Comparison of Symbolic Maximal End Component Decomposition Algorithms

by
Felix Faber
(394522)

Bachelor Thesis at RWTH Aachen University,
Lehrstuhl für Informatik 2

Submitted to:	Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen
Submission Date:	September 7, 2023
First examiner:	Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen
Second examiner:	apl. Prof. Dr. Thomas Noll
Thesis advisor:	Tim Quatmann, M. Sc.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Declaration of Academic Integrity

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting). I have not used any other sources or aids than those indicated. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding three years or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

The computation of maximal end components (MECs) is extensively used to model check Büchi and Streett objectives on Markov decision processes (MDPs). As the systems which are modelled by MDPs become increasingly large, symbolic algorithms are often employed in which vertices and edges are processed in sets using symbolic operations. Three symbolic MEC decomposition algorithms have been proposed in the literature. Given an MDP containing n vertices and m edges, these algorithms require $O(n^2)$, $O(n \cdot \sqrt{m})$ and $O(n^{2-\epsilon} \cdot \log(n))$ symbolic operations respectively (with $0 < \epsilon \leq 0.5$).

In this thesis, we implement and compare the proposed algorithms empirically using a variety of real world benchmark MDPs from the quantitive verification benchmark set (QVBS). The results show that the $O(n^2)$ MEC decomposition often has the fastest runtime and the lowest amount of symbolic operations, while only the $O(n^{2-\epsilon} \cdot \log(n))$ algorithm is competitive in runtime, but loses its applicability on MDPs using an edge-based actions implementation. While the traversal of the MDP is usually done on a graph-like structure, one can choose whether to encode its actions into the transitions or as distinct vertices of the structure. This thesis also provides and implements a symbolic algorithm to convert from an edge-based action to a vertex-based action representation.

Contents

Chapter 1

Introduction

The field of *model checking* aims to automatically prove properties of complex systems [BK08]. A guarantee of certain properties, e.g. the (un)reachability of certain states, can ensure the correctness of a system and is thus crucial for systems which cannot afford failure. The model to check is usually represented using a transition system.

One popular way to model a system containing uncertainty is to construct a *Markov decision process* (MDP), which is made up of states and governable actions but with uncertain outcomes in conjunction with rewards [Put94]. Here, model checking can help the user reason about the actions to take at any given state, but also e.g. inform them about the probabilities of certain outcomes. This results in MDPs being used in a wide variety of application areas to model real-world systems such as maintenance, finance or communication systems [Whi85, Whi88, Whi93, BVD17]. Long-term rewards are especially interesting for systems running in a loop, whose verification usually requires decomposing the system's MDP into its *maximal end components* (MECs) [CH14, CDHL16]. Büchi objectives check whether a set of states can be visited infinitely often, while Streett objectives work with infinite paths of a transition system. An MEC, which consists of states and actions, describes actions which can be used to loop infinitely on the states of the MEC. Therefore an MEC decomposition is often computed as a preprocessing step when model checking Büchi or Streett objectives [CH14, CDHS19].

Due to the inherent statefulness of most complex systems, the amount of states which need to be checked quickly explodes and becomes difficult to manage. Even with the ongoing advancements in computational power, *explicit* algorithms, in which each state is processed individually, become infeasible when working with systems consisting of billions or trillions of states. *Symbolic* model checking combats this problem by exclusively working with *sets* of states represented by *Binary Decision Diagrams* (BDDs) [Lee59, Ake78, Bry86, Bry92].

While this approach enables model checking on large systems, it also requires the usage of symbolic algorithms in order to function efficiently [EFT93, DB13, XB00]. Naturally, an improvement to symbolic MEC decomposition algorithms is always desired in order to verify some properties of a complex system more efficiently. Three symbolic MEC decomposition algorithms have been proposed in the literature. Naive is the symbolic implementation of a basic MEC decomposition algorithm described in [DA98]. LockStep extends Naive by using a symbolic lockstep search on the transition system [CHL+18]. Collapsing [CDHS21], whose first implementation is provided by this thesis, focuses on the quicker detection of *end components* (ECs), from which all MECs can be computed. These symbolic algorithms, in the worst-case, require $O(n^2)$, $O(n \cdot \sqrt{m})$ and $O(n^{2-\epsilon} \cdot \log(n))$ symbolic operations respectively (with $0 < \epsilon \leq 0.5$) [CHL+18, CDHS21]. But as neither theoretical improvements nor disadvantages always translate directly into measurable practical impacts, any theoretical advancements should be experimentally evaluated such that real-world problems can be pragmatically solved to the best of our abilities [EFT93, BW96, LSS+23].

1.1 Contributions of the Thesis

The main contribution of this thesis is the experimental evaluation of three existing symbolic MEC decomposition algorithms Naive, LockStep and Collapsing [CHL+18, CDHS21] by implementing them into the model checker Storm [HJK+21]. Using the *quantitative verification benchmark set* (QVBS) [HKP+19], both the runtime and the amount of symbolic *Pre*/*Post* operations are compared empirically. The authors of the recent symbolic MEC decomposition algorithms assume a graph-like structure in which the actions of a given MDP are encoded into vertices distinct from states. In contrast, the model checkers Storm and Prism [HJK+21, KNP02] as well as e.g. the Jani model specification [BDH+17] usually encode an MDP's actions into the edges of the transition system. We demonstrate that one of the symbolic MEC decomposition algorithms is not applicable to the edge-based actions representation without significantly adjusting the algorithm. We believe this thesis presents the first implementation of Collapsing, as well as the first implementation of LockStep for an edge-based action representation.

This thesis also formalizes the differences of these two MDP representations. We present and implement a (to the best of our knowledge) novel symbolic algorithm to convert from the edge-based to the vertex-based actions representation into Storm. While the runtime cost of the conversion process itself is cheap, the conversion is not optimal as it significantly increases the transition BDD size, which leads to slower symbolic operations and thus a slower symbolic MEC decomposition overall. Using this conversion, the symbolic MEC decomposition algorithms are compared for both MDP representations (when possible) by comparing the amount of performed symbolic operations.

Our experimental results show that, in practice, neither LockStep nor Collapsing exhibit the theoretical improvents in the amount of symbolic operations required for most of the benchmarks when compared to Naive. The performance of LockStep is worse than Naive in regard to both runtime and symbolic operations in most instances. The runtime performance of Collapsing is competitive to Naive when ran on the converted MDP representation, but Collapsing performed more symbolic operations than Naive in all of the benchmarks. Therefore the runtime advantages of Collapsing are partially due to a varying cost of each symbolic operation due to the modification of the transition BDD. But as the transition BDDs are constructed using the suboptimal conversion process, the presented evidence for real-world performance improvements is inconclusive. Comparing the amount of symbolic operations, we argue that the edge-based representation is more efficient in the context of symbolic MEC decomposition algorithms.

To summarize, this thesis contributes:

- the (to the best of our knowledge) first implementation of LockStep using an MDP representation with edge-based actions and the first implementation of Collapsing,
- an empirical evaluation of three symbolic MEC decomposition algorithms Naive, LockStep, Collapsing, and
- a novel symbolic algorithm to convert an edge-based action representation of an MDP into using vertex-based actions.

1.2 Related Work

Given a graph of an MDP consisting of n vertices and m edges, previous results regarding the computation of MECs can be categorized by the usage of explicit and symbolic decomposition algorithms.

Explicit Algorithms. The basic MEC decomposition algorithm described in [DA98] takes $O(n \cdot m)$ operations in the worst-case [CH14]. In the works of [CH11], this algorithm is improved upon by first introducing a lockstep search based on the depth-first search algorithm for strongly connected components by Tarjan [Tar72]. While this lockstep algorithm requires $O(m \cdot \sqrt{m})$ operations, they provide a second algorithm which reduces the number of searches in a lockstep search with a runtime of $O(m \cdot n^{2/3})$. In [CH14], another algorithm is presented with a runtime of $O(n^2)$ operations, where an algorithm to find the winning set of Büchi games is adjusted to find bottom *strongly connected components* (SCCs) instead. While these algorithms are deterministic, a randomized explicit algorithm exists which utilizes a decremental SCC algorithm, which recomputes SCCs based on deleted edges [CDHS19]. This randomized algorithm has a runtime of $\tilde{O}(m)$ operations (where $\tilde{O}$ hides logarithmic factors). The works of [WKB14] focused on a faster implementation of the basic $O(n^2)$ decomposition algorithm by presenting a GPU based implementation. Here, empirical

results have shown up to 79 times faster MEC decompositions when compared to a CPU based implementation.

Symbolic Algorithms. The symbolic implementation of the MEC decomposition algorithm described in [DA98] requires $O(n^2)$ symbolic operations when utilizing a linear time symbolic SCC decomposition algorithm [CHL+18]. Similar to the explicit algorithm improvements, [CHL+18] introduces a novel symbolic lockstep algorithm which is then used to improve the MEC decomposition algorithm to $O(n \cdot \sqrt{m})$ symbolic operations. The symbolic algorithm shown in [CDHS21] focuses on detecting and collapsing ECs quickly with a space-time tradeoff. Their proposed MEC decomposition algorithm requires $O(n^{2-\epsilon} \cdot \log(n))$ symbolic operations, in which they set $\epsilon = 0.5$ to obtain a decomposition in $\tilde{O}(n \cdot \sqrt{n})$ symbolic operations.

Empirical Evaluation. To the best of our knowledge, no experiments to compare the runtime performance of the symbolic or explicit MEC decomposition algorithms in practice exist[1]. The importance of benchmarks can be seen in the recent works of [LSS+23], where various symbolic SCC decomposition algorithms are benchmarked: although one of the SCC algorithms has an optimal runtime complexity of $O(n)$ symbolic operations, it has an unusually high space requirement, which in practice can lead to worse runtime performance than the naive $O(n^2)$ algorithm.

1.3 Structure of the Thesis

This thesis is structured as follows: MDPs, their graph-like structures and all the notions used for MEC decompositions are formally defined in Chapter 2. Chapter 3 outlines BDDs and shows how BDD operations work exclusively with sets by looking at an implementation of the logical And operation. In Chapter 4, we look at how an MDP is implemented in Storm by matching the symbolic version with the sparse representation. The novel symbolic algorithm to convert the MDP representation is found at the beginning of Chapter 5, in which the core idea of each symbolic algorithm is also outlined. The main contributions are found in Chapter 6, where both the symbolic conversion and MEC decomposition algorithms are experimentally evaluated. In addition to the runtime performance, we also benchmark the amount of symbolic operations to see whether the theoretical algorithm improvements on symbolic operations hold in practice and evaluate which symbolic MDP representation is preferable for computing MEC decompositions.

[1] [CHL+18] contains benchmarks utilizing the symbolic lockstep algorithm on MDPs, but their measurements begin after preprocessing the graphs, which includes computing all SCCs and MECs of the graph.

Chapter 2

Preliminaries

2.1 Strongly Connected Components

Definition 2.1 Given a directed graph $G = (V, E)$ and two vertices $v_1, v_n \in V$ with $v_1 \neq v_n$, the vertex $v_n \in V$ is considered to be *reachable from* $v_1 \in V$ iff the vertices $\{v_1, ..., v_n\} = V' \subseteq V$ exist such that $(v_i, v_{i+1}) \in E \ \forall 1 \leq i < n$. If $|V'|$ is minimal, then we consider the *distance* between the vertices v_1, v_n to be $d(v_1, v_n) = |V'| - 1$.

Definition 2.2 A directed graph $G = (V, E)$ is considered to be *strongly connected* iff for all pairs of vertices $v_1, v_2 \in V$, the vertex v_2 is reachable from v_1. $U \subseteq V$ is a *strongly connected component* (SCC) of G iff the induced subgraph $G[U] := (U, (U \times U) \cap E)$ is strongly connected and U is maximal; that is no U' with $U \subsetneq U' \subseteq V$ exists for which $G[U']$ is strongly connected.

Additionally, if the subgraph of an SCC consists of only one vertex without a self-loop then it is considered to be a *trivial* SCC. The SCC decomposition of a graph is a partition $U_1, ..., U_n$ of V such that U_i is an SCC $\forall 1 \leq i \leq n$, and every vertex $v \in V$ belongs to precisely one U_i.

Definition 2.3 An SCC U is a *bottom* SCC iff the SCC has no outgoing edges, meaning $(U \times (V \setminus U)) \cap E = \emptyset$. U is a *top* SCC iff U has no incoming edges, i.e. $((V \setminus U) \times U) \cap E = \emptyset$.

Definition 2.4 An SCC with its vertices V' is said to have a *diameter* of n if $n = \max\{d(v_a, v_b) \mid v_a, v_b \in V' : \ v_a \neq v_b\}$.

2.2 Separators

Definition 2.5 Let $G = (V, E)$ be a graph with $|V| = n$ vertices and an SCC $U \subseteq V$. For $q \in \mathbb{N}$, a *q-Separator* is a set of vertices T with $\emptyset \subsetneq T \subseteq U$ such that each SCC of the subgraph $G[U \setminus T]$ contains at most $n - q \cdot |T|$ vertices.

In other words, a q-Separator is a set of vertices which, when removed from U, splits U into smaller SCCs, where q denotes the "quality" of the separation: the higher q, the smaller the separated SCCs will be (see e.g. Fig. 2.1).

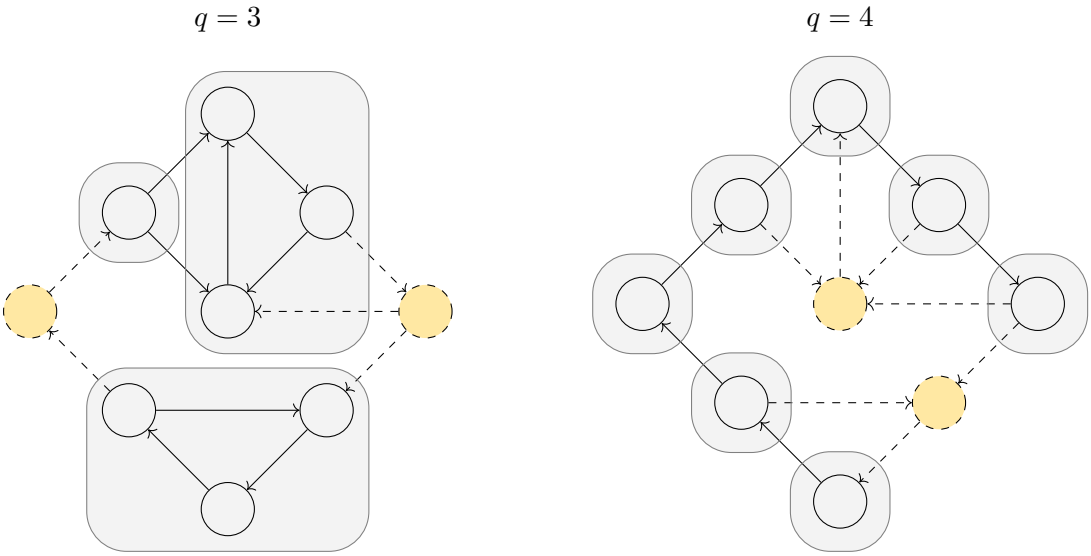


Figure 2.1: Two separators (yellow) of varying quality q on the same directed graph. After the removal of the separator, the resulting SCCs of the graph are marked in gray.

2.3 Markov Decision Process

A *Markov decision process* (MDP) is a transition system with states, on which an action can be chosen. The current state and the chosen action yield the next state using a known probability distribution.

Definition 2.6 An MDP is a tuple $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$, where S is a finite set of states, A is a finite set of actions, $d_{\text{init}} : S \to [0, 1]$ is the initial state distribution function, $\delta : S \times A \times S \to [0, 1]$ is the transition probability function, and $r : S \times A \times S \to \mathbb{R}$ is the reward function. We assume that each action is *unique*, meaning $\forall s_1, s_1', s_2, s_2' \in S,\ \alpha \in A$: if $\delta(s_1, \alpha, s_1') > 0$ and $\delta(s_2, \alpha, s_2') > 0$, then $s_1 = s_2$.

Furthermore, we will use the notion of reachability between states of an MDP similar to reachability between vertices on a graph (see Def. 2.1):

Definition 2.7 Given an MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ the state $s_n \in S$ is considered to be *reachable from* $s_1 \in S$ iff there exists a sequence of states $s_2, s_3, ... \in S$ and actions $\alpha_1, \alpha_2, ... \in A$ such that $\delta(s_i, \alpha_i, s_{i+1}) > 0 \ \forall 1 \leq i < n$.

The initial state distribution function with $\sum_{s \in S} d_{\text{init}}(s) = 1$ describes the probability $d_{\text{init}}(s)$ of starting in state $s \in S$. The transition probabilty function δ describes the probability $\delta(s, \alpha, s')$ of transitioning from state $s \in S$ into $s' \in S$ using action $\alpha \in A$. In other words, for $s \in S$ with $\alpha \in A$: $\sum_{s' \in S} \delta(s, \alpha, s') \in \{0, 1\}$. $\alpha \in A$ is said to be *enabled* on $s \in S$ iff a state $s' \in S$ exists with $\delta(s, \alpha, s') > 0$. Let $A[s] \subseteq A$ denote the set of actions enabled on state $s \in S$. It is required that $A[s] \neq \emptyset$ for all $s \in S$. The function r yields the reward $r(s, \alpha, s')$ after transitioning from state $s \in S$ into state $s' \in S$ using action $\alpha \in A$.

Informally, one starts in a state randomly determined using the probabilities given by d_{init}. At every state $s \in S$, an enabled action $\alpha \in A[s]$ is chosen nondeterministically. Afterwards, one is transitioned into a successor state $s' \in S$, which is randomly chosen using the probablities given by δ, and a reward $r(s, \alpha, s')$ is given.

MDPs provide a simple, yet powerful abstraction to model probabilistic behaviours in combination with governable decisions. As such, MDPs have been employed to model real-world problems, a variety of which are listed in the works of [Whi85, Whi88, Whi93, BVD17].

2.4 Policies and Properties

When modelling problems using MDPs, one is often interested in *policies* dictacting which action should be taken at any specific state. Policies which are only dependent on the current state are often referred to as *memoryless* or *positional* policies, which can be formally described using a function $f_{\text{Policy}} : S \to A$. Some objectives such as visiting all possible states in a finite MDP greatly benefit from policies which take the history of previous states into account.

However, the field of *model checking* goes beyond just policies: model checking systematically checks *properties* of all states of a model, which can be of varying complexity [BK08]. The properties of an MDP can in turn provide insight into the real-world systems being modelled. For example, simple reachability properties on MDPs might be:

- Which states are reachable?
- What is the probability of reaching any or all states of a subset $S' \subsetneq S$?
- Can a policy guarantee that a specific state $s \in S$ is never reached?

Büchi and *Streett* objectives extend on this notion: a Büchi objective checks whether a given set of states can be visited infinitely often, while Streett objectives are concerned with infinite paths, which often arise in the field of verification [CDHS19]. Other long term properties might check what the minimum or maximum expected average rewards of an MDP can be [CDHL16].

While the listed properties are only a small selection, they make heavy use of *maximal end components* (MECs) [CDHL16], which will be formally introduced in Chapter 2.6. In short, MECs describe parts of an MDP which, using certain policies, will never be left once entered.

2.5 Graph-Like Structures of MDPs

In the context of model checking, properties of a given MDP $\mathcal{M}$ are verified using graphs or graph-like structures which model $\mathcal{M}$. The definition of the (often simplified) structure varies depending on the application (see e.g. [Put94, BK08, CH11, SLL09]).

In this thesis, we are only interested in the computation of *maximal end components* (MECs), which will be formally introduced in Chapter 2.6. The computation of MECs is usually performed on one of the following graph-like structures [CHL$^+$18, CDHS21, HM18, BCC$^+$14], which model and simplify $\mathcal{M}$:

Definition 2.8 Let $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ be an MDP. The graph-like structure $G_{EBA} = (V, E)$ models $\mathcal{M}$ with

$$\begin{aligned} V &= S, \\ E &= \{\ (s, \alpha, s') \mid s, s' \in S,\ \alpha \in A :\ \delta(s, \alpha, s') > 0\ \}, \end{aligned}$$

and the graph-like structure $G_{VBA} = (V, E)$ models $\mathcal{M}$ with

$$\begin{aligned} V &= \underbrace{V_P}_{\{v_s \mid s \in S\}} \cup \underbrace{V_R}_{\{v_\alpha \mid \alpha \in A\}}, \\ E &= \underbrace{\{(v_s, v_\alpha) \mid s \in S, \alpha \in A[s]\}}_{V_P \text{ to } V_R} \cup \underbrace{\{(v_\alpha, v_{s'}) \mid s, s' \in S, \alpha \in A[s] : \delta(s, \alpha, s') > 0\}}_{V_R \text{ to } V_P}, \end{aligned}$$

where V_P are the *player vertices* and V_R are the *random vertices*. Edges which originate from V_P are *player edges*, while edges which originate from V_R are considered *random edges*.

Both G_{EBA} and G_{VBA} simplify $\mathcal{M}$ as can be seen in Fig. 2.2: the initial state distribution d_{init} and the rewards r are omitted entirely. For the transition probabilities δ using $s, s' \in S$ and $\alpha \in A$, it is only relevant whether $\delta(s, \alpha, s') > 0$ or $\delta(s, \alpha, s') = 0$.

To the best of our knowledge, no terminology has been established for these two different graph-like constructions. As such, we will refer to G_{EBA} as a graph-like structure using *edge-based actions* and similarly to G_{VBA} using *vertex-based actions*.

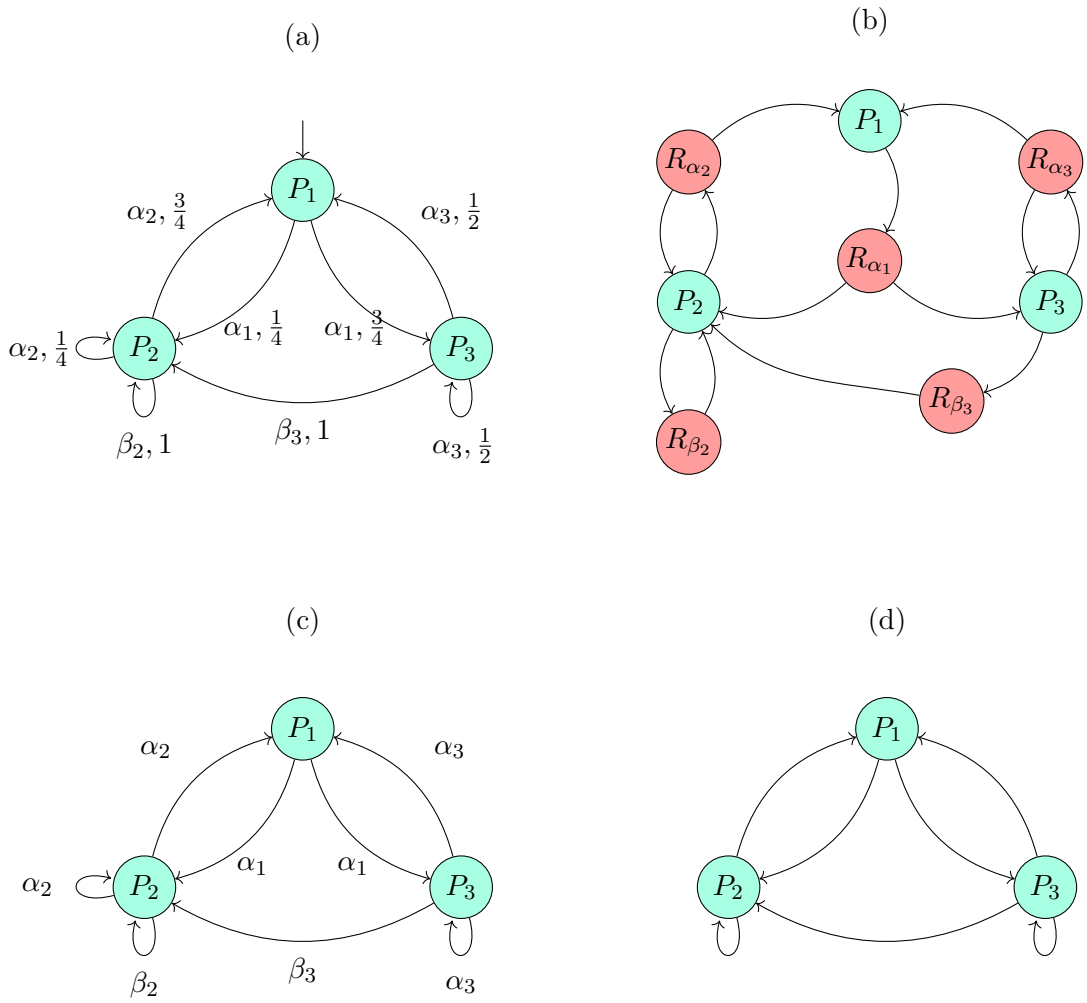


Figure 2.2: The same MDP (a) depicted using their G_{VBA} (b) and G_{EBA} (c). When performing *Post*/*Pre* operations, G_{VBA} can be used directly, while G_{EBA} is first converted to a directed graph (d).

Remark 2.9 For G_{EBA}, which might contain self-loops, the removal of actions refers to the removal of edges. In comparison, G_{VBA} cannot contain self-loops and the removal of actions refers to the removal of vertices.

When required, these graph-like structures can be converted to directed graphs (e.g. when computing SCCs). For G_{VBA}, no additional processing is required as the graph-like structure is also a directed graph. For G_{EBA}, edges with equal source and destination vertices but with differing actions are merged. More formally, the edges of the converted graph are as follows:

$$E = \{(s, s') \mid s, s' \in S,\ \exists \alpha \in A :\ \delta(s, \alpha, s') > 0\}$$

We will use to $Pre(X)$ to refer to the set of vertices $V' \subseteq V$ which have an edge to X, and $Post(X)$ to refer to the vertices $V' \subseteq V$ which have an incoming edge from X.

2.6 Maximal End Components

Definition 2.10 Given an MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$, an *end component* (EC) consists of a non-empty set $S' \cup A'$ of states $S' \subseteq S$ and actions $A' \subseteq A$ such that

1. $A'[s] \neq \emptyset\ \forall s \in S'$ and $\bigcup_{s \in S'} A'[s] = A'$,
2. for each $s_1, s_n \in S'$: s_n is reachable from s_1 using only states $s_2, s_3, ... \in S'$ with actions $\alpha_1, \alpha_2, ... \in A'$,
3. for each $s, s' \in S, \alpha \in A'$: if $\delta(s, \alpha, s') > 0$ and $s \in S'$, then $s' \in S'$

all hold. An EC X is *maximal* (MEC) iff S' and A' are maximal with regard to set inclusion.

Each state belongs to at most one MEC. Informally, an EC X describes a part of an MDP which, once entered and using only actions part of X, will never be left again. Therefore MECs are especially interesting for long term properties and are heavily used during the processing of e.g. Büchi and Streett objectives [CDHL16], which usually require computing an *MEC decomposition*:

Maximal End Component Decomposition	
Given:	An MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$.
Task:	Find all MECs of $\mathcal{M}$. Each MEC consists of a set of states and actions.

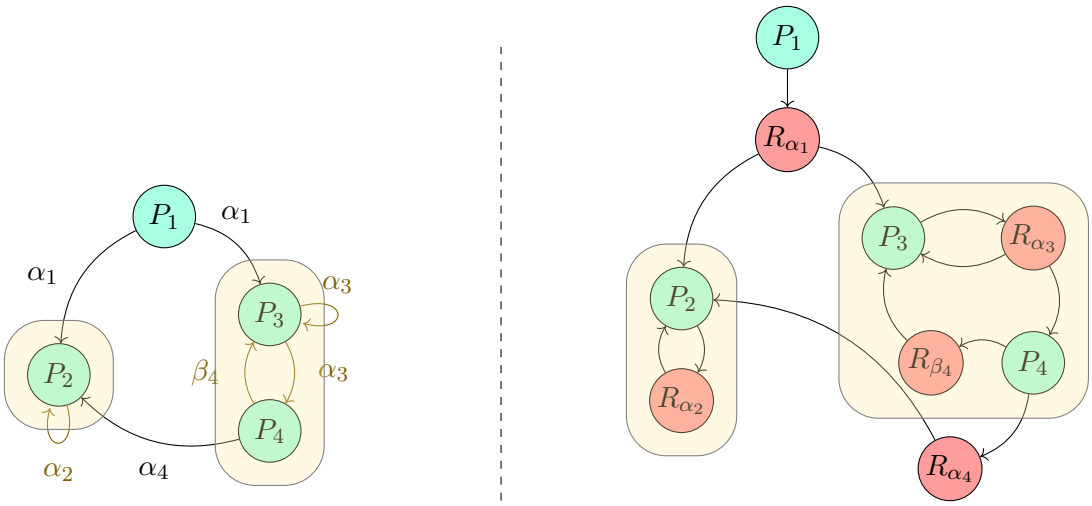


Figure 2.3: MEC decomposition of an MDP for both of its graph-like structures G_{EBA} (left) and G_{VBA} (right). Each MEC is marked in yellow. For G_{VBA}, each MEC consists of a set of vertices. For G_{EBA}, each MEC consists of a set of vertices and actions (also marked in yellow).

In the context of this thesis, we compute MEC decompositions using the graph-like structures G_{EBA} and G_{VBA} of $\mathcal{M}$ (see Fig. 2.3). The algorithms to compute these decompositions are shown later in Chapter 5.5. Using the graph-like structures, an EC X exhibits the following properties:

- For G_{VBA}, X consists of vertices. For G_{EBA}, X consists of vertices and actions.
- Let G be the (converted) directed graph of the graph-like structure of $\mathcal{M}$. Then $G[X]$ is strongly connected.
- For G_{EBA}, there are no transitions with an action $\alpha \in X$ which point to a vertex $v \notin X$. For G_{VBA}, there are no random edges of X which point to a vertex $v \notin X$. For both structures, we will refer to this property as $ROut(X) = \emptyset$, where $ROut(X)$ returns the actions or vertices of X which would violate this property.

2.7 Random Attractors

Definition 2.11 Let $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ be an MDP and X a set of states and actions. The *Random Attractor* of X ($Attr_R(X)$) is defined as a set containing:

1. All elements of X,
2. Actions $\alpha \in A$ with $\delta(s, \alpha, s') > 0$ for states $s \in S, s' \in Attr_R(X)$ and
3. States $s \in S$ for which all actions $\alpha \in A[s]$ are contained in $Attr_R(X)$.

In other words, X describes a section of an MDP, and $Attr_R(X)$ is the part of the MDP in which, once entered, there is always a positive probability to reach X, no matter which actions are chosen (see e.g. Fig. 2.4).

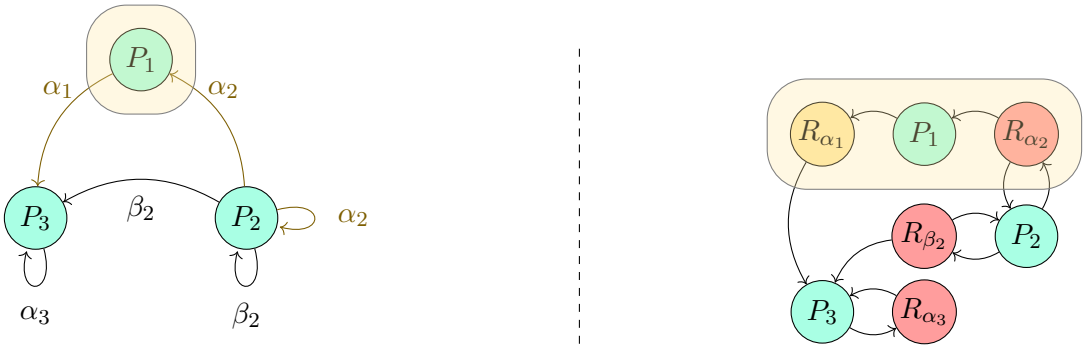


Figure 2.4: Random attractor of action α_1 on an MDP visualized on both of its graph-like structures G_{EBA} (left) and G_{VBA} (right).

Chapter 3

Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are structures which can encode arbitrary Boolean functions. While first developed in the context of logic circuit modelling ([Lee59, Ake78] as outlined by [DB13]), BDDs are extensively used in model checking. After covering notations and operations used in this thesis for Boolean terms and functions, this chapter will provide an introduction into BDDs and operations on BDDs.

3.1 Notation

The value of a Boolean variable x is defined to be either 0 ("false") or 1 ("true"). For two (or more) Boolean variables $x_1, x_2 \in \{0, 1\}$, logical operations are denoted as follows:

- Conjunction: $x_1 \cdot x_2$, or $x_1 x_2$
- Disjunction: $x_1 + x_2$
- Negation: $\overline{x_1}$

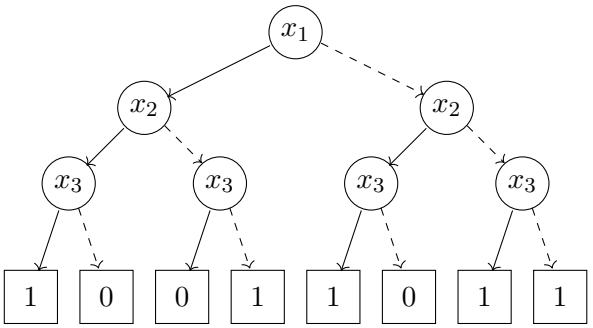
For Boolean functions $f : \{0, 1\}^n \to \{0, 1\}$, for example $f(x_1, x_2, x_3) = \overline{x_1} + x_2 x_3$, the following additional operations are defined:

- Restriction: let $b \in \{0, 1\}$. Then, $f|_{x_i=b} := f(x_1, ..., x_{i-1}, b, x_{i+1}, ..., x_n)$
- Existential quantification: $\exists x_i(f) := f|_{x_i=0} + f|_{x_i=1}$, which yields a new function $f'(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$. Multiple existential quantifications $\exists x_1 ... \exists x_n(f)$ will be written as $\exists\{x_1, ..., x_n\}(f)$

3.2 Naive BDDs and Reduced Ordered BDDs

The simplest implementation of a BDD is a tree, where each internal node with children represents a Boolean variable x_i with two outgoing edges: a *high edge* and a *low edge*. Here, we will write $var(v)$ to refer to the variable x_i encoded in node v, while $low(v)$

and *high*(v) refers to the children of v. Each leaf of the BDD represents either *true* or *false*. Thus a path from the root of the BDD to a leaf represents a function evaluation, where the high edge at node x_i indicates that x_i is set.



x_1	x_2	x_3	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 3.1: An ordered BDD with the truth table of the Boolean function being modelled.

Such a BDD can be seen in Fig. 3.1. However, the number of nodes grow exponentially with the number of Boolean variables: a function with t Boolean arguments results in a tree with $2^{t+1} - 1$ nodes. To address this issue, *Reduced Ordered Binary Decision Diagrams* (ROBDDs) are usually used when BDDs are mentioned [Bry86]:

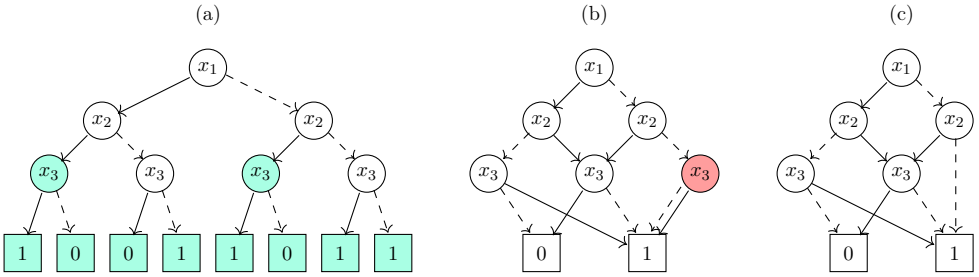


Figure 3.2: The initial BDD contains identical terminal leaves and duplicate sub-trees (marked in mint, (a)). The intermediate BDD now contains a node with identical outgoing edges (marked in red, (b)). After its removal, the BDD is now fully reduced and is thus an ROBDD (c).

A *reduced* BDD is derived from a BDD by applying the following three reduction rules repeatedly:

1. Duplicate terminal nodes: merge identical terminal leaves.
2. Duplicate, isomorph sub-trees: merge two nodes representing the same Boolean variable x_i iff their sub-trees are equivalent.
3. $low(v) = high(v)$: a node v can be skipped over and omitted if both of its edges point to the same node.

The resulting decision diagram generally is no longer a tree but an acyclic directed graph (see Fig. 3.2).

An *ordered* BDD is a BDD with an established variable ordering $x_1 < x_2 < ... < x_t$, meaning the variables encountered on a path from the root to a leaf of a BDD are in a specific order. While the variable ordering can be arbitrary, it has a great influence on the compressibility (see Fig. 3.3). Additionally, a fixed variable ordering makes ROBDDs unique: two Boolean functions $f(x_1, ..., x_n), g(x_1, ..., x_n)$ are logically equivalent ($f = g$) iff the ROBDDs of f and g are the same.

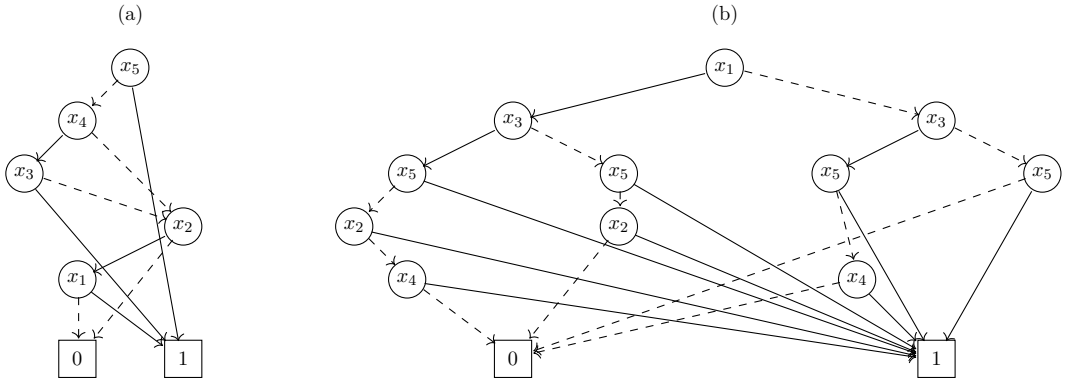


Figure 3.3: Two ROBDDs of the same Boolean function $x_1x_2 + x_3x_4 + x_5$ using two different variable orderings.

Further optimizations of ROBDDs exist. For example, some implementations add *Complement Edges* such that a BDD for a function f can also easily represent $\overline{f}$ by setting a flag which logically inverts the leaf values (e.g. [BRB91]). Further space savings can be achieved by introducing the notion of a *Multi-Rooted* or *Shared* BDDs, in which multiple BDDs share parts of their graph with another [MSS07].

Additionally, the concept of a BDD can be extended: *Multi-Terminal* BDDs (MTBDDs) or also referred to as *Algebric* DDs (ADDs) are DDs with non-Boolean terminal nodes ([BFG+97, FMY97]). On these DDs, leaves can have integer or real values, and additional operations such as addition or multiplication on the DDs can be implemented.

3.3 AND Operation on ROBDDs

Crucially, most operations relevant for model checking do can be performed directly on ROBDDs. Here, one possible algorithm as shown by [Bry92] is explained which calculates $f \cdot g$ given the ROBDDs of f and g, provided that f, g take the same Boolean arguments $(x_1, ..., x_n)$. Let in the following be $x_1 < x_2 < ... < x_n$ and let r_f, r_g be the root nodes of the ROBDDs of f, g.

Most $f \cdot g$ implementations rely heavily on the Shannon expansion, which states that for any variable $x_i \in \{x_1, ..., x_n\}$:

$$f(x_1, ..., x_n) = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}.$$

First, note that the ROBDD of the restriction $f|_{x_i=b}$ can be trivially computed if $x_i \leq var(r_f)$:

$$f|_{x_i=b} = \begin{cases} r_f, & x_i < var(r_f) \\ high(r_f), & x_i = var(r_f) \text{ and } b = 1 \\ low(r_f), & x_i = var(r_f) \text{ and } b = 0 \end{cases} \tag{3.1}$$

where the resulting node is the root node of the resulting ROBDD. In the context of $f \cdot g$, it follows that

$$\underbrace{f \cdot g}_{\text{node } v} = x_i \cdot \underbrace{(f|_{x_i=1} \cdot g|_{x_i=1})}_{high(v)} + \overline{x_i} \cdot \underbrace{(f|_{x_i=0} \cdot g|_{x_i=0})}_{low(v)}. \tag{3.2}$$

Therefore using Shannon expansion, the resulting BDD of $f \cdot g$ can be computed as outlined in Algorithm 1. The idea is to recursively walk both f and g in the order of $x_1 < ... < x_n$ to be able to compute each restriction $f|_{x_i=b}$ as shown in equation 3.1.

Algorithm 1: AND operation $f \cdot g$ on two ROBDDs.

Input : Vertices: v_f, v_g of the ROBDDs of f, g
Output: A node of the resulting BDD $f \cdot g$

if *v_f or v_g is terminal node* 0 **then**
  return *terminal node* 0
if *v_f and v_g are terminal nodes* **then**
  return *terminal node* $v_f \cdot v_g$

Splitting variable $x_i := \min\{var(v_f),\ var(v_g)\}$
Create new node v' with
  $var(v') := x_i$
  $high(v') :=$AND$(f|_{x_i=1},\ g|_{x_i=1})$
  $low(v') :=$AND$(f|_{x_i=0},\ g|_{x_i=0})$
return v'

The algorithm starts at the root nodes r_f, r_g: as we only process one Boolean variable x_i in each recursion step, we compute the *splitting variable* $x_i = \min\{var(r_f),\ var(r_g)\}$, which ensures that the resulting BDD of $f \cdot g$ remains ordered. Then we create a new node v' which encodes x_i: using x_i with the Shannon expansion in equation (3.2), the children $high(v')/low(v')$ of v' need to be computed by obtaining the next pair of nodes v_f, v_g using equation (3.1) on r_f, r_g. This process is repeated recursively until

the obtained nodes v'_f, v'_g are both terminal leaves, as the resulting node will also be a terminal node computed by $v'_f \cdot v'_g$.

Therefore the algorithm builds the resulting BDD in a depth-first manner, an example of which can be seen in Fig. 3.4: at recursion step A_6B_7, the algorithm is at node A_6 and at node B_7 of the ROBDDs of f and g. As both A_6 and B_7 are terminal nodes, the recursion stops and returns the leaf $A_6 \cdot B_7 = 1 \cdot 1 = 1$. Now that the high edge of A_4B_3 is computed, the low edge is next. As the splitting variable of A_4B_3 is $\min\{var(A_4),\ var(B_3)\} = \min\{x_3, x_3\} = x_3$, equation (3.1) indicates that we follow the low edge of A_4 to A_5, and the low edge of B_3 to B_5. Here we can apply an optimization: Given that A_5 is a 0-leaf, any computation of the terminal nodes $A_5 \cdot B_i$ results to 0. Therefore A_5B_5 is a 0-leaf, which finishes the computation of A_4B_3, after which the low edge A_6B_4 of A_2B_2 would be computed next.

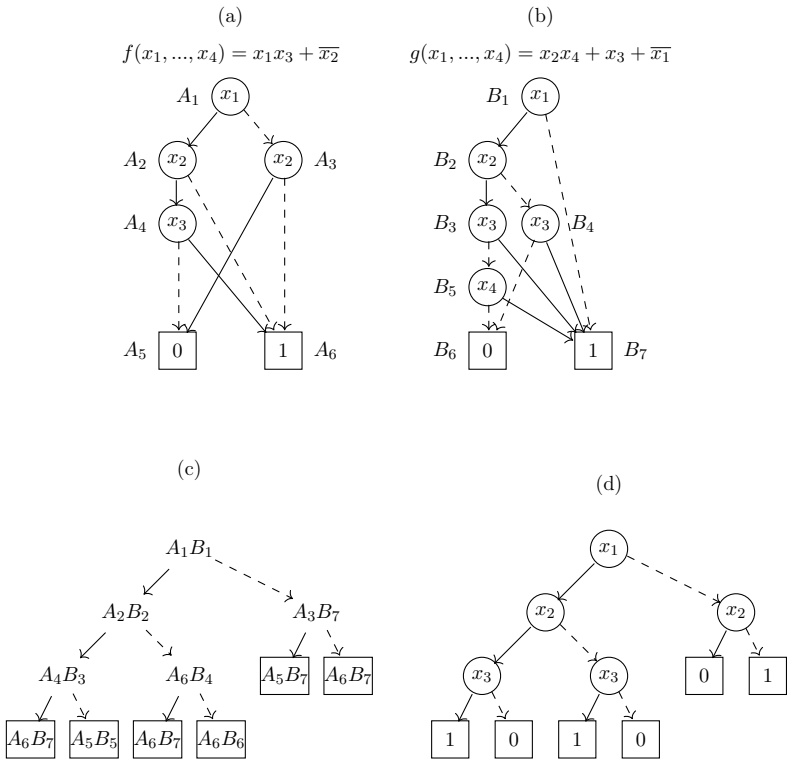


Figure 3.4: Example AND operation on ROBDDs. Given two Boolean functions f (a) and g (b), their ROBDDs (with labelled nodes) are traversed recursively (c), from which the resulting (unreduced) BDD can be constructed (d).

After all recursion steps are evaluated, the resulting BDD of $f \cdot g$ is fully computed, but still needs to be reduced. But with little adjustments as listed in [Bry86], the algorithm will directly yield the final ROBDD, as the reduction rules listed in Chapter 3.2 can be incorporated directly into the algorithm:

1. Prevention of duplicate terminal nodes: both a 0-leaf and a 1-leaf are kept track of during generation. Once a terminal node $A_i B_j$ is being generated, the stored 0- or 1-leaf is returned instead.
2. Prevention of duplicate, logically equivalent sub-trees: all generated triplets $(var(v), low(v), high(v))$ are stored in a hash map. If a node v' yielding a duplicate triplet is being generated, the existing node v will be re-used. As the algorithm builds the BDD depth-first, duplicate sub-trees are discarded in a bottom-up process.
3. Omitting nodes with $low(v) = high(v)$: if a node v is being generated with $low(v) = high(v)$, simply return $low(v)$ directly, discarding node v.

While these operations introduce a runtime and memory overhead, the amount of nodes present in the BDD during the computation is potentially greatly reduced. Furthermore, the discussed algorithm can also be adjusted to implement e.g. $f + g$. As previously noted, this is just one approach of computing $f \cdot g$ ROBDDs. For example, the approach of [YO97] constructs new ROBDDs in a breadth-first, parallelizable manner.

3.4 List of BDD operations

Several more operations on BDDs are used later on, whose implementation are outside of the scope of this thesis (see e.g. [Bry86, vD16] for possible implementations). As such, we will assume that existential quantification, restriction as well as the following additional operations are implemented on BDDs:

- Swap variables: given a BDD of $f(x_1, ..., x_n)$, a BDD of $g(y_1, ..., y_n)$ is created, where each variable x_i is renamed to y_i.
- (Inverse) Relational product: let S be a set, $R = \{(s_a, s_b), ...\}$ be a relation on S, and f_S, f_R be the corresponding Boolean functions. Then the relational product yields a Boolean function $f_{S'}$ representing the set $S' = \{s' \mid s \in S : \ (s, s') \in R\}$ (and for the inverse relational product the set $S' = \{s' \mid s \in S : \ (s', s) \in R\}$). The implementation of relations will be discussed in Chapter 4.2.
- Identity: given a set S, create a BDD representing the identity relation $R = \{(s, s) \mid s \in S\}$.

Chapter 4

MDP Representation

Automatically analyzing MDPs using computers requires storage for states and transitions within a limited amount of memory, and operations on these structures can be computionally expensive. Therefore, the implementation of them is detrimental to the practical runtime performance of model checking algorithms.

In this section, an overview of two representations, *sparse* and *symbolic*, is given. These representations assume an MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ with $|S| = n$ states and support storing and modifying vertices, edges and actions of G_{VBA} and G_{EBA}. Additionally, we will outline how $Post(V')/Pre(V')$ operations on a set of vertices V' are implemented. While these parts are sufficient to compute the MEC decomposition, we will support encoding the probabilities given by δ, as they are often required to model check properties of $\mathcal{M}$. The following descriptions for G_{EBA} are based on the implementations in the model checker Storm [HJK+21].

4.1 Sparse Representation

In the sparse representation, each single state and transition is explicitly stored: when referring to a set of vertices, a flag is stored for each vertex of the graph-like structure. As these flags can be packed into a set of ordered bits, $\lceil \frac{n}{8} \rceil$ Bytes are required for each set of vertices, where the i-th bit set indicates that vertex i is part of the set. Operations such as the intersection and union of two sets can be implemented by performing the equivalent logical operations (bitwise AND, bitwise OR) on each pair of bytes.

For G_{VBA}, the edges can be stored using an $n \times n$ transition matrix T, where the entry $t_{i,j} \neq 0$ indicates that s_i has an edge to s_j. If $s_i \in V_P$, then $t_{i,j} \in \{0, 1\}$ indicates whether a certain action $\alpha \in A$ can be chosen from state $s_i \in S$. Otherwise, $s_i \in V_R$ represents an action in A and $t_{i,j} \in [0, 1]$ encodes a probability given by δ.

For G_{EBA}, we need to be able to differentiate between two edges $(s_i, \alpha, s_j), (s_i, \beta, s_j) \in E$ using a transition matrix T consisting of n "row groups". Each state $s \in S$ has a

corresponding row group containing $|A[s]|$ rows. Each row encodes an enabled action $A[s]$ of s and each entry $t_{i,j}$ is storing a probability given by δ. For G_{EBA}, we are only concerned whether $t_{i,j} > 0$ or $t_{i,j} = 0$.

For both graph-like structures, the transition matrices of most MDPs are sparsely populated, but its size grows quadratically with the amount of states n of the MDP. As such, most implementations employ *sparse* matrices, in which only non-zero elements are stored.

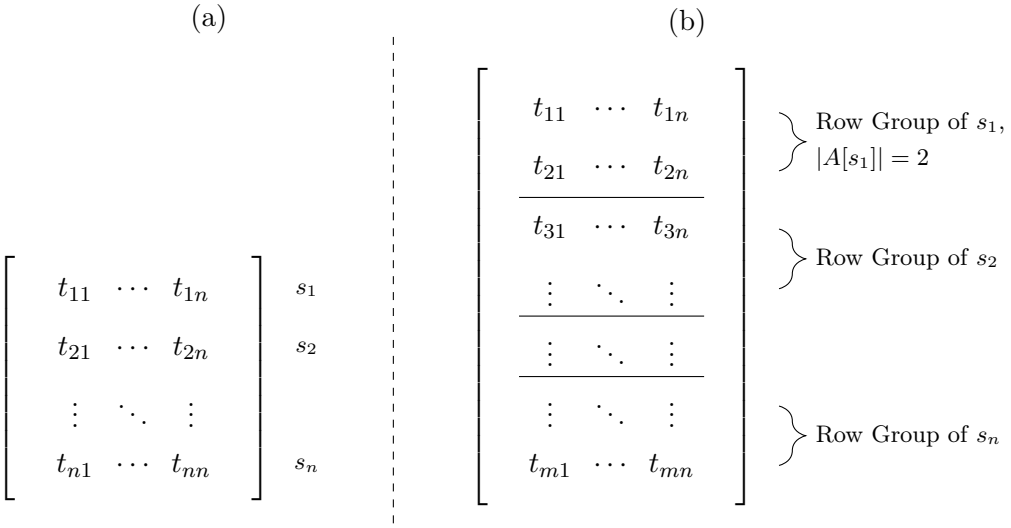


Figure 4.1: The transition matrices of G_{VBA} (a) and G_{EBA} (b).

To implement the $Post(V')$ operations, we need to process each vertex $v \in V'$ individually by storing all non-zero columns of the corresponding row (group) of T. The Pre operations can be implemented similarly by processing the corresponding column and collecting all rows (or row groups).

While this approach is easy to implement, its scalability is limited: storing all SCCs of a graph-like structure can quickly become infeasibly due to memory requirements, given that large MDPs can consist of billions of states with thousands of SCCs. Additionally, $Post(V')/Pre(V')$ operations become more computationally expensive the larger the set V' is, as each vertex (and edge) is processed individually.

4.2 Symbolic Representation

In the symbolic representation, one works with *sets* of vertices or edges, which are stored using BDDs. In the context of this thesis, we work with the assumption[1] that each vertex is assigned to a unique index $0 \leq i < n = |V|$, which can be encoded

[1]This assumption is a simplification of program variable valuations [JM09].

into $t := \lceil \log_2(n) \rceil$ Boolean variables $x_1, ..., x_t$ by using the bits of i. A set of vertices $V' \subseteq V$ can then be encoded into a BDD:

$$f_{V'} : \{0,1\}^t \to \{0,1\}, \quad f_{V'}(\underbrace{x_1, ..., x_t}_{\text{encodes } s \in V}) = \begin{cases} 1, & s \in V' \\ 0, & s \notin V' \end{cases}$$

For G_{VBA}, a set of edges can be encoded into an ADD similar to a transition matrix of a sparse representation: the ADD takes a set of Boolean arguments for the source vertex $s_i \in V$ (row) and another set of Boolean arguments for the destination vertex s_j (column) of an edge. If the ADD evaluates to a non-zero value, then an edge from s_i to s_j exists, and if $s_i \in V_R, s_j \in V_P$, then the ADD evaluates to a probability given by δ. For the computation of MECs, the ADD of all transitions can be converted to an (often smaller) BDD, representing the following Boolean function:

$$t_{VBA}(\underbrace{x_1, ..., x_t}_{\substack{s \in V \\ \text{(Row)}}}, \underbrace{x'_1, ..., x'_t}_{\substack{s' \in V \\ \text{(Column)}}}) = \begin{cases} 1, & (s, s') \in E \\ 0, & (s, s') \notin E \end{cases}$$

This effectively implements a relation R for vertices $v, v' \in V$, in which each element $(v, v') \in R$ represents an edge in G_{VBA}. As such, we can compute $Post(V')/Pre(V')$ of $V' \subseteq V$ with the (inverse) relational product operation using t_{VBA} and the BDD $f_{V'}$ of V'.

As G_{EBA} encodes actions into the edges of the graph, an additional set of Boolean arguments $y_1, ..., y_u$ is required for the transition BDD t_{EBA}. Here, the actions are encoded in conjunction with the vertex given by $x_1, ..., x_t$: instead of representing all $|A|$ actions, we only use u Boolean variables to encode $\max_{s \in S}\{|A[s]|\}$ different values, which results in a smaller BDD.

$$t_{EBA}(\underbrace{x_1, ..., x_t}_{\substack{s \in V \\ \text{(Row-Group)}}}, \underbrace{y_1, ..., y_u}_{\alpha \in A[s]}, \underbrace{x'_1, ..., x'_t}_{\substack{s' \in V \\ \text{(Column)}}}) = \begin{cases} 1, & (s, \alpha, s') \in E \\ 0, & (s, \alpha, s') \notin E \end{cases}$$

Remark 4.1 We convert t_{EBA} to the transition BDD of a directed graph by computing $\exists\{y_1, ..., y_u\}(t_{EBA})$. The (inverse) relational product on this modified transition BDD can then be used to obtain $Post(V')/Pre(V')$ for some $V' \subseteq V$.

While the implementation of BDDs and operations on them are more complex compared to the sparse approach, the major benefit of the symbolic representation is its scalability: as $Pre(V')$ and $Post(V')$ operations are performed on BDDs, all elements of $V' \subseteq V$ are processed at once. Additionally, the ability to compress BDDs (see Chapter 3) is crucial to be able to store large sets of vertices or transitions.

But as outlined by [BK08], no single data structure is able to compactly represent all Boolean functions, which means that there are MDPs which require very large BDDs. The size of each BDD is also heavily influenced by the chosen Boolean variable ordering. Although the optimal ordering for a given Boolean function is an NP-hard problem (see [BW96]), various heuristics exist depending on the problem being modelled. Here for transition functions, an interleaved ordering $x_1, x'_1, ..., x_t, x'_t(, y_1, y'_1, ..., y_u, y'_u)$ tends to yield good results [EFT93].

Chapter 5

Symbolic Algorithms

In this section we will cover three existing symbolic MEC decomposition algorithms. As we will rely on the computation of SCCs and additional sets such as $ROut(...)$, we will also outline their symbolic computations for both G_{EBA} and G_{VBA}. Additionally, a novel symbolic algorithm converting G_{EBA} into G_{VBA} is provided. Note that space and runtime complexity of symbolic algorithms is measured in *symbolic operations* and *symbolic space*, as these algorithms work independent of the underlying symbolic implementation.

5.1 G_{EBA} Conversion

In the following, it is assumed that the BDDs of G_{EBA} are structured as outlined in Chapter 4.2. The algorithm we propose for converting from G_{EBA} to G_{VBA} is described in Algorithm 2. Informally, this conversion requires either obtaining or converting to the following four parts of G_{VBA}:

- the transitions from V_R to V_P,
- the transitions from V_P to V_R,
- the player vertices V_P, and
- the random vertices V_R,

The core idea of the algorithm is the extension of the existing transition BDD t_{EBA} to have the same structure as t_{VBA} by adding additional Boolean variables (marked in red):

$$\text{Have:} \quad t_{EBA}(\underbrace{x_1, ..., x_t}_{\text{Source Vertex}}, \underbrace{y_1, ..., y_u}_{\text{Action}}, \underbrace{x'_1, ..., x'_t}_{\text{Target Vertex}})$$

$$\text{Want:} \quad t_{VBA}(\underbrace{x_1, ..., ..., ..., ..., x_s}_{\text{Source Vertex}}, \underbrace{x'_1, ..., ..., ..., ..., x'_s}_{\text{Target Vertex}})$$

$$t_{EBA} \text{ after conversion:} \quad t_{VBA}(\underbrace{x_1, ..., x_t, y_1, ..., y_u, z}_{\text{Source Vertex}}, \underbrace{x'_1, ..., x'_t, y'_1, ..., y'_u, z'}_{\text{Target Vertex}})$$

Intuitively, instead of limiting the use of the Boolean variables $y_1, ..., y_n$ to the transition BDD, we will use both $x_1, ..., x_t$ and $y_1, ..., y_u$ at all times to describe a vertex in the converted graph-like structure G_{VBA}. To avoid the edge case in which an action and a state are mapped to the same vertex, we use the additional Boolean variable z to indicate whether a vertex belongs to V_R or V_P. More formally, let $f_{V_{EBA}}$ be the BDD of the vertices of G_{EBA}, and let $f_{V_{VBA}}$ be the converted BDD for the vertices of G_{VBA}. The conversion algorithm uses the following convention:

$$f_{V_{VBA}}(\underbrace{x_1, ..., x_t,\ y_1, ..., y_u,\ z}_{\text{Vertex } v}) = \begin{cases} 1, & v \in V_P \wedge z = 0 \wedge \bigwedge\limits_{1 \leq i \leq u} (y_i = 0) \text{ and} \\ & f_{V_{EBA}}(x_1, ..., x_t) = 1 \\ 1, & v \in V_R \wedge z = 1 \text{ and} \\ & \text{there exists } x'_1, ..., x'_t \text{ such that} \\ & t_{EBA}(x_1, ..., x_t, y_1, ..., y_t, x'_1, ..., x'_t) = 1 \\ 0, & v \notin (V_P \cup V_R) \end{cases}$$

The original transition BDD t_{EBA} can be extended to follow these conventions, which will result in the transitions from V_R to V_P (see L. 15-19). To obtain the missing transitions from V_P to V_R, we first extract the actions of t_{EBA} (L. 6). By creating an identity BDD (L. 7), we can create a set of transitions consisting of self-edges on V_R (L. 7-8), which can be transformed to originate from their corresponding states (L. 9-13). This yields the transitions from V_P to V_R. These two sets of transitions can be combined to form the converted transition BDD t_{VBA} of G_{VBA} (L. 20).

The player vertices V_P are represented by the BDD $f_{V_{EBA}}$, which needs to be extended to follow our conventions (L. 23-26). The BDD of the random vertices V_R can be retrieved from t_{VBA} using the transitions whose origins start from a vertex with the z flag set to 1 (L. 28-29). Together with t_{VBA}, this results in a completed conversion from G_{EBA} to G_{VBA}.

Algorithm 2: Conversion of G_{EBA} to G_{VBA}.

Input : The vertices V_{EBA} and the transition BDD t_{EBA} of G_{EBA}

Output: The converted graph-like structure $G_{VBA} = (V, V_P, V_R, t_{VBA})$

```
// Create new Boolean variables not present in t_EBA
    z, z' := createBddVars(2)    // Flags
    y'_1, ..., y'_u := createBddVars(u)    // Missing column vars of y_1, ..., y_u

// Convert transitions
    // Create Player to Random vertices transitions T_{P→R}
        A := ∃{x'_1, ..., x'_t}(t_EBA)    // Actions of G_EBA
        I_0 := createIdentityRelationBdd((x_1, x'_1), ..., (y_1, y'_1), ...)
        I_1 := I_0 ∧ A.swapVariables((x_1, x'_1), ..., (y_1, y'_1), ...)
        T_{P→R} := ∃{y_1, ..., y_t}(I_1)
        foreach y_i ∈ {y_1, ..., y_u} do
            T_{P→R} := T_{P→R} ∧ createBdd(y_i = 0)
        T_{P→R} := T_{P→R} ∧ createBdd(z = 0)
        T_{P→R} := T_{P→R} ∧ createBdd(z' = 1)

    // Create Random to Player vertices transitions T_{R→P}
        T_{R→P} := t_EBA
        foreach y'_i ∈ {y'_1, ..., y'_u} do
            T_{R→P} := T_{R→P} ∧ createBdd(y'_i = 0)
        T_{R→P} := T_{R→P} ∧ createBdd(z = 1)
        T_{R→P} := T_{R→P} ∧ createBdd(z' = 0)

    t_VBA := T_{P→R} ∨ T_{R→P}

// Create vertices
    // Player vertices V_P
        V_P := V_EBA    // Vertices of G_EBA
        foreach y_i ∈ {y_1, ..., y_u} do
            V_P := V_P ∧ createBdd(y_i = 0)
        V_P := V_P ∧ createBdd(z = 0)

    // Random vertices V_R
        V_R := ∃{x'_1, ..., y'_1, ..., z'}(t_VBA)
        V_R := V_R ∧ createBdd(z = 1)

    V := V_R ∨ V_P

return (V, V_P, V_R, t_VBA)
```

Using the presented conversion process, the results on G_{VBA} can be trivially converted back into G_{EBA}: given the BDD $f_{V_1} = (x_1, ..., x_t, y_1, ..., y_u, z)$ of a set of player vertices $V_1 \subseteq V_P$, the corresponding vertices in G_{EBA} are given by $\exists\{y_1, ..., y_t, z\}(f_{V_1})$. For a set of random vertices $V_2 \subseteq V_R$ with the BDD $f_{V_2} = (x_1, ..., x_t, y_1, ..., y_u, z)$, the transitions t_{EBA} can be constrained to the corresponding actions of f_{V_2} using $t_{EBA} \wedge \exists z(f_B)$.

During this conversion process, we add $u + 2$ additional Boolean variables to the transition BDD, which will increase its size. Due to the usage of ROBDDs, it is difficult to tell how much the size of the converted transition BDD of G_{VBA} deviates from the original transition BDD of G_{EBA}. We will evaluate the BDD sizes before and after the conversion experimentally in Chapter 6.2.

Remark 5.1 The introduction of the Boolean variables z, z' might not be required in all instances. Furthermore, we can easily construct an MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ such that $2^{(u+t)} \gg (|S| + |A|)$, meaning the converted transition BDD (as well as every BDD of a set of vertices) uses more Boolean variables than required. Therefore, the BDDs of the converted graph-like structure G_{VBA} will likely contain more nodes than a "native" implementation, meaning an implementation of G_{VBA} which does not rely on the conversion from G_{EBA} to G_{VBA}. As such, we say that this conversion algorithm is *suboptimal* in regard to the size of the resulting BDDs of G_{VBA}.

5.2 SCC Decomposition

For sparse representations, the usual approach to build an SCC decomposition is to perform a depth-first search while labelling and storing information for each individual node. As described by [Tar72], all SCCs of a graph $G = (V, E)$ can be computed in $O(|V| + |E|)$ operations using $O(|V|)$ space. However, this approach is unsuitable for symbolic graphs, as the primary advantage of symbolic operations using BDDs is to work on sets of nodes simultaneously.

Symbolic algorithms generally compute SCCs using forward and backward sets. In the worst-case, a naive SCC decomposition algorithm runs in $O(|V|^2)$ steps on a directed graph G: starting from a vertex $v \in V$, all reachable vertices are computed by repeatedly performing *Post* operations, forming the forward set. Likewise, this process is repeated on v to compute its backwards set using *Pre* operations. Then, the SCC C which v is part of is the overlap between the forward and backward set (see Fig. 5.1). After removing C, this process is repeated until all vertices of G have been processed.

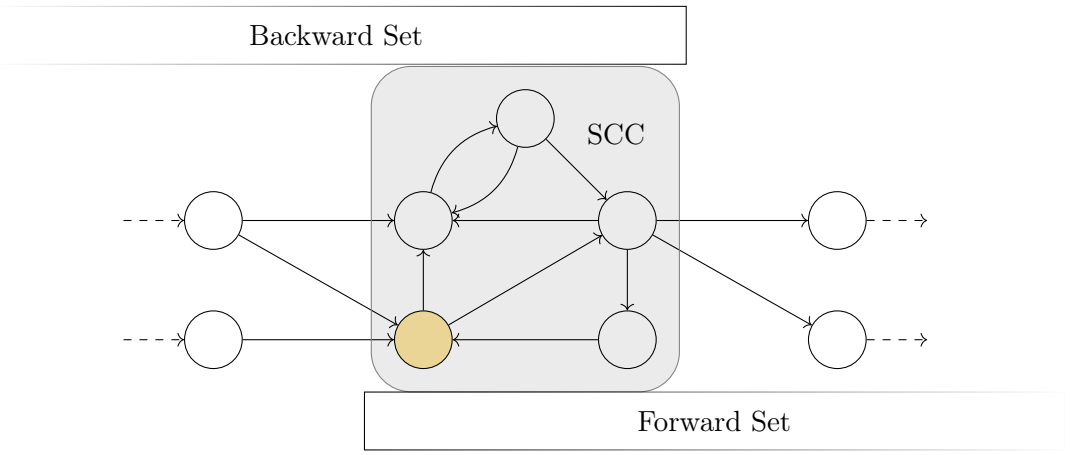


Figure 5.1: Excerpt of a directed graph. The SCC containing the vertex marked in yellow can be computed by retrieving the overlap between the forward set and the backward set of the marked vertex.

Various SCC decomposition algorithms exist which aim to improve the worst-case runtime of this naive approach (see [LSS+23] for an overview). But generally, they all build upon the computation of forward and backward sets while reducing the amount of redundant *Post*/*Pre* operations, e.g. by terminating the computation of a forward or backward set early if possible. To date, the best symbolic SCC decomposition algorithms run in $O(|V|)$ symbolic steps in the worst-case [GPP03, LSS+23]. Analysis by [CDHL18] provides lower bounds for the symbolic computability of SCCs, which proves the computational runtime complexity of $O(|V|)$ to be "essentially optimal"[1].

5.3 Random Attractor

On G_{VBA}, [CDHS21] computes the random attractor (see Def. 2.11) $Attr_R(S)$ of $S \subseteq V$ on G_{VBA} iteratively using

$$A_0 = S$$
$$A_{i+1} = A_i \cup \Big(\underbrace{Pre(A_i)}_{\substack{\text{All vertices which} \\ \text{point into } A_i}} \setminus \underbrace{(V_P \cap Pre(V \setminus A_i))}_{\substack{\text{Player vertices which} \\ \text{do not all point into } A_i}} \Big)$$

until the last set A_j equals the previous set A_{j-1}. We modify this algorithm to be usable on G_{EBA} by computing the set of states and the set of actions for each iteration separately. In Algorithm 3, we follow the assumption that we only compute the random attractor of $ROut(S)$ of an SCC S and that the BDDs of G_{EBA} are structured as outlined in Chapter 4.2.

[1] [CDHL18], p. 2349

We start by computing the new set of states to add, meaning all states whose enabled actions are *all* part of $Attr_R$. In Line 6, we store all states S_1 which have at least one enabled action *not* in $Attr_R$. We then compute all states S_2 which have an enabled action in $Attr_R$ and remove all states of S_1 from it. Afterwards we can add $S \cap S_2$ to $Attr_R$. With the updated set of states, we continue to compute a new set of actions to add: if any transition of S leads into a state contained in $Attr_R$, we can retrieve its action (A_1) and add it to $Attr_R$. If no new actions have been added, no new states will be added either and we can terminate this loop.

Algorithm 3: Computation of $Attr_R$ on G_{EBA}.

Input : G_{EBA},
the set of states S of the current SCC,
the set of actions A of which to compute the attractor of.

Output: R_n, which contains a set of states and a set of actions.

1 $R_n := (states := \emptyset, actions := A)$ `// Iteration i`
2 $R_c := R_n$ `// Iteration i+1`
3 **do**
4 $\quad R_c := R_n$
5 $\quad$ `// Update States`
6 $\quad S_1 := \exists\{y_1, ..., y_u, x'_1, ..., x'_t\}(t_{EBA} \wedge \neg R_c.actions)$
7 $\quad S_2 := (\exists\{y_1, ..., y_u\}(R_c.actions)) \wedge \neg S_1$
8 $\quad R_n.states := (R_c.states \vee S_2) \wedge S$
9 $\quad$ `// Update Actions`
10 $\quad S'_n := R_n.states.swapVariables((x_1, x'_1), ..., (x_t, x'_t))$
11 $\quad A_1 := \exists\{x'_1, ..., x'_t\}(S \wedge t_{EBA} \wedge S'_n)$
12 $\quad R_n.actions := R_c.actions \vee A_1$
13 **while** $R_c.actions \neq R_n.actions$
14 **return** R_n

5.4 Random Out

To identify whether an SCC S of G is an (M)EC, we need to check whether $ROut(S) = \emptyset$ (see Chapter 2.6). For G_{VBA}, we can retrieve these outgoing actions as vertices ([CHL+18, CDHS21]):

$$ROut(S) = \underbrace{Pre(V \setminus S)}_{\substack{\text{All vertices with} \\ \text{edge leaving } S}} \cap \underbrace{(S \cap V_R)}_{\substack{\text{Random vertices} \\ \text{of } S}}$$

For G_{EBA} with a transition BDD $t_{EBA}(x_1, ..., x_t, y_1, ..., y_u, x'_1, ..., x'_t)$, we adjust the computation as follows: let $f_S(x_1, ..., x_t), f'_S(x'_1, ..., x'_t)$ each be a BDD describing the vertices of S. Then the BDD of $ROut(S)$ containing outgoing actions can be computed with

$$f_{ROut(S)}(x_1, ..., x_t, y_1, ..., y_u) = \exists\{x'_1, ..., x'_t\}(t_{EBA} \wedge f_S \wedge \overline{f_{S'}}).$$

5.5 MEC Decomposition

In this subsection, an overview of three MEC decomposition algorithms Naive, Lock-Step and Collapsing on G_{EBA} and G_{VBA} is given [CHL+18, CDHS21]. We will assume that the graph-like structure G contains $|V| = n$ vertices and $|E| = m$ edges and that an SCC decomposition is computed in a linear amount of symbolic steps (using e.g. [GPP03, LSS+23]).

5.5.1 Algorithm Naive

The Naive MEC decomposition algorithm is a symbolic implementation of a basic MEC decomposition described in [DA98] (see e.g. [CDHS21]). Naive decomposes G into maximal, non-trivial SCCs and processes each SCC S as follows: if S has no leaving random edges ($ROut(S) = \emptyset$) then S is an MEC. Otherwise, the associated actions $\alpha \in ROut(S)$ are removed from G. As this might change the connectivity of S, S is recursively processed by re-computing its SCC decomposition. An execution of this algorithm can be seen in Fig. 5.2.

During the MEC decomposition of Naive, the same vertices (and edges) are likely to be traversed several times due to the recomputation of SCCs. As an optimization described and formally proven in [CH11, CHL+18], the random attractor $Attr_R(ROut(S))$ can be computed and removed from G instead (see Fig. 5.3). Regardless of the optimization, the algorithm requires $O(n^2)$ symbolic steps with $O(\log n)$ symbolic space in the worst-case [CDHS21].

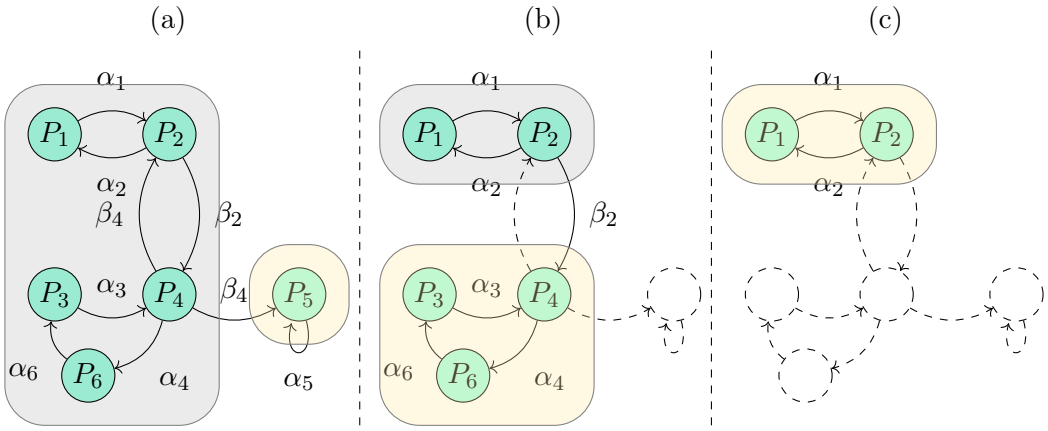


Figure 5.2: Execution of NAIVE:
(a): G_{EBA} is decomposed into SCCs. The SCC $\{P_5\}$ with the action α_5 is identified as an MEC (yellow) due to $ROut(\{P_5\}) = \emptyset$. The other SCC C is not an MEC (gray) due to the outgoing action β_4.
(b): After the removal of β_4, another SCC decomposition on C is performed. The SCC $\{P_3, P_4, P_6\}$ has no outgoing actions and is thus identified as an MEC with its actions $\{\alpha_3, \alpha_4, \alpha_6\}$. The other SCC $\{P_1, P_2\}$ is not an MEC due to the outgoing action β_2.
(c): After the removal of β_2, the final SCC decomposition on $\{P_1, P_2\}$ is performed. The resulting SCC has no outgoing edges and is identified as an MEC with its actions $\{\alpha_1, \alpha_2\}$.

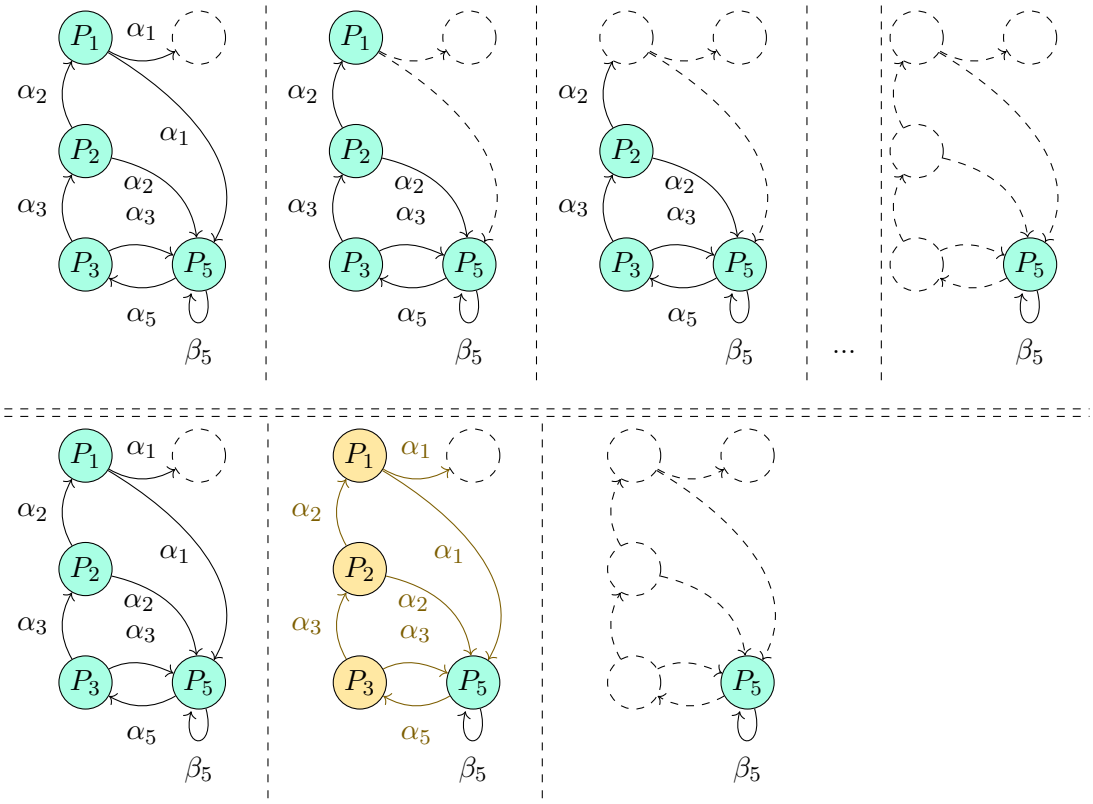


Figure 5.3: In the most basic version of NAIVE, the algorithm only removes the outgoing actions $ROut(...)$ and then recomputes an SCC decomposition. This process is repeated several times until the MEC is identified (top). By removing $Attr_R(ROut(...))$ (yellow) instead, the vertices $\{P_1, P_2, P_3\}$ and the actions $\{\alpha_1, ..., \alpha_5\}$ are removed, which reduces the amount of SCC decompositions performed to achieve the same result (bottom).

5.5.2 Algorithm LockStep

The algorithm LockStep by Chatterjee et al. [CHL+18] improves the worst-case runtime of Naive to $O(n \cdot \sqrt{m})$ symbolic operations with $O(\sqrt{m})$ symbolic space ([CDHS21]) by providing a symbolic implementation of a strategy used for sparse MEC decomposition (see [CH11]). The core idea of LockStep is to identify bottom SCCs (see Def. 2.3) quickly using a lockstep search.

To understand in which scenario Naive can be improved upon, let us take a look at the example shown in Fig. 5.4 of an SCC S. Naive proceeds as follows:

- We identify and remove the outgoing actions γ_1, β_4 of S.
- We perform an SCC decomposition on S, where we *traverse over the large EC* and identify the two SCCs A, B.
- When processing A, we identify the outgoing action β_1 and need to perform another SCC decomposition of A, where we *traverse over the large EC once more.*

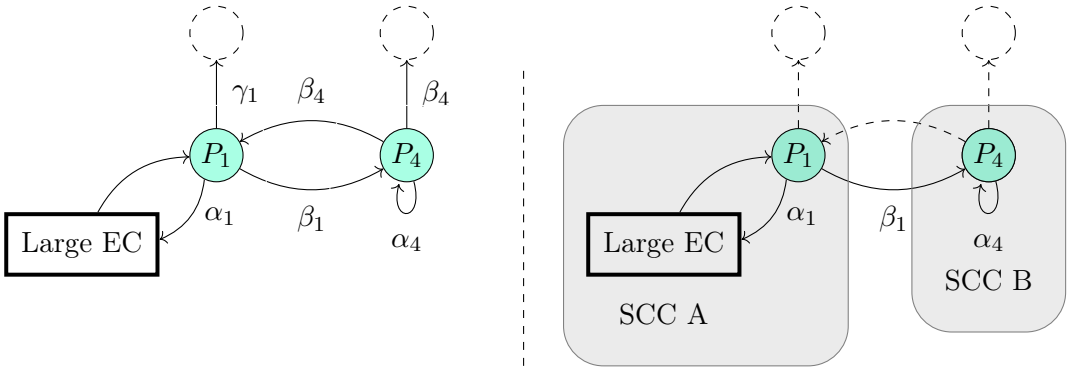


Figure 5.4: The pictured SCC S on G_{EBA} has two outgoing actions γ_1, β_4, whose removal will split S into two SCCs A, B (right).

Here, the early identification of the bottom SCC A is beneficial, as LockStep runs as follows:

- Like Naive, LockStep identifies and removes the outgoing actions γ_1, β_4.
- Now a lockstep-search is started on every vertex which have lost an edge (P_1, P_4) to find a bottom SCC.
- The search terminates before traversing most of the large EC and yields the bottom SCC B, which is identified as an MEC and removed from C.
- C is now the same as A in Naive, but we avoided one traversal of the large EC.

LockStep exploits the fact that, after the removal of all outgoing of an SCC, at least one bottom SCC exists (which might be trivial): let S be obtained after removing $Attr_R(ROut(...))$ from some SCC which is not an MEC. If S is still strongly connected, then S has no outgoing actions, which means S has no outgoing edges and is thus a bottom SCC. If S is not strongly connected, then the SCC decomposition of S must contain at least one top and one bottom SCC which are disjoint (as noted by [CHL+18]). Crucially, if $T_S \subseteq S$ is the set of vertices which have lost an outgoing edge, then each bottom SCC of S has at least one vertex contained in T_S. If $v_b \in T_S$ is a vertex of a bottom SCC, then exploring all reachable vertices from v_b using repeated *Post* operations yields the bottom SCC. The lockstep search conducts $|T_S|$ searches in parallel by performing one *Post*-operation in each iteration for each search and returns the earliest found bottom SCC.

Chatterjee et al. further optimize the lockstep search by aborting a search early if possible. Consider the searches from two vertices $v_A, v_B \in T_S$. where v_B is reachable from v_A. If v_A and v_B belong to the same SCC, then v_A is reachable from v_B as well. If v_A and v_B belong to disjoint SCCs, then the SCC of v_A has an outgoing edge, meaning it cannot be a bottom SCC. In both cases, the search from v_A and does not need to be reconsidered until the vertex v_A has lost another edge.

A complete execution of LockStep can be seen in Fig. 5.5. To summarize: LockStep first computes the SCC decomposition of the graph-like structure G and processes each SCC S as follows: $Attr_R(ROut(S))$ is computed and removed from S, after which T_S is updated accordingly. Given $|E| = m$,

- if $|T_S| = 0$, S has no outgoing actions and is identified as an MEC (if S is non-trivial).
- if $|T_S| < \sqrt{m}$, a lockstep search is started on T_S until a bottom SCC C is found. If C is non-trivial, it is identified as an MEC. C is removed from S and T_S is updated accordingly, after which S is reprocessed.
- if $|T_S| \geq \sqrt{m}$, then the lockstep search would be too expensive. Instead, we compute the SCC decomposition of S and process each SCC recursively.

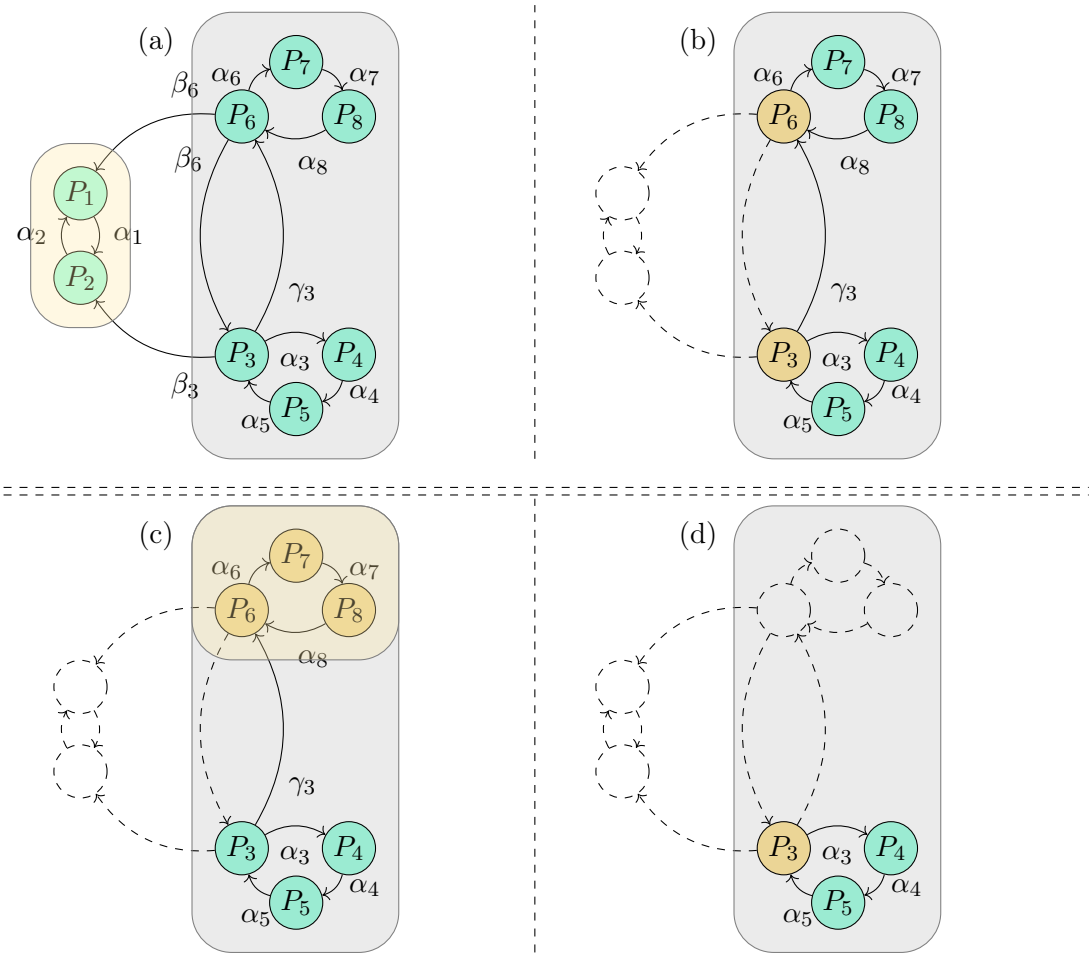


Figure 5.5: Execution of LockStep:
(a): G_{EBA} is decomposed into SCCs. The SCC $\{P_1, P_2\}$ is identified as an MEC with the actions $\{\alpha_1, \alpha_2\}$. The other SCC is not an MEC (gray) due to the outgoing actions $ROut(...) = \{\beta_3, \beta_6\}$.
(b): After the removal of β_3 and β_6, a lockstep search is started from the vertices $\{P_3, P_6\}$ which have lost an outgoing edge.
(c): The search from P_3 is aborted as it encounters the starting vertex P_6 of another search. The search of P_6 continues and returns a non-trivial bottom SCC $\{P_6, P_7, P_8\}$, which is identified as an MEC with the actions $\{\alpha_6, \alpha_7, \alpha_8\}$.
(d): The remaining SCC is not an MEC due to $ROut(...) = \{\gamma_3\}$. The action γ_3 is removed and a lockstep search is started from P_3, which will later return the final MEC with its actions $\{\alpha_3, \alpha_4, \alpha_5\}$.

The authors of LockStep assume a graph-like structure G_{VBA}. To compute the set of vertices T_S of the SCC S which have just lost an outgoing edge, let $A = Attr_R(ROut(S))$ be the set of vertices which have been removed from S. Then, T_S can be computed using $T_S = Pre(A) \cap S$ [CHL+18]. Assuming a graph-like structure as shown in Chapter 4.2, we adjust the computation for G_{EBA} as follows: Let $f_A(x_1, ..., x_t, y_1, ..., y_u)$ be the BDD of the actions from the random attractor (see 5.3) and let $f_S = (x_1, ..., x_t)$ be the BDD of the vertices of S. Then we can compute the BDD f_{T_S} of T_S with

$$f_{T_S}(x_1, ..., x_t) = (\exists\{y_1, ..., y_u\}(f_A)) \wedge f_S$$

Additionally, we need to preprocess the transition BDD of G_{EBA} before we can perform $Pre/Post$ operations (Remark 4.1). The original implementation of LockStep does not check whether $Attr(ROut(S)) = \emptyset$; instead it only checks the set T_S. On G_{EBA}, we add an additional check to see whether $Attr(ROut(S)) = \emptyset$ in order to only preprocess the transition BDD again if an action was removed. This avoids redundant symbolic operations in an edge case where multiple lockstep searches are performed consecutively without the removal of any actions (see Fig. 5.6).

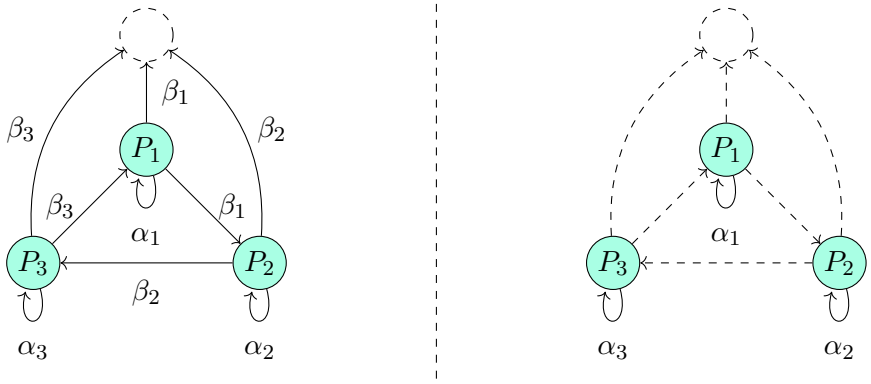


Figure 5.6: An instance in which LockStep would perform multiple lockstep searches consecutively without removing any actions from the transition BDD of G_{EBA}.

5.5.3 Algorithm Collapsing

In [CDHS21], a symbolic algorithm for MEC decomposition on G_{VBA} is presented with a runtime of $O(n^{2-\varepsilon} \log n)$ symbolic steps and $O(n^{\varepsilon} \log n)$ symbolic space in the worst-case, with $0 < \varepsilon \leq 0.5$ as a trade-off parameter. The algorithm computes all MECs in two passes: in the first pass, all vertices of non-trivial ECs of G_{VBA} are detected and stored. Afterwards in the second pass, the SCC decomposition of the vertices of all ECs is computed, which yields the MEC decomposition. The main speedup of the algorithm comes from the fast detection of ECs. Given a set of vertices X of G_{VBA}, one can decide whether X is an EC or not. In contrast, we might have to consider all of G_{VBA} in order to decide whether X is a maximal EC (see Fig. 5.7).

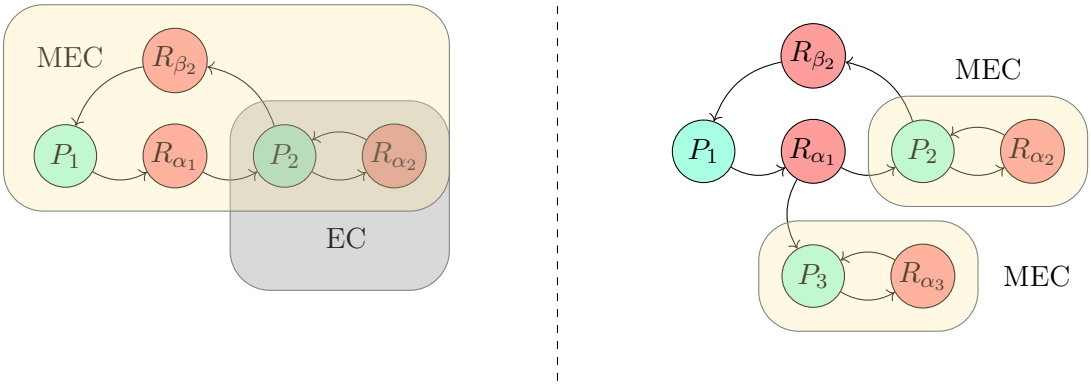


Figure 5.7: Two similar graph-like structures G_{VBA}. Here, the set of vertices $\{P_2, R_{\alpha_2}\}$ always make up an EC, but whether they are an MEC depends on the rest of G_{VBA}.

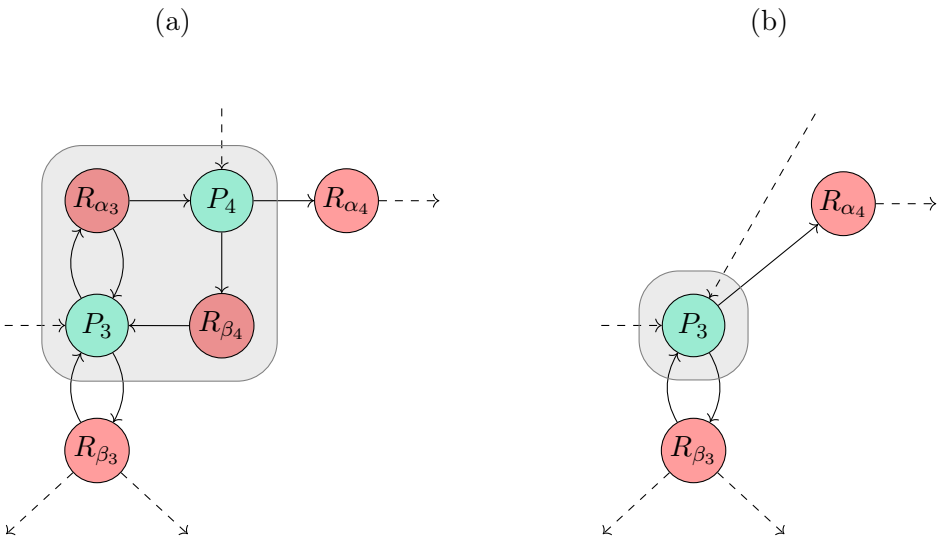


Figure 5.8: Collapsing of a detected EC (a) in G_{VBA} into a single vertex (b).

One fundamental operation used in this algorithm is the notion of *collapsing* an EC: given G_{VBA}, let an EC be given by a set of vertices X. The EC is collapsed by picking a player vertex $v \in X \cap V_P$, redirecting all incoming edges of X into v, redirecting all outgoing edges of X from v, and removing all other edges within X as well as the vertices $X \setminus \{v\}$ from G_{VBA}. An example of this operation can be seen in Fig. 5.8.

The faster detection of ECs is achieved by reducing the cost of duplicate vertex traversals on large SCCs by detecting and collapsing ECs incrementally using q-separators (Def. 2.5). An example of the first pass, which works with a copy of the original structure of G_{VBA}, can be seen in Fig. 5.9: whenever an SCC S is identified as an EC, it is collapsed. Otherwise, the attractor of $ROut(S)$ is removed from S and a q-separator T is computed if possible:

- If T was computed, all ECs of the smaller SCCs in $S \setminus Attr_R(T)$ are detected and collapsed. The remaining ECs of S are incrementally identified by iteratively removing a single vertex $v \in T$ from T and searching for an emerged EC in the updated $S \setminus T$. Due to the collapsed ECs within S, the re-traversal of S is now faster compared to the original graph.
- If T cannot be computed, then S has a small diameter (Def. 2.4) and is recursively processed using its SCC decomposition.

Given a set of vertices S, the computation of T can be performed symbolically: as explained in [CHI+16], the q-separator can be computed by building a BFS tree from a vertex $v \in S$. Removing a layer L_i of said tree from S necessarily splits S into (at least) two SCCs. Given a large enough diameter of S, a layer L_i exists which is a q-separator. [CDHS21] describes how to symbolically compute said layer using $Post/Pre$ operations. The parameter ε of the algorithm determines the required diameter of S and the quality q of the separator.

Remark 5.2 If Collapsing cannot compute a single q-separator throughout the entire MEC decomposition, then Collapsing essentially degrades into Naive: In the first pass, all (M)ECs are detected and collapsed on a copy of G_{VBA}. The detection of (M)ECs is using the same strategy as Naive, as no q-separator is computed and therefore no incremental EC detection on collapsed ECs is performed. In the second pass, the SCC decomposition of the detected vertices yield the MEC decomposition.

However using the symbolic representation shown in Chapter 4.2, this algorithm is not applicable on G_{EBA} due to the collapse operation on ECs: to collapse an EC X, we need to adjust the incoming and outgoing edges of X, which means we need to modify the transition BDD t_{EBA} of $G_{EBA} = (V, E)$:

$$t_{EBA}(\underbrace{x_1, ..., x_t}_{\substack{\text{Source Vertex}\\ s \in V}}, \underbrace{y_1, ..., y_u}_{\substack{\text{Action}\\ \alpha \in A[s]}}, \underbrace{x'_1, ..., x'_t}_{\substack{\text{Destination Vertex}\\ s' \in V}}) = \begin{cases} 1, & (s, \alpha, s') \in E \\ 0, & (s, \alpha, s') \notin E \end{cases}$$

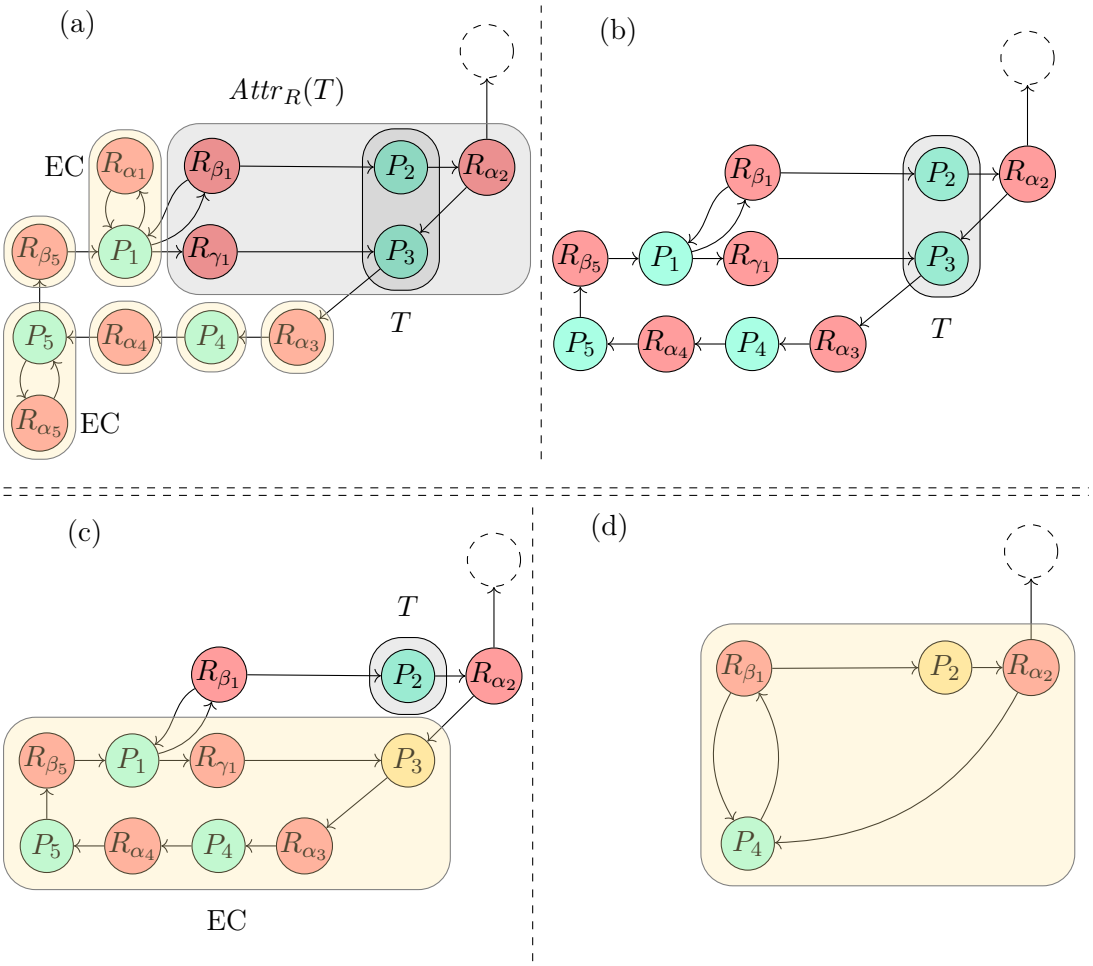


Figure 5.9: Execution of the first pass of Collapsing:
(a): Given the SCC S, a separator $T := \{P_2, P_3\}$ is computed and $S \setminus Attr_R(T)$ is decomposed into SCCs (yellow), which are individually processed. Two of the SCCs are identified as ECs.
(b): The identified ECs are each collapsed into a single vertex. Now T will be added back into S one vertex at a time.
(c): P_3 is removed from T and a search for an SCC is started in P_3 on $S \setminus T$. The SCC is identified as an EC.
(d): The identified EC is collapsed into a single vertex. P_2 is removed from T and a search for an SCC is started in P_2 on $S \setminus T$. The SCC is not an EC. As T is empty, S is finished processing.

But to keep the transition BDD small, an action $\alpha \in A$ of the MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta, r)$ is represented by t_{EBA} using *both* the Boolean variables $x_1, ..., x_t$ of a state $s \in S$ and the Boolean variables $y_1, ..., y_u$ of an action $\alpha \in A[s]$. When adjusting the source vertex of an outgoing edge of X, we inadvertently change the action of the transition. This change can result in the merge of two originally separate actions, as can be seen in Fig. 5.10. After a merge of actions, the correctness of detecting the ECs and therefore the correctness of the computed MEC decomposition cannot be guaranteed.

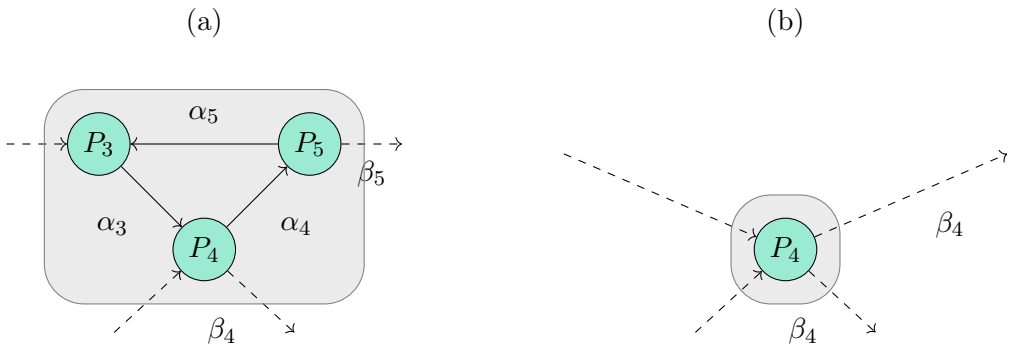


Figure 5.10: Collapsing of a detected EC in G_{EBA} (a) into a single vertex (b) leads to the merging of the distinct actions β_4, β_5 due to the construction of the transition BDD t_{EBA}.

There are several strategies to circumvent this issue, for example: the overlap between actions can be avoided if the symbolic representation of G_{EBA} is changed such that the actions are not encoded in conjunction with the vertices. However, the size of the transition BDD would likely be significantly larger and scale worse with the amount of states in $\mathcal{M}$. Another approach could be to convert one of the colliding actions into a different action, which would be unused or added to the transition BDD by extending G_{EBA} with additional Boolean variables. This would also require additional logic to create, update and maintain a mapping of the modified actions, as the same action might be part of several collapse operations. In this thesis we explore the approach of symbolically converting G_{EBA} into G_{VBA}, which leads to a larger transition BDD (Remark 5.1). As discussed in Chapter 5.1, the results obtained on the converted structure G_{VBA} can be translated back into the original edge-based representation G_{EBA}.

Chapter 6

Evaluation

The two MEC decomposition algorithms LockStep and Collapsing of Chatterjee et al. [CHL+18, CDHS21] improve upon the Naive algorithm by lowering the required amount of symbolic operations in a worst-case scenario for sufficiently large inputs. Due to the lack of benchmarks comparing symbolic MEC algorithms, it is unknown how they compare to another in practice. As such, we aim to answer the following research questions:

1. Is the symbolic conversion algorithm (Chapter 5.1) a viable strategy to use algorithms designed for G_{VBA} on implementations using G_{EBA}?
2. When compared to Naive, do empirical results reflect the worst-case theoretical improvements of the symbolic MEC decomposition algorithms LockStep and Collapsing in regard to runtime and the amount of symbolic operations performed?
3. Which graph-like structure is more efficient in regard to the symbolic computation of an MEC decomposition: G_{EBA} or G_{VBA}?

We aim to answer these questions in order by experimentally evaluating Naive, LockStep and Collapsing for both graph-like structures (when possible) by comparing both the runtime and the amount of symbolic operations of the algorithms. Similarly, we will evaluate the algorithm which converts G_{EBA} into G_{VBA} by looking at the runtime performance as well as the size of the generated transition BDDs.

6.1 Setup

Implementation. All three symbolic MEC algorithms were implemented into a custom build of the model checker Storm [HJK+21] using the SCC decomposition algorithm of Gentilini et al. [GPP03], which is the algorithm referenced in the LockStep / Collapsing publications [CHL+18, CDHS21]. For Collapsing, the authors obtain

a space-time tradeoff[1] by setting a parameter γ such that $(2\sqrt{n}+2)\log(n) \leq \gamma \leq n$. Here, we use $\gamma := (2\sqrt{n}+2)\log(n)$ (unless noted otherwise) to achieve the theoretical best runtime performance in a worst-case scenario. All code, benchmark files and generated logs are available at [Fab23].

Benchmarks. The benchmarks consist of the *quantitative verification benchmark set* [HKP+19] of which all MDP models and the underlying MDPs of the Markov automata models were considered. Storm constructs the MDPs as graph-like structures G_{EBA}. The BDD operations are implemented using BDD libraries, where Storm supports using either Cudd [Som97] or the multi-threaded library Sylvan [vD16]. Here, we ran benchmarks for both BDD libraries, where Sylvan was configured to use 1, 4 and 8 threads. To evaluate the benchmarks as G_{VBA}, each benchmark is symbolically converted into using vertex-based actions as shown in Chapter 5.1.

For each of the 379 benchmarks, Storm constructs G_{EBA}, optionally performs a conversion to G_{VBA} and computes an MEC decomposition with a total time limit of an hour. In the following sections when referring to runtime data, we will mostly focus on the 189 benchmarks of which at least one MEC decomposition was computed in time. For each benchmark, the runtime of conversion to G_{VBA} as well as the MEC decomposition itself were both measured independently of the time it took for Storm to construct G_{EBA}. Here, "TO" is used to indicate that a benchmark ran out of time, while "ME" indicates that a benchmark ran out of memory.

To gather data about the amount of symbolic operations, we restrict the benchmark set to the 177 benchmarks of which all MEC decompositions algorithms terminated in time. These benchmarks were re-run without a timelimit, where the amount of symbolic operations during the MEC decomposition algorithms were counted, which includes the operations performed during the SCC decomposition. Complexity analyses of symbolic algorithms usually focus on the amount of *Pre*/*Post* operations, which tend to be more computationally expensive as they operate on the larger transition BDD [CHL+18]. As such, we will focus on these operations.

Hardware. All benchmarks were performed on machines equipped with Intel Xeon Platinum 8160 Processors, on which 8 threads with 32GB of RAM were allocated for each benchmark.

Quantile and Scatter Plots. The span of our results is quite large. As an example, some benchmarks contain transition BDDs consisting of a few dozen nodes, while other transition BDDs contain millions of nodes. We will therefore be using *quantile plots* and *scatter plots* extensively to evaluate and visualize the results of these benchmarks.

A quantile plot is best explained by example; see e.g. Fig. 6.1 where the size of transition BDDs are evaluated: the transition BDD of G_{VBA} is measured on a set of

[1] [CDHS21], p. 12

benchmarks, which are sorted by the amount of nodes of the BDD. The results are visualized by the blue line: a point (x, y) on the line means x benchmarks contain a transition BDD with *at most* y nodes. This is repeated for i.e. the transition BDDs of G_{EBA}, which yields two additional lines. The resulting plot can then be used as an overview of the resulting BDD sizes over all benchmarks.

To compare the results of individiual benchmarks directly, scatter plots are provided in addition to the quantile plots. Resuming from the previous example, both axes of the left plot in Fig. 6.3 now indicate the size of the BDD. Each marked point (x, y) represents a single benchmark, where x is the amount of nodes of the transition BDD in G_{EBA}, and y is the amount after the conversion to G_{VBA} using the same benchmark.

6.2 MDP Conversion

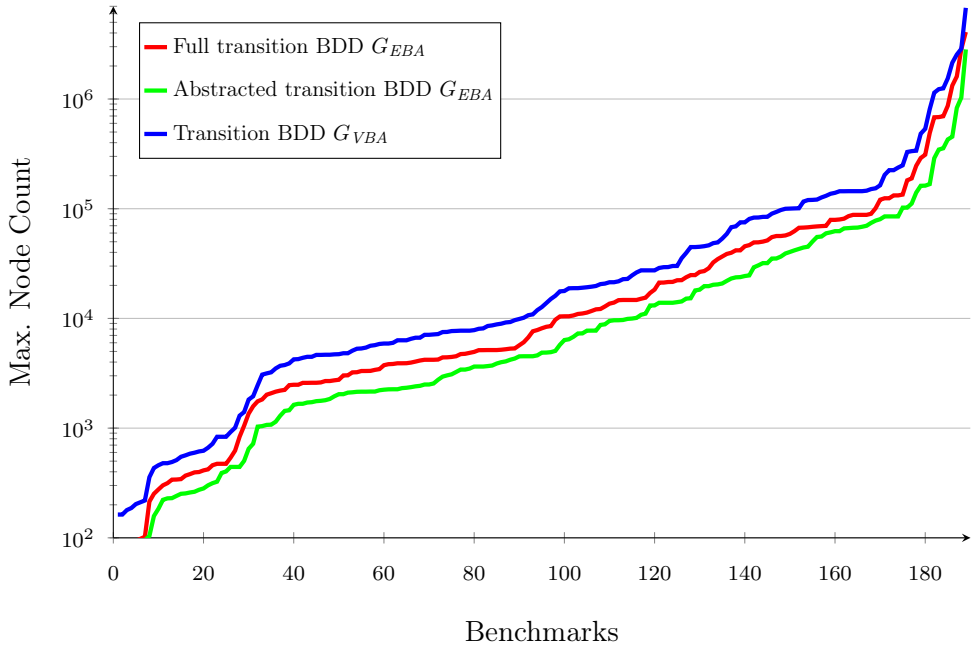


Figure 6.1: Quantile plot of transition BDD sizes.

For G_{EBA}, we differentiate between two transition BDDs: the *full* transition BDD contains information about each action of G_{EBA}. Set operations such as the removal of actions are performed on this larger BDD. *Pre*/*Post* operations are performed using the *abstracted* transition BDD (Remark 4.1). The abstracted BDD does not differentiate between actions and is computed from the full BDD using an *existsAbstract* operation (see Chapter 4.2). For G_{VBA}, no such differentiation exists as the actions are encoded into the vertices.

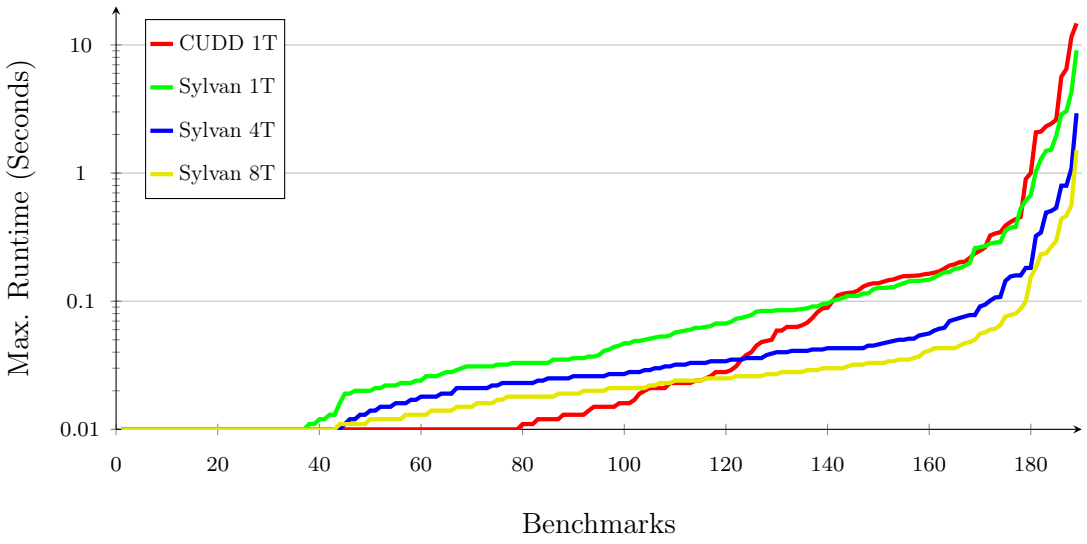


Figure 6.2: Quantile plot of the symbolic $G_{EBA} \rightarrow G_{VBA}$ conversion for multiple BDD library configurations.

To get an overview of the various MDP sizes, the quantile plot of Fig. 6.1 shows the node count of the transition BDDs. As the conversion from G_{EBA} to G_{VBA} is done symbolically, the conversion process itself runs quickly, as shown in Fig. 6.2 for various BDD library configurations. In most benchmarks, the conversion can be performed within 100 milliseconds, while only the largest transition BDDs require more than a second to convert. As we will see in the following section, the longest MEC decompositions of this set of benchmarks are close to the time limit of an hour. Consequently, the execution time of the conversion process is negligible when performing MEC decompositions on large BDDs.

In Fig. 6.3, we can observe that the converted transition BDDs of G_{VBA} tend to be roughly twice the size compared to the full transition BDD of G_{EBA}. This effect only worsens when comparing to the abstracted BDD, while some benchmarks show an increased BDD size by one order of magnitude. We should therefore expect a noticably higher runtime cost per $Pre/Post$ operation when working with the converted G_{VBA}.

In summary to answer research question 1, the conversion process itself runs fast due to its symbolic nature, but subsequent symbolic operations on the converted graph-like structure G_{VBA} perform slower due to a significant increase in the size of the transition BDD. The conversion algorithm should therefore only be used if either the algorithm on G_{VBA} performs several times fewer symbolic operations than its equivalent on G_{EBA} or if no algorithm on G_{EBA} exists in the first place.

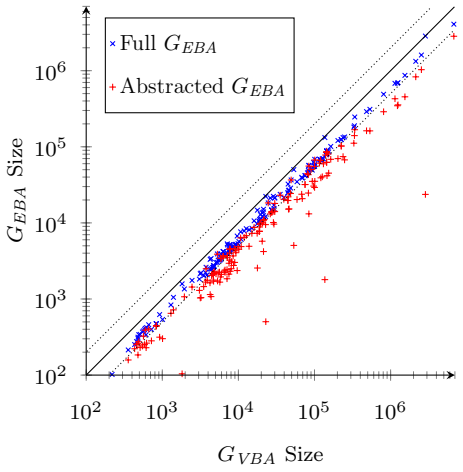


Figure 6.3: Comparison of the transition BDDs used in the "native" G_{EBA} and after the conversion to G_{VBA} by BDD node count.

6.3 MEC Decomposition Algorithms

The runtime results for each algorithm for both CUDD and SYLVAN can be seen in the quantile plots of Fig. 6.4. While the conversion algorithm has been used to obtain G_{VBA} from G_{EBA}, the runtime as well as the amount of symbolic operations of the conversion process itself are excluded from all figures of this section. We observe the general trend that SYLVAN with four threads seems to be one of the fastest configurations for this set of benchmarks; the benefits of adding four additional threads seem to be negligible. In comparison, CUDD outperforms multi-threaded SYLVAN mostly on the fastest MEC decomposition benchmarks. Notably, the gap in performance between SYLVAN and CUDD seems to be especially large when using LOCKSTEP.

To compare the runtimes of the algorithms against each other, we focus on the configuration using SYLVAN with four CPU threads. The quantile plot of Fig. 6.5 compares the overall runtime performance of the MEC decomposition algorithms for both graph-like structures.

Overall, LOCKSTEP seems to have the worst performance for either graph-like structure. When comparing LOCKSTEP to NAIVE in Fig. 6.6, we can see that LOCKSTEP is only slightly more performant in very few benchmarks, while NAIVE seems to be more performant in most other instances, regardless of the graph-like structure used or the overall runtime duration. Similiar results can be seen when measuring the amount of symbolic operations. This indicates that the worst-case scenarios in which LOCKSTEP theoretically improves upon NAIVE do not occur often enough in practice in order to yield a performance improvement.

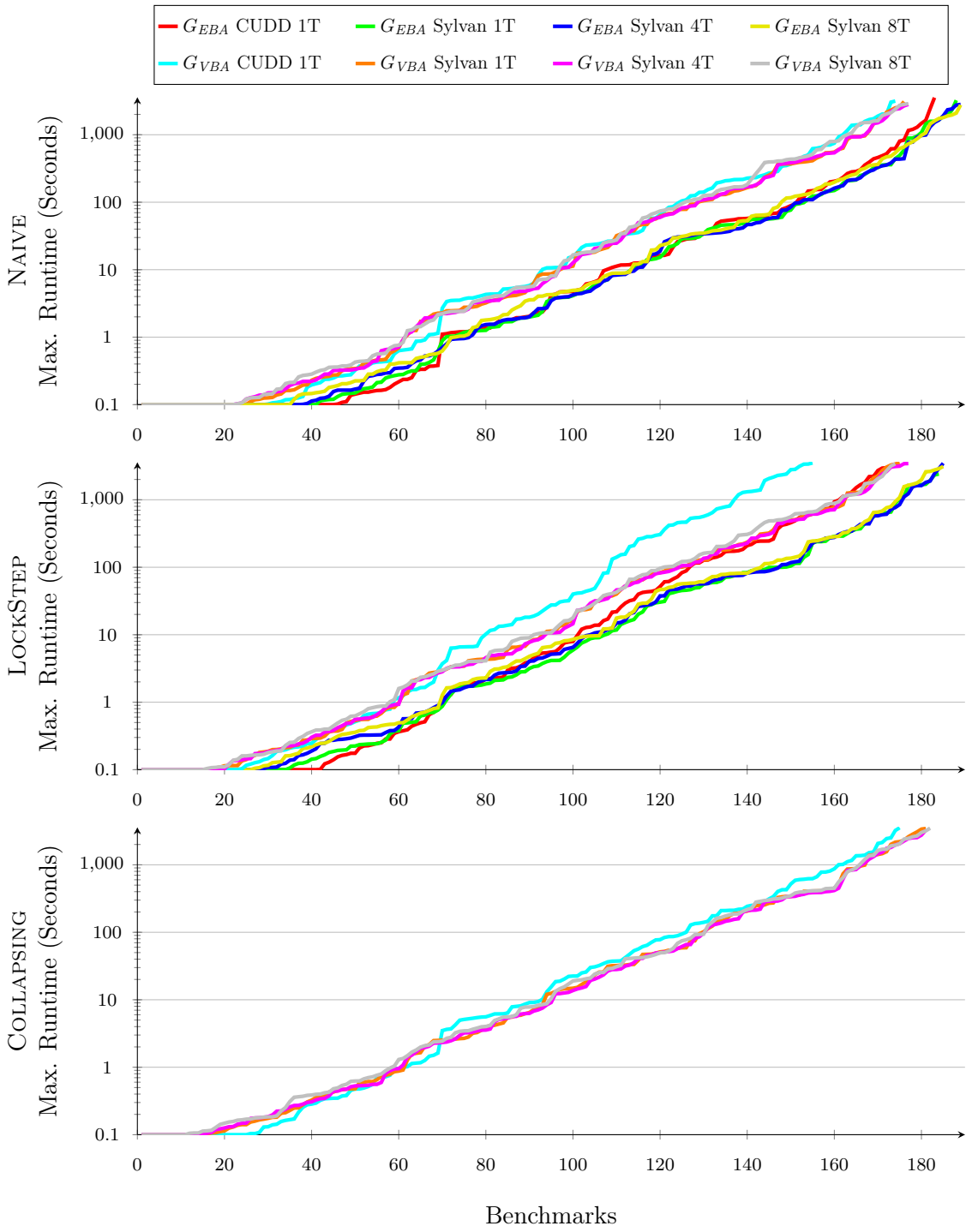


Figure 6.4: Quantile plots showing the benchmarked runtimes of all three symbolic MEC decomposition algorithms for both G_{EBA} and G_{VBA} (when possible) using various BDD library configurations.

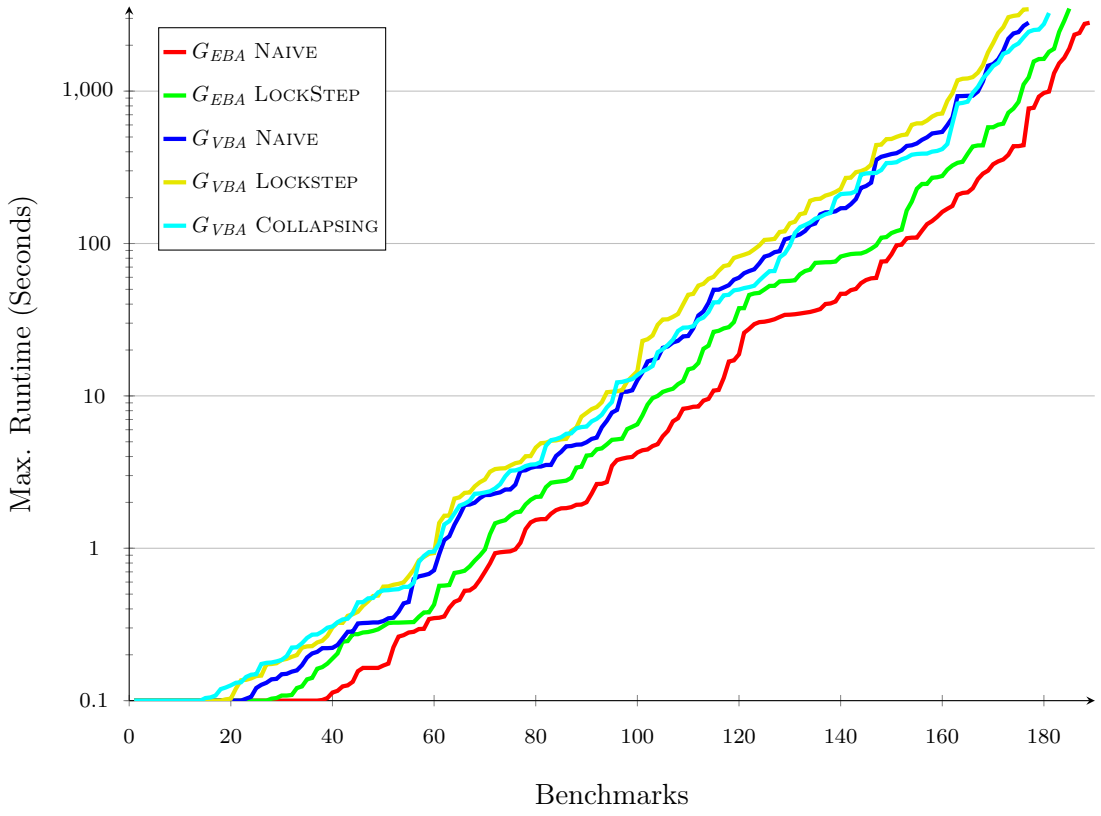


Figure 6.5: Quantile plot showing the runtimes of the symbolic MEC decomposition algorithms for both G_{EBA} and G_{VBA}.

In contrast, the results of Collapsing seem more mixed: the runtime scatter plot in Fig. 6.7 shows that Naive is more performant on quick MEC decompositions, while Collapsing becomes more competitive with longer runtime. Most interestingly, Naive still uses less symbolic *Pre*/*Post* operations than Collapsing in *all* benchmarks.

In Fig. 6.8, we compare the runtimes and the amount of symbolic operations of Collapsing using various γ. Collapsing requires an SCC to have a diameter (Def. 2.5) of at least γ in order to be able to compute a q-separator [CDHS21]. Therefore for $\gamma = n = |V|$, no q-separator will be (successfully) computed. When looking at the amount of symbolic operations, over 80% of the benchmarks of Collapsing with $\gamma = (2\sqrt{n} + 2)\log(n)$ perform an exactly equal amount of symbolic operations as Collapsing with $\gamma = n$. This indicates that in most instances, Collapsing fails to compute any q-separator and therefore follows the same strategy as Naive while performing strictly more work (see Remark 5.2). This raises the question of where the performance improvements of Collapsing over Naive are coming from.

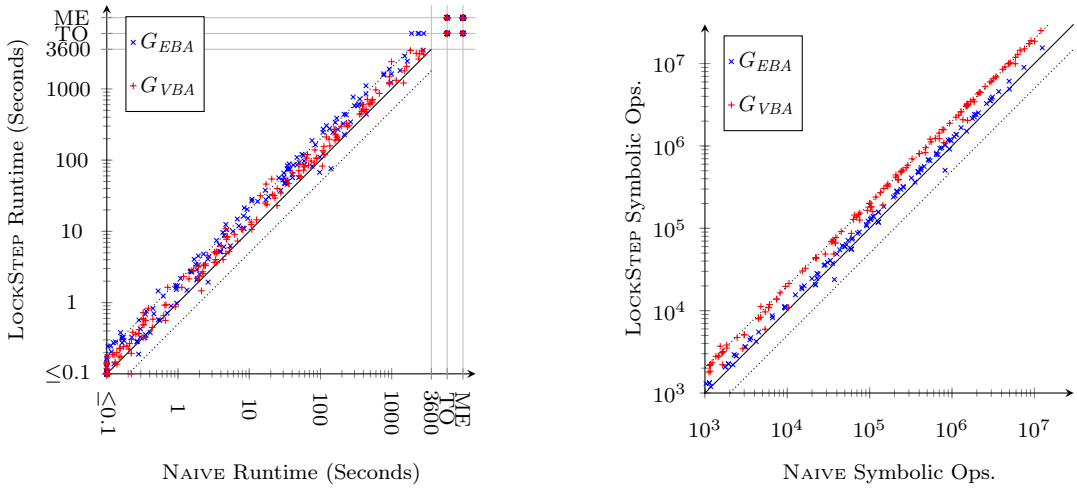


Figure 6.6: Comparison of LockStep to Naive for both G_{EBA} and G_{VBA} by runtime (in seconds, left) and symbolic operations ($Pre + Post$, right).

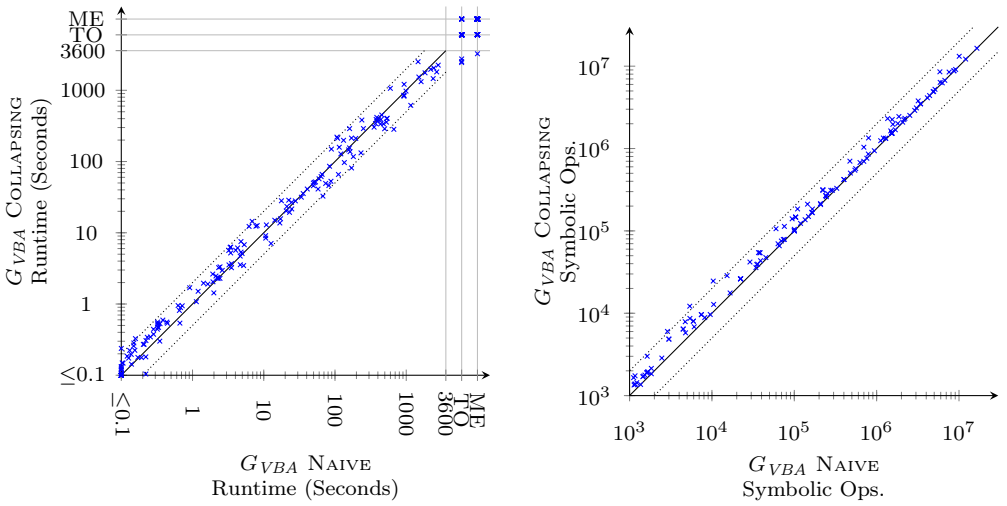


Figure 6.7: Comparison of Collapsing and Naive on G_{VBA}. Runtimes (in seconds) are compared in the left plot, the amount of symbolic $Pre + Post$ operations are compared on the right.

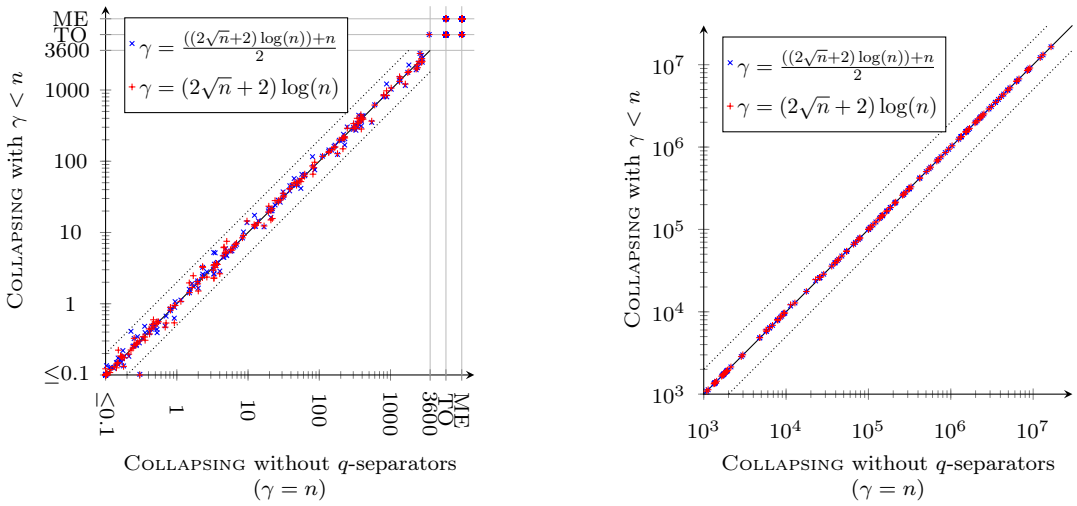


Figure 6.8: Comparison of Collapsing runtime (left) and amount of symbolic *Pre* + *Post* operations (right) using various γ parameters.

A direct comparison of the amount of symbolic operations of Collapsing and either Naive or LockStep on G_{VBA} is partially inaccurate: both Naive and LockStep do not modify the transition BDD of G_{VBA} in contrast to Collapsing. Each time a detected EC is collapsed into a single vertex, the transition BDD is modified, which can lead to a non-uniform runtime cost per *Pre*/*Post* operation. The transition BDD size changes during the decomposition are visualized in Fig. 6.9, but it is unclear to which extent this effect can be seen on a "native" G_{VBA} implementation (Remark 5.1).

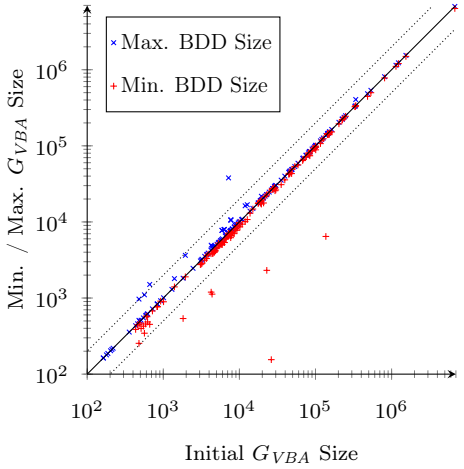


Figure 6.9: Maximum and minimum size (in node count) of the transition BDD in the first phase of Collapsing.

For G_{EBA}, both NAIVE and LOCKSTEP modify the transition BDD whenever an action is removed (Remark 2.9). If the varying transition BDD size of COLLAPSING is the reason for its lower runtime, we would like to try to replicate this effect on G_{EBA} as best as possible. As such, two additional variants of NAIVE have been benchmarked on G_{EBA}: NAIVEREMOVENONTRIVIAL removes all actions of each identified MEC from the transition BDD, while NAIVEREMOVEALL also removes the actions of trivial SCCs.

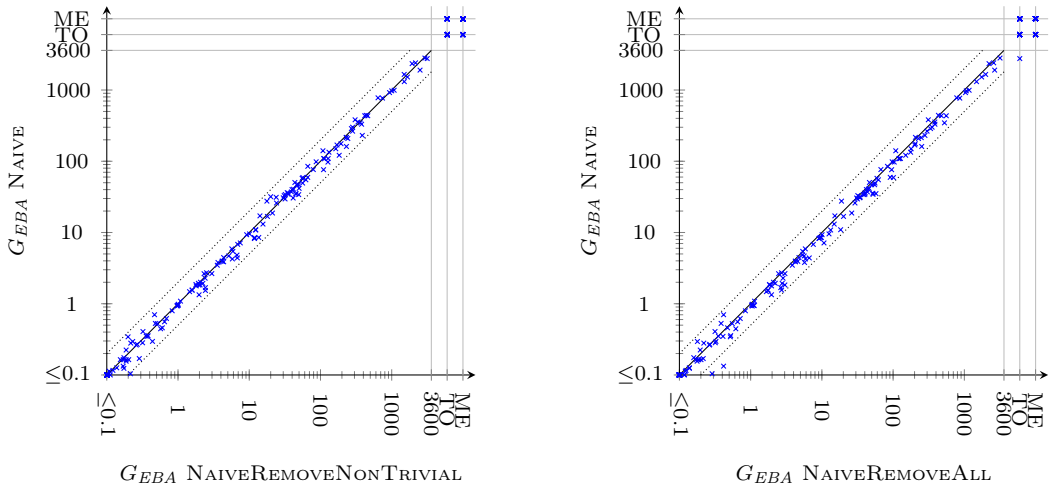


Figure 6.10: Runtime (in seconds) comparisons of NAIVE to the variants NAIVEREMOVENONTRIVIAL and NAIVEREMOVEALL.

The results of these benchmarks can be seen in Fig. 6.10 and are mixed overall: NAIVEREMOVEALL is only faster in rare instances. NAIVEREMOVENONTRIVIAL shows slightly better results, but overall, NAIVE remains faster most of the time.

To summarize and answer research question 2, our empirical evidence implies that the worst-case instances in which LOCKSTEP and COLLAPSING improve upon NAIVE do not occur often enough in order to yield a lower amount of symbolic operations. When focusing on the real-world runtime performance, COLLAPSING seems competitive with NAIVE when ran on a converted G_{VBA} structure, but whether COLLAPSING remains competitive in a "native" G_{VBA} implementation (Remark 5.1) requires further research.

6.4 G_{EBA} vs G_{VBA}

Given that the used conversion process from G_{EBA} to G_{VBA} is suboptimal (Remark 5.1), it is misleading to simply compare the benchmarked runtimes directly. However for any given model, the amount of symbolic operations is not influenced by the conversion process, which we can use to argue which graph-like structure is preferable for computing MEC decompositions. Assuming that each action is encoded as a distinct vertex in G_{VBA}, any given MDP will require at least twice the amount of vertices in comparison to G_{EBA}. This will result in at least two more Boolean variables being used within the transition BDD of G_{VBA} in comparison to the abstracted transition BDD of G_{EBA}.

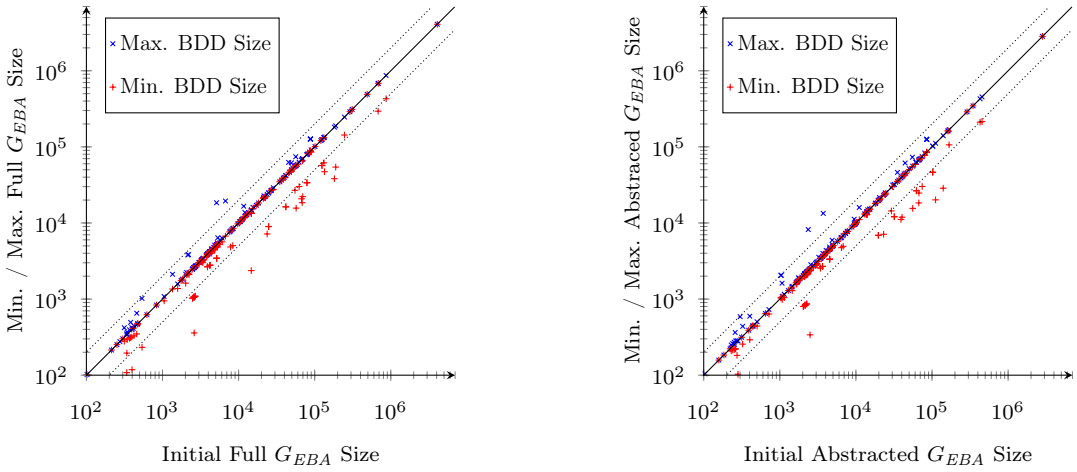


Figure 6.11: Maximum and minimum size (in node count) of the full and abstracted transition BDD of Naive on G_{EBA}.

Therefore it is likely that a single *Pre*/*Post* operation on G_{EBA} is comparable to an operation performed on G_{VBA}, if not faster. Additionally, the cost is non-uniform for Naive and LockStep on G_{EBA} in contrast to G_{VBA} due to the removal of actions from the transition BDD (Remark 2.9). The extent of the varying BDD size on G_{EBA} when using Naive can be seen in Fig. 6.11. For this set of benchmarks, the varying cost seems to work out in favour of G_{EBA} by often reducing transition BDD size which should lead to faster *Pre*/*Post* operations.

Given these assumptions, we can argue that G_{EBA} generally has a better runtime performance for MEC decompositions than G_{VBA}, even if G_{VBA} is constructed "natively": in Fig. 6.12, G_{EBA} uses less symbolic *Pre*/*Post* operations than any of the algorithms used on G_{VBA}. In Fig. 6.13, we can see that Naive uses roughly twice the amount of symbolic operations on G_{VBA} in comparison to G_{EBA}, while this effect only worsens when using LockStep.

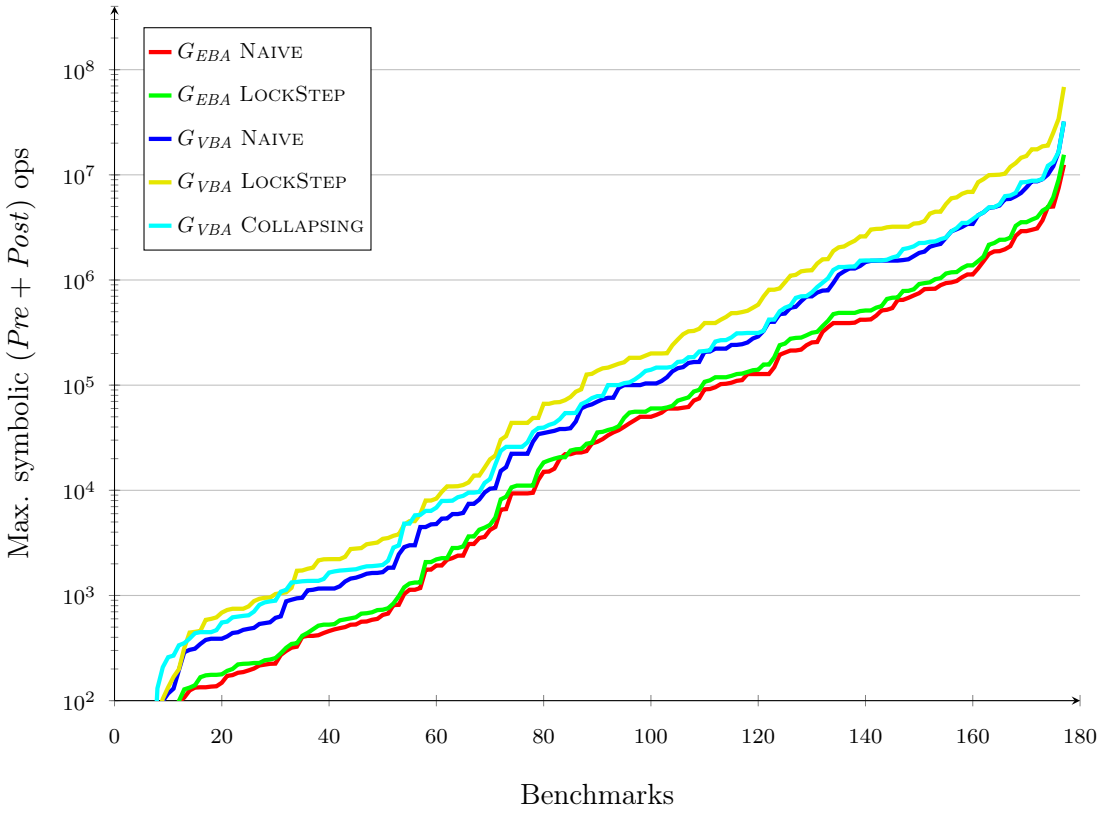


Figure 6.12: Quantile plot showing the amount of symbolic $Pre + Post$ operations of the symbolic MEC decomposition algorithms for both G_{EBA} and G_{VBA}.

To truly capture the difference in runtime performance, further benchmarks with a "native" G_{VBA} are required, as these comparisons also disregard additional symbolic operations unique to G_{EBA}, such as the calculation of the abstracted BDD from the full transition BDD. The general hierarchy of required symbolic operations is still present even if *all* symbolic operations are measured (see Fig. 6.14), which indicates that the comparisons of symbolic $Pre/Post$ operations should be representative and indicate a better runtime performance on G_{EBA}.

To answer research question 3, a symbolic MEC decomposition computation requires fewer symbolic operations on G_{EBA} than on G_{VBA}. As the cost of each symbolic operation on G_{EBA} is roughly equal to (if not lower than) a symbolic operation on G_{VBA}, the computation of an MEC decomposition is more efficient on G_{EBA} than G_{VBA}.

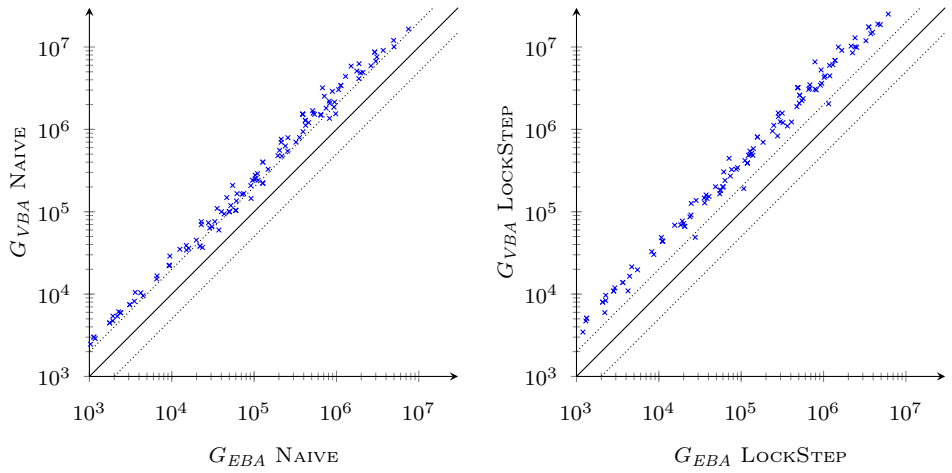


Figure 6.13: Comparison of G_{EBA} to G_{VBA} by the amount of symbolic operations ($Pre + Post$) used for both Naive (left) and LockStep (right).

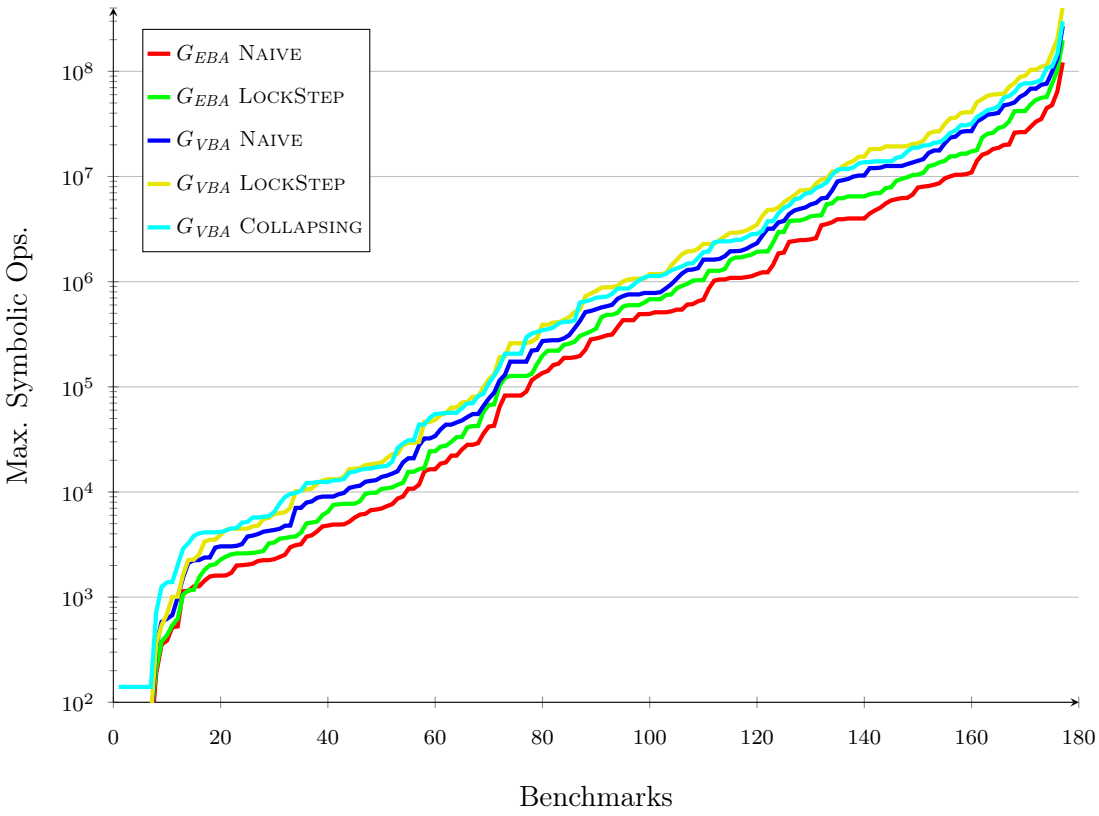


Figure 6.14: Quantile plot showing the amount of all symbolic operations of the symbolic MEC decomposition algorithms for both G_{EBA} and G_{VBA}.

Chapter 7

Conclusion and Outlook

The primary goal of this thesis was to evaluate how three different symbolic MEC decomposition algorithms perform and compare to each other in practice. To achieve this, we have started from the abstract definitions of MDPs and formalized two differing graph-like structures G_{EBA} and G_{VBA}. In the literature, the definitions and representations of MDPs tend to differ depending on the context in which they are used. As we have seen in Chapter 5, these different representations not only require slightly different implementations of various algorithms, but can also lead to some algorithms becoming impractical if not infeasible to use. To understand the limitations of symbolic algorithms and operations, we discussed (RO)BDDs and how they work by explaining an implementation of the AND operation as an example. We covered both the explicit and the symbolic implementation of MDPs of G_{EBA} found in STORM, compared them to another to make them easier to grasp and discussed the differences to a G_{VBA} implementation. We used these differences to present a novel symbolic algorithm to convert from G_{EBA} to G_{VBA} and showed how results on the converted G_{VBA} can be translated back into the G_{EBA} representation.

To experimentally evaluate the symbolic MEC decomposition algorithms for both graph-like structures, the conversion algorithm was extensively used. While the process itself runs quickly, the resulting BDDs contain more nodes and the runtime cost of each symbolic operation increases substantially. In theory for the decomposition algorithms, both LOCKSTEP and COLLAPSING improve upon NAIVE by reducing the amount of required symbolic *Pre*/*Post* operations in a worst-case scenario. Our empirical evaluation showed that NAIVE remains faster when compared to LOCKSTEP, but the runtime results become more mixed when comparing NAIVE to COLLAPSING. Most notably, NAIVE almost always uses the least amount of symbolic *Pre*/*Post* operations, which indicates that any of the advantages in runtime performance of COLLAPSING might be dependant on the non-uniform cost of the symbolic *Pre*/*Post* operations due to the modifications of the transition BDD.

Whether this effect can be seen on a "native" implementation of G_{VBA} might be of interest for future work. While the amount of symbolic operations indicate that G_{EBA} is the more performant option for symbolically computing MECs, the true runtime difference to a "native" G_{VBA} implementation has yet to be measured, as the suboptimal size of the of the converted transition BDD also hinders a direct comparison of the runtime performance between graph-like structures. Additionally, the recent works of [LSS+23] provide a faster symbolic SCC decomposition algorithm. It is unclear whether all MEC decomposition algorithms benefit equally from a faster SCC computation. As the results of this thesis reiterates the difference between theoretical and practical impact, a similar performance comparison using the various *explicit* MEC decomposition algorithms [CH14, CH11, CH14, CDHS19] could also be of interest for future work.

Bibliography

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 27(06):509–516, 1978. (Cited on pages 1 and 13.)

[BCC+14] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtěch Forejt, Jan Křetínskỳ, Marta Kwiatkowska, David Parker, and Mateusz Ujma. Verification of Markov decision processes using learning algorithms. In *Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings 12*, pages 98–114. Springer, 2014. (Cited on page 8.)

[BDH+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: Quantitative Model and Tool Interaction. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 151–168, 2017. (Cited on page 2.)

[BFG+97] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2):171–206, 1997. (Cited on page 15.)

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008. (Cited on pages 1, 7, 8, and 22.)

[BRB91] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991. (Cited on page 15.)

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. (Cited on pages 1, 14, and 18.)

[Bry92] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992. (Cited on pages 1 and 15.)

[BVD17] Richard J Boucherie and Nico M Van Dijk. *Markov decision processes in practice*, volume 248. Springer, 2017. (Cited on pages 1 and 7.)

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers*, 45(9):993–1002, 1996. (Cited on pages 2 and 22.)

[CDHL16] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Veronika Loitzenbauer. Model and objective separation with conditional lower bounds: Disjunction is harder than conjunction. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 197–206, 2016. (Cited on pages 1, 8, and 10.)

[CDHL18] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Veronika Loitzenbauer. Lower bounds for symbolic computation on graphs: Strongly connected components, liveness, safety, and diameter. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2341–2356. SIAM, 2018. (Cited on page 27.)

[CDHS19] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Alexander Svozil. Near-linear time algorithms for streett objectives in graphs and MDPS. *arXiv preprint arXiv:1909.05539*, 2019. (Cited on pages 1, 3, 8, and 56.)

[CDHS21] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Alexander Svozil. Symbolic time and space tradeoffs for probabilistic verification. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021. (Cited on pages 2, 4, 8, 27, 29, 32, 35, 37, 41, 42, and 47.)

[CH11] Krishnendu Chatterjee and Monika Henzinger. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 1318–1336. SIAM, 2011. (Cited on pages 3, 8, 29, 32, and 56.)

[CH14] Krishnendu Chatterjee and Monika Henzinger. Efficient and dynamic algorithms for alternating Büchi games and maximal end-component decomposition. *Journal of the ACM (JACM)*, 61(3):1–40, 2014. (Cited on pages 1, 3, and 56.)

[CHI+16] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{O}(m\sqrt{n})$ total update time. In *2016 IEEE 57th*

Annual Symposium on Foundations of Computer Science (FOCS), pages 315–324. IEEE, 2016. (Cited on page 37.)

[CHL+18] Krishnendu Chatterjee, Monika Henzinger, Veronika Loitzenbauer, Simin Oraee, and Viktor Toman. Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In *International Conference on Computer Aided Verification*, pages 178–197. Springer, 2018. (Cited on pages 2, 4, 8, 29, 32, 33, 35, 41, and 42.)

[DA98] Luca De Alfaro. *Formal verification of probabilistic systems.* stanford university, 1998. (Cited on pages 2, 3, 4, and 29.)

[DB13] Rolf Drechsler and Bernd Becker. *Binary decision diagrams: theory and implementation.* Springer Science & Business Media, 2013. (Cited on pages 2 and 13.)

[EFT93] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993. (Cited on pages 2 and 22.)

[Fab23] Felix Faber. Comparison of Maximal End Component Decomposition Algorithms: Data and Code, September 2023. (Cited on page 42.)

[FMY97] Masahiro Fujita, Patrick C. McGeer, and JC-Y Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal methods in system design*, 10(2):149–169, 1997. (Cited on page 15.)

[GPP03] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. Computing strongly connected components in a linear number of symbolic steps. In *SODA*, volume 3, pages 573–582, 2003. (Cited on pages 27, 29, and 41.)

[HJK+21] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2021. (Cited on pages 2, 19, and 41.)

[HKP+19] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 344–350. Springer, 2019. (Cited on pages 2 and 42.)

[HM18] Serge Haddad and Benjamin Monmege. Interval iteration algorithm for MDPs and IMDPs. *Theoretical Computer Science*, 735:111–131, 2018. (Cited on page 8.)

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009. (Cited on page 20.)

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002. (Cited on page 2.)

[Lee59] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959. (Cited on pages 1 and 13.)

[LSS+23] Casper Abild Larsen, Simon Meldahl Schmidt, Jesper Steensgaard, Anna Blume Jakobsen, Jaco van de Pol, and Andreas Pavlogiannis. A Truly Symbolic Linear-Time Algorithm for SCC Decomposition. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 353–371. Springer, 2023. (Cited on pages 2, 4, 27, 29, and 56.)

[MSS07] Shin-ichi Minato, Ken Satoh, and Taisuke Sato. Compiling Bayesian Networks by Symbolic Probability Calculation Based on Zero-Suppressed BDDs. In *IJCAI*, volume 2007, pages 2550–2555, 2007. (Cited on page 15.)

[Put94] Martin L Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1994. (Cited on pages 1 and 8.)

[SLL09] Alexander L Strehl, Lihong Li, and Michael L Littman. Reinforcement Learning in Finite MDPs: PAC Analysis. *Journal of Machine Learning Research*, 10(11), 2009. (Cited on page 8.)

[Som97] Fabio Somenzi. CUDD: CU decision diagram package. *Public Software, University of Colorado*, 1997. (Cited on page 42.)

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. (Cited on pages 3 and 26.)

[vD16] Tom van Dijk. Sylvan: multi-core decision diagrams. 2016. (Cited on pages 18 and 42.)

[Whi85] Douglas J White. Real applications of Markov decision processes. *Interfaces*, 15(6):73–83, 1985. (Cited on pages 1 and 7.)

[Whi88] Douglas J White. Further real applications of Markov decision processes. *Interfaces*, 18(5):55–61, 1988. (Cited on pages 1 and 7.)

[Whi93] Douglas J White. A survey of applications of Markov decision processes. *Journal of the operational research society*, 44(11):1073–1096, 1993. (Cited on pages 1 and 7.)

[WKB14] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački. GPU-based graph decomposition into strongly connected and maximal end components. In *Computer Aided Verification: 26th International Conference, CAV 2014,*

Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26, pages 310–326. Springer, 2014. (Cited on page 3.)

[XB00] Aiguo Xie and Peter A Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1225–1230, 2000. (Cited on page 2.)

[YO97] Bwolen Yang and David R O'hallaron. Parallel breadth-first BDD construction. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–156, 1997. (Cited on page 18.)