Syntaxbasierte Reduktion von PRISM Modellen

Maximilian Starke

Fakultät für Informatik, Technische Universität Dresden

26. November 2021

Inhaltsverzeichnis

1 Kurzfassung

In vielen die Sicherheit von komplexen Systemen betreffenden Problemstellungen spielen probabilistische Modelle, beispielsweise Markow-Ketten mit diskreter Zeitauffassung, eine wichtige Rolle. Im konkreten Beispiel kann die Modellierung einer Flugzeugsteuerung und die Analyse resultierender Modelle auf gewisse Eigenschaften dazu beitragen, Abstürze von Flugzeugen zu verhindern. Der PRSIM model checker [7] ist ein bekanntes Werkzeug, um jene Modelle auf gewisse Eigenschaften zu untersuchen. Dabei ist die Größe des Modells ein limitierender Faktor, weil mit Modellen zunehmender Größe der Speicher- und Rechenaufwand des PRISM model checkers steigt. Methoden zur intelligenten Variablenumordnung, welche der PRISM model checker bereits mitbringt, erschöpfen noch lange nicht die Möglichkeiten, Modelle in äquivalente Modelle zu verkleinern [4]. Es gibt Reduktionsmethoden, welche auf semantischer Ebene agieren, beispielsweise basierend auf Bisimulation [5] oder Symmetrie [8]. Methoden zur Reduktion von Modellen basierend auf rein syntaktischer Ebene zu finden, ist bis zu unserem Wissensstand ein noch recht offenes Forschungsfeld. In bisheriger Forschungsarbeit [4] wurde bereits ein Machbarkeitsbeweis geführt, dass tatsächlich eine Reduktion motiviert durch die Methode der Live Range Analysis aus dem Compilerbau möglich ist und diese eine erhebliche Verkleinerung von Modellen bewirken kann. Ziel soll es in diesem Forschungsprojekt sein, vorhandene Methoden syntaktischer Reduktion weiter zu untersuchen, mit welchen jene Modelle in solche äquivalente kleinere Modelle umgewandelt werden können, die invariant bzgl. der zu analysierenden bzw. verifizierenden Eigenschaft zueinander und zum ursprünglichen Modell sind. Im Konkreten werden wir zunächst die bereits vorgestellte Methode der syntaktischen Reduktion [4] erweitern, anschließend die Erweiterung in C++ implementieren und schlussendlich ausgewählte Heuristiken für Graphfärbungen vorstellen und evaluieren hinsichtlich des Einflusses auf die Reduktion von Kontrollflussprogrammen. Dabei verkleinern wir Modelle durch eine Reduktion der Anzahl der Variablen mittels Zusammenfassung redundanter Variablen. Mittels der Zahl der nodes und states, wie sie der PRISM model checker bei der Verarbeitung des verkleinerten Modells im Log ausgibt, werden wir beurteilen, wie erfolgsbringend eine solche Reduktion ist. Die Zahl der states bezeichnet dabei die Zahl der Zustände des gesamten Modells. Die Zahl der nodes ist nicht immer relevant, jedoch entscheidend für die MTBDD engine von PRISM [9], welche auf Basis von binären Entscheidungsdiagrammen beruht. Um die Evaluation von Heuristiken für Graphfärbungen zu ermöglichen, entwickeln wir eine geschickte Implementierung, um all jene minimalen reduzierten Modelle zu finden, welche mit kleinster Anzahl an Variablen auskommen. Wir werden empirisch an beispielhaften Modellen zeigen, dass die Zahl der nodes und states bei der anschließenden Verarbeitung mit PRISM unter allen erhaltenen minimalen reduzierten Modellen variiert.

2 Grundlagen

2.1 PRISMs Eingabesprache

Wir betrachten für unsere praktische Implementierung gewählte Beispiele, die als Markow-Ketten in der Eingabesprache von PRISM vorliegen. Da unsere

Abbildung 2.1: Beispiel Eingabesprache

```
1  dtmc
2
3  global cf : [0 .. 5];
4  global a : [0 .. 4];
5  global b : [0 .. 4];
6  global c : [0 .. 4];
7  global d : [0 .. 4];
8  global e : [0 .. 4];
9
10 init
11 (cf=0) & (a=0) & (b=1) & (c=2) &
12 (d=0 | d=1) & (e=4 | e=2)
13 endinit
14
15 module some_module
16 [goto_1] cf = 0 & a = 0 ->   0.4 : cf' = 1 & a' = 1
17 + 0.6 : cf' = 2 & b' = 3;
18 [goto_2] cf = 0 & a = 4 ->   0.2 : cf' = 2 & b' = 4
19 + 0.8 : cf' = 4 & e' = 0;
20 ...
21 ...
22 [reset1] cf = 3 & e = 0 ->   0.3 : cf' = 0 & (e' = 4 | e' = 2)
23 + 0.7 : cf' = 2 & e' = 0;
24 endmodule
25
26 formula failure = (e = 0);
```

Hauptmotivation darin besteht, kleine Modelle zu erreichen, um diese dann mit Software wie PRISM in vernünftiger Zeit verarbeiten zu können, beginnen wir mit einem Beispielmodell in der Eingabesprache von PRISM. Davon ausgehend abstrahieren wir und bestimmen unsere Annahmen. Betrachten wir beispielhaft den Ausschnitt eines Modells in Abbildung 2.1.

Wie wir sehen, bestehen unsere Modelle zunächst aus einer Menge von Variablen mit endlichem Zustandsraum sowie einer nichtleeren endlichen Menge von Startzuständen. Ein Zustand des Modells ist hierbei eine konkrete Zuweisung von Werten zu jeder einzelnen Variablen. Wir gehen weiterhin von genau einem sogenanntem Modul aus, welches aus einzelnen Transitionen besteht, die der Lesbarkeit wegen Namen wie *goto_1* oder *reset1* tragen können. Eine Transition besteht aus einer Eintrittsbedingung, z.B. `cf = 0 & a = 0` im Falle von *goto_1*, welche erfüllt sein muss (cf und a werden gelesen), damit die Transition auf einen Zustand anwendbar ist sowie einer Wahrscheinlichkeitsverteilung für Folgezustandsbedingungen: Beispielsweise wird bei Eintreten von Transition *goto_1* mit Wahrscheinlichkeit 0.4 der Zustand eingenommen, in welchem sich cf zu 1 ändert und a zu 1, mit Wahrscheinlichkeit 0.6 der Zustand, in welchem sich cf zu 2 ändert und b zu 3. In letzterem Fall werden cf und b geschrieben. Mit der Zeile `formula failure = (e = 0)` wird eine Eigenschaft beschrieben, ein Zustand des Modells jeweils erfüllt oder nicht. Somit wird eine letztlich Teilmenge von Zuständen ausgezeichnet. Mit der Verwendung dieser in PCTL-querys lässt sich ein Modell mit dem PRISM model checker auf die gewünschte Eigenschaft hin untersuchen.

Annahmen an das Modell Wir können zusammenfassend sagen, dass die Semantik eines betrachteten Modells jeweils eine Markow-Kette ist. In folgenden Abschnitten werden wir uns nur noch auf die syntaktische Ebene beziehen und die Modelle schlicht als Programme mit Variablen auffassen. Während für allgemeine Betrachtungen Wahrscheinlichkeiten als ein Wesensmerkmal von Markow-Ketten von Relevanz sind, können wir diese für unser Vorhaben der Modellreduktion vernachlässigen. Die Reduktion, welche wir betrachten wollen, basiert auf dem Zusammenführen von Variablen mit disjunkten live ranges, wie es aus dem Compilerbau bekannt ist. Um die Invarianz bezüglich der durch PRISM zu untersuchenden Eigenschaft herzustellen, ermöglichen wir es, ausgewählte Variablen von der Zusammenführung auszuschließen. Wir wollen die folgenden Annahmen für unsere Reduktion treffen:

(a) Es gibt eine Variable, die als Kontrollflussvariable dient, wir nennen diese hier cf (engl. für *control flow*).

(b) In jeder Transition ist cf in der Eintrittsbedingung eindeutig festgelegt. In allen Folgezustandsbedingungen steht cf eindeutig fest: Dazu bekommt cf entweder explizit einen eindeutigen Wert zugewiesen oder bleibt unverändert.

(c) Es gibt eine Liste von Variablen, die von der Reduktion auszuschließen sind.

Man kann solche Programme auch als *control flow programs* bezeichnen. Das Beschränken auf die syntaktische Ebene entspricht einer Überapproximation der Liveness. Führen wir Modellreduktionen mittels live ranges auf der syntaktischen Ebene durch, so führt dies zu keinen unzulässigen Reduktionen im originalen Modell. Wir verzichten darauf, dies hier formal zu zeigen, weil die Korrektheit intuitiv anhand nachfolgender Darstellungen klar ist. Für eine formale Betrachtung verweisen wir auf [3].

Um mit der Reduktion von einem Modell zu beginnen, gehen wir also davon aus, dass gegeben sind:

- Variablen mit jeweils endlichem Wertebereich $cf \in D_{cf}, x_1 \in D_1, x_2, \in D_2, \ldots, x_n \in D_n$.
- endlich viele Transitionen $t_1, t_2, \ldots, t_m$ als Paare von Eintrittsbedingung und Tupel der Folgezustandsbedingungen:

$$t_i = (f_i^{enter}, (f_{i,1}^{leave}, f_{i,2}^{leave}, \ldots, f_{i,k_i}^{leave}))$$

- eine Startzustandsbedingung siehe Zeile 10 des Beispielmodells (Abbildung 2.1). D_{cf}

In der vorangegangenen Forschungsarbeit [4] wurden alle Folgezustandsbedingungen einer Transition zu einer Transition zusammengefasst. Die erste, jedoch natürlich offensichtliche Optimierung, die wir vornehmen wollen, ist es, jede Transition aus der Eingabesprache von PRISM mit mehr als einer Folgezustandsbedingung anhand ebendieser in eigenständige Transitionen aufzusplitten. Dies können wir ohne Weiteres tun, da für unsere auf rein syntaktischer Basis funktionierende Reduktion den Wahrscheinlichkeiten keine Bedeutung zukommt. Aus

$$t_i = (f_i^{enter}, (f_{i,1}^{leave}, f_{i,2}^{leave}, \ldots, f_{i,k_i}^{leave}))$$

werden somit die eigenständigen Transitionen

$$\begin{aligned} t_i^1 &= (f_i^{enter}, (f_{i,1}^{leave})) \\ t_i^2 &= (f_i^{enter}, (f_{i,2}^{leave})) \\ &\dots \\ t_i^{k_i} &= (f_i^{enter}, (f_{i,k_i}^{leave})) \end{aligned}$$

Daraus können wir unmittelbar einen Kontrollflussgraphen aufbauen, welcher für sich gesehen nur noch die syntaktische Ebene des Modells repräsentiert.

Definition 2.1 (Multigraph) Sei $V \neq \emptyset$ eine nichtleere Menge, aufgefasst als Menge von Knoten und E eine Menge, aufgefasst als Menge gerichteter Kanten. Seien weiterhin

$$\begin{aligned} \cdot_{\text{from}} &: E \to V : e \mapsto e_{\text{from}}, \\ \cdot_{\text{to}} &: E \to V : e \mapsto e_{\text{to}} \end{aligned}$$

zwei Abbildungen, die jeder Kante e ihren Startknoten e_{from} sowie ihren Endknoten e_{to} zuordnen. Dann nennen wir $G = (V, E, \cdot_{\text{from}}, \cdot_{\text{to}})$ einen (gerichteten) Multigraph.

Wir identifizieren jede resultierende Transition als ein $e^j = (f_j^{enter}, f_j^{leave}) \in E$. Dann erhalten wir unmittelbar einen Multigraphen $G = (V, E, \cdot_{\text{from}}, \cdot_{\text{to}})$ als Programmgraphen, wie wir ihn als Ausgangspunkt für die Live Range Analysis benötigen, indem wir weiterhin definieren:

(a) $V \coloneqq D_{cf}$

(b) $\forall e^j \in E : e^j_{\text{from}} \coloneqq v_1$, falls $f_j^{enter} \implies cf = v_1$

(c) $\forall e^j \in E : e^j_{\text{to}} \coloneqq v_2$, falls $f_j^{leave} \implies cf = v_2$

Es sei bemerkt, dass (b) und (c) eindeutig definiert sind aufgrund unserer Annahmen. Wir operieren insofern auf einem semantischen Aspekt des Modells, als dass wir den Zustandsraum D_{cf} einbeziehen müssen. Ansonsten haben wir das Modell auf seinen syntaktischen Aspekt, bestehend aus einem System von Transitionen, reduziert.

Es sei noch angemerkt, warum an dieser Stelle ein Multigraph und kein gewöhnlicher gerichteter Graph unter Verzicht auf Mehrfachkanten verwendet wird: Der Grund liegt in der Eigabesprache. Die Elemente von D_{cf} entsprechen der Gesamtheit aller Knoten des Programmgraphen, vergleichbar mit gesetzten Haltepunkten bei der Benutzung eines gewöhnlichen Debuggers für imperative Programmiersprachen. Es kann nun mehrere Transitionen des ursprünglichen Modells zwischen konkreten Knoten mit Werten `cf =` c_1 und `cf =` c_2 geben oder auch eine Transition mit `cf =` c_1 in der Eintrittsbedingung und `cf =` c_2 in mindestens zwei Folgezustandsbedingungen. In beiden Fällen ergeben sich zwei oder mehr Kanten von c_1 nach c_2 in G. Natürlich lässt sich ein solcher Multigraph durch Einfügen zusätzlicher Knoten in einen Graphen ohne Mehrfachkanten überführen. Dies bei der Implementierung so umzusetzen, bringt jedoch keine Vorteile hinsichtlich Performance.

2.2 Live Range Analysis

Wenn ein Compiler in der Phase der Codegenerierung für die finale Plattform lokale Variablen aus Funktionen den Registern einer CPU zuweisen muss, so ist es für die Performance zur Laufzeit der entstehenden Anwendung sehr bedeutend, wie geschickt stets diejenigen Variablen in Registern gehalten werden, die als nächstes gelesen werden müssen [6]. Zugriffe auf Cache oder Arbeitsspeicher dagegen gehen mit erheblichen Wartezeiten einher. Der auf der Hand liegende erste Schritt zur Steigerung der Laufzeitperformance ist es daher, keine Register mit solchen Variablen, die gar nicht mehr gebraucht werden, belegt und weiterhin reserviert zu halten, während dort schon neue Variablen gespeichert werden können. Wir beschreiben im Folgenden allgemein die Live Range Analysis, die genau dieses Problem löst und beziehen das Verfahren dabei auf unsere jeweils als Multigraph vorliegenden Programme:

Ein Programm kann als (gerichteter) Graph mit Mehrfachkanten, so genannter Multigraph $G = (V, E, \cdot_{\text{from}}, \cdot_{\text{to}})$ aufgefasst werden, unabhängig davon, ob wir uns auf unsere Modelle beziehen oder auf den vorliegenden Fall bei Compilern für gewöhnliche imperative Programmiersprachen. Jede einzelne Position im Programm zwischen Transitionen fassen wir als einen Knoten $v \in V \neq \emptyset$ auf, jeder Sprung bzw. Übergang von einer Position $v_1 \in V$ über eine Transition zu einer nächsten Position $v_2 \in V$ wird als Kante $e \in E$ zwischen den Knoten $v_1 = e_{\text{from}}$ und $v_2 = e_{\text{to}}$ aufgefasst. Dann sagen wir:

Definition 2.2 (Liveness) Eine Variable x heißt *live* in einem Knoten v, wenn es von v einen Pfad in G gibt, auf welchem x gelesen wird, ohne dass zuvor auf diesem Pfad x geschrieben wird. Wir schreiben dann $\text{live}_x(v)$ und sonst $\neg\text{live}_x(v)$.

Auch wenn der Begriff Pfad allgemein bekannt ist, soll dieser für Multigraphen einmal kurz definiert werden:

Definition 2.3 Sei $G = (V, E, \cdot_{\text{from}}, \cdot_{\text{to}})$ ein Multigraph. Wir nennen ein Tupel $(v_0, v_1, \ldots, v_n)$ von Knoten einen Pfad in G, wenn für jede zwei aufeinander folgenden Knoten eine solche Kante existiert:

$$\forall i \in \{0, \ldots, n-1\} \exists e \in E : v_i = e_{\text{from}} \wedge v_{i+1} = e_{\text{to}}$$

Aber kommen wir zurück zum Begriff Liveness. Man könnte auch sagen, ist eine Variable live, so wird sie eventuell noch gebraucht. Für den Fall, dass eine Variable auf allen Pfaden jeweils entweder nicht mehr gelesen wird, oder vor jedem Lesen erneut geschrieben wird, so wird offensichtlich das Zwischenergebnis, welches in der Variablen abgelegt ist, auf allen weiterführenden Pfaden nicht mehr gebraucht. Per Definition ist diese Variable dann nicht live, wir nennen sie dann auch *dead*. In so einem Fall ließe sich ein Register wiederverwenden ab dem Zeitpunkt, zu dem die dort abgelegte Variable dead wird, man kann auch sagen, wenn sie ihre *live range* verlässt.

Definition 2.4 (Live Range) Zu einem Programmgraph $G = (V, E, \cdot_{\text{from}}, \cdot_{\text{to}})$ bezeichnen wir mit live_range(x) für eine Variable x die Menge aller Knoten $v \in V$, sodass x live ist in v:

$$\text{live\_range}(x) \coloneqq \{v \in V \mid \text{live}_x(v)\}$$

Zweck der Live Range Analysis ist es nun, durch Berechnung von live_range(x) für jede Variable x festzustellen, dass sich Variablen $x_1, x_2, x_3 \ldots$ mit paarweise disjunkten Mengen live_range(x_1), live_range(x_2), live_range(x_3), $\ldots$ ein und dasselbe Register teilen können, wenn wir uns im Kontext der Registerallokation im Compilerbau befinden. Bezogen auf unseren Anwendungsfall fassen wir solche Variablen zu einer neuen Variablen zusammen. Eine allgemein bekannte, einfache iterative Methode zur Berechnung der Mengen $\mathrm{LIVE}_v \coloneqq \{x \in \mathrm{Var} \mid \mathrm{live}_x(v)\}$ wollen wir der Vollständigkeit halber hier kurz wiedergeben [1]:

In jeder Transition e können einerseits Variablen gelesen werden, womit diese unmittelbar vor diesem Schritt live werden. Andererseits können Variablen geschrieben werden. Dadurch werden diese an der Programmposition zuvor dead. Dieses Generieren bzw. Aufheben von Liveness drücken wir mithilfe von Mengen GEN_e und KILL_e für jede Kante e aus:

$$\mathrm{GEN}_e \coloneqq \{x \in \mathrm{Var} \mid x \text{ gelesen, nicht zuvor } x \text{ geschrieben in Transition } e\}$$
$$\mathrm{KILL}_e \coloneqq \{x \in \mathrm{Var} \mid x \text{ geschrieben, nicht zuvor } x \text{ gelesen in Transition } e\}$$

Bezeichnen wir weiterhin mit LIVE_e^{in} die Menge der Variablen, welche live sind unmittelbar vor Schritt e und mit LIVE_e^{out} die Menge der Variablen, welche live sind unmittelbar nach Schritt e im Programmfluss. Dann können wir LIVE_e^{in} mithilfe von GEN_e und KILL_e aus der Menge der live Variablen nach Schritt e ausdrücken:

$$\mathrm{LIVE}_e^{in} = \mathrm{GEN}_e \cup \left(\mathrm{LIVE}_e^{out} \setminus \mathrm{KILL}_e\right)$$

An einer Position werden verschiedene Möglichkeiten des weiteren Programmverlaufs entsprechend akkumuliert:

$$\mathrm{LIVE}_e^{out} = \bigcup_{f \in E \text{ mit } f_1 = e_{\mathrm{to}}} \mathrm{LIVE}_f^{in}$$

Für den Fall eines Terminalzustandes $v \in V$ und einer dort endenden Kante $e \in E, e_{\mathrm{to}} = v$ ergibt sich die Menge $\mathrm{LIVE}_e^{out} = \emptyset$ entsprechend zur leeren Menge. Es ist aus der Literatur wohlbekannt, dass bei endlichen Graphen mit Mehrfachkanten G es genügt mit $\mathrm{LIVE}_e^{out} \coloneqq \emptyset$ und $\mathrm{LIVE}_e^{in} \coloneqq \emptyset$ für alle Kanten $e \in E$ zu starten und schrittweise die Mengen anhand der Gleichungen zu aktualisieren bis keine Änderungen mehr auftreten. Das Verfahren terminiert nach endlichen Schritten und wir erhalten zu jeder Programmposition, d.h. zu jedem Knoten die Menge

$$\mathrm{LIVE}_v = \bigcup_{f \in E \text{ mit } f_{\mathrm{from}} = v} \mathrm{LIVE}_f^{in}$$

der Variablen, die dort live sind. Das Bestimmen von live_range(x) für eine Variable x anschließend ist trivial, da stets gilt $x \in \mathrm{LIVE}_v \Leftrightarrow \mathrm{live}_x(v) \Leftrightarrow v \in$ live_range(x). Wir können zwei verschiedene Variablen x, y stets dann zu einem Register (bzw. zu einer Variablen) zusammenfassen, wenn live_range(x) $\cap$ live_range(y) $= \emptyset$.

Gehen wir von unseren Modellen aus, so erfolgt das Zusammenfassen von Variablen $x_1, \ldots, x_n$ durch Festlegen eines neuen gemeinsamen Namens und

durch Vereinigung der Definitionsbereiche: Wir ersetzen die Deklarationen von $x_1, \ldots, x_n$ durch die Deklaration von $y \in (D_{x_1} \cup \cdots \cup D_{x_n})$. Wir ersetzen im originalen Modell jedes Vorkommen von x_i durch y. Nach der sytaktischen Ersetzung aller Variablennamen können logische Widersprüche entstehen, einerseits in der Startzustandsbedingung (siehe Zeile 10 in Abbildung 2.1), andererseits in jeder Folgezustandsbedigung einer Transition: Stellen wir uns vor eine solche Bedingung hat die Form $x_1 = 1 \wedge x_2 = 5$ und seien x_1 und x_2 Variablen, die wir zusammenfassen zu y. Nach stupidem Ersetzen der Variablennamen ergibt sich die immer falsche Bedingung $y = 1 \wedge y = 5$. Jedoch bemerken wir, dass an jeder solchen Stelle immer nur entweder x_1 oder x_2 live ist. Der Wert der jeweils anderen Variable ist irrelevant und kann aus der ursprünglichen Bedingung gestrichen werden. So bleibt von der ursprünglichen Bedingung beispielsweise $x_1 = 1$ übrig und im resultierenden Modell erhalten wir $y = 1$ als Bedingung.

2.3 Graphfärbungen

Nun soll es jedoch nicht Ziel sein, lediglich zwei Variablen herauszugreifen und zusammenzuführen, sondern solange Variablen zusammenzuführen, bis es keine weiteren Reduktionen mehr gibt. Von diesen maximalen Reduktionen wollen wir dann diese betrachten, bei denen die Zahl der übrigen Variablen am kleinsten ist. Dazu bilden wir einen ungerichteten Graphen $H = (\mathrm{Var}, F)$. Dabei ist die Menge der Knoten lediglich die Menge Var von Variablen. Zwei Variablen verbindet eine Kante genau dann, wenn diese an irgendeinem Knoten des Programmgraphen gleichzeitig live sind. Dies entspricht genau dem Fall, wo die zwei Variablen nicht zusammengeführt werden können.

$$\{x, y\} \in F \Leftrightarrow \exists v \in D_{cf} : \{x, y\} \subseteq \mathrm{LIVE}_v$$

Das Finden einer maximalen Reduktion der Variablen ist identisch mit dem Problem, eine Färbung mit minimal vielen Farben für H zu finden. Um wie zuvor beschrieben zu erreichen, dass ausgewählte Variablen der Erhaltung gewünschter Eigenschaften des Modells wegen von der Reduktion ausgeschlossen werden, muss jede solche Variable als Knoten des Graphen H lediglich mit allen anderen Variablen über eine Kante verbunden werden. Dies erzwingt eine eigene Farbe für den Knoten.

3 Details zur Implementierung

Wir wollen nun wesentliche Schritte des Vorgehens bei der Implementierung in C++ darstellen. Der Quellcode kann im Detail online eingesehen werden [10]. Zunächst war es als nicht unwesentlicher Teil der Implementierung erforderlich, die Eingabesprache des PRISM model checkers zu parsen. Im Rahmen dieses Forschungsprojektes ist daher ein Parser für PRISM dtmcs entstanden, der auch für zukünftige Softwareanwendungen mit wenig Aufwand übernommen oder auch erweitert werden kann zum Beispiel für MDPs. Nach dem Parsen wurde der beschriebene Multigraph aufgestellt mit anschließender Berechnung mittels des Standardverfahrens zur Live Range Analysis. Ein Zwischenergebnis stellt sodann der im letzen Abschnitt beschriebene Graph H dar. Für das Berechnen aller Graphfärbungen mit minimaler Zahl an Farben im nächsten Schritt ist geschicktes Vorgehen erforderlich. Viele Probleme bezüglich Graphfärbung sind

bereits NP-schwer, so zum Beispiel die Frage zu einem gegebenem Graph, ob dieser 4-färbbar ist. Wir erwarten also in der Praxis exponentielle Laufzeiten zur Bestimmung aller optimalen Graphfärbungen.

3.1 Berechnung von maximalen Variablenmengen gleicher Färbung

Wir wollen hier eine Implementierung vorstellen, um die Menge aller größtmöglichen Reduktionen, d.h. die Menge aller k-Färbungen des Graphen H mit kleinstmöglichem k geschickt zu berechnen, um später die Ergebnisse von Heuristiken zum Finden einer Graphfärbung mit allen k-Färbungen zu vergleichen. Dazu nutzen wir bei gegebenem ungerichteten Grapen $H = (\mathrm{Var}, F)$ Paare von Bitsets der Länge $|\mathrm{Var}|$:

Definition 3.1 (Variablenkombination) Wir nennen k eine *Variablenkombination*, wenn es ein Paar zweier boolscher Funktionen $k = (id, neighbours)$ ist mit

$$id, neighbours : \mathrm{Var} \to \{0, 1\}$$

und $\neg\exists x \in \mathrm{Var} : id(x) = neighbours(x) = 1$. Wir nennen $size(k) \coloneqq |id|$ die *Größe* der Variablenkombination. Für ein Paar k bezeichnen wir deren Elemente als $\mathrm{id}(k)$ und $\mathrm{neighbours}(k)$. Entsprechend gilt $k = (\mathrm{id}(k), \mathrm{neighbours}(k))$.

Bei der Implementierung nutzen wir die C++ Klasse `std::bitset` für die Funktionen id und $neighbours$ aus Performancegründen. Man könnte id und $neighbours$ alternativ als Teilmengen von Var auffassen. Eine Variablenkombination $k = (id, neighbours)$ dient jeweils der Darstellung einer Teilmenge der Variablen, die mit derselben Farbe gefärbt werden können (id). Dabei beschreibt $neighbours$ die Menge der Variablen, die in H adjazent zu einem Knoten aus id sind und somit nicht mehr mit derselben Farbe gefärbt werden können. Jeder ursprünglichen Variablen $x \in \mathrm{Var}$ ordnen wir die Variablenkombination $k_x \coloneqq (\{x\}, \{y \in \mathrm{Var} \mid \{x, y\} \in F\})$ zu.

Definition 3.2 (zusammenführbar) Wir nennen zwei Variablenkombinationen k, l *zusammenführbar*, in Symbolen $k \downarrow l$, wenn $\mathrm{id}(k) \cap \mathrm{neighbours}(l) = \emptyset$ und $\mathrm{id}(l) \cap \mathrm{neighbours}(k) = \emptyset$. Mit $k \cup l$ bezeichnen wir in diesem Fall die aus k und l nach Zusammenführen entstandene Variablenkombination $k \cup l \coloneqq (\mathrm{id}(k) \cup \mathrm{id}(l), \mathrm{neighbours}(k) \cup \mathrm{neighbours}(l))$. Aus Lesbarkeitsgründen verwenden wir hier die Schreibweise als Teilmengen von Var.

Die Definition zur Zusammenführbarkeit erlaubt die Zusammenführung offensichtlich genau in dem Fall, wenn keine zwei Variablen $x \in \mathrm{id}(k)$ und $y \in \mathrm{id}(l)$ existieren, welche eine Kante in H verbindet. Somit handelt es sich bei zwei Teilmengen der Knoten Var, welche jeweils für sich in ein und derselben Farbe gefärbt werden können, dann auch bei der Vereinigung um eine Menge an Knoten, die gleich gefärbt werden können.

Wir gehen nun algorithmisch folgendermaßen vor: Sei K_1 die Menge aller Variablenkombinationen der Größe 1, die sich unmittelbar aus H ergeben: $K_1 \coloneqq \{k_x \mid x \in \mathrm{Var}\}$. Wir berechnen nun nacheinander $K_2, K_3, K_4, \ldots$ durch Zusammenführen bereits bekannter Variablenkombinationen: Haben wir zuletzt K_i bestimmt als die Menge aller Variablenkombinationen k mit $size(k) = i$, so berechnen wir K_{i+1} aus K_i und K_1 wie folgt:

```
1  K_{i+1} := ∅
2  for(b in K_1):
3      for (k in K_i):
4          if (b ↓ k and size(b ∪ k) = i + 1):
5              K_{i+1} := K_{i+1} ∪ (b ∪ k)
```

Durch das Zusammenführen mit einer Kombination b aus K_1 wird die Kombination k um genau 1 größer. In Zeile 4 prüfen wir, dass eine Variable hinzugefügt wird, die bereits in k enthalten ist. Nach endlich vielen Schritten wird ein $K_i = \emptyset$ erreicht, sobald sich aus den Variablenkombinationen K_{i-1} keine größeren mehr ableiten lassen. Alle Elemente von K_{i-1} sind dann maximal und wir benennen mit $max\_size := i - 1$ deren Größe. In C++ nutzen wir aus Performancegründen für jedes K_i einen `std::vector`, dessen Elemente $k = (id, neighbours)$ wir stets nach id sortiert halten.

Definition 3.3 (maximale Variablenkombination) Wir nennen eine Variablenkombination k maximal, wenn es keine Variablenkombination $l \in K_1$ gibt, sodass k, l zusammenführbar sind und $k \cup l \neq k$.

Es genügt im nun folgenden Schritt für jedes i alle maximalen Variablenkombinationen aus K_i zu betrachten. Aus diesen bilden wir Abdeckungen aller Variablen unter Verwendung von so wenig wie möglich Variablenkombinationen. Seien $L_i := \{k \in K_i \mid k \text{ maximal}\}$. Insbesondere ist dann L_1 die Menge aller Variablen, die mit keiner anderen Variablen in einer Färbung dieselbe Farbe zugeordnet bekommen können. Wir bilden die endliche Menge aller maximalen Variablenkombinationen $L := L_1 \cup L_2 \cup \cdots \cup L_{max\_size}$.

3.2 Abdeckung aller Variablen

Eine k-Färbung des Graphen H erhalten wir genau dann, wenn es eine Auswahl $C \subseteq L$ mit $|C| = k$ gibt, sodass es für jede Variable x eine Variablenkombination $(id, neighbours) \in C$ gibt, die x enthält, d.h. $id(x) = 1$. Wir wollen also das minimale k ermitteln, für welches eine k-Färbung existiert und dann alle k-Färbungen auflisten. Dafür benötigen wir erneut eine geschickte iterative Implementierung.

Definition 3.4 (Teilauswahl über L) Sei L eine endliche Menge von Variablenkombinationen. Wir bezeichnen eine Abbildung

$$c : L \to \{0, \#, 1\}$$

als *Teilauswahl über* L. Dabei nennen wir $l \in L$ *ausgewählt* für den Fall $c(l) = 1$, *nicht ausgewählt* für den Fall $c(l) = 0$ und *nicht entschieden* für den Fall $c(l) = \#$.

Wir benötigen für unseren Algorithmus den Begriff Teilauswahl über L als eine Art Fallunterscheidung, sodass wir noch offene Fälle in einer Warteschlange vorhalten können. Die Abbildung c, welche jedem $l \in L$ den Wert $c(l) = \#$ zuordnet, repräsentiert die Gesamtheit aller möglichen Teilmengen von L. Sobald wir einem l den Wert 1 zuweisen, legen wir uns fest, dass l enthalten sein soll, mit dem Wert 0 wiederum schließen wir l aus. Im Laufe des Algorithmus werden wir stets Werte # ersetzen und die zwei Teilfälle 0 und 1 stattdessen aufnehmen. Der folgende Algorithmus, hier in Pseudo-Code dargestellt, erwies sich bei den verwendeten Beispielen als praktikabel:

```
1  partial_selections = []; // Liste aller offenen Teilauswahlen
2  partial_selections.push_back({(#,#,...,#)}); // nichts entschieden
3  choices = []; // kleinste Auswahlen
4  last_size = ∞;
5  again: while (! partial_selections.empty()) {
6   s = partial_selections.pop_front();
7   if (covering_all(s)){
8    if (size(s) > last_size)
9     goto again;
10   if (size(s) < last_size) {
11    last_size = size(s);
12    choices.clear();
13   }
14   choices.push_back(s);
15   goto again;
16  }
17  if (size(s) >= last_size)
18   goto again;
19  if (cannot_fill(s, L))
20   goto again;
21  select = greatest_fill(s, L);
22  if (gap_size(s) / fill_size(select, s) < last_size - size(s))
23   goto again;
24  s_yes = s;
25  s_yes[select] = 1;
26  partial_selections.push_front(s_yes);
27  s_no = s;
28  s_no[select] = 0;
29  partial_selections.push_back(s_no);
30 }
```

Der Pseudo-Code bedarf einiger Erläuterungen: Mit `partial_selections` führen wir eine Liste von Teilauswahlen über L, welche noch betrachtet werden müssen, um keine Auswahl $C \subseteq L$ zu vergessen. Eine solche Teilauswahl sagt stets aus, welche Variablenkombinationen ausgewählt, welche nicht ausgewählt sind und über welche noch nicht entschieden ist. Wir starten mit der Teilauswahl, bei welcher jedes $l \in L$ nicht entschieden ist (Zeile 2). In der Schleife ab Zeile 5 wird bei jedem Schleifendurchlauf die vorderste Teilauswahl `s` herausgenommen (Zeile 6).

Nun kann es einerseits sein, dass `s` bereits alle Variablen abdeckt (Zeile 7), d.h. für jede Variable x gibt es ein $l \in L$ mit $\mathtt{s}(l) = 1$ und $id_l(x) = 1$. Ist dies der Fall, so haben wir eine Auswahl gefunden, die bereits alle Variablen abdeckt und wir fügen diese der Liste `choices` hinzu (Zeile 14), welche die Ergebnisse dieses Algorithmus hält. Es wird mit dem nächsten Schleifendurchlauf fortgefahren (Zeile 15). Ausnahmen bestehen jedoch noch im Fall, dass die gefundene Auswahl mehr Variablenkombinationen auswählt als eine bereits gefundene (Zeile 8). In diesem Fall ignorieren wir `s`. Mit `last_size` merken wir uns immer die Größe der bisher kleinsten gefundenen Auswahl. Wir wollen keine größeren Auswahlen mehr akzeptieren, denn die Größe der Auswahl entspricht der Zahl der verwendeten Farben in der Graphfärbung und wir sind nur an den optimalen Färbungen interessiert. Sehr wohl kann es jedoch passieren, das wir einmal eine kleinere Auswahl, d.h. Färbung mit weniger Farben als bisher gefunden haben (Zeile 10). In diesem Fall verwerfen wir alle bisher gefundenen, nicht optimalen Auswahlen (Zeile 12) und merken uns die neue kleinste Auswahlgröße (Zeile 11).

Andererseits kann es sein, dass wir noch nicht alle Variablen abdecken. Es

folgen sodann die Schritte ab Zeile 17. Da noch nicht alle Variablen abgedeckt sind, muss noch mindestens eine Variablenkombination hinzugezogen werden. Wenn wir damit bereits mehr Variablenkombinationen (Farben bei der Graphfärbung) benötigen als in einer bereits gefundenen Auswahl (Zeile 17), dann brauchen wir `s` nicht weiter betrachten (Zeile 18). Genauso kann es sein, dass wir gar keine Abdeckung aller Variablen mehr erreichen können, da zu viele Variablenkombinationen in `s` bereits als nicht ausgewählt markiert sind (Zeile 19). Auch in diesem Fall verwerfen wir `s` (Zeile 20). Sofern die zwei eben dargestellten Abbruchbedingungen für den aktuellen Schleifendurchlauf nicht zutreffen, wählen wir mit `select` eine der nicht entschiedenen Variablenkombinationen von `s` aus, mit der ein Maximum an noch nicht abgedeckten Variablen getroffen wird (Zeile 21). Nun kann es sein (Zeile 22), dass wir noch höchstens $n \coloneqq \texttt{last\_size} - \texttt{size}(\texttt{s})$ Variablenkombinationen hinzunehmen können, um noch bei einer Auswahl bisher kleinster gesehener Größe zu landen, dabei noch `gap_size(s)` bisher nicht abgedeckte Variablen treffen müssen, im aktuellen Schritt jedoch höchstens $d \coloneqq \texttt{fill\_size}(\texttt{select}, \texttt{s})$ verbleibende Variablen abdecken können, sodass die Bedingung in Zeile 22 erfüllt ist. Auch in den verbleibenden Schritten können wir sodann jeweils höchstens weitere d Variablen abdecken und wir werden `last_size` vor dem Erreichen einer Abdeckung aller Variablen überschreiten. Daher können wir (Zeile 23) `s` in diesem Fall wieder verwerfen. Traf keine Abbruchbedingung zu, erreichen wir Zeile 24. Die neu in Betracht gezogene Variablenkombination `select` können wir nun entweder definitiv auswählen und `s_yes` erreichen (Zeile 25) oder mit `s_no` ausschließen (Zeile 28). Die ursprüngliche Teilauswahl entspricht der disjunkten Vereinigung der zwei Entscheidungen `s_yes` und `s_no`. Wir haben `s` am Anfang des Schleifendurchlaufs aus der Warteschlange `partial_selections` entfernt (Zeile 6), nun fügen wir `s_yes` vorn (Zeile 26) und `s_no` hinten (Zeile 29) hinzu. Die Priorität in der Warteschlange ist auf diese Weise so gewählt, um nach Möglichkeit rasch zu optimalen Färbungen zu finden.

Angemerkt sei, es ist an der Stelle nicht auszuschließen, dass es auch pathologische Beispiele geben kann, für welche diese Herangehensweise im Vergleich zu anderen zu langen Laufzeiten führt. Eine Optimierung des Algorithmus sei an dieser Stelle noch erwähnt: Anstatt in Zeile 2 wie bisher mit $(\#, \#, \ldots, \#)$ zu starten, können alle Variablenkombinationen in L_1 bereits ausgewählt werden. Schließlich sind dies diejenigen Variablen, welche mit keiner anderen zusammengeführt werden dürfen, bzw. die mit keiner anderen Variablen gleich gefärbt werden dürfen, um es in der Terminologie der Graphfärbungen zu formulieren.

3.3 Auflistung konkreter Graphfärbungen

Im eben beschriebenen Schritt endeten wir mit einer Auflistung von Mengen von Variablenkombinationen. Jede Menge von Variablenkombinationen deckt dabei jede Variable mindestens einmal ab. Aber es kann der Fall auftreten, dass Variablen von mehreren Variablenkombinationen abgedeckt werden. Dies lässt sich so vorstellen als hätten wir einen Graph mit k-Färbung mit kleinstmöglichem k, wobei es Knoten gibt, die mehrere Farben zugewiesen bekommen haben. Zum Erreichen einer k-Färbung im ursprünglichen Sinne dürfen wir uns nun in jedem uneindeutig gefärbten Knoten für eine beliebige vorhandene Farbe entscheiden. Dies muss im letzten Schritt der Implementierung zur Auflistung aller optimalen Graphfärbungen erfolgen. Dieser Schritt ist jedoch recht simpel: Zu einer Menge

von Variablenkombinationen können wir einfach alle möglichen resultierenden k-Färbungen bestimmen. Die resultierenden k-Färbungen aller Mengen von Variablenkombinationen vereinen wir in einer Menge und sind fertig.

4 Analyseergebnisse

Es werden nun drei Heuristiken zum Finden einer möglichst wenig Farben benötigenden Graphfärbung separat auf ein gegebenes originales Modell angewandt. Die drei resultierenden Modelle mit reduzierter Anzahl an Variablen vergleichen wir mit der Menge aller reduzierten Modelle mit kleinstmöglicher Zahl an Variablen.

Genutzt wurden zur Validierung der Heuristiken mehrere ausgewählte, randomisiert generierte Modelle als originale Modelle. Der anschließende Vergleich beruht darauf, den PRISM model checker jeweils auf den verkleinerten Modellen auszuführen und die Zahl der nodes der MTBDD engine von PRISM sowie die Zahl der states aus dem Log auszulesen. Wir wollen zunächst die drei Heuristiken kurz beschreiben.

4.1 Heuristiken für Graphfärbungen

Zunächst dient uns als Vorlage, der bereits in der unmittelbar vorangegangenen Forschungsarbeit [4] erwähnte Welsh-Powell Algorithmus zur Graphfärbung:

```
1   input: G = (V, E), V = {1, ..., n}
2   output: WP(G) : V → {1, ..., n}
3   for each v ∈ V
4     f(v) ≔ 0
5     d(v) ≔ |{{v, v'} ∈ E : v' V}|
6   Let o : {1, ..., n} → V such that
7     d(o(i)) ≥ d(o(j)) for all 1 ≤ i ≤ j ≤ n
8   for c ≔ 1, ..., n
9     for i ≔ 1, ..., n, f(o(i)) = 0
10      if ¬∃v : {o(i), v} ∈ E ∧ f(v) = c
11        f(o(i)) ≔ c
12  return f
```

Der Welsh-Powell Algorithmus basiert kurz gefasst darauf, die Knoten in einer Liste absteigend nach Zahl inzidenter Kanten zu ordnen und in dieser Reihenfolge jedem Knoten die kleinste noch freie (nicht durch Nachbarknoten verwendete) Farbe zuzuordnen. Wird einmal ein Knoten gefärbt, so wird in einer Subroutine zunächst in Reihenfolge der absteigend sortierenten Liste geschaut, welchen Knoten dieselbe Farbe noch zugeordnet werden kann. In den noch folgenden Diagrammen werden wir Welsh-Powell mit dem Kürzel *WelPow* markieren.

Als erste Alternative soll hier ein Algorithmus basierend auf Chaitin's graph coloring algorithm [2] gegenübergestellt werden. Letzterer wird bei Compilern verwendet, geht davon aus, nur eine fixe Zahl von Farben zur Verfügung zu haben und setzt dann sogenannten Spill Code ein, wenn die Farben nicht ausreichen. Die fixe Zahl ergibt sich aus der fixen Zahl von verfügbaren Registern, welche ein Compiler mit Variablen belegen kann. Schließlich entsprechen in diesem Anwendungsfall die Farben den Registern, deren Zuordnung zu Variablen sich während der Laufzeit ändern lässt, wenn dafür Zwischenergebnisse in den Arbeitsspeicher geschrieben werden, um Platz zu schaffen, andere Variablen in Registern vorhalten zu können.

Trotz des etwas andersgearteten Anwendungsfalls lässt sich die folgende Idee aus dem Algorithmus auch anwenden, wenn die Zahl der Farben nicht vorab festgelegt ist: Stellen wir uns vor, wir betrachten einen Graphen, den wir mit k Farben färben wollen und wir finden einen Knoten v mit weniger als k Nachbarn. Wenn wir v entfernen und den restlichen Graphen erfolgreich färben können, ist es am Ende leicht für v eine von den k Farben zu wählen, die durch keinen der Nachbarn von v belegt ist. Ist der restliche Graph ohne v also k-färbbar, so auch der ursprüngliche Graph. Wir nutzen nun die Idee folgendermaßen: Wir enfernen solange Knoten mit minimaler Zahl inzidenter Kanten und betrachten den verbleibenden induzierten Subgraphen, bis alle Knoten entfernt sind. Dann iterieren wir in umgekehrter Reihenfolge über die Knoten und weisen jeweils die kleinste Farbe zu, die noch nicht von Nachbarknoten verwendet wird. Wir wollen das Verfahren *Remove Nodes* nennen und kürzen es in den später folgenden Diagrammen mit *RemNod* ab:

```
1  input: G = (V, E), V = {1, ..., n}
2  output: f : V → {1, ..., n}
3  Let o : {1, ..., n} → V
4  W := V
5  while W ≠ ∅
6   H := G[W] //induced subgraph
7   Let v be any node of H with minimal number of incident edges.
8   o(|W|) := v
9   W := W \ {v}
10 while W ≠ V
11  v := o(|W| + 1)
12  W := W ∪ {v}
13  Let f(v) be the minimal number x ≥ 1 such that
14   in the induced subgraph G[W] there is no neighbour u of v
15   such that f(u) = x
16 return f
```

Als dritte Methode zum Auswählen einer konkreten Färbung soll uns *Maximum Merge First* - wie wir sie hier nennen wollen - dienen, abgekürzt *MaxMer*: Wir bestimmen die erste Farbe und eine größtmögliche Teilmenge aller Knoten, denen dieselbe Farbe zugewiesen werden darf. Wir färben alle diese Knoten in der ersten Farbe und wissen zugleich, dass kein weiterer Knoten mehr in dieser Farbe erscheinen darf. Wir entfernen alle gefärbten Knoten aus dem Graphen, suchen uns die nächste freie Farbe und beginnen an dieser Stelle immer wieder von vorn, bis alle Knoten gefärbt sind.

```
1  input: G = (V, E), V = {1, ..., n}
2  output: f : V → {1, ..., n}
3  next_color := 1
4  while not G empty
5   Find a set W ⊆ V such that
6    all v ∈ W can have the same color in Graph G and
7    W has maximal size
8   Let f(v) := next_color for all v ∈ W
9   Remove all nodes in W from G.
10  Let furthermore G be the remaining induced sub graph
11  next_color := next_color + 1
12 return f
```

Für diese Methode, die als zusätzlicher Vergleich dienen soll, lassen wir offen, wie konkret sie algorithmisch umgesetzt werden soll in Bezug auf Zeilen 5 bis 7. Es ist trivial die Methode anzuwenden, wenn man Zwischenberechnungen kennt, wie wir sie zum Finden aller optimalen Färbungen beschrieben haben.

4.2 Ergebnisse aus Vergleichen

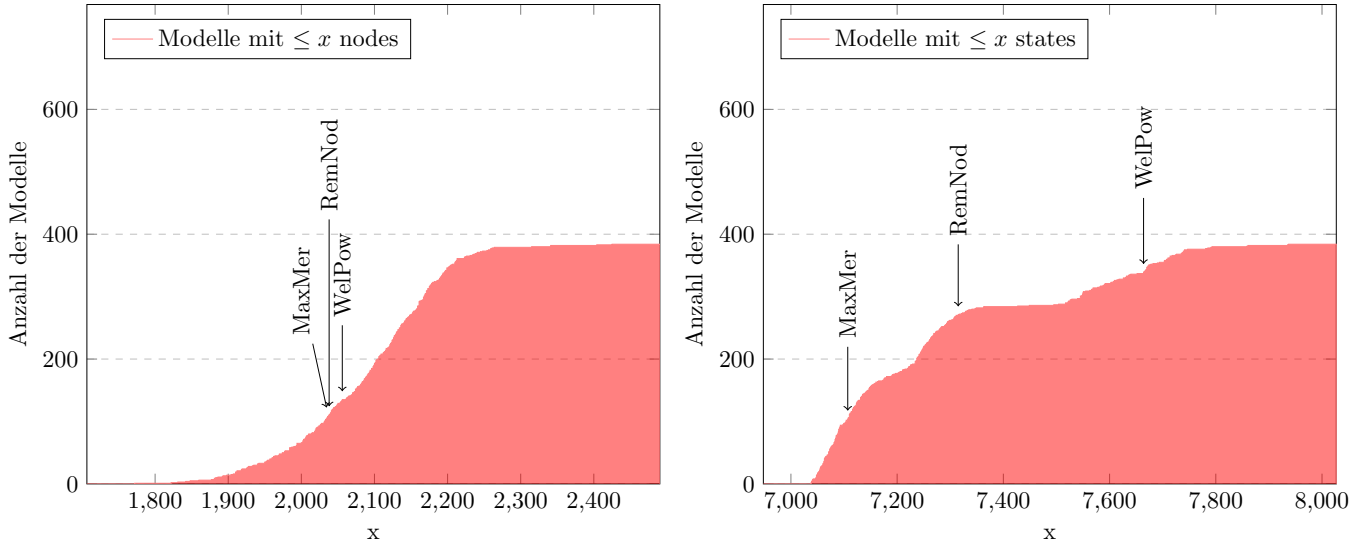
Wir betrachten nun die Abbildungen 4.1 bis 4.4. Dargestellt sind die Ergebnisse für sechs verschiedene ausgewählte Modelle. Die Nummerierung der Modelle entspricht dabei den Dateinamen im zugehörigen git repository und sollen daher hier nicht irritieren. Für jedes Modell gibt es zwei Diagramme, im ersten ist jeweils die Verteilung der Anzahl der nodes, im zweiten die Verteilung der Anzahl der states zu sehen. Die Art und Weise der Darstellung ist ansonsten in allen Diagrammen gleich.

Werfen wir beispielhaft zunächst einen Blick auf Abbildung 4.1. Auf der Abzissenachse ist die Zahl der nodes abgetragen, die Ordinate wiederum misst die Anzahl von Modellen. Letztere bezieht sich nun auf alle Modelle, die durch optimale Graphfärbungen aus dem originalen Modell 6 entstanden sind. Die Zahl der nodes zu einem Modell wurde bestimmt, indem PRISM auf dem jeweiligen Modell aufgerufen wurde und der entsprechende Wert aus dem Log abgelesen wurde. Dabei wurde mit den Argumentden beim Aufruf von PRSIM `-reorder -reorderoptions converge -mtbdd` das Reordering der MTBDD engine genutzt. Die Zahl der nodes bezieht sich immer auf die Transitionsmatrix nach dem Reordering. Die dargestellte Funktion ordnet jeder Zahl von nodes x die Anzahl all jener Modelle zu, welche zu einer Anzahl von höchstens x nodes geführt haben. Somit beträgt der Funktionswert am linken Ende der Grafik null und am rechten Ende die Gesamtzahl der durch Live Range Analysis und Zusammenfassen von Variablen entstandenen Modelle. In jedem Diagramm sind mit drei Pfeilen die Ergebnisse der reduzierten Modelle markiert, die durch Anwendung der drei zuvor beschriebenen Heuristiken enstanden sind. Tatsächlich führten bei allen bisher untersuchten Modellen alle der Heuristiken tatsächlich zu einem reduzierten Modell mit kleinstmöglicher Zahl an Variablen, also zu einem Modell, was sich auch unter allen generierten befindet. Alle weiteren Diagramme sind zu Abbildung 4.1 analog.

Zu jeden zwei Diagrammen gibt es noch eine Tabelle mit den konkreten Werten für die drei Heuristiken. Gegenübergestellt sind die Werte für das originale Modell sowie die minimalen Werte bezogen auf alle reduzierten Modelle. Die Zeitangaben sind die Dauer der Ausführung von PRISM und sollen nur als grobe Richtwerte verstanden werden.

Zunächst fällt auf, dass sowohl bei der Zahl der nodes als auch der states deutlich erkennbar ist, dass nicht alle reduzierten Modelle eines Originalmodells etwa die gleiche Anzahl an nodes oder states besitzen. Es besteht tatsächlich die Möglichkeit dahingehend zu optimieren, dass man nach Modellen mit wenig states bzw. nodes sucht. Das reduzierte Modell zu Modell 6 mit den meisten nodes weist beispielsweise über 35% mehr auf als das reduzierte Modell mit den wenigsten nodes. Die drei Heuristiken, von denen man sich erhofft, dass sie möglichst zu Modellen führen, mit geringer Zahl an nodes bzw. states, können in dieser Hinsicht alle nicht überzeugen. Maximum Merge First schnitt beispielsweise bei Modell 18 wie auch Modell 6 sowohl bei der Zahl der nodes als auch der Zahl der states am besten ab, teilweise nur knapp. Bei der Zahl der states zu Modell 11 jedoch enttäuschten sowohl Maximum Merge First als auch Welsh-Powell auf ganzer Linie und erreichten den höchstmöglichen Wert unter allen reduzierten Modellen. Remove Nodes erreichte dagegen bei den states zu Modell 11 eines der Modelle mit minimaler Zahl an states. Schauen wir nur für Modell 11 stattdessen die nodes an, so liegen plötzlich Maximum Merge First und Welsh-Powell vorn.

Abbildung 4.1: Verteilung der nodes und states von Modell 6



	#nodes	**% reduction**	**#states**	**% reduction**	**Zeit**
originales Modell	XXX	0	XXX	0	TTT
Maximum Merge First	XXX	YY	XXX	YY	TTT
Remove Nodes	XXX	YY	XXX	YY	TTT
Welsh-Powell	XXX	YY	XXX	YY	TTT
Minimum (red. Modelle)	XXX	YY	XXX	YY	

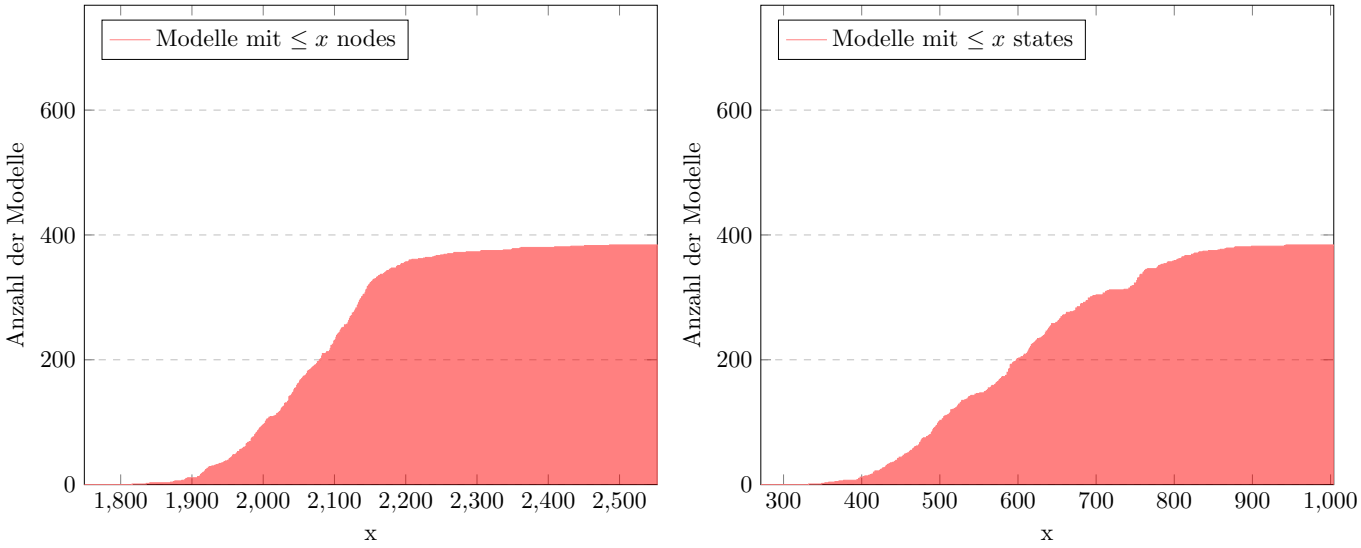
Tatsächlich führten beide Heuristiken sogar zum selben reduzierten Modell.

Wie sich unschwer erkennen lässt, sind alle Heuristiken bezogen auf die ausgewählten Beispielmodelle nur wenig brauchbar. Es scheint genauso gut zu sein, einfach irgendein Modell mit kleinster Zahl an Variablen zu nehmen. Es sei an dieser Stelle noch darauf hingewiesen, dass keine der 3 Heuristiken im Allgemeinen deterministisch zu einem eindeutigen Modell führt: Maximum Merge First führt zu einer nichtdeterministischen Auswahl für den Fall, dass einmal mindestens zwei maximale Mengen an Knoten existieren, die in derselben Farbe gefärbt werden können. Die beiden anderen Heuristiken basieren auf einer Reihenfolge von Knoten, die auch nicht eindeutig ist. Bei der Implementierung der Heuristiken wurde jeweils nur ein resultierendes Modell ermittelt.

4.3 Offene Fragestellungen

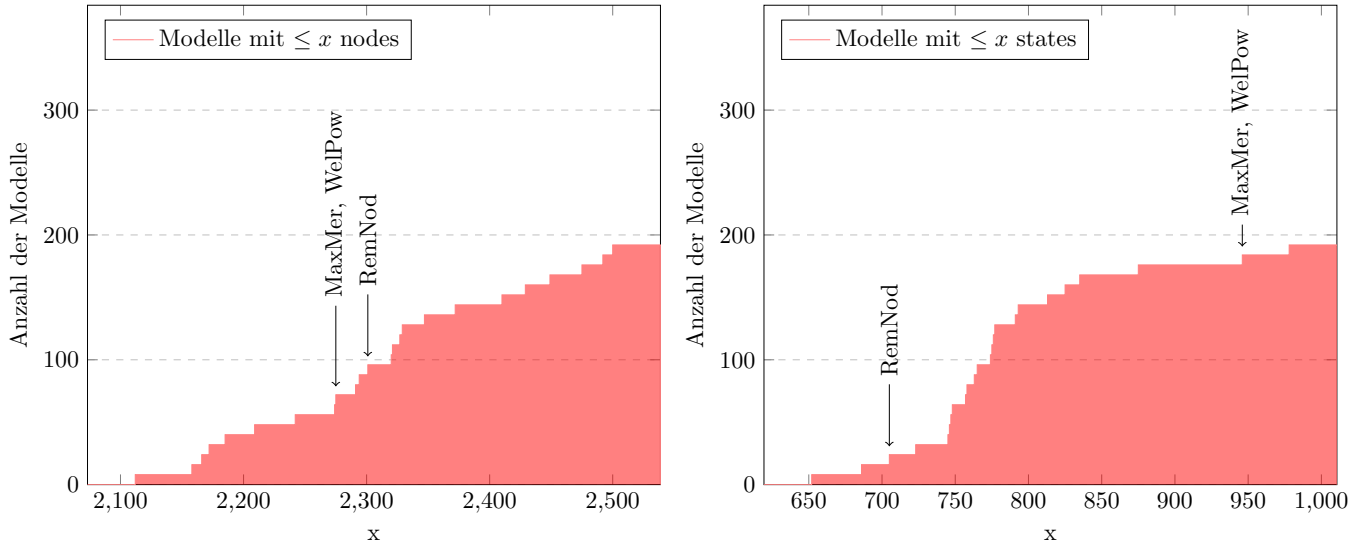
Wir wollen einmal kurz auf eine offene Fragestellungen eingehen, die näher untersucht werden könnte: Die vorgestellen Heuristiken zum finden einer Graphfärbung führen nicht zu eine eindeutigen Ergebnis. Vielmehr erlauben sie nichtdeterministische Entscheidungen zum Beispiel beim Auswählen eines Knotens mit kleinster Nachbarschaft bei Variante [Remove Nodes]. Daher wäre es interessant zu einer Heuristik einmal alle möglichen entstehenden Modelle aufzulisten, die bei verschiedener Auflösung des enthaltenenen Nichtdeterminismus entstehen

Abbildung 4.2: Verteilung der nodes und states von Modell 6



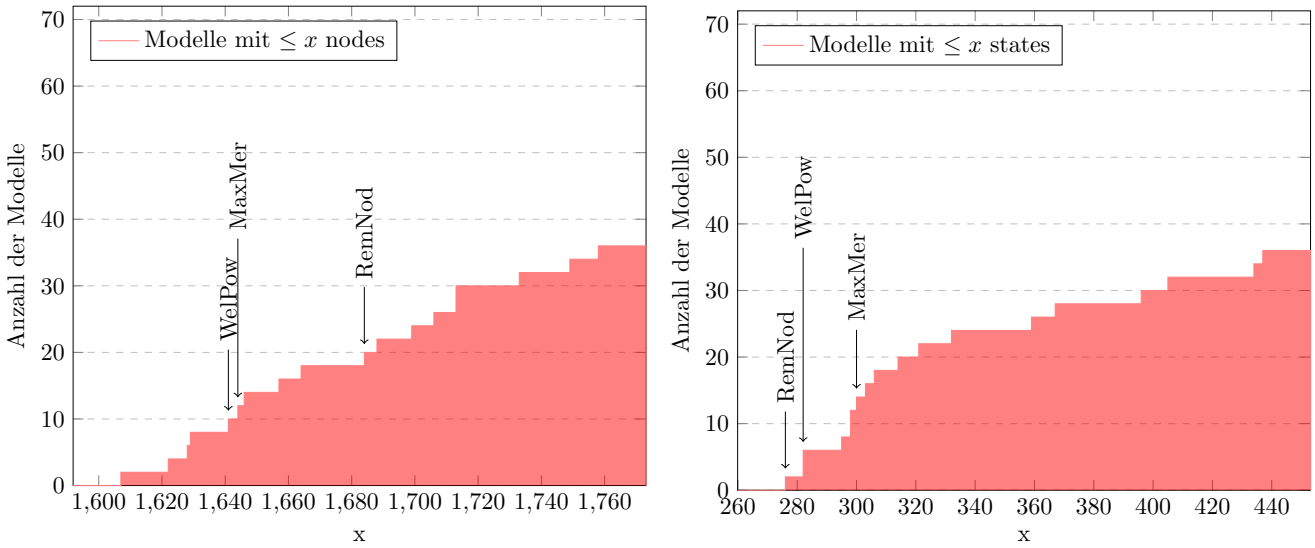
	#nodes	**% reduction**	**#states**	**% reduction**	**Zeit**
originales Modell	XXX	0%	XXX	0%	TTT
Maximum Merge First	XXX	YY%	XXX	YY%	TTT
Remove Nodes	XXX	YY%	XXX	YY%	TTT
Welsh-Powell	XXX	YY%	XXX	YY%	TTT
max. Reduktion	XXX	YY%	XXX	YY%	

Abbildung 4.3: Verteilung der nodes und states von Modell 11



	#nodes	**% reduction**	**#states**	**% reduction**	**Zeit**
originales Modell	2285	0%	3440	0%	8s
Maximum Merge First	2275	0.4%	946	73%	8s
Remove Nodes	2301	-1%	705	80%	9s
Welsh-Powell	2275	0.4%	946	73%	8s
max. Reduktion	2112	8%	652	81%	

Abbildung 4.4: Verteilung der nodes und states von Modell 18



	#nodes	**% reduction**	**#states**	**% reduction**	**Zeit**
originales Modell	2215	0%	1010	0%	5s
Maximum Merge First	1644	26%	300	70%	3s
Remove Nodes	1684	24%	276	73%	2s
Welsh-Powell	1641	26%	282	72%	3s
max. Reduktion	1607	27%	276	73%	

können. Ein Extremfall, welcher dabei auftreten könnte, wäre, dass eine Heuristik allein bereits Modelle erzeugt, die sich über das gesamte Intervall der Anzahl von nodes bzw. states erstreckt. Eine solche Heuristik wäre nutzlos innerhalb aller Modelle mit minimaler Zahl an Variablen, ein optimales zu finden, wohl aber eventuell nützlich, um überhaupt ein Modell mit minimaler Zahl an Variablen zu finden.

Literatur

[1] Andrew Appel. „Modern compiler implementation in Java. With Jens Palsberg. 2nd ed“. In: (Okt. 2002). DOI: 10.1017/CBO9780511811432.

[2] G. J. Chaitin. „Register Allocation and Spilling via Graph Coloring“. In: *SIGPLAN Not.* 17.6 (Juni 1982), S. 98–101. ISSN: 0362-1340. DOI: 10.1145/872726.806984. URL: https://doi.org/10.1145/872726.806984.

[3] Clemens Dubslaff. „Quantitative Analysis of Configurable and Reconfigurable Systems“. In: (2021), S. 127.

[4] Clemens Dubslaff u. a. *Breaking the Limits of Redundancy Systems Analysis.* 2019. arXiv: 1912.05364 [eess.SY].

[5] Robert Givan, Thomas Dean und Matthew Greig. „Equivalence notions and model minimization in Markov decision processes“. In: *Artificial Intelligence* 147.1 (2003). Planning with Uncertainty and Incomplete Information, S. 163–223. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(02)00376-4. URL: https://www.sciencedirect.com/science/article/pii/S0004370202003764.

[6] Philipp Krause. „The complexity of register allocation“. In: *Discrete Applied Mathematics* 168 (Mai 2014), S. 51–59. DOI: 10.1016/j.dam.2013.03.015.

[7] M. Kwiatkowska, G. Norman und D. Parker. „PRISM 4.0: Verification of Probabilistic Real-time Systems“. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Hrsg. von G. Gopalakrishnan und S. Qadeer. Bd. 6806. LNCS. Springer, 2011, S. 585–591.

[8] Marta Kwiatkowska, Gethin Norman und David Parker. „Symmetry Reduction for Probabilistic Model Checking“. In: *Computer Aided Verification.* Hrsg. von Thomas Ball und Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 234–248. ISBN: 978-3-540-37411-4.

[9] D. Parker. „Implementation of Symbolic Model Checking for Probabilistic Systems“. University of Birmingham, 2002.

[10] *Prism Syntactic Reduction.* URL: https://github.com/Necktschnagge/prism-syntactic-reduction.