classification of set functors I. *Comment. Math. Univ. Carolin.*, 12:143–174, 1971.

43 Antti Valmari. Bisimilarity minimization in $\mathcal{O}(m \log n)$ time. In *Applications and Theory of Petri Nets, PETRI NETS 2009*, volume 5606 of *Lecture Notes in Comput. Sci.*, pages 123–142. Springer, 2009. doi:10.1007/978-3-642-02424-5.

44 Antti Valmari and Giuliana Franceschinis. Simple $\mathcal{O}(m \log n)$ time Markov chain lumping. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010*, volume 6015 of *Lecture Notes in Comput. Sci.*, pages 38–52. Springer, 2010.

45 Tom van Dijk and Jaco van de Pol. Multi-core symbolic bisimulation minimization. *J. Softw. Tools Technol. Transfer*, 20(2):157–177, 2018.

46 Boris Weisfeiler. *On Construction and Identification of Graphs.* Springer, 1976. doi:10.1007/bfb0089374.

47 Anton Wijs. Gpu accelerated strong and branching bisimilarity checking. In Christel Baier and Cesare Tinelli, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Comput. Sci.*, pages 368–383. Springer, 2015.

48 Thorsten Wißmann, Ulrich Dorsch, Stefan Milius, and Lutz Schröder. Efficient and modular coalgebraic partition refinement. *Log. Methods. Comput. Sci.*, 16(1):8:1–8:63, 2020.

49 Thorsten Wißmann, Stefan Milius, Jérémy Dubut, and Shin-ya Katsumata. A coalgebraic view on reachability. *Comment. Math. Univ. Carolin.*, 60(4), 2019.

50 Thorsten Wißmann. *Coalgebraic Semantics and Minimization in Sets and Beyond*. Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2020. URL: `https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/14222`.

51 Thorsten Wißmann, Hans-Peter Deifel, Stefan Milius, and Lutz Schröder. From generic partition refinement to weighted tree automata minimization, 2020. accepted for publication in *Formal Aspects of Computing*; available online at `arXiv:2004.01250`. `arXiv:2004.01250`.

52 Lijun Zhang, Holger Hermanns, Friedrich Eisenbrand, and David Jansen. Flow Faster: Efficient decision algorithms for probabilistic simulations. *Log. Meth. Comput. Sci.*, 4(4), 2008. `doi:10.2168/LMCS-4(4:6)2008`.