Friedrich-Alexander-Universität
Erlangen-Nürnberg

Chair for Computer Science 8
Theoretical Computer Science

Implementation and Evaluation of Efficient Partition Refinement Algorithms

Master Thesis in Computer Science

Hans-Peter Deifel

Advisors:

PD Dr. Stefan Milius   Thorsten Wißmann

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Erlangen, January 15, 2019

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 15.01.2019

Hans-Peter Deifel

Abstract

Minimization under bisimilarity is a way of reducing the state space of transition systems by identifying behavioral equivalent states. The resulting equivalence relation on states can be computed by partition refinement.

This thesis presents an implementation of a recently developed partition refinement algorithm that is generic in the type of transition system. It achieves this generality by regarding system types as endofunctors on sets and modeling transition systems as coalgebras for the given type functor. The resulting algorithm runs in time either $O(m \cdot \log n)$ or $O(m \cdot \log m \cdot \log n)$ for systems with n states and m edges, which matches the running time of the best known algorithms for some specific system types and in some other cases even improves the asymptotic running time.

The presented implementation provides a ready-to-use partition refinement tool for a wide range of system types out of the box, including deterministic finite automata, weighted tree automata, labeled transition systems and weighted transition systems. Additionally, users can quickly obtain a partition refiner for new types of systems by composing pre-implemented basic functors and also extend the program with new basic functors by implementing a self-contained refinement interface in Haskell.

Contents

1 Introduction

Minimization is the task of identifying behaviorally equivalent states in a state-based system to reduce the size of the system as much as possible while preserving its behavior. It has been extensively studied for different system types and is often used as a pre-processing step to further analysis, such as model checking.

Typically, minimization of state based systems consists of two steps:

(1) Elimination of unreachable states.

(2) Identification of behaviorally equivalent states.

While the first part can be achieved with a simple graph search, the second part depends heavily on the exact notion of behavioral equivalence. In this thesis, we work with *coalgebraic behavioral equivalence*, which coincides with the standard notion of *bisimilarity* on labeled transition systems. Classically, two states x and y are considered *bisimilar* if for any step x can make to a state x', y can make a step with the same effect to a state y', such that y' is bisimilar to x', and vice versa. Since this defines a greatest fixpoint, it follows immediately from Kleene's fixpoint theorem that it can be calculated iteratively in polynomial time by approximating the fixpoint from above. The task of calculating this fixpoint is often referred to as *partition refinement* or *lumping* in the literature. It proceeds by first tentatively considering all states to be equivalent and then successively distinguishing states that exhibit different behavior with regard to the current partition.

Instances of such algorithms have been extensively studied over the past 50 years. In fact, Kanellakis and Smolka [KS90] introduced an algorithm to minimize ordinary transition systems, which runs in $\mathcal{O}(n \cdot m)$ time, where n is the number of states and m the number of transitions. A more efficient algorithm for the same type of system was later described by Paige and Tarjan [PT87] with time complexity $\mathcal{O}((m + n) \cdot \log n)$ and generalized to labeled transition system with the same complexity [Val10]. Hopcroft's classical automata minimization algorithm [Hop71] for deterministic automata with a fixed input alphabet also falls into this category and has a running time of $\mathcal{O}(n \cdot \log n)$. It was later generalized to an unbounded input alphabet $|A|$ with running time $\mathcal{O}(|A| \cdot n \cdot \log n)$ [Gri73, Knu01]. Valmari and Franceschinis [VF10] have developed an algorithm for Markov chains and weighted systems with rational weights that has the same $\mathcal{O}((m + n) \cdot \log n)$ time complexity as the Paige-Tarjan algorithm.

This thesis describes an implementation of the efficient coalgebraic partition refinement algorithm developed in [WDMS18, DMSW17], which uses abstractions from category theory to achieve a highly generic and modular algorithmic framework that can be readily instantiated to concrete types of systems. Specifically, in this abstraction, transition systems are modeled as coalgebras following the paradigm of universal coalgebra [Rut00], in which the transition type is encapsulated as a functor, the *type functor*. This covers not only classical relational and weighted systems, but also systems that mix different transition types, such as Segala systems [Seg95]. The algorithm itself is described abstractly in the type functor, using a functor specific *refinement interface* that allows to represent coalgebras as graphs and provides the basic system-specific steps of the algorithm for each functor. Under suitable conditions on the type functor and refinement interface, the algorithm runs in $\mathcal{O}(m \cdot \log n)$ time where n is the number of states and m the number of edges in the graph representation of a given coalgebra.

The present thesis provides a working tool called CoPaR[1] as an implementation of the abstract framework outlined above, which shares many of its properties. The program can be used as an off-the-shelf partition refinement tool for a wide range of different systems, including all examples mentioned above, and additionally provides a framework for adding new type functors by implementing the equivalent of a refinement interface in Haskell. All existing and newly implemented basic system types can be freely mixed and combined by the user and an intuitive input syntax for coalgebras of the selected type is automatically devised. Our implementation deviates in some details from the generic algorithm given in [WDMS18] and we provide correctness proofs and performance analysis for those. Also, the complexity analysis of the generic algorithm is generalized to accommodate for refinement interfaces with non-constant running time.

Genericity usually comes at a cost, and thus it is not surprising that our program is slower by a constant factor than specialized implementations for specific type functors. We provide an extensive benchmark collection that includes converted models from the PRISM model checker and randomly generated automata. We find that the running time of CoPaR, when compared with a specialized implementation by Valmari, based on an algorithm by Valmari and Franceschinis [VF10] can be slower by a factor of up to twenty. To remedy some of that slowness, we have developed different optimization techniques and analyzed their impact on the benchmark set above.

1.1 Overview of This Thesis

Chapter 2 recalls the generic coalgebraic algorithm that is implemented in our tool CoPaR. First, in Section 2.1, the necessary technical background material is revisited, giving an introduction to equivalence relations and coalgebras. The generic shape of partition refinement algorithms is developed informally in Section 2.2 and subsequently made precise on the abstraction level of coalgebras in Section 2.3, which also develops a way to iteratively compute the algorithm. This is the prerequisite for Section 2.4, which introduces refinement interfaces for functors, gives concrete pseudocode listings for our algorithm and analyzes its running time complexity. Finally, Section 2.5 explains how to achieve modularity of the implementation by combining refinement interfaces.

In Chapter 3, a concrete implementation of the above algorithm in Haskell is presented. First, Section 3.1 explains how functors and refinement interfaces are realized in the implementation and Section 3.2 shows how to implement the modularity mentioned above by using existential types to combine functors and refinement interfaces. Section 3.3 gives an overview of the input file syntax for our program and Section 3.4 presents a refinable partition data structure and queue implementation used in our implementation. Lastly, some selected implementation details including optimizations are presented.

Finally, the performance of our implementation is analyzed in Chapter 4, first on models from the PRISM model checker in Section 4.1, and then on randomly generated automata in Section 4.2. Finally, the impact of different optimizations implemented for our program is analyzed in Section 4.3.

[1]Available at `https://gitlab.cs.fau.de/i8/copar`

2 Theoretical Foundations

This chapter recalls the theoretical foundation for the concrete implementation in Chapter 3 from [WDMS18, DMSW17]. Our framework achieves its genericity by using coalgebras for a functor on sets as an abstraction for state-based systems with a certain transition type. We use notions from basic category theory to achieve the necessary abstraction level but restrict ourselves to the category of sets in this thesis. We assume some familiarity with basic category theoretical notation like functors, although the most important concepts should be understandable even with knowledge of basic set theory.

Section 2.1 gives an introduction to equivalence classes and coalgebras. In Section 2.2, a partition refinement algorithm is developed informally and subsequently made precise in 2.3. Finally, the mathematical abstractions presented in those sections are then developed into efficient and ready to use data structures and pseudo-code in Section 2.4. Except for the results on non-cancellative Monoids and the respective complexity (Remark 2.25, Example 2.26, Proposition 2.27), all results in Chapter 2 can be found in [WDMS18, DMSW17] in full detail.

2.1 Preliminaries

We now recall the necessary notions from basic category theory.

Notation 2.1. Given maps $f\colon X \to Y$ and $g\colon X \to Z$ the injection into the cartesian product $\langle f, g\rangle\colon X \to Y \times Z$ is defined as $\langle f, g\rangle(x) = (f(x), g(x))$. The *kernel* of a map $f\colon X \to Y$ is the equivalence relation $\ker(f) = \{(x_1, x_2) \in X \times X \mid f(x_1) = f(x_2)\}$, and the canonical surjective map corresponding to an equivalence relation $E \subseteq X \times X$ is denoted by $\kappa_E\colon X \twoheadrightarrow X/E$.

We identify equivalences on X with partitions (and quotients) of X and use the maps κ_E to denote them.

Remark 2.2. (1) For given maps $f\colon X \to Y$ and $g\colon X \to Z$, the intersection of the equivalence relations is $\ker f \cap \ker g = \ker\langle f, g\rangle$.

(2) Whenever $\ker(a\colon D \to A) = \ker(b\colon D \to B)$, then $\ker(a \cdot g) = \ker(b \cdot g)$ for every $g\colon C \to D$.

We model state based transition systems by coalgebras for endofunctors on set.

Definition 2.3 (Coalgebra)**.** Given an endofunctor $F\colon \mathsf{Set} \to \mathsf{Set}$ on sets, a *coalgebra* is a pair (C, c) where C is a set of states, the *carrier*, and $c\colon C \to FC$ is a map called the *structure* of the coalgebra.

Example 2.4. (1) Labeled transition systems for a set of labels A are coalgebras for the functor given by $FX = \mathcal{P}(A \times X)$. Explicitly, the coalgebra $c\colon X \to FX$ assigns to each state $x \in X$ a set of *labeled successor states* $c(x) \subseteq A \times X$. Unlabeled transition systems are coalgebras for the powerset functor $\mathcal{P}$. For the rest of this thesis we will restrict ourselves to the finite powerset $\mathcal{P}_f X = \{S \in \mathcal{P}X \mid S \text{ finite}\}$. Coalgebras for the functor $\mathcal{P}_f X$ correspond to finitely branching transition systems.

(2) Finitely branching weighted transition systems with weights from a commutative monoid $(M, +, 0)$ are modeled as coalgebras for the monoid-valued functor $FX = M^{(X)}$, defined on sets by

$$M^{(X)} := \{f\colon X \to M \mid f(x) \neq 0 \text{ for finitely many } x\},$$

and for a map $h\colon X \to Y$ by

$$M^{(h)}(f)(x) := \sum_{h(x)=y} f(x).$$

We refer to the elements $M^{(X)}$ as (finitely supported) *M-valued measures* on X, e.g., $\mathbb{R}_{\geq 0}$-valued measures are measures in the standard sense and $\mathbb{R}$-valued measures are signed measures. When regarded as a weighted transition system, a coalgebra $c\colon C \to M^{(C)}$ maps states $x, y \in C$ to the weight $c(x)(y)$ of the transition from x to y.

(3) The finite powerset functor $\mathcal{P}_\mathrm{f}$ mentioned above is the monoid-valued functor for the Boolean monoid $\mathbb{B} = (2, \vee, 0)$. Analogously, the *bag functor $\mathcal{B}_f$*, which assigns to each set X a set of bags (i.e., finite multisets) on X, is the monoid-valued functor for the additive monoid of natural numbers $(\mathbb{N}, +, 0)$.

(4) Probabilistic transition systems (e.g., Markov chains) are modeled coalgebraically using the distribution functor $\mathcal{D}$. This is a subfunctor of the monoid-valued functor $\mathbb{R}_{\geq 0}^{(X)}$ for the additive monoid of non-negative reals, given by

$$\mathcal{D}X = \left\{ f \in \mathbb{R}_{\geq 0}^{(X)} \,\middle|\, \sum_{x \in X} f(x) = 1 \right\}.$$

A coalgebra $c\colon C \to \mathcal{D}C$ for this functor assigns to every state x a finitely supported probability distribution over successor states.

(5) Deterministic finite automata for a fixed input alphabet A can be modeled as coalgebras for the functor $FX = 2 \times X^A$. A coalgebra $c\colon C \to FC$ assigns to each state x its finality $\pi_1 \cdot c(x)$ and an a-successor $\pi_2 \cdot c(x)(a)$ for each $a \in A$.

(6) Markov decision processes combine nondeterministic and probabilistic choice, and the following two examples give a transition system representation of them. Segala systems [Seg95] strictly alternate between non-deterministic and probabilistic transitions. Simple Segala systems can be modeled by the functor $\mathcal{P}_\mathrm{f}(A \times \mathcal{D}(-))$, which means that for each $a \in A$, a state non-deterministically proceeds to one of a finite number of probability distributions over successor states. General Segala systems are coalgebras for the functor $\mathcal{P}_\mathrm{f}\mathcal{D}(A \times -)$, which means that a state s non-deterministically proceeds to one of a finite number of probability distributions over A-labeled transitions from s.

A *coalgebra morphism* from a coalgebra (C, c) to a coalgebra (D, d) is a function $h\colon C \to D$ for which the following diagram commutes:

$$\begin{tikzcd}
C \arrow[r, "c"] \arrow[d, "h"] & HC \arrow[d, "Hh"] \\
D \arrow[r, "d"] & HD
\end{tikzcd}$$

Intuitively, coalgebra morphisms preserve observable behavior. A coalgebra (C, c) is called *simple*, if every coalgebra morphism with domain (C, c) is injective. In simple coalgebras, all states with the same observable behavior in C are already equal and the coalgebra is therefore minimal with respect to behavioral equivalence.

One can prove that every F-coalgebra (C, c) has a simple quotient, i.e., there exists a surjective coalgebra morphism $h\colon (C, c) \to (D, d)$ with D simple. Moreover, the simple quotient is unique up to isomorphism [Gum03], and we can therefore call (D, d) *the* simple quotient of (C, c).

Definition 2.5. Two elements x and y of a coalgebra (C, c) are called *behaviorally equivalent* if they are identified by some coalgebra morphism, i.e., there exists a coalgebra morphism $h\colon (C, c) \to (D, d)$ such that $h(x) = h(y)$.

Identifying all behaviorally equivalent states without introducing new ones is the task of finding a surjective coalgebra morphism to a simple coalgebra, i.e., the simple quotient. Minimizing a coalgebra is therefore the process of finding the simple quotient. The next section presents an abstract partition refinement algorithm that achieves this task on the level of F-coalgebras.

2.2 Partition Refinement in State-Based Systems

Before going into the details of the coalgebraic Algorithm 2.9 in Section 2.3, we will first develop some informal intuition for the process of partition refinement for state-based systems in general. One of the core ideas of the previously published partition refinement algorithms mentioned in the introduction, in particular the algorithms by Hopcroft [Hop71] and Paige-Tarjan [PT87], is to maintain two partitions of the state set X, X/P and X/Q, where X/P is "one transition step ahead of X/Q" and so the relation P is a refinement of Q. We call the elements of X/P *subblocks* and the elements of X/Q *compound blocks*.

Algorithm 2.6 (Informal Partition Refinement)**.** Given a state based system on X, we initially put $X/Q = \{X\}$ and let X/P be the initial partition on the "output behavior" of states in X. For example, for deterministic finite automata this output behavior is given by the finality of states, i.e., X/P initially separates final from non-final states. For labeled transition systems, states with successors are separated from deadlock states.

Iterate the following steps while P is properly finer than Q:

(1) Pick a subblock $S \in X/P$ that is properly contained in a compound block $C \in X/Q$, i.e., $S \subsetneqq C$. Note that this choice represents a quotient $q\colon X \to \{\, S,\ C\backslash S,\ X\backslash C \,\}$.

(2) Refine X/Q by this quotient q by splitting C into new blocks S and $C \setminus S$. This is just the (block-wise) intersection of the partitions X/Q and $\{\, S,\ C\backslash S,\ X\backslash C \,\}$ on X.

(3) Choose X/P as coarse as possible such that two states are identified in X/P if their transition structure is the same up to Q. This is sometimes referred to as refining the partition X/P such that the transition structure is *stable* w.r.t. X/Q.

Note that steps (1) and (2) only perform basic operations on quotients and are therefore independent of the transition type. In contrast, the initialization of X/P and step (3) depend on the "output behavior" and transition structure of the given system X.

2.3 Coalgebraic Partition Refinement

This section formalizes the pattern of Algorithm 2.6 as a coalgebraic refinement algorithm that computes the simple quotient of a given coalgebra.

Assumption 2.7. For the rest of this section we fix an endofunctor $F\colon \mathsf{Set} \to \mathsf{Set}$.

Given a coalgebra $\xi\colon X \to FX$, a coalgebraic partition refinement algorithm should maintain a quotient $q\colon X \to X/Q$ that distinguishes some but not necessarily all states with different behavior. Initially, q typically identifies all states and is then successively refined over the course of the algorithm. This quotient corresponds to the partition X/Q in Algorithm 2.6 and the idea of X/P being "one step ahead" of X/Q is captured by the map $Hq \cdot \xi\colon X \xrightarrow{\xi} HX \xrightarrow{Hq} H\,X/Q$. The transition type specific steps of Algorithm 2.6 analyze this map to identify equivalence classes w.r.t. q containing states that exhibit distinguishable behavior after one more step of the transition structure ξ. For example, to obtain an initial partition X/P of states in the automaton $\xi\colon X \to 2 \times X^A$, i.e., to distinguish final from non-final states, we can separate states w.r.t. the partition $!\colon X \to 1$ that identifies all states. Indeed, two states X have the same finality iff they are identified by the map $(2\times !^A)\cdot\xi\colon X \to 2\times 1^A \cong 2$.

We model step (2) of Algorithm 2.6 using the following map. Having a map, we will be able to model step (3) by what the given functor does on maps.

Definition 2.8. Given a chain of subsets $S \subseteq C \subseteq X$, define the following characteristic function:

$$\chi_S^C\colon X \to 3 \qquad \chi_S^C = \begin{cases} 2 & \text{if } x \in S \\ 1 & \text{if } x \in C \setminus S \\ 0 & \text{if } x \in X \setminus C \end{cases}$$

Note that the above definition is obtained from $\langle \chi_S, \chi_C\rangle\colon X \to 2\times 2 \cong 4$, where χ_S and χ_C are the ordinary characteristic functions for S and C respectively, by omitting the impossible case $x \in X \setminus C$.

With this, the coalgebraic refinement algorithm works as follows.

Algorithm 2.9. Given a finite coalgebra $\xi\colon X \to FX$, we successively refine equivalence relations Q and P on X, maintaining the invariant that P is finer than Q. In each step, we represent the split of a compound block $C \in X/Q$ into a subblock $S \in X/P$ and $C \setminus S$, by the map $q = \chi_S^C$ and accumulate this information in another map $\bar{q}$.

For easier reference in the following sections, all variables are indexed over loop iterations.

Initially define

$$\bar{q}_0 := q_0 := !\colon X \to 1 \qquad Q_0 := \ker q = X \times X$$

and compute

$$P_0 := \ker(X \xrightarrow{\xi} FX \xrightarrow{F!} F1).$$

While P_i is properly finer than Q_i, witnessed by the canonical quotient $\kappa_{P_i}\colon X/P_i \twoheadrightarrow X/Q_i$, iterate the following steps:

(1) Pick a subblock $S_i \in X/P_i$, such that $2\cdot|S_i| \le |C_i|$, where $C_i := \kappa_{P_i}(S_i)$ and define

$$q_{i+1} := \chi_{S_i}^{C_i}\colon X \to 3 \qquad \text{and} \qquad \bar{q}_{i+1} := \langle \bar{q}_i, q_{i+1}\rangle : X \to 3^{i+1}.$$

(2) Define $Q_{i+1} := \ker(\bar{q}_{i+1}) \qquad (= \ker\langle \bar{q}_i, q_{i+1}\rangle = \ker \bar{q}_i \cap \ker q_{i+1})$.

(3) Compute $P_{i+1} := \ker(X \xrightarrow{\xi} FX \xrightarrow{F\bar{q}_{i+1}} F(3^{i+1}))$.

Upon termination, return $X/P_i = X/Q_i$ as the simple quotient of (X, ξ).

Note that this corresponds precisely to the informal Algorithm 2.6. Indeed, in the initialization phase, X/Q is defined to consist of a single block containing all states of X. X/P is initialized by considering the "output behavior" of the states. The choice of a subblock in step (1) is made more precise than in Algorithm 2.6 and the characteristic function χ_S^C is used, which is addressed in detail below. In step (2), the partition X/Q is refined by χ_S^C. Finally, step (3) does the same for X/P by taking the coalgebra structure into account. The definition of X/P_i depends on the type functor F and require more involved computations, which will be described in Section 2.4. All other steps are basic operations on partitions (respectively kernels).

Remark 2.10. Most classical partition refinement algorithms are parameterized by an initial partition $\kappa_I \colon X \to X/I$. By replacing the coalgebra (X, ξ) with $(X, \langle \kappa_I, \xi\rangle)$ for the functor $X/I \times F(-)$, Algorithm 2.9 can also be made to support initial partitions. In this case, P_0 will already be finer than I.

Theorem 2.11. *For a finite F-coalgebra (X, ξ), Algorithm 2.9 terminates after at most $|X|$ steps and computes the simple quotient of a given coalgebra.*

Intuitively, X/Q becomes properly finer from iteration to iteration and hence can only be refined at most $|X|$ times. In each iteration, Q contains all pairs of behaviorally equivalent states and upon termination, $X \twoheadrightarrow X/Q$ is a coalgebra morphism and so identifies only behaviorally equivalent states.

Remark 2.12. The choice of a subblock S in step (1) of Algorithm 2.9 deserves some special attention. Requiring S to be at most half the size of the compound block that it is contained in is crucial for the good performance of the algorithm. This “select the smaller half” trick goes back to Hopcroft [Hop71] and is used in all the known efficient partition refinement algorithms mentioned in Chapter 1. In Chapter 3, we describe how to implement this operation with a carefully maintained queue of subblocks.

Algorithm 2.9 cannot be implemented efficiently without requiring additional assumptions on the type functor F. To see why, consider steps (2) and (3) that compute the partitions X/Q_i and X/P_i respectively. The relation $Q_i \coloneqq \ker \bar{q}_i = \ker\langle \bar{q}_i, q_{i+1}\rangle$ is defined as the intersection of kernels $\ker \bar{q}_i$ and $\ker q_{i+1}$, which can be computed iteratively by successive intersections. However, the same trick cannot be immediately applied for X/P_i, because of the functor F inside the computation of the kernel $P_i \coloneqq \ker(F\langle \bar{q}_i, q_{i+1}\rangle \cdot \xi)$ and so X/P_i has to be computed anew from the information in $\bar{q}_i$ in each iteration step. In the following, we will provide sufficient conditions on F to be able to optimize Algorithm 2.9 by changing step (3) to

$$P'_{i+1} \coloneqq \ker\langle F\bar{q}_i \cdot \xi,\ Fq_{i+1} \cdot \xi\rangle \qquad (= P_i \cap \ker(Fq_{i+1} \cdot \xi)). \tag{2.1}$$

Definition 2.13. A functor $F \colon \mathsf{Set} \to \mathsf{Set}$ is *zippable* if the following map is injective for all sets X, Y:

$$\mathrm{unzip}_{F,X,Y} \colon F(X + Y) \xrightarrow{\langle F(X+!),\, F(!+Y)\rangle} F(X + 1) \times F(1 + Y).$$

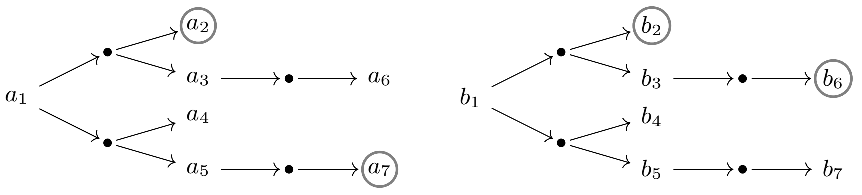
Lemma 2.14. *Zippable endofunctors are closed under products, coproducts and subfunctors.*

All functors from Example 2.4 are zippable. Nevertheless, zippable functors fail to be closed under composition and quotients as witnessed by the counterexample $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$, for which

$$\begin{aligned} &\mathrm{unzip}(\{\{a_1, b_1\}, \{a_2, b_2\}\}) \\ &\qquad = (\{\{a_1, \bullet\}, \{a_2, \bullet\}\}, \{\{\bullet, b_1\}, \{\bullet, b_2\}\}) \\ &\qquad = \mathrm{unzip}(\{\{a_1, b_2\}, \{a_2, b_1\}\}) \end{aligned}$$

and therefore unzip is not injective. Indeed, the optimized Algorithm 2.9 is not correct for the functor $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$, as demonstrated by the following example:

Example 2.15. Consider the following coalgebra $\xi : X \to FX$ for $FX = 2 \times \mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}X$:



In this picture, the outer sets are depicted as $\bullet$ and states x with $\pi_1(\xi(x)) = 1$ are circled. When computing P_{i+1} with the optimized version (2.1) of Algorithm 2.9, a_1 and b_1 are not distinguished, although they are behaviorally equivalent and are indeed separated correctly in the original version of the algorithm.

For this coalgebra, the initial partition distinguishes the three sets of states

$$A \coloneqq \{a_2, a_7, b_2, b_6\}\,, \quad B \coloneqq \{a_4, a_6, b_4, b_7\} \quad \text{and} \quad C \coloneqq \{a_1, a_3, a_5, b_1, b_3, b_5\}\,,$$

of circled states, non-circled states without successors and the rest. The optimized algorithm then computes the following sequence of partitions Q_i and P'_i:

q_i	X/Q_i	X/P'_i
$! : X \to 1$	$\{X\}$	$\{A, B, C\}$
$\vdots$	$\{A, B, C\}$	$\{A, B, \{a_1, b_1\}, \{a_3, b_5\}, \{a_5, b_3\}\}$
$\chi^C_{\{a_1,b_1\}} : X \to 3$	$\{A, B, \{a_1, b_1\}, \{a_3, b_5, a_5, b_3\}\}$	$\{A, B, \{a_1, b_1\}, \{a_3, b_5\}, \{a_5, b_3\}\}$
$\chi^{\{a_3,b_5,a_5,b_3\}}_{\{a_3,b_5\}} : X \to 3$	$\{A, B, \{a_1, b_1\}, \{a_3, b_5\}, \{a_5, b_3\}\}$	$\{A, B, \{a_1, b_1\}, \{a_3, b_5\}, \{a_5, b_3\}\}$

Note that in the last step for $S = \{a_3, b_3\}$, $\{a_1, b_1\}$ is not split in X/P', because:

$$\begin{aligned}
F\chi^{\{a_3,b_5,a_5,b_3\}}_{\{a_3,b_5\}} \cdot \xi(a_1) &= F\chi^{\{a_3,b_5,a_5,b_3\}}_{\{a_3,b_5\}}\{\{a_2, a_3\}, \{a_4, a_5\}\} \\
&= \qquad\qquad \{\{0, 2\}, \{0, 1\}\} \\
&= \qquad\qquad \{\{0, 1\}, \{0, 2\}\} \\
&= F\chi^{\{a_3,b_5,a_5,b_3\}}_{\{a_3,b_5\}}\{\{b_2, b_3\}, \{b_4, b_5\}\} = F\chi^{\{a_3,b_5,a_5,b_3\}}_{\{a_3,b_5\}} \cdot \xi(b_1)
\end{aligned}$$

At this point, the algorithm terminates because $X/Q_i = X/P'_i$ without distinguishing a_1 and b_1.

Another condition for the correctness of optimization (2.1) is already noted by Paige and Tarjan [PT87]. When refining the partition X/P, one needs to split by $C \setminus S$ in addition to S. Our algorithm achieves this by splitting by χ^C_S to refine X/P, instead of just the ordinary characteristic function χ_S. See [WDMS18, Example 5.11] for an example where this fails. Formally, this condition is captured by the following proposition:

Proposition 2.16. *Le $q\colon X \twoheadrightarrow X/Q$ be a partition and $S \subseteq C \in X/Q$. Then* $\ker q \cup \ker \chi^C_S$ *is a kernel.*

Proof. Let $p = \chi^C_S$. Since reflexive and symmetric relations are closed under unions in Set, we only need to show that $\ker q \cup \ker p$ is transitive.

Given $a, b, c \in X$ with $q(a) = q(b)$ and $p(b) = p(c)$, we show that either $q(a) = q(b) = q(c)$ or $p(a) = p(b) = p(c)$.

(1) If $b \in S$, then $c \in S$ by the definition of p and $a \in C$ by q. Since $S \subseteq C$, we have $a, b, c \in C$, and therefore $q(a) = q(b) = q(c)$.

(2) If $b \in (C \setminus S) \subseteq C$, then $a \in C$ by q and also $c \in (C \setminus S) \subseteq C$ by the definition of p. Therefore, $q(a) = q(b) = q(c)$.

(3) Finally if $b \in X \setminus C$, then $c \in X \setminus C$ by the definition of p. Also, a and b are in some block $C' \in X/Q$ with $C' \neq C$, and therefore $p(a) = 0 = p(b) = p(c)$. $\square$

Proposition 2.17. *Let $a\colon D \to A$ and $b\colon D \to A$ be such that $\ker a \cup \ker b$ is a kernel and let $F\colon \mathsf{Set} \to \mathsf{Set}$ be a zippable functor. Then we have*

$$\ker\langle\, Fa, Fb \,\rangle = \ker F\langle a, b\rangle.$$

Combining Propositions 2.16 and 2.17, we get the following theorem.

Theorem 2.18. *If $F\colon \mathsf{Set} \to \mathsf{Set}$ is zippable, then optimization* (2.1) *is correct.*

Proof. Correctness of (2.1) means that the second equality below holds:

$$P_{i+1} = \ker(F\langle \bar{q}_i, q_{i+1}\rangle \cdot \xi) = P_i \cap \ker(F q_{i+1} \cdot \xi) = P'_{i+1}.$$

If F is zippable and $q_{i+1} = \chi_S^C$ for $C \in X/Q_i$ and $S \in X/P_i$ as in Algorithm 2.9, we have:

$$\begin{aligned} &\ker \bar{q}_i \cup \ker q_{i+1} \text{ is a kernel} && \text{by Proposition 2.16}\\ \Rightarrow\, &\ker\langle F\bar{q}_i, Fq_{i+1}\rangle = \ker F\langle \bar{q}_i, q_{i+1}\rangle = \ker F\bar{q}_{i+1} && \text{by Proposition 2.17}\\ \Rightarrow\, &\ker(\langle F\bar{q}_i, Fq_{i+1}\rangle \cdot \xi) = \ker(F\bar{q}_{i+1} \cdot \xi) && \text{by Remark 2.2(2)}\\ \Rightarrow\, &P_{i+1} = P'_{i+1} && \square \end{aligned}$$

Corollary 2.19. *Suppose that F is a zippable endofunctor on* Set. *Then Algorithm 2.9 with optimization* (2.1) *terminates and computes the simple quotient of a given finite F-coalgebra.*

2.4 Efficient Calculation of Kernels

We have seen in Algorithm 2.9 with optimization (2.1) that the computation of kernels can be performed iteratively by successively refining the corresponding equivalence relations, but we have not yet specified how to actually compute kernels efficiently. We now define a more detailed algorithm parameterized by a *refinement interface* for the given type functor. This interface is aimed towards an efficient implementation of the *refinement step* in the algorithm. Specifically, from now on we split along a map $\xi\colon X \to FX$ w.r.t. a subblock $S \subseteq C \in X/Q$ and need to compute how the partition X/P has to be changed when the compound block C is split into S and $C \setminus S$ in X/Q.

The low complexity of Hopcroft- and Paige-Tarjan-style algorithms hinges on the refinement step running in $\mathcal{O}(|\mathrm{pred}[S]|)$ time, where $\mathrm{pred}[S]$ denotes the set of predecessors of the states s in the subblock S. In order to speak about "predecessors" w.r.t. a map $\xi\colon X \to FX$, we also provide for a given type functor F an encoding of ξ as a set of states with successor structures encoded as bags of A-labeled edges, where A is an appropriate label alphabet. Moreover, The refinement interface will allow us to analyze the behavior of elements of X w.r.t. transitions to states in $S \subseteq C$ and $C \setminus S$ by looking only at states in S.

Definition 2.20. An *encoding* of a functor F consists of a set A of *labels* and for every X a map $\flat\colon FX \to \mathcal{B}_\mathrm{f}(A \times X)$. The *encoding* of an F-coalgebra $\xi\colon X \to FX$ is then given by a set E of edges and maps

$$\texttt{graph}\colon E \to X \times A \times X, \qquad\qquad \texttt{type}\colon X \to F1,$$

such that $(\flat \cdot \xi(X))(a, y) = |\{e \in E \mid \texttt{graph}(e) = (x, a, y)\}|$ and $\texttt{type} = F! \cdot \xi$.

Intuitively, an encoding presents the map ξ as a graph with edge labels from A and node labels from $F1$.

Notation 2.21. In the following, we write $e = x \xrightarrow{a} y$ instead of $\texttt{graph}(e) = (x, a, y)$.

Definition 2.22. Given the encoding $(A, \flat)$ of the Set-functor F, a *refinement interface* for F consists of a set W of *weights* and functions

$$\texttt{init}\colon F1 \times \mathcal{B}_\mathrm{f}A \to W, \qquad\qquad \texttt{update}\colon \mathcal{B}_\mathrm{f}A \times W \to W \times F3 \times W$$

satisfying the following coherence condition: there exists a map $w\colon \mathcal{P}_\mathrm{f}X \to FX \to W$ such that for all $t \in FX$ and all $S \subseteq C \subseteq X$:

$$\begin{aligned} w(X)(t) &= \texttt{init}(F!(t), \mathcal{B}_\mathrm{f}\pi_1(\flat(t))) \\ \langle w(S), F\chi_S^C, w(C \setminus S)\rangle(t) &= \texttt{update}(\{a \mid (a, x) \in \flat(t), x \in S\}, w(C)) \end{aligned}$$

Note that the comprehension in the first argument of `update` is to be read as a *multiset* comprehension. Intuitively, $w(C)(\xi(x)) \in W$ captures the overall weight of the block C for a state $x \in X$. The idea is that `init` computes the weight of the whole set X and `update` produces the weights of S and $C \setminus S$ when given the weight of C. The purpose of the set W itself is to save enough information to allow `update` to efficiently compute $F\chi_S^C(\xi(x))$ from a list of edge labels into S.

We say that a refinement interface for the functor F has time complexity $c(m, n)$, if $\texttt{init}(t, \ell)$ and $\texttt{update}(\ell, w)$ run in time $\mathcal{O}(|\ell| \cdot c(m, n))$ for every $t \in F1, w \in W$ and $\ell \in \mathcal{B}_\mathrm{f}A$, and additionally the comparison of two elements $w_1, w_2 \in F3$ or two elements $t_1, t_2 \in F1$ runs in time $\mathcal{O}(c(m, n))$ for every coalgebra $\xi : X \to FX$ with $n = |X|$ and $m = \sum_{x \in X} |\flat \cdot \xi(x)|$.

Example 2.23. (1) For the functor $F = \mathbb{R}^{(-)}$, we have $W = \mathbb{R}^2$ and $w(C)(t)$ carries the accumulated weight of $X \setminus C$ and C, i.e.,

$$w(C)(t) = F\chi_C = \Big(\sum_{x \in X \setminus C} t(x), \sum_{x \in C} t(c)\Big) \in F2.$$

Then the remaining functions are

$$\begin{aligned} \texttt{init}(f_1, e) &= (0, \Sigma e), \\ \texttt{update}(e, (r, c)) &= ((r + c - \Sigma e, \Sigma e), (r, c - \Sigma e, \Sigma e), (\Sigma e + r, c - \Sigma e)), \end{aligned}$$

where $\Sigma\colon \mathcal{B}_\mathrm{f}M \to M$ is the summation of bags defined as $\Sigma(f) = \{f(m) \mid m \in M\}$.

In fact, this refinement interface is not specific to $\mathbb{R}^{(-)}$ but applies to any group-valued functor $G^{(-)}$ for an abelian group $(G, +, 0)$ by putting $W = (G, G)$.

(2) Let F be any of the functors $\mathcal{B}_\mathrm{f}, \mathcal{D}, F_\Sigma$, where F_Σ is the polynomial functor for a finite signature. Then $W = F2$ and $w(C) = F\chi_C\colon FX \to F2$, as for $\mathbb{R}^{(-)}$. Full definitions of the refinement interface of those functors can be found in [WDMS18] and in the source code of CoPaR.

(3) The refinement interface for the powerset functor uses W to count edges into S and C in order to know whether there exist edges into $C \setminus S$, as described by Paige and Tarjan [PT87]. With $w(C)(Y) = (|Y \setminus C|, |C \cap M|)$, we have

$$\begin{aligned} \texttt{init}(f_1, \ell) &= (0, |\ell|), \\ \texttt{update}(\ell, (r, c)) &= \Big((r + c - |\ell|, |\ell|), (r \overset{?}{>} 0, c - |\ell| \overset{?}{>} 0, |\ell| \overset{?}{>} 0), (r + |\ell|, c - |\ell|)\Big), \end{aligned}$$

where the notation $\overset{?}{>}$ is used to test for zeros, i.e., $(x \overset{?}{>} 0) \in 2$ is 0 if $x = 0$ and 1 otherwise.

Proposition 2.24. *All refinement interfaces in Example 2.23 have time complexity* $\mathsf{c}(n, m) = 1$.

This is easy to see for the functors $G^{(-)}, \mathcal{B}_\mathrm{f}, \mathcal{D}$ and $\mathcal{P}$. A proof for the polynomial functor F_Σ is given in [WDMS18, p. 27].

Remark 2.25. In [HMM07], weighted automata with weights from *cancellative* and *non-cancellative* monoids are considered. A commutative monoid $(M, +, 0)$ is called cancellative, if for every $a, b, c \in M$, $(a + c = b + c)$ implies $a = b$. In this case, the refinement interface for groups can still be used since every commutative, cancellative monoid $(M, +, 0)$ embeds into an abelian group by the Grothendieck construction [nLa19] as follows: Define $G(M) = (M \times M)/\!\sim$ as the set of equivalence classes under the equivalence relation

$$((a_+, a_-) \sim (b_+, b_-)) \Leftrightarrow (a_+ + b_- = b_+ + a_-).$$

Then the following group structure defines an abelian group $(G(M), +)$:

$$[a_+, a_-] + [b_+, b_-] = [a_+ + b_+, a_- + b_-], \qquad -[a_+, a_-] = [a_-, a_+].$$

For the non-cancellative case, the following refinement interface can be used, which works for arbitrary commutative monoids:

Example 2.26. For the monoid-valued functor $FX = M^{(X)}$, for an arbitrary commutative monoid $(M, +, 0)$, we take labels $A = M_{\neq 0} \subset M$ being all elements except the unit and define $\flat(f) = \{\, (f(x), x) \mid x \in X, f(x) \neq 0 \,\}$. The refinement interface for F has weights $W = M \times \mathcal{B}_\mathrm{f}(M_{\neq 0})$ and

$$w(C)(f) = \Big(\sum_{x \in X \setminus C} f(x),\ (m \mapsto |\{x \in C \mid f(x) = m\}|)\Big).$$

So every $w(C)(f)$ consists of the accumulated weight of $X \setminus C$ in f and a list of the coefficients of C-elements in the M-valued measure f. The remaining functions are

$$\begin{aligned} &\texttt{init}(f_1, \ell) = (0, \ell) \\ &\texttt{update}(\ell, (r, c)) = ((r + \Sigma(c - \ell), \ell), (r, \Sigma(c - \ell), \Sigma(\ell)), (r + \Sigma(\ell), c - \ell)), \end{aligned}$$

where $\Sigma\colon \mathcal{B}_\mathrm{f} M \to M$ is the previously defined summation map, and we define $a - b$ for bags $a, b \in \mathcal{B}_\mathrm{f} Y$ by $(a - b)(y) = \max(0, a(y) - b(y))$.

Proposition 2.27. *For an arbitrary (possibly infinite) monoid M, the function* $\texttt{update}(\ell, w)$ *defined in Example 2.26 can be computed in $\mathcal{O}(|\ell| \cdot \log \min(|M|, m))$, where m is the total number of edges in the given coalgebra encoding.*

Proof. We implement the bags $\mathcal{B}_\mathrm{f}(M_{\neq 0})$ used in the definition of $W = M \times \mathcal{B}_\mathrm{f}(M_{\neq 0})$ as balanced search trees with keys $M_{\neq 0}$ and values $\mathbb{N}$, where every node additionally stores the sum of its sub-trees. Then, the sum of a bag is immediately available at the root node and all other basic operations have logarithmic time complexity in the bag's size, which is bounded by $\min(|M|, m)$. □

Assumption 2.28. Until the end of this section, we assume that $F\colon \mathsf{Set} \to \mathsf{Set}$ is given together with the encoding for an F-coalgebra $\xi\colon X \to FX$ and a refinement interface that has time complexity $c(|E|, |X|)$, where $|E| = \sum_{x\in X} |\flat \cdot \xi(x)|$.

The following code listings use square bracket notation for array lookups and updates in order to emphasize that they run in constant time. We assume that `graph` and `init` are implemented as arrays and are initialized before the initialization step, e.g., by parsing an input file as described in Section 3.3. Sets and bags can both be implemented as lists, since our code only adds elements to sets not yet containing them.

Our algorithm maintains the following mutable data structures:

- An array `toSub`: $X \to \mathcal{B}_\mathrm{f} E$, temporarily mapping $x \in X$ to a list of its outgoing edges ending in the currently processed subblock during a splitting operation.
- A pointer mapping edges to memory addresses: `lastW`: $E \to \mathbb{N}$.
- A store of memory cells `deref`: $\mathbb{N} \to W$ of type W. This is meant to be indexed by the memory locations returned by `lastW`, i.e., `deref` $\cdot$ `lastW`: $E \to W$.
- A set of marked states (paired with a memory address) $\mathtt{mark}_B \subseteq B \times \mathbb{N}$ for each block B.

The array `toSub` is used only internally in the INITIALIZE and SPLIT operations below while processing a single subblock and is reset after each iteration of the algorithm. The invariant maintained for `lastW` is that for each edge $e\colon x \xrightarrow{a} y$ with $y \in C \in X/Q$, we have that `deref` $\cdot$ `lastW` $= w(C)(\xi(x))$ and any two edges $e_1\colon x \xrightarrow{a_1} y_1$ and $e_2\colon x \xrightarrow{a_2} y_2$, $\mathtt{lastW}[y_1] = \mathtt{lastW}[y_2]$ iff y_1 and y_2 are in the same block of X/Q.

Additionally, the partition X/P is implemented as a refinable partition data structure described in detail in Subsection 3.4.2 that handles marking of states and splitting of blocks efficiently. The partition X/Q is maintained only implicitly in the array `lastW` and in a queue of blocks that need to be split. See Subsection 3.4.1 for details on this.

Remark 2.29. We say that we *group* a finite set Z by $f\colon Z \to Z'$ to indicate that we compute $[-]_f$. This is done by first sorting the elements $z \in Z$ by $f(z)$ using a standard $\mathcal{O}(|Z| \cdot \log |Z|)$ sorting algorithm and then grouping the elements with equal $f(z)$ into blocks. In order to keep the overall complexity of this grouping operation low enough, one needs to compute a possible majority before sorting, following Valmari and Franceschinis [VF10]. The grouping operation itself is implemented as part of the refinable partition presented in Subsection 3.4.2, and Section 3.5 describes an algorithm to compute the possible majority candidate in $\mathcal{O}(|Z|)$ time.

```
INITIALIZE
for $e \in E$, $e = x \xrightarrow{a} y$ do
    add $e$ to toSub$[x]$ and pred$[y]$
for $x \in X$ do
    $p_X$ := new cell in deref containing init(type$[x]$, $\mathcal{B}_\mathrm{f}(\pi_2 \cdot$ graph$)($toSub$[x]))$
    for $e \in$ toSub$[x]$ do lastW$[e] = p_X$
    toSub$[x] := \emptyset$
$X/P$ := group $X$ by type: $X \to F1$
$X/Q := \{X\}$
```

Figure 2.1: The initialization procedure

The initialization procedure of our algorithm is listed in Figure 2.1. It first initializes the arrays `pred` and `toSub` from the coalgebra encoding and then stores $w(X)(\xi(x))$ in `deref` $\cdot$ `lastW`$[x]$

```
SPLIT($S \in X/P$)
 1: $\mathrm{M} := \emptyset \subseteq X/P \times H3$
 2: for $y \in S, e \in$ pred[$y$] do
 3:     $x \xrightarrow{a} y := e$
 4:     $B :=$ block with $x \in B \in X/P$
 5:     if mark$_B$ is empty then
 6:         $w_C^x :=$ deref $\cdot$ lastW[$e$]
 7:         $v_\emptyset := \pi_2 \cdot$ update($\emptyset, w_C^x$)
 8:         add $(B, v_\emptyset)$ to M
 9:     if toSub[$x$] $= \emptyset$ then
10:         add ($x$, lastW[$e$]) to mark$_B$
11:     add $e$ to toSub[$x$]
```

(a) Collecting predecessor blocks

```
12: for $(B, v_\emptyset) \in \mathrm{M}$ do
13:     $B_{\neq\emptyset} := \emptyset \subseteq X \times H3$
14:     for $(x, p_C)$ in mark$_B$ do
15:         $\ell := \mathcal{B}_{\mathrm{f}}(\pi_2 \cdot$ graph)(toSub[$x$])
16:         $(w_S^x, v^x, w_{C\setminus S}^x) :=$ update($\ell$, deref[$p_C$])
17:         deref[$p_C$] $:= w_{C\setminus S}^x$
18:         $p_S :=$ new cell containing $w_S^x$
19:         for $e \in$ toSub[$x$] do lastW[$e$] $:= p_S$
20:         toSub[$x$] $:= \emptyset$
21:         if $v^x \neq v_\emptyset$ then
22:             remove $x$ from $B$
23:             insert $(x, v^x)$ into $B_{\neq\emptyset}$
24:     $B_1 \times \{v_1\}, \dots, B_\ell \times \{v_\ell\} :=$
            group $B_{\neq\emptyset}$ by $\pi_2 \colon X \times H3 \to H3$
25:     insert $B_1, \dots, B_\ell :=$ into $X/P$
```

(b) Splitting predecessor blocks

Figure 2.2: A single refinement step

by calling `init` for each $x \in X$. Finally, it creates X/P by identifying elements with the same $F1$ value as returned by `type`.

Lemma 2.30. *The initialization procedure runs in time $\mathcal{O}((|E| + |X| \cdot \log |X|) \cdot c(|E|, |X|))$.*

Proof. The grouping in line 7 takes $\mathcal{O}(|X| \cdot \log |X| \cdot c(|E|, |X|))$ time, since comparison of $F1$-values has complexity $c(|E|, |X|)$. The first loop has $|E|$ iterations that run in constant time each. Lines 6 and 4 are executed $|X|$ times; the former runs in constant time and the second one in $\mathcal{O}(|\,\mathrm{toSub}[x]| \cdot c(|E|, |X|))$ for each x, since `init`(f, ℓ) is assumed to have complexity $\mathcal{O}(|\ell| \cdot c(|E|, |X|))$, leading to $\mathcal{O}(|E| \cdot c(|E|, |X|) + |X|)$ overall. Finally, the assignment in line 5 runs $|E|$ times in total. □

The algorithm for a single refinement step along $\xi : X \to FX$ is shown in Figure 2.2. It receives as input a subblock $S \in X/P$ with $S \subset C$ for a $C \in X/Q$, splits C into S and $C \setminus S$ and uses `update` to efficiently compute $F\chi_S^C$. This corresponds to steps (2) and (3) of Algorithm 2.9.

In lines 1 to 11, all blocks $B \in X/P$ that have an edge into S are collected into M, together with $v_\emptyset \in F3$, that represents $F\chi_S^C \cdot \xi(x)$ for all $x \in B$ that do not have an edge into S. Also, for each $x \in X$ that has edges into S, those edges e_i are collected into `toSub`$[x]$ and x is added to `mark`$_B$ together with a pointer to $w(C, \xi(x))$, which is available as `lastW`$[e_0]$.

In the second part, each of the previously collected blocks B is split w.r.t. $F\chi_S^C \cdot \xi$. First, for all $x \in \pi_1($`mark`$_B)$, the function `update`$(\ell, w(C)(\xi(x)))$ is called, where ℓ contains the labels of all edges from x to S. This computes $v^x \in F3$ as well as the weights $w(S)(\xi(x))$ and $w(C \setminus S)(\xi(x))$, which are then used to update `lastW`. Since all edges $e\colon x \to y$ with $y \in C$ store $w(C)(\xi(x))$ at the same address `lastW`$[e]$, we only need to write to this memory location to update weights for those edges with $y \in C \setminus S$ and update `lastW` for edges with $y \in S$. Finally, B is split into new blocks w.r.t. v^x.

Theorem 2.31 (Correctness, [WDMS18, Theorem 6.17])**.** *SPLIT computes the correct partitions. That is for $S \in X/P$ and $C \in X/Q$ with $S \subseteq C$, SPLIT(S) refines X/P by $F\chi_S^C \cdot \xi$.*

Lemma 2.32. *For $S_i \subseteq C_i \in X/Q_i$, $0 \leq i < k$, with $2 \cdot |S_i| \leq |C_i|$ and $Q_{i+1} = \ker\langle \kappa_{Q_i}, \chi_{S_i}^{Q_i} \rangle$:*

(1) For each $x \in X$, $|\{ i < k \mid x \in S_i\}| \leq \log_2 |X| + 1$.

(2) $\textsc{split}(S_i)$ for all $0 \leq i < k$ takes at most $\mathcal{O}(|E| \cdot \log |X| \cdot c(|E|, |X|))$ time in total.

Proof. A proof for $c(|E|, |X|) = 1$ is given in [WDMS18, Lemma 6.21], which means that under the assumption that $F1$- and $F3$-values can be compared in constant time and all functions of the refinement interface run in linear time in the number $|\ell|$ of labels passed to them, the overall complexity of split is $\mathcal{O}(|E| \cdot \log |X|)$. Of course, this proof also assumes the usual computational model where each basic machine operation takes constant time.

Instantiating the proof on a machine where each basic operation takes $\mathcal{O}(c(|E|, |X|))$ time, we get a proof of the desired overall complexity of $\mathcal{O}(|E| \cdot \log |X| \cdot c(|E|, |X|))$ under the assumptions on the refinement interface complexity given in Assumption 2.28. □

Bringing Sections 2.3 and 2.4 together, take a coalgebra $\xi\colon X \to FX$ for a zippable Set-Functor F with a refinement interface of time complexity $c(|E|, |X|)$. Replace step (3) of Algorithm 2.9 with

$$X/P_{i+1} = \textsc{split}(S_i). \tag{2.1'}$$

By Theorem 2.31 this is equivalent to (2.1), which is in turn equivalent to the original step (3) by Theorem 2.18. Then by Lemma 2.30 and Lemma 2.32, we have

Theorem 2.33. *The above instance of Algorithm 2.9 computes the quotient modulo behavioral equivalence in time $\mathcal{O}((m + n) \cdot \log n \cdot c(m, n))$, for $m = |E|$ and $n = |X|$.*

2.5 Combining Refinement Interfaces

The instance of Algorithm 2.9 given in Section 2.4 is generic in the zippable functor F, respectively a refinement interface for F. However, to apply the algorithm to a wide range of system types, one has to provide a refinement interface for every individual type functor. For example, coalgebras for $\mathcal{P}_\mathrm{f}$ and $I \times \mathcal{P}_\mathrm{f}(-)$ both correspond to (unlabelled) transition systems, once without and once with an initial partition, but have different refinement interfaces. In addition, zippable functors are not closed under composition, rendering the algorithm unusable for coalgebras of functors that are composed from simple functors, e.g., $\mathcal{P}_\mathrm{f}\mathcal{P}_\mathrm{f}$.

In this section, we show how to transform coalgebras for functors that are composed from simpler basic functors into coalgebras for the coproduct of those basic functors. We also provide a refinement interface for this coproduct, which uses the refinement interfaces of basic functors as subroutines.

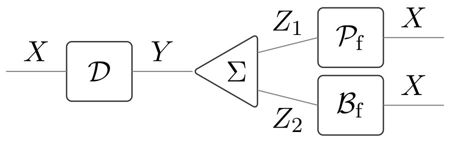


Figure 2.3: Example of a combined functor for $F_\Sigma Z = \mathbb{N} \times Z \times Z$

2.5.1 Explicit Intermediate States

As an example, consider the functor $FX = \mathcal{D}(\mathbb{N} \times \mathcal{P}_\mathrm{f}X \times \mathcal{B}_\mathrm{f}X)$, visualized in Figure 2.3. It can be understood as a composition of basic functors $\mathcal{D}$, $\mathcal{P}_\mathrm{f}$, $\mathcal{B}_\mathrm{f}$ and $F_\Sigma Z = \mathbb{N} \times Z \times Z$, i.e., the nodes in the visualization. Our construction takes coalgebras for the functor F and transforms them into coalgebras for the sum of all basic functors

$$F'X = \mathcal{D}X + \mathcal{P}_\mathrm{f}X + \mathcal{B}_\mathrm{f}X + (\mathbb{N} \times X \times X).$$

This transformation introduces a set of intermediate states, for each non-leaf edge in the above visualization, e.g., for a coalgebra $\xi\colon X \to FX$, we first introduce the set Y for every outgoing $\mathcal{D}$-edge for every $x \in X$, i.e., $Y = \coprod_{x\in X}\{y \mid \mu(x)(y) \neq 0\}$. The successors of Y lie in $\mathbb{N}\times Z_1 \times Z_1$, for the additional intermediate sets Z_1 and Z_2, whose successors in turn lie in $\mathcal{P}_\mathrm{f}X$ and $\mathcal{B}_\mathrm{f}X$ respectively. The latter two do not need additional states, since their successors lie in the original X again. Finally, we can then define the new coalgebra $\xi'\colon X' \to F'X'$, where $X' = X + Y + Z_1 + Z_2$, whose minimization w.r.t F' yields a minimization for the original coalgebra w.r.t F. We refer to the individual sets that make up the coproduct as *sorts*, to the coproduct formation as *desorting* and to the introduction of intermediate states, if the term is unambiguous from context, as *sorting*. The theoretical foundations of this construction and a proof of its correctness are provided in [WDMS18, Section 7].

However, while the above construction is theoretically sound and has no effect on the running time complexity, the introduction of additional states can lead to a substantial blow up of the state space and hence affect the performance and memory consumption of our algorithm. Fortunately, this can be partially avoided by flattening out polynomial parts of the functor term and thus avoiding the corresponding intermediate states. In the above example, the functor is transformed to $\mathcal{D}X + (\mathbb{N} \times \mathcal{P}_\mathrm{f}X \times \mathcal{B}_\mathrm{f}X)$, which avoids the intermediate state sets Z_1 and Z_2 but still needs the set Y. Section 3.5 describes how this additional transformation step is realized in our implementation and Section 4.3 provides benchmarks for its performance benefits.

Example 2.34. Hopcroft's classical automata minimization algorithm [Hop71] is obtained by instantiating our algorithm for the functor $FX = 2\times X^A$ for fixed input alphabet A with running time $\mathcal{O}(n\cdot\log n)$. For non-fixed A, the best known complexity is $\mathcal{O}(|A|\cdot n\cdot\log n)$ [Gri73, Knu01]. To regard the alphabet as part of the input instead of the functor, we encode the letters of A as natural numbers and consider coalgebras for the functor $FX = 2 \times \mathcal{P}_\mathrm{f}(\mathbb{N} \times X)$.

Using the above construction, we decompose the type functor into $F_1 = 2\times\mathcal{P}_\mathrm{f}$ and $F_2 = A\times(-)$ and finally recombine them to $F' = F_1 + F_2$. To transform an F-coalgebra $\xi\colon X \to FX$ into an F'-coalgebra, we need to introduce an additional set Y of intermediate states, containing one state for each outgoing edge of each $x \in X$.

With $|X| = n$, the resulting system has $n \cdot |A|$ states and $n \cdot |A| + n \cdot |A|$ edges. Thus, the running time becomes $\mathcal{O}((|A| \cdot n) \cdot \log(|A| \cdot n))$.

2.5.2 Coproducts of Refinement Interfaces

Let $F_i : \mathsf{Set} \to \mathsf{Set}$, $i \in I = \{1, \dots, n\}$ be zippable functors with encoding $(A_i, \flat_i)$ and refinement interfaces with weights W_i and associated functions w_i, $\mathtt{init}_i$ and $\mathtt{update}_i$. By Lemma 2.14, the coproduct $F = \coprod_{i\in I} F_i$ is also zippable. We proceed to show that we can also construct a refinement interface for F.

First, we define an encoding for F by taking the set of labels $A = \coprod\limits_{i\in I} A_i$ and (writing $\coprod$ for $\coprod_{i\in I}$):

$$\flat = (\coprod F_iX \xrightarrow{\coprod \flat_i} \coprod \mathcal{B}_\mathrm{f}(A_i \times X) \xrightarrow{[\mathcal{B}_\mathrm{f}(\mathrm{in}_i \times X)]_{i\in I}} \mathcal{B}_\mathrm{f}(\coprod A_i \times X)).$$

For the refinement interface, we need the following helper function, which filters a bag of labels $\coprod_{j=1}^n A_j$ for a particular type of label A_i:

$$\mathrm{filter}_i\colon \mathcal{B}_\mathrm{f}(\coprod_{j=1}^{n} A_j) \to \mathcal{B}_\mathrm{f}A_i \qquad\qquad \mathrm{filter}_i(f)(a) = f(\mathrm{in}_i(a)).$$

A refinement interface is then given by taking $W = \coprod_{i \in I} W_i$ and defining the associated functions as follows:

$$\begin{aligned} w(S) &= \textstyle\coprod_{i \in I} w_i(S), \\ \texttt{init}(\mathrm{in}_i\, f_1, \ell) &= \mathrm{in}_i\, \texttt{init}_i(f_1, \mathrm{filter}_i(\ell)), \\ \texttt{update}(\ell, \mathrm{in}_i\, w) &= (\mathrm{in}_i \times \mathrm{in}_i \times \mathrm{in}_i)(\texttt{update}_i(\mathrm{filter}_i(\ell), w)). \end{aligned}$$

Proposition 2.35. *If the individual refinement interfaces for F_i have time complexity $c_i(m, n)$, then the data $A, \flat, W, w,$ `init` and `update` as constructed above forms a refinement interface with time complexity $\max_{i \in I}(c_i(m, n))$.*

Proof. A proof that this data satisfies the coherence condition of Definition 2.22 and therefore forms a refinement interface is given in [WDMS18, p. 41]. It remains to show its time complexity.

Both `init` and `update` preprocess their parameters in linear time (via filter) before calling $\texttt{init}_i$ and $\texttt{update}_i$ of the corresponding F_i. Therefore, if $\texttt{init}_i(f_1, \ell_i)$ takes $\mathcal{O}(|\ell_i| \cdot c_i(m, n))$ time, $\texttt{init}(\mathrm{in}_i\, f_1, \ell)$ takes $\mathcal{O}(|\ell| + |\ell_i| \cdot c_i(m, n)) = \mathcal{O}(|\ell| \cdot c_i(m, n))$, since $|\ell_i| \leq |\ell|$. Maximizing over all possible $c_i(m, n)$ we get the desired result. The proof for `update` works analogously.

We order $F3$- and $F1$-values lexicographically, i.e., $\mathrm{in}_i(x) < \mathrm{in}_j(y)$ iff either $i < j$ or $i = j$ and $x < y$. This comparison then clearly runs in $\mathcal{O}(\max c_i(m, n))$, if the individual comparisons run in $\mathcal{O}(c_i(m, n))$. □

3 Implementation

The implementation of the algorithm described above is written in Haskell, with a focus on being modular and generic. It is implemented as a standalone program that can do partition refinement for all transition system types listed in Example 2.4 and many more. The system type is defined at runtime as part of the program input and the system itself is specified in an intuitive syntax, inspired by the formal notation for the chosen type functor.

Despite having a neat input syntax and being able to handle complex system types reasonably efficiently (see Chapter 4), the implementation is still generic and modular just like the algorithm given in Section 2.4. Basic functors can be implemented independently of the core algorithm and functor composition is handled automatically, following Section 2.5.

This chapter gives an overview over the architecture of the implementation, discusses some design decisions made in the process of writing it and examines a few aspects in detail such as input file parsing and key data structures.

3.1 Functors and the Refinement Interface

A basic functor F in CoPaR is defined as a Haskell type of kind `* -> *` that implements a variety of type classes. Although `Functor` is one of those classes, the semantics of the `Functor` implementation in Haskell does not generally coincide with the functor F. Instead, the type `F` serves as a mere tag for type class instance selection and as a means to build heterogeneous functor expressions, as described in Section 3.2.

The actual algorithm is implemented abstractly in the choice of functor, just like the pseudocode formulation is, requiring only that it implements a type class `RefinementInterface` which is a direct translation of the theoretical concepts presented in Definition 2.20 and Definition 2.22 and is reproduced below.

```
class (Ord (F1 f), Ord (F3 f)) ⇒ RefinementInterface f where
  init   :: F1 f → [Label f] → Weight f
  update :: [Label f] → Weight f → (Weight f, F3 f, Weight f)

type family Label (f :: ∗ → ∗) :: ∗
type family Weight (f :: ∗ → ∗) :: ∗
type family F1 (f :: ∗ → ∗) :: ∗
type family F3 (f :: ∗ → ∗) :: ∗
```

Listing 3.1: Refinement Interface[1].

Apart from the type class itself this also defines the associated types `F1`, `F3`, `Label` and `Weight` as type families corresponding to the sets $F1$, $F3$, A and W from Definition 2.20 and 2.22 respectively. Since open type families in Haskell are not injective and `init` as well as `update` mention the type variable `f` only under type family applications, call sites for these functions have to employ either explicit type annotations or visual type application [EWA16]. This

[1]From file `src/Copar/RefinementInterface.hs` in the source code

reaffirms the fact that the functor type itself is just used as tag for the instance selection of the Haskell compiler.

As an example, consider the following implementation of the refinement interface for the powerset functor, as defined in Example 2.23 (3):

```
type instance F1 Powerset = Bool
type instance F3 Powerset = (Bool, Bool, Bool)
type instance Label Powerset = ()
type instance Weight Powerset = (Int, Int)

instance RefinementInterface Powerset where
  init _ labels = (0, length labels)
  update labels (toRest, toC) =
    let toS = length labels
        f3 = (toRest > 0, toC > toS, toS > 0)
    in ((toC + toRest, toS), f3, (toS + toRest, toC − toS))
```

Listing 3.2: Refinement Interface implementation for $\mathcal{P}_\mathrm{f}$

This implementation is pleasantly similar to the mathematical notation used in Example 2.23 and can be easily derived from the theory, even without proficiency in the Haskell language[2]. Note that the actual source code for the powerset functor in CoPaR[3] is highly optimized for performance and therefore not as easy to read.

3.2 Heterogeneous Functor Expressions

After having seen how the type class `RefinementInterface` is defined and implemented for basic functors, we need to substantiate the claim that we support arbitrary compositions of such functors. To do this, we translate the construction from Section 2.5 into Haskell by employing a mix of different techniques from the Haskell folklore: Extensible recursive data using a fixpoint-of-a-functor technique and existential types to model open sums.

Recall from Section 2.5 that we start out with a nested term involving different functors of potentially different arity. Representing this as a data type in an extensible way that allows more functors to be defined later without changing the data type itself, is an instance of the expression problem [Wad98]. Our solution takes the fixpoint-of-a-functor trick of [Swi08], but replaces the subtyping based open union with an existential type:

```
data FunctorExpression f a
  = Functor a (f (FunctorExpression f a))
  | Variable

data SomeFunctor a
  = ∀ f. (RefinementInterface f, ...) ⇒ SomeFunctor (f a)
```

Listing 3.3: Definition of `SomeFunctor` type[4]

This augments the fixpoint type `FunctorExpression` with an additional type variable `a` that can be used as a placeholder for arbitrary annotations in the AST. Using the existential type

[2]See the file `doc/HOWTO_IMPLEMENT_A_FUNCTOR.org` in CoPaR's source for a gentle introduction on how to write `RefinementInterface` instances.

[3]In the file `src/Copar/Functors/Powerset.hs`.

[4]From file `src/Copar/Functors/SomeFunctor.hs` in the source

`SomeFunctor` as an open sum over basic functors has less performance overhead than Swierstra's approach and does not require the `OverlappingInstances` GHC extension, but gives up the concept of extensible interpreters. Basically, we have to mention every typeclass that we may ever need implemented for `f` in the constructor definition of `SomeFunctor`, and these classes currently include `RefinementInterface`, `PrettyShow` and many others. While this has proven to be annoying, it does not outweigh the cost of runtime sum indexing.

To obtain a refinement interface implementation for the whole functor expression, we wrap each individual functor type in an annotation containing a fresh sort number per functor and finally recombine the individual refinement interfaces in the implementation of `RefinementInterface` for `SomeFunctor`. This closely resembles the sum of refinement interfaces described in Section 2.5, and as such, requires new existential types `SomeF1`, `SomeF3`, `SomeLabel` and `SomeWeight`, that carry additional runtime type information to be able to implement `init` and `update`, the former of which is shown below.

```
init (SomeF1 (f :: TypeRep tf) f1) labels =
  let myLabels = mapMaybe isSameType labels
  in SomeWeight f (init @tf f1 myLabels)

  where
    isSameType :: SomeLabel → Maybe (Label tf)
    isSameType (SomeLabel f2 l) = case eqTypeRep f f2 of
      Nothing → Nothing
      Just HRefl → Just l
```

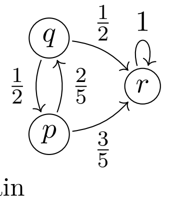
Listing 3.4: `init` implementation for `SomeFunctor`[5]

`update` is defined analogously. These runtime type checks are necessary from an isolated perspective of the `RefinementInterface`, since Haskell's type system does not prevent a caller from calling `init` with a list of labels from different basic functors. But our implementation of Algorithm 2.9 does not do this, and consequently, the checks can be disabled at compile time to gain some performance.

3.3 Parsing

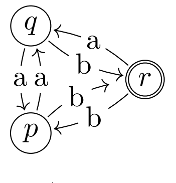
The first step of the minimization implementation is to read a text file and parse its contents representing a coalgebra $\xi\colon X \to FX$ from a custom syntax to an in-memory representation of the coalgebra encoding described in Section 2.4. The input syntax has two separate parts. The first part specifies the functor F on a single line and the remaining lines each describe a single state $x \in X$ and its behavior $\xi(x)$. Examples of input files are shown in Figure 3.5 and the following paragraphs describe the syntax in detail.

```
DX

q: {p: 0.5, r: 0.5}
p: {q: 0.4, r: 0.6}
r: {r: 1}
```



(a) Markov chain

```
{f,n} x X^{a,b}

q: (n, {a: p, b: r})
p: (n, {a: q, b: r})
r: (f, {a: q, b: p})
```



(b) Deterministic finite automaton

Figure 3.5: Examples of input files with encoded coalgebras

[5]From file `src/Copar/Functors/SomeFunctor.hs` in the source

3.3.1 Functor Syntax

The first line of the input file (or the value of the command line argument `--functor`) contains a functor expression defined abstractly by the grammar

$$T ::= \mathtt{X} \mid F(T,\dots,T) \qquad (F\colon \mathsf{Set}^k \to \mathsf{Set}) \in \mathcal{F}, \tag{3.1}$$

where X is a terminal symbol and $\mathcal{F}$ is a set of predefined symbols for the basic functors of the form $F\colon \mathsf{Set}^k \to \mathsf{Set}$. This presentation highlights the fact that as with refinement interfaces, only the syntax of individual basic functors has to be implemented explicitly and the composition is derived automatically. Including the syntax for all currently implemented basic functors, the complete grammar is effectively

$$\begin{aligned}
&T ::= \mathtt{X} \mid \mathcal{P}_{\mathrm{f}}\,T \mid \mathcal{B}_{\mathrm{f}}\,T \mid \mathcal{D}\,T \mid M^{(T)} \mid \overbrace{C \mid T+T \mid T\times T \mid T^A}^{\text{Polynomial constructs}},\\
&C ::= \mathbb{N} \mid A \qquad A ::= \{s_1,\dots,s_n\},\\
&M ::= \mathbb{N} \mid \mathbb{R} \mid \mathbb{C} \mid \mathbb{Q} \mid (\mathbb{Z},\max) \mid (\mathbb{R},\max),
\end{aligned}$$

where the s_i are strings subject to the usual conventions of C-style identifiers, exponents T^A are written `T^A` and M is one of the monoids $(\mathbb{N},+)$, $(\mathbb{R},+)$, $(\mathbb{C},+)$, $(\mathbb{Q},+)$, $(\mathbb{N},\max)$ and $(\mathbb{R},\max)$ the latter two being the additive monoid of the tropical semiring. Note that C effectively ranges over at most countable sets and A over finite sets. The polynomial constructs highlighted above all belong to the same polynomial functor, which can have multiple arguments and the operations $\times$ and $+$ are written infix.

3.3.2 Coalgebra Syntax

After the functor F is determined in the first line of the input, the remaining lines constitute the given F-coalgebra. Each of these lines defines a state $x \in X$ and its behavior $\xi(x) \in FX$ with the syntax `x: t`, where `x` is a C-style identifier that defines a name for x and `t` represents $\xi(x)$. The syntax for `t` follows the structure of the specified functor F inductively as follows: For the base case `X`, t can be one of the defined names for the elements of X. In the inductive step $F(T)$, where F is one of the implemented basic functors $\mathcal{F}$, the syntax depends on the functor. The syntax for the currently implemented functors is described in detail below and a brief overview is given in tabular form in Table 3.6.

- For $F = \mathcal{P}_{\mathrm{f}}(T)$ or $F = \mathcal{B}_{\mathrm{f}}(T)$, `t` is in standard braces notation $\{t_1,t_2,\dots,t_n\}$ for $t_1,t_2 \in T$. The difference is only in whether they check for duplicate elements.
- For $F = T_1 \times T_2$, the syntax is (t_1,t_2) for $t_1 \in T_1, t_2 \in T_2$. For $F = T_1 + T_2$, we use `inj0` t and `inj1` t for $t \in T_1$ and $t \in T_2$ respectively.
- If F is T^A, we use map notation $\{\, a_1 : t_1, a_2 : t_2, \dots, a_n : t_n \,\}$, where $t_i \in T$ and $a_i \in A$, which uses the same syntax for constants as $T = C$.
- For $F = M^{(T)}$, we use the same map notation as for T^A with the tuples reversed, that is $\{\, t_1 : m_1, t_2 : m_2, \dots, t_n : m_n \,\}$ with $t_i \in T$ and $m_i \in M$. The exact syntax of the m_i depends on M: For $M = \mathbb{N}$ and $M = \mathbb{R}$ we use the same syntax as for constants, for $M = \mathbb{C}$, we use standard notation for complex numbers $a + bi$ and for $M = (\mathbb{N},\max)$ the syntax for m_i is the same as for $\mathbb{N}$, just the semantics is different.
- For $F = \mathcal{D}(T)$ the term syntax is the same as for $\mathbb{R}^{(T)}$.
- Finally, for the constant functors $F = \mathbb{R}$, $F = \mathbb{N}$ and $F = N \in \mathbb{N}$ the term syntax is just the usual notation for elements of the respective set, e.g., $0.23 \in \mathbb{R}$, $5 \in \mathbb{N}$ and $5 \in \mathbf{6}$.

Explicit constant sets $A = \{ a_1, a_2, \dots, a_n \}$ have the C-style identifiers a_i as coalgebra syntax.

Functor	Syntax
X	Name $x \in X$
$\mathcal{P}_\mathrm{f}(T), \mathcal{B}_\mathrm{f}(T)$	$\{ t_1, t_2, \dots \}$
$\mathcal{D}(T)$	$\{ t_1 \colon w_1, t_2 \colon w_2, \dots \}, \quad w_i \in [0,1] \subset \mathbb{R}, \sum w_i = 1, t_i \in T$
$T_1 \times T_2$	(t_1, t_2)
$T_1 + T_2$	`inj` i␣$t_i, \quad i \in \{ 1, 2 \}$
C	$c, \quad c \in C$
T^A	$\{ a_1 \colon t_1, a_2 \colon t_2, \dots \}, \quad a_i \in A, t_i \in T$
$M^{(T)}$	$\{ t_1 \colon m_1, t_2 \colon m_2, \dots \}, \quad m_i \in M, t_i \in T$

Table 3.6: Coalgebra Syntax

The polynomial constructs $\times$ and $+$ extend to higher arities $T_1 \times \cdots \times T_n$ and $T_1 + \cdots + T_n$. Also, many functor implementations perform additional sanity checks on the input beyond syntax. For example, all functors with map notation $\{ x : a, \dots \}$ check that the resulting map is total and the powerset functor obviously does not allow duplicate successors. These sanity checks can be optionally disabled for some performance gains if the input is known to be well-formed.

As an example input, the coalgebra for the functor $\mathbb{N} + \mathcal{P}_\mathrm{f}(\{a, b\} \times \mathbb{R}^{(X)})$ defined by the input file

```
x: inj1 {(a, {x: 0.5, y: 1.7}), (b, {})}
y: inj0 4     # chosen by fair dice roll
```

has two states x and y. The latter state y has a constant label of 4 with no successors and x has one a- and one b-successor, which are both $\mathbb{R}$-valued measures on X. The a-successor assigns weights 0.5 to x and 1.7 to y, while the b-successor is the zero measure. Figure 3.5 contains two more examples.

3.4 Required Data Structures

The complexity result for Algorithm 2.9 given in Theorem 2.33 relies heavily on the good performance of a few key data structures that we have only glossed over so far, in particular a queue of subblocks and a refinable partition data structure. Both of these data structures are also described by Valmari and Franceschinis in [VF10] and this section follows their presentation, adapted to our setting.

3.4.1 Main Loop and Queue

Step (1) of Algorithm 2.9 picks a subblock $S \in X/P$ that is subsequently used to split its containing compound block $C \in X/Q$. We require that S fulfills the property $2 \cdot |S| \leq |C|$, but we have not yet made precise how to find such a block. The actual implementation differs a little from the above condition and is both simpler to implement and more difficult to formalize. In this section, we will first explain the implementation and then show that it does in fact have the same properties as the condition outlined above.

The main loop of our algorithm as shown in Figure 3.7 maintains a FIFO-queue of subblocks `W`. After the initial partition X/P_0 is computed in line 1, `W` is initialized to contain all blocks of

```
1: INITIALIZE
2: W := $X/P$ except a largest block
3: while W is not empty do
4:     Remove $S$ from W
5:     Split $C$ in $X/Q$ into $S$ and $C \setminus S$
6:     SPLIT($S \subseteq C$)                         ▷ This adds new blocks to W
```

Figure 3.7: Main Loop of Implementation

X/P_0 except a largest one in line 2. The algorithm then proceeds to remove a block S from `W` and calls SPLIT with S until `W` is empty. SPLIT itself is modified to insert new blocks $B_1, \ldots, B_n$, which result from splitting a block B, into `W` subject to the following conditions:

(1) If $B \in$ `W`, replace B in `W` with $B_1, \ldots, B_n$.

(2) If $B \notin$ `W`, insert $B_1, \ldots, B_n$ except a largest one into `W`.

Since the maximum number of blocks for each coalgebra is just the number of states $|X|$ and thus known after parsing, the queue can be implemented as a ring buffer of size $|X|$, leading to efficient queue operations. Also, the membership tests in (1) and (2) above can be made to run in constant time with a bit array. We now show that this queue implementation is correct in the context of Algorithm 2.9.

Proposition 3.1. *Substituting the subblock selection step of Algorithm 2.9 with the above queue implementation does not affect correctness.*

Proof. Since the condition $2 \cdot |S| \leq |C|$ is not necessary for correctness [WDMS18], we only need to show that `W` indeed selects a proper subblock $S \in X/P$ iff such a block is available. It is obvious that the queue can only contain elements of X/P. To show that it contains at least one proper subblock if $X/P \neq X/Q$, we establish the following invariant:

For each $C \in X/Q$, `W` contains all but exactly one of the subblocks S with $S \subseteq C$.

This invariant holds by construction after line 2 of the main loop. It also holds after line 5 for the two new compound blocks S and $C \setminus S$: By the invariant, there exists a subblock $S' \subseteq C$ with $S' \neq S$ that is not in `W` and therefore $S' \subseteq C \setminus S$ holds as required. In addition, S has only one trivial subblock, S itself, which is removed from the queue in line 4. After line 6 the invariant also holds: For each block $B \in X/P$ that is split into $B_1, \ldots, B_n$ by SPLIT, let C_B be the compound block containing B. If B was the one subblock of C not in the queue, exactly one of the $B_1, \ldots, B_n$ is also not put into `W` by the above condition (1), thereby maintaining the invariant. If B was in `W` before being split, it is replaced by all the B_i due to condition (2). Thus, the invariant holds.

It follows from the invariant that the queue always contains a proper subblock, except if each compound block consists of only one subblock, in which case we have $X/P = X/Q$ and the queue is empty, which is our termination condition. □

Remarkably, our queue implementation does not satisfy the condition that $2 \cdot |S| \leq |C|$ when a block S is removed from the queue in line 4. Consider, for example, the following situation:

$$C = \{\overbrace{1, 2, 3}^{B_1}, \overbrace{4, 5, 6}^{B_2}\} \qquad B_1 \in W,\ B_2 \notin W$$

Here, the compound block C contains two subblocks B_1 and B_2 where B_1 is in the queue and B_2 is not. Now suppose B_2 is split into three single-element subblocks B_{21}, B_{22} and B_{23}. Since at this point B_2 is not in W, only two of the subblocks are put into the queue. Now suppose B_1

is also split into $B_{11} = \{1\}$ and $B_{12} = \{2,3\}$ and both of them are added to W. If the queue maintains strict FIFO-order, at this point it contains the elements $B_{21}, B_{22}, B_{11}, B_{12}$ in order. If no further splittings of these blocks occur, and they are removed successively from W, the compound block C shrinks by one element every time one of those blocks is removed. By the time $B_{12} = \{2,3\}$ is removed, its compound block C' is $\{2,3,6\}$ and therefore $2 \cdot |B_{12}| > |C'|$.

Nevertheless, we have the following

Proposition 3.2. *Substituting the subblock selection step of Algorithm 2.9 with the above queue implementation does not affect its running time complexity.*

Proof. The complexity proof for Algorithm 2.9 given in Section 2.4 and [WDMS18] relies on Lemma 2.32(1), which states that for each of the $S_i,\ 0 \leq i < k$ that are selected in step (1) of the algorithm and each $x \in X$, we have that $|\{\, i < k \mid x \in S_i\}| \leq \log_2 |X| + 1$.

We now proof this exact statement for our queue implementation without relying on the assumption that $2 \cdot |S_i| \leq |C_i|$.

Assume that a block S_i is extracted from `W` and passed to SPLIT, and that later a block $S_j \subseteq S_i$ is the one that will be removed next from the queue. We know that S_j has been created by successive splittings $S_i = S_{i_1} \supseteq \ldots \supseteq S_{i_n} = S_j$ with $n > 1$. Since we originally removed S_i from `W` but S_j is in the queue again, it follows that at some point an S_{i_k} has been put into the queue where $S_{i_{k-1}}$ was not in the queue. In this case we have $2 \cdot |S_{i_k}| \leq |S_{i_{k-1}}|$ by condition (2) and thus $2 \cdot |S_j| \leq 2 \cdot |S_{i_k}| \leq |S_{i_{k-1}}| \leq |S_i|$. This implies for each $x \in X$ that each time a block S with $x \in S$ is removed from the queue and passed to split, that block is only half as large as the previous time. Thus, a state x can only appear in such a block $\log_2 |X| + 1$ number of times. $\square$

3.4.2 Refinable Partition

As blocks of states are gradually being split into smaller blocks throughout the course of our algorithm, we need to keep track of the current partition X/P. In Algorithm 2.9, each iteration of the main loop produces a new partition X/P_{i+1}, but creating a new data structure each time would be entirely too wasteful in an actual implementation. Instead, we use a mutable data structure called *refinable partition* that maintains a partition of states into blocks and supports efficient in-place splitting of those blocks. This block splitting must additionally be exposed in two different interfaces. First, it must be possible to flag individual states as *marked* to later split blocks into marked and unmarked states. Secondly, we need to be able to group all states of a block according to some predicate P and create new blocks for the equivalence classes that P induces. All in all, we require the following operations and their worst-case running time complexities:

mark(s) Mark a state s for later splitting in $\mathcal{O}(1)$.

split(B) Split a block B into one block B_m containing all marked states of block B and one block B_u with all unmarked states and subsequently unmark all states in B_u. It is important for the good performance of the algorithm that this operation runs in $\mathcal{O}(|B_m|)$, meaning that the running time must not depend on $|B|$.

groupBy(B, P) Group a block B by $P \colon X \to Z$ as detailed in Remark 2.29 with running time $\mathcal{O}(|B| \cdot \log |B| \cdot c)$, provided the comparison on Z takes $\mathcal{O}(c)$ time.

blockOf(s) Find the block that the state s belongs to in constant time.

statesOf(B) Return the list of states in block B in $\mathcal{O}(|B|)$.

There are several ways to implement such a data structure, for example with two doubly linked lists for the marked and unmarked states for each block [AHU74]. Instead, our implementation is based on mutable arrays and is presented in detail in [VF10] and [Val10].

Throughout our implementation, states and blocks are both represented as integers counting from zero which enables their use as array indices. In the following section, we will use lowercase letters for states, blocks and their representations interchangeably. Our refinable partition data structure consists of the three arrays of size $|X|$ listed below.

elems is an array of state numbers. It is ordered such that all states belonging to the same block are adjacent.

states contains an entry for each state s at $\mathit{states}[s]$. Its entries are structures with two fields: A field *loc* that contains the location of a state in the *elems* array, i.e., $\mathit{elems}[\mathit{states}[s].\mathit{loc}] = s$. The other field *block* contains the number of the block that the state belongs to.

blocks contains an entry for each block b at $\mathit{blocks}[b]$ of structural type with the fields *start*, *unmarked* and *end* containing indices into *elems*. For ease of reading, we will use the notation field_b instead of $\mathit{blocks}[b].\mathit{field}$ in the following paragraphs. The two indices start_b and end_b mark a range inside *elems* that contain the elements of block b, i.e., the elements of b are

$$\mathit{elems}[\mathit{start}_b], \mathit{elems}[\mathit{start}_b + 1], \ldots, \mathit{elems}[\mathit{end}_b - 1].$$

Like the other two arrays, *blocks* is of size $|X|$, even though the number of blocks may be much smaller. In contrast to states, the block count is not fixed after parsing, and so we use the maximum number of blocks possible – one block for every state. Unused array entries are simply left uninitialized.

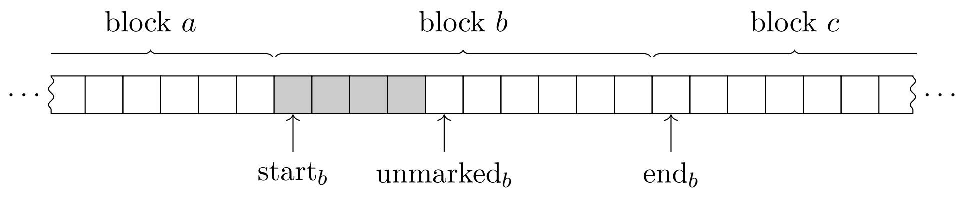


Figure 3.8: Array of states of refinable partition data structure. Marked states are shaded gray.

Figure 3.8 shows how the states of block b would be laid out in the *elems* array with the three fields of the block data structure for b displayed as arrows and marked states shaded gray. The names of the adjacent blocks a and c highlight the fact that neither states nor block numbers appear in sequence in the *elems* array.

With the data structure in place, the individual operations are rather simple to implement:

mark(s) first retrieves the position of state s in the *elems* array and the block b that s belongs to by reading the states array entry at index s. It can then optionally check if the state is already marked by comparing its position with the $\mathit{unmarked}_b$ offset. This is not strictly required if our algorithm is implemented carefully since a single state never has to be marked twice without being unmarked in between, but can nevertheless be good programming practice. If the state is unmarked, it can now be marked by swapping it with the first unmarked state of b in the *elems* array, and incrementing the value of $\mathit{unmarked}_b$.

split(b) splits the block b at its unmarked offset. This entails creating a new block b' with $\mathit{start}_{b'}$ set to $\mathit{unmarked}_b$ and $\mathit{end}_{b'}$ set to end_b. The old block b needs to be shrunk by setting end_b to $\mathit{unmarked}_b$ and finally, the unmarked fields of both blocks must be reset to the value of start, which unmarks all states in both blocks.

groupBy(b, P) simply sorts the *elems* array between $start_b$ and end_b according to P with a standard $\mathcal{O}(n \cdot \log n)$ sorting algorithm and subsequently splits the block into the equivalence classes of P. This can be done conveniently by calling *mark* and *split* repeatedly.

The remaining operations *blockOf* and *statesOf* are implemented as simple array lookups in *blocks* and *states*.

3.5 Details of the Algorithm Implementation

This section presents a few minor details that may not seem all that important but are nevertheless vital to understand our implementation.

Possible majority candidate A possible majority candidate (pmc) of a non-empty sequence A is the value of the element that occupies more than half of the places in A or, if no such value exists, any value from A. This value can be calculated in $\mathcal{O}(|A|)$ with the Boyer–Moore majority vote algorithm [BM91], which is displayed as pseudo-code in Figure 3.9. To see that it works, let $x \in A$ and $c(x)$ be a value defined at each step of the algorithm to be k if $cand = x$ and $-k$ otherwise. If $A[i] = x$ in step i of the algorithm, then $c(x)$ increases independently of the value of *cand* and if $A[i] \neq x$, $c(x)$ either increases or decreases. Thus, if x occupies more than half of the positions, $c(x)$ increases more often than it decreases, implying that at the end of the algorithm $c(x) > 0$ and therefore $cand = x$.

```
1:  $k := 0$
2:  **for** $x \in A$ **do**
3:      **if** $k = 0$ **then**
4:          $k := 1$
5:          $cand := x$
6:      **else if** $x = cand$ **then**
7:          $k := k + 1$
8:      **else**
9:          $k := k - 1$
10: **return** $cand$
```

Figure 3.9: Boyer-Moore majority vote algorithm

Skipping blocks of one element Our implementation spends a large chunk of its processing time splitting individual blocks by calling `update` from the refinement interface and `split` or `groupBy` on the refinable partition data structure. For blocks of exactly one element, these operations all run in $\mathcal{O}(1)$ time and ultimately do not have any visible effect, since those blocks cannot be split any further, but induce a significant constant overhead. Detecting this condition early and not even attempting to split the block has thus proven to be beneficial for our implementation on some inputs. Especially coalgebras which are already minimal benefit the most, since in this case all blocks are split maximally often until only one-element blocks remain. These inputs would normally be the ones where our algorithm needs the most iterations. Benchmarks of the impact of this optimization on different input files are given in Section 4.3.

Mitigating the state space blow-up One reason why our implementation lacks behind the performance of specialized partition refinement tools tailored to a specific system type is the proliferation of auxiliary states that arise from our approach to modularity outlined in Section 2.5. For example, coalgebras for the commonly used functor $I \times \mathcal{P}_\mathrm{f}(-)$ are transformed into coalgebras for $(I \times (-) + \mathcal{P}_\mathrm{f})$ that have twice as many states as the original coalgebra. As mentioned in Section 2.5, we can avoid this state space blow-up in some cases by flattening out polynomial parts of the functor.

Concretely, this flattening is implemented as a post-processing step that is applied after parsing the functor expression but before parsing the coalgebra. It traverses the functor expression tree described in Section 3.1 in a bottom up manner, recursively replacing every instance of the polynomial functor together with its immediate children with a new type `AbsorbingPolynomial` as shown in Figure 3.10. The coalgebra parser for this type then manually calls the parsers of its sub-expressions and therefore creates only a single state in the coalgebra in each parsing step instead of relying on the usual parsing procedure to create new states for each sort. The refinement interface is handled similarly by manually calling `init` and `update` of the inner functors, much like the coproduct described in Subsection 2.5.2.

$$\begin{tikzcd}
A \times (-) \times (-) \arrow[d, no head] \arrow[dr, no head] & & & A \times \mathcal{P}_\mathrm{f}(-) \times \mathcal{D}(-) \arrow[d, no head] \\
\mathcal{P}_\mathrm{f} \arrow[d, no head] & \mathcal{D} & \Rightarrow & B \times (-) \\
B \times (-) & & &
\end{tikzcd}$$

Figure 3.10: Flattening out polynomials in $FX = A \times \mathcal{P}_\mathrm{f}(B \times X) \times \mathcal{D}X$

4 Benchmarking and Evaluation

Beware of slow code. I have only proved it asymptotically optimal, not run it.
— Donald Knuth, paraphrased

It is not obvious how to verify the correctness of the results that our tool produces, especially for the wide range of supported systems. To increase the confidence in our implementation, we have included an extensive suite of unit and integration tests as well as micro benchmarks in the source code. Assessing the running time performance of the whole tool has its own challenges. For many of our supported system types, no specialized partition refinement software exists and if it exists, it is bound to beat our tool in performance due to the overhead of generality and modularity on our side. We have nevertheless compared the running time of our implementation with another specialized refinement tool by Antti Valmari. This section presents the testing environment, describes our two benchmark suites and discusses the results.[1]

All benchmarks below were executed on an Intel® Core™ i5-6500 processor with 3.20 GHz clock rate and 16 GiB of RAM. For large inputs, memory usage quickly becomes the limiting factor and for this reason, all timing results in this chapter are in the range of a few seconds to minutes. The two benchmarking subjects are our own refinement tool CoPaR, compiled with the GHC compiler version 8.4.4 and Valmari's tool written in C++, compiled with GCC 8.2.1. All timing results are given in seconds.

We have created two collections of benchmarks, one adapted from the PRISM [KNP11] model checker and one with randomly generated deterministic finite automata. We will next describe those benchmark suites, present and discuss the results and finally evaluate the impact of some specific code optimizations in our tool.

4.1 PRISM Benchmark Suite

PRISM is a probabilistic model checker with a large suite of published models for real-world case studies that we can adapt to our setting. The model files and PRISM code that we use are extracted from the source of PRISM 4.4. We will now first describe how to extract coalgebras from those models, which can then also be translated into Valmari's input format.

PRISM supports four model types, Markov decision processes (MDPs), Discrete-time Markov chains (DTMCs), Continuous-time Markov chains (CTMCs) and probabilistic timed automata (PTAs), an extension to MDPs with real-time behavior. The source code of CoPaR includes a translation tool which converts models of all types except PTAs into coalgebras. The objective of our translations is to generate a set of large enough inputs for our implementation with some non-uniformity to them. In particular, that we do not cover all features of those supported system types and choose arbitrary initial partitions, which means that the minimized coalgebras do not necessarily have any useful properties. To convert models to coalgebras, we first use PRISM

[1]More detailed results and extensively commented tooling can be found online at `https://gitlab.cs.fau.de/hpd/copar-benchmarks`

Model type	Functor
Markov decision process	$\mathbb{N} \times \mathcal{P}_\mathrm{f}(\mathbb{N} \times \mathcal{D}X)$
Discrete-time Markov chain	$\mathbb{N} \times \mathcal{D}(X)$
Continuous-time Markov chain	$\mathbb{N} \times \mathbb{R}^{(X)}$

Table 4.1: Functors for PRISM Models

itself to extract states and transition files from the model.[2] The states file contains variable assignments for each state, which we use to generate initial partitions by identifying states with equal values for an arbitrary set of variables. The transitions file already provides a transition structure on those states and can be readily translated into CoPaR's input format, with some specific considerations for each system type. See Table 4.1 for a summary of the functors used to encode the individual model types.

Markov decision processes have already been introduced in Example 2.4 and are translated to coalgebras for the functor $\mathbb{N} \times \mathcal{P}_f(\mathbb{N} \times \mathcal{D}X)$. The first $\mathbb{N}$ represents the initial partition and the second $\mathbb{N}$ labels successors with a choice number. Valmari's tool does not support MDPs directly, but allows mixing labeled and weighted transitions. We use this to encode MDPs as a bipartite graph with labeled edges going from the set S of states in the original MDP to a set of auxiliary states $S' = \{s_l := (s, l) \in S \times \mathbb{N} \mid \exists s' \in S.(s, l, s') \text{ is an edge in the MDP}\}$ and weighted edges going from S' back to S. With this, the example above is translated to the graph $s \xrightarrow{\alpha} s_\alpha \xrightarrow{p} t$, where s_α is the auxiliary state for s and the label α. This construction resembles the way coalgebras for MDPs are encoded internally in CoPaR after desorting and polynomial rewriting have been performed. MDPs in general and PRISM in particular also include so called rewards, which assign real values to states or transitions depending on the definition. For simplicity and because they are not supported by Valmari's tool, we have chosen to ignore rewards in our translation.

Discrete- and Continuous-time Markov chains simply correspond to weighted transition systems for the functors $\mathbb{N} \times \mathcal{D}(X)$ and $\mathbb{N} \times \mathbb{R}^{(X)}$ respectively, where the first $\mathbb{N}$ again determines the initial partition. PRISM also supports reward structures on Markov chains as for MDPs, but they are again ignored by our converter.

For each benchmark from our PRISM set, the input size as well as various timing results are given in Table 4.2 and Table 4.3. In particular, the following attributes of each benchmark are included:

$\lvert X\rvert$	is the number of states in the coalgebra that is passed to the refinement algorithm. In particular, this includes auxiliary states that arise from desorting.
$\lvert X_0\rvert$	is the number of states in the first sort, i.e., the named states that are explicitly defined in the input file.
$\lvert X/P_0\rvert$	is the size of the initial partition, i.e., the partition computed by INITIALIZE. In all PRISM benchmarks, the initial partition is chosen rather arbitrarily by grouping together states that share the same value for a subset of the variables in the original PRISM model.
$\lvert X/Q\rvert$	is the size of the final partition computed by the whole algorithm. This is not an input but computed by our program. It can nevertheless be useful to assess how many main loop iterations and SPLIT invocations were necessary to compute the result on this particular input. The more blocks the final partition has, the more work our program had to do.

[2]See https://www.prismmodelchecker.org/manual/RunningPRISM/ExportingTheModel for details on states and transition files.

In addition to the size information listed below, the timing results of each benchmark are given. This includes the complete running time t of CoPaR and t_v of Valmari's tool, as well as fine-grained timing results for our tool t_p, t_i and t_r for the time spent in the input parser, the INITIALIZE routine and the refinement loop respectively. Perhaps unintuitively, the time t is usually more than just $t_p + t_i + t_r$ since it also includes the overhead of measuring the fine-grained timings themselves as well as outputting the final result and other smaller parts of the program.

The first set of benchmarks to consider is extracted from PRISM models for aspects of the real-world protocols IEEE 802.11 Wireless LAN, Zeroconf and FireWire, all of which are modeled as Markov decision processes. From the results in Table 4.2 it is immediately obvious that Valmari's tool is a lot faster than CoPaR, by a factor of 15 to 20 on these examples. The factor can vary between models depending on how well our optimizations are applicable. This large performance difference can be attributed partly to the fact that our implementation is written in Haskell, a high level language with garbage collection, and Valmari's is written in hand optimized C++. Additionally, CoPaR being more general and modular paired with the complex functor $\mathbb{N} \times \mathcal{P}_\mathrm{f}(\mathbb{N} \times \mathcal{D}X)$ for MDPs introduces indirections into our code that take a toll on performance. The time to parse the input t_p itself is also over five times higher than the total running time of Valmari's tool t_v. Reasons for this are the same as above and additionally our input format is much more complex.

#	$\lvert X\rvert$	Edges	$\lvert X_0\rvert$	$\lvert X/P_1\rvert$	$\lvert X/Q\rvert$	t	t_p	t_i	t_r	t_v
1	65718	94452	28598	9	49602	1.42s	0.47s	0.28s	0.57s	0.12s
2	582327	771088	248503	9	236170	14.76s	4.89s	3.01s	5.45s	0.91s
3	1408676	1963522	607727	9	543056	38.04s	12.59s	6.17s	15.12s	2.45s
4	876995	1281359	307768	1722	435504	22.68s	7.00s	4.57s	8.28s	1.49s
5	548465	718048	220410	15	231876	14.47s	4.49s	3.22s	5.56s	0.80s
6	744965	978048	298010	15	327376	19.94s	6.17s	4.41s	7.66s	1.14s
7	941465	1238048	375610	15	422876	27.90s	8.42s	5.96s	11.26s	1.54s

Benchmark 1 corresponds to the PRISM model `wlan3_collide` with parameter assignments COL = 2, TRANS_TIME_MAX = 10 and $k = 2$. Benchmarks 2 and 3 correspond to `wlan0_time_bounded` and `wlan1_time_bounded` respectively, both with the same parameters TRANS_TIME_MAX = 10 and DEADLINE = 100.
Benchmark 4 is taken from the model `zeroconf` with parameters $N = 1000$, $K = 4$, err = 0 and reset = `false`.
Benchmarks 5 to 7 come from the `firewire_abst_deadline` model with parameters delay = 36 and fast = 0.5 each as well as values 400, 500 and 600 respectively for the deadline parameter.

Table 4.2: Benchmark results for Markov decision processes

The next set of benchmarks, displayed in Table 4.3, is extracted from PRISM models that are one of the two variants of Markov chains. The first model is a DTMC based on Herman's self-stabilization algorithm [Her90]. The second one is based on the flexible manufacturing system of [CT93] and the third one on a simple embedded control system described in [MCT94]. The latter two are modeled as CTMCs. The functor for Markov chains is much simpler than for MDPs, which is made evident by the fact that $|X_0| = |X|$ meaning that there are no auxiliary states in the coalgebra. This in turn not only leads to fewer iterations of the refinement loop but also fewer indirections from the functor abstraction at every step. The timing results reflect this. In contrast to the results of Table 4.2, CoPaR is now two to five times faster and also only about three to four times slower than Valmari's tool for the first two benchmarks. Parsing the input file now requires equal or more time than the actual refinement steps t_i and t_r. In addition, note that the `embedded` benchmarks are slower than `herman` and `fms` for comparable input sizes. This is due to specific properties of the model as well as the applicability of our optimizations, which are examined in detail in Section 4.3.

#	$\|X\|$	Edges	$\|X_0\|$	$\|X/P_1\|$	$\|X/Q\|$	t	t_p	t_i	t_r	t_v
1	2048	177148	2048	4	2048	0.53s	0.31s	0.05s	0.15s	0.15s
2	8192	1594324	8192	4	8192	4.80s	2.72s	0.36s	1.13s	1.34s
3	32768	14348908	32768	4	32768	48.98s	31.29s	3.52s	12.18s	15.05s
4	35910	237120	35910	815	35910	0.88s	0.48s	0.12s	0.16s	0.21s
5	152712	1111482	152712	1299	152712	4.81s	2.39s	0.66s	1.09s	1.19s
6	537768	4205670	537768	1766	537768	20.75s	9.66s	2.83s	5.43s	5.58s
7	86288	364205	86288	157	8996	3.07s	1.39s	0.43s	0.83s	0.32s
8	424288	1791005	424288	157	43396	18.52s	7.33s	3.63s	4.97s	1.66s
9	846788	3574505	846788	157	86396	39.35s	17.38s	7.88s	10.76s	3.31s

Benchmarks 1 to 3 correspond to PRISM's example model `herman11`, `herman13` and `herman15` respectively, each with parameter values $K = 0$ and $k = 0$.
Benchmarks 4 to 6 are taken from the model `fms` with parameters $n = 4$, $n = 5$ and $n = 6$.
Benchmarks 7 to 8 correspond to the `embedded` model with $T = 0$ and MAX_COUNT values of 100, 500 and 1000 respectively.

Table 4.3: Benchmark results for Markov chains

4.2 Randomly Generated DFAs

While the PRISM models of the previous section were intended to be somewhat close to real-world scenarios, this section presents a set of randomly generated benchmarks with simple semantics that can be used to examine the running time behavior of CoPaR more closely. In particular, we have chosen labeled transition systems with an initial partition of size two (also known as deterministic finite automata without distinguished initial state) as models to generate randomly. They can be easily parameterized by their size and are also supported by Valmari's tool, which make them ideal candidates for random samples.

Let us briefly recall the notation for DFAs. Let $A = (X, x_0, \Sigma, \mathcal{F})$ be a deterministic finite automaton, where X is a finite nonempty set of states, $x_0 \in X$ is the initial state, Σ is an alphabet and $\mathcal{F} \subseteq X$ is the set of final or accepting states. Since the state x_0 is irrelevant for partition refinement, we ignore it for the rest of this section and focus on the transitions and final states instead. This leads to the comparatively simple functor $FX = 2 \times X^\Sigma$ and coalgebras for that functor with $|X|$ states and $|X| \cdot |\Sigma|$ edges. An important caveat of this functor is that our algorithm does not have the same complexity as Valmari's implementation when Σ is regarded as part of the input instead of being fixed as property of the functor. See Example 2.34 for details. Consequently, in this section all benchmarks of a single set share the same alphabet size, since otherwise comparisons of the asymptotic running time behavior between our tool and Valmari's would not be useful.

Our random model generation is parameterized by the number of states n, the size of the input alphabet m and the probability P_f of a state being final. It proceeds by taking $X = \{0, 1, \dots, n-1\} \subset \mathbb{N}$ as the set of states and $\Sigma = \{0, 1, \dots, m-1\} \subset \mathbb{N}$ as the alphabet. For each $x \in X$ and $\sigma \in \Sigma$, we randomly choose an $x' \in X$ uniformly distributed as the successor and for each $x \in X$, we decide whether it is final with probability P_f. An important aspect of each model is how many refinement steps our algorithm will perform until it terminates. In general, an automaton that is already minimal requires the maximum number of steps since the initial partition has to be refined until all blocks contain only a single element and the least amount of work is performed on automata where all final and all non-final states have the same behavior. On the other hand, our optimization for one-element blocks applies more often on inputs that are already minimal. For variety, we have included two sets of benchmarks below, one with $m = 1$ and large n where all samples are not already minimal and one with $m = 1000$ and smaller n where almost all samples are minimal.

Contrary to our initial assumption, different values for P_f do not have a noticeable effect on our benchmark sets, neither on the running times nor the size of the final partition (except for $P_f = 0$ or $P_f = 1$, where all states are identified). Thus, all models in this section are generated with $P_f = 0.5$, which means that about half of the states are final. For each individual benchmark with fixed n and m, each timing result is averaged from the individual results on ten random models. The sample variation, specifically the corrected sample standard deviation, is shown in Figures 4.4 and 4.5 in the form of error bars. These are satisfyingly small, despite the low sample count.

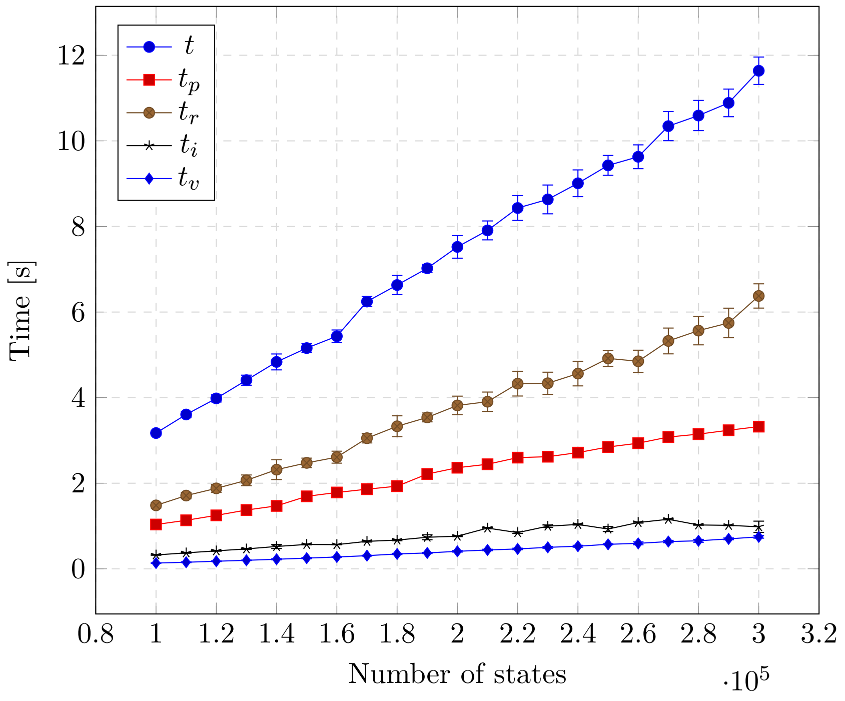


Figure 4.4: Benchmark results for DFAs with $m = 1$ and $P_f = 0.5$. For each n, the mean of 10 samples is shown, with corrected standard deviation as error bars.

Figure 4.4 shows the running times for DFAs with $m = 1$, where all models are minimizable. This is reflected by the fact that the number of times the optimization of one-element blocks in CoPaR is applicable is much lower than for the models of Figure 4.5, for which $m = 1000$. The result is that for the first set, CoPaR is significantly slower than the competition to the extent that even our INITIALIZE procedure runs slower than Valmari's complete program. For $m = 1000$, the performance of CoPaR is not as bad, with both t_i and t_r being lower than t_v and parsing dominating the running time.

In both cases, due to time and memory constraints, the number of data points is too small for a confident assessment of the asymptotic behavior of the measured running times, but the curves also do not suggest anything else than the theoretical complexity of $\mathcal{O}(n \log n)$. Irritatingly, the curves for CoPaR's timing results exhibit abnormal values for specific n in both figures, even though the corresponding standard deviations are not unusually high, which suggests consistent behavior of all samples for this n. We currently do not have a plausible explanation for this phenomenon.

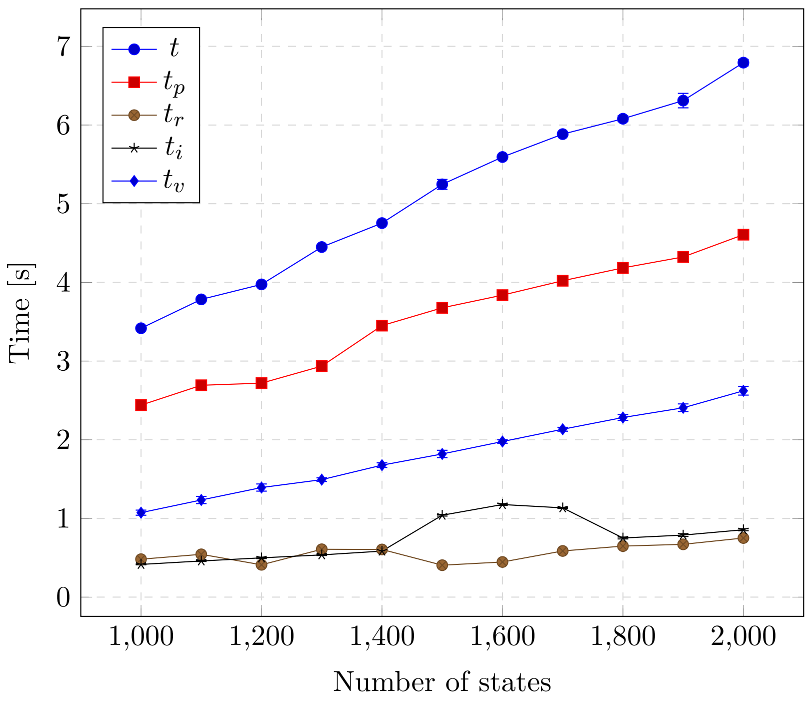


Figure 4.5: Benchmark results for DFAs with $m = 1000$ and $P_f = 0.5$. For each n, the mean of 10 samples is shown, with corrected standard deviation as error bars.

4.3 Impact of Optimizations

Two optimizations that speed up our algorithm implementation by constant factors have been mentioned a few times already in the preceding sections as they can have drastic influence on benchmark results. They are described in detail in Section 3.5. On the one hand, our first optimization skips the splitting step of the algorithm for blocks containing exactly one element and on the other hand, functor expression rewriting is a pre-processing step on functor expressions to mitigate some of the state space inflation induced by our desorting procedure. This section tries to quantify these effects by comparing CoPaR's running time with and without each optimization enabled.

As inputs, we have chosen three benchmarks that have already been explored in the preceding sections: From the PRISM benchmark collection, the models `wlan1_time_bounded` with parameters `TRANS_TIME_MAX`=10 and `DEADLINE`=100 as well as `fms` with parameter $n = 6$ are used. Additionally, a random DFA with $n = m = 1000$ and $P_f = 0.5$ is included. The complete running time of CoPaR is measured on each of those input files in four configurations. Once without any optimization, once with just the one-element block optimization, once with only functor expression rewriting and once with both optimizations enabled. The results are shown in Figure 4.6 as a bar chart, where each individual bar represents the mean of the total running time for three runs on the same input files. Error bars show the sample standard error scaled by a factor of ten.

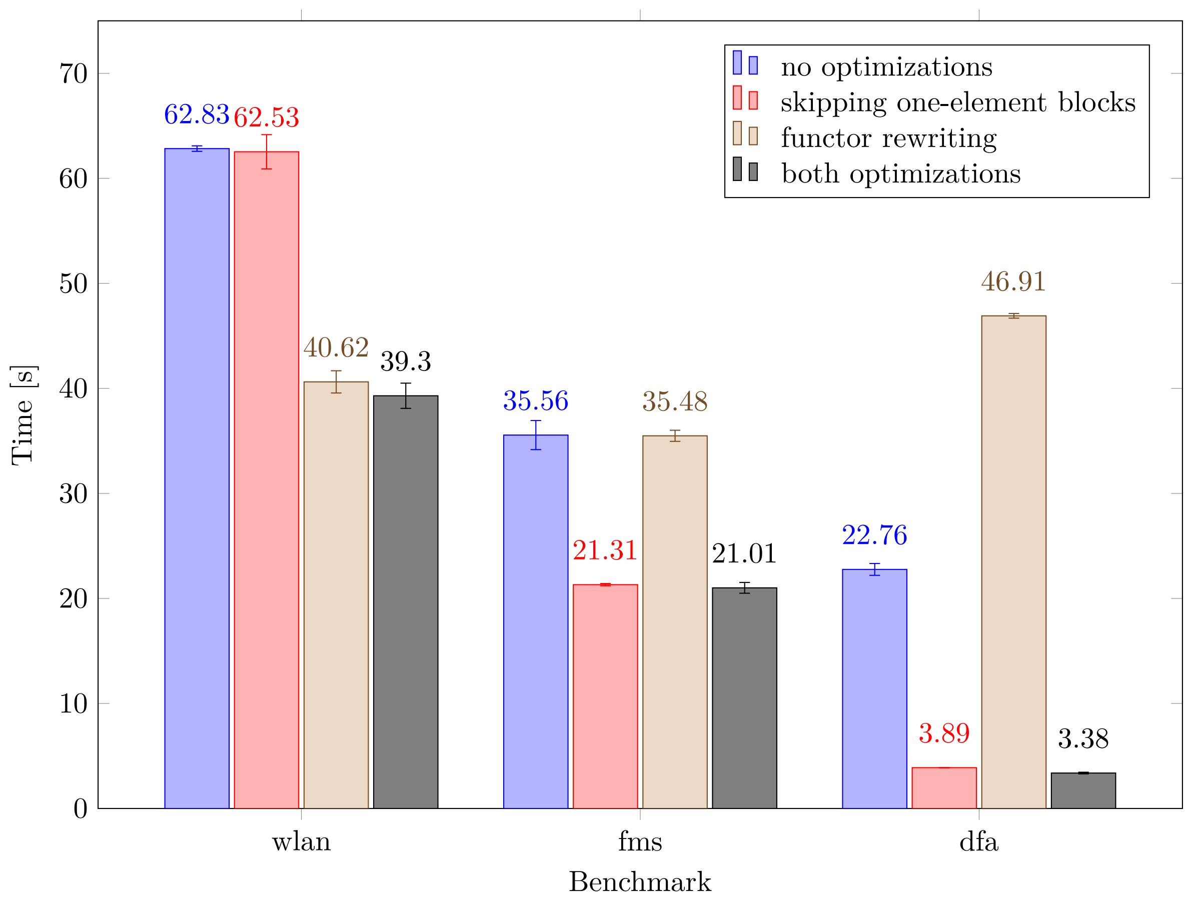


Figure 4.6: Total running time of CoPaR on three different inputs, each with optimization disabled, only the one-element block optimization, only functor rewriting and both enabled. Bars show mean and ten times the standard error of three runs.

One-element block optimization It can be seen from Figure 4.6 that the impact of the one-element block optimization ranges from unremarkable to a speed-up factor of about six, depending on the input. One important difference between the coalgebra for WLAN and the others is that the former is actually minimizable, meaning that the final partition is coarser, leading to fewer blocks with one element in turn. In contrast, if the partition can be refined maximally often (as is the case for the FMS and DFA benchmarks), the final partition consists only of one-element blocks and consequently gives our optimization more opportunities to apply. This is not the only criterion though, as the large difference between FMS and DFA is inexplicable by this measure. As explained in Section 3.5, the optimization does not have an influence on the theoretical asymptotic upper bound but reduces the constant factor by skipping the overhead of calling `update` and various sort procedures for lists of one element. The consequence is that the impact of our optimization also depends heavily on the efficiency of the refinement interface implementation for the functor in question. For DFAs, the running time of `update` depends linearly on the size of the input alphabet Σ which, even when Σ is fixed, results in a noticeable overhead.

Functor expression rewriting Pre-processing the functor expression as described in Section 3.5 is designed to reduce the overhead of complex functors that are compositions of multiple basic functors by eliminating auxiliary states. Recall that it works by replacing instances of the `Polynomial` functor with a new functor `AbsorbingPolynomial` that directly calls the refinement interface of its subexpressions. For simpler functor expressions without any compositions, this can introduce its own overhead since the refinement interface implementation for `AbsorbingPolynomial` has to be more complicated than the one of `Polynomial`. The result can be seen in Figure 4.6. WLAN, being modeled as a Markov decision process, has the most complex functor and consequently sees a performance increase of about 30% from functor expression rewriting. Our DFA benchmark lies on the other end of the spectrum. Its functor $2 \times X^{\Sigma}$ is not a composite, since the polynomial is implemented by a single basic functor. As a result, rewriting actually *slows down* refinement by a factor of two. The functor $\mathbb{N} \times \mathbb{R}^{(X)}$ of continuous time Markov chains is a composition of the polynomial and the real-valued monoid functor. In this case the overhead of state space inflation and `AbsorbingPolynomial` cancel each other out and the running time of the FMS benchmark is not noticeably affected by this optimization.

Curiously, even though functor expression rewriting by itself is detrimental to the performance on DFAs, both optimizations combined are still a bit faster than just skipping one-element blocks. This can be explained by observing that the overhead of `AbsorbingPolynomial` is contained in its refinement interface implementation and `update` in particular. Since this is exactly what the one-element block optimization tries to avoid calling, the inefficiency of the refinement interface implementation carries less weight for some inputs like DFAs. In contrast, the slight increase in overall performance is due to `AbsorbingPolynomial`'s parser implementation being slightly more efficient than the one for `Polynomial`.

5 Conclusions and Future Work

We have presented an implementation for a highly generic and efficient partition refinement algorithm, which retains much of the same genericity. Our tool makes use of the functor and coalgebra abstractions from category theory to model state-based transition systems and provides an of-the-shelf minimizer w.r.t. coalgebraic behavioral equivalence for a wide range of different system types.

New transition types can be added by implementing a simple functor interface in the form of a single Haskell data type and associated type classes. Moreover, these implemented basic functors can be freely combined by the user to form more complex types. Since many basic functors are already provided out of the box, our tool can be readily used to minimize transition systems including deterministic automata, ordinary weighted and probabilistic transition systems, weighted tree automata and Segala systems.

Furthermore, we have created an extensive benchmark suite for our program from a collection of real-world models for the PRISM model checker and by generating random automata. We have used these benchmarks to assess the running time performance of our implementation on different system types and to compare the results with an existing specialized implementation. Moreover, we have devised and implemented different ad hoc optimizations for our program and analyzed their effectiveness.

Future Work The generic algorithm itself provides room for several further developments, e.g., the class of supported functors can be broadened to include functors like the monotone neighborhood functor, which fails to be zippable. In addition, support for categories other than Set is a possible extension, where the category of nominal sets is of particular interest.

Since the main drawback of our generic implementation is its worse performance when compared to specialized implementations, further optimization of our program is of particular importance. Such optimizations include simplification of the refinement interface for system types with trivial weights and finding ways to mitigate the state space blow-up introduced by our approach to modularity.

Bibliography

[AHU74] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[BM91] Robert S Boyer and J Strother Moore. Mjrty - a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.

[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[DMSW17] 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 *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[EWA16] Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. Visible type application. In *European Symposium on Programming Languages and Systems*, pages 229–254. Springer, 2016.

[Gri73] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[Gum03] H Peter Gumm. Universelle coalgebra. *Allgemeine Algebra*, 10:155–207, 2003.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[HMM07] Johanna Högberg, Andreas Maletti, and Jonathan May. Bisimulation minimisation for weighted tree automata. In *Proceedings of the 11th International Conference on Developments in Language Theory*, DLT'07, pages 229–241, Berlin, Heidelberg, 2007. Springer-Verlag.

[Hop71] 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.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Knu01] Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, 250:333 – 363, 2001.

[KS90] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

[MCT94] J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.

[nLa19] nLab authors. Grothendieck group of a commutative monoid. `http://ncatlab.org/nlab/show/Grothendieck%20group%20of%20a%20commutative%20monoid`, January 2019. Revision 2.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[Rut00] Jan Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.

[Seg95] Roberto Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[Swi08] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.

[Val10] Antti Valmari. Simple bisimilarity minimization in o (m log n) time. *Fundamenta Informaticae*, 105(3):319–339, 2010.

[VF10] Antti Valmari and Giuliana Franceschinis. Simple o(m logn) time markov chain lumping. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–52. Springer Berlin Heidelberg, 2010.

[Wad98] Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[WDMS18] Thorsten Wißmann, Ulrich Dorsch, Stefan Milius, and Lutz Schröder. Efficient and modular coalgebraic partition refinement. submitted, `https://arxiv.org/abs/1806.05654`, 2018.