ROBIN BERGENTHUM

VERIFIKATION VON HALBGEORDNETEN ABLÄUFEN IN PETRINETZEN

VERIFIKATION VON HALBGEORDNETEN ABLÄUFEN IN PETRINETZEN

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

der Fakultät für Mathematik und Informatik
der FernUniversität in Hagen

vorgelegt von Robin Bergenthum
geb. in Düsseldorf

Hagen, 22. Februar 2013

Robin Bergenthum:
Verifikation von halbgeordneten Abläufen in Petrinetzen,
Dissertation, 22. Februar 2013

BERICHTERSTATTER:
Prof. Dr. Jörg Desel
Prof. Dr. Robert Lorenz

ABSTRACT

The present thesis deals with verifying a partially ordered run in a Petri net. Petri nets are a well established formalism for modeling concurrent behavior of systems. A run serves as a behavioral-specification and the question is whether this specification is true or fulfilled in the given Petri net. In this thesis Petri nets are referred to by marked place/transition-nets and the run is given by a scenario. A scenario is a partially ordered set of events. In contrast to sequentially ordered runs a scenario includes arbitrary dependencies and independencies of events. Consequently, a scenario allows a precise and intuitive specification of the behavior of a concurrent or distributed system. The scenario-verification-problem comprises the decision whether a given scenario coincides with the behavior of a place/transition-net.
A scenario will coincide with the behavior of the place/transition-net if the scenario is included in the scenario-language of the net. Three different approaches exist to define this language in the literature, each yielding a different verification-algorithm. The structure of the scenario determines the runtime of these algorithms. The present thesis will introduce the different characterizations of the scenario-language and will discuss the runtime of the respective verification-algorithms. A new characterization of the scenario-language of a marked place/transition-net, which is optimized for the scenario-verification-problem, will be introduced. The goal of this thesis is to develop an algorithm, efficiently solving the scenario-verification-problem in any case.

ZUSAMMENFASSUNG

Die vorliegende Arbeit behandelt das Problem der Verifikation eines halbgeordneten Ablaufs in einem Petrinetz. Petrinetze sind ein anerkannter Formalismus, um nebenläufige Systeme integriert darzustellen. Ein Ablauf formuliert eine Verhaltens-Spezifikation und es stellt sich die Frage, ob diese im Petrinetz wahr bzw. erfüllt ist. Petrinetze werden in dieser Arbeit in ihrer allgemeinen Form der markierten Stellen/Transitions-Netze verwendet, der Ablauf wird als Szenario dargestellt. Ein Szenario ist eine Menge von Ereignissen zusammen mit einer Halbordnung auf dieser Menge. Ein Szenario beschreibt im Gegensatz zu sequentiellen Ablaufmodellen nicht nur eine Folge von Ereignissen, sondern stellt beliebige Abhängigkeiten und Unabhängigkeiten zwischen Ereignissen dar. Aus diesem Grund lässt sich nebenläufiges oder verteiltes Verhalten mithilfe eines Szenarios intuitiv und leicht spezifizieren. Die Frage danach, ob ein Szenario Verhalten eines Stellen/Transitions-Netz beschreibt, ist das Szenario-Verifikations-Problem.
Ein Szenario beschreibt Verhalten eines markierten Stellen/Transitions-Netzes, wenn es in dessen Szenario-Sprache enthalten ist. In der Literatur existieren drei verschiedene Möglichkeiten, diese Sprache zu charakterisieren. Aus jeder Charakterisierung kann man einen Verifikations-Algorithmus ableiten, mit dem sich das Szenario-Verifikations-Problem

entscheiden lässt. Je nach Probleminstanz besitzen diese Algorithmen sehr unterschiedliche Laufzeiten. In dieser Arbeit werden die Charakterisierungen der Szenario-Sprache vorgestellt und die sich aus ihnen ergebenden Verifikations-Algorithmen verglichen. Außerdem wird eine vierte, für das Problem der Verifikation optimierte Charakterisierung der Szenario-Sprache entwickelt, und mit deren Hilfe ein Verfahren vorgestellt, dass das Szenario-Verifikations-Problem für jede Probleminstanz effizient entscheidet.

INHALTSVERZEICHNIS

Abbildungsverzeichnis viii

1 EINLEITUNG 1
- 1.1 Einführendes Beispiel 1
- 1.2 Anwendung 11
  - 1.2.1 Geschäftsprozesse der AUDI AG 12
  - 1.2.2 Geschäftsprozesse im Therapaedicum Medifit 14
- 1.3 Szenariobasierte Modellierung 17
- 1.4 Agile Geschäftsprozessmodellierung 20
- 1.5 Problemstellung und Literaturübersicht 23
- 1.6 Gliederung 26

2 FORMALE GRUNDLAGEN 29
- 2.1 Notationen 29
- 2.2 Komplexitätsbetrachtungen 32

3 FLÜSSE IN NETZWERKEN 35
- 3.1 Flussnetzwerke 35
- 3.2 Der Algorithmus von Ford und Fulkerson 38
- 3.3 Der Algorithmus von Dinic 45
- 3.4 Der Forward-Backward-Propagation Algorithmus 49
- 3.5 Der Preflow-Push Algorithmus 54
- 3.6 Vergleich der Fluss-Maximierungs-Algorithmen 61

4 ABLÄUFE UND PETRINETZE 69
- 4.1 Stellen/Transitions-Netze 69
- 4.2 Szenarien 73
- 4.3 Der aktivierte Schnitte Algorithmus 76
- 4.4 Der Prozessnetz Algorithmus 79
- 4.5 Der Markenfluss Algorithmus 87
  - 4.5.1 Der iterative Test 90
  - 4.5.2 Der direkte Test 98

5 KOMPAKTE MARKENKENFLÜSSE UND PETRINETZE 105
- 5.1 Kompakte Markenflüsse 105
- 5.2 Der kompakte Markenfluss Algorithmus 115

6 VERGLEICH DER SZENARIO-VERIFIKATIONS-ALGORITHMEN 123
- 6.1 Laufzeitexperimente 123
  - 6.1.1 Test des Prozessnetz Algorithmus 125
  - 6.1.2 Verifikation von dichten Szenarien 127
  - 6.1.3 Verifikation von lichten Szenarien 131
  - 6.1.4 Verifikation ressourcenabhängiger Modelle 134
  - 6.1.5 Bewertung der Verifikations-Algorithmen 137
- 6.2 Integration in das VipTool 138

7 ABSCHLUSSBETRACHTUNGEN 141
- 7.1 Zusammenfassung 141
- 7.2 Ausblick 143

LITERATURVERZEICHNIS 145

ABBILDUNGSVERZEICHNIS

Abbildung 1 Unsere Kaffee-Druckbrühmaschine. 1
Abbildung 2 Ein Szenario der Kaffeemaschine. 3
Abbildung 3 Ein zweites Szenario der Kaffeemaschine. 6
Abbildung 4 Das Petrinetzmodell der Kaffeemaschine. 7

Abbildung 5 Ein Foto des Mississippi. 35
Abbildung 6 Ein Flussnetzwerk. 36
Abbildung 7 Die Matrix-Darstellung des Flussnetzwerks. 37
Abbildung 8 Zwei Flüsse in einem Flussnetzwerk. 38
Abbildung 9 Links: Ein flussvergrößernder Weg. Rechts: Das zugehörige Restnetzwerk. 39
Abbildung 10 Ein Schnitt im Flussnetzwerk. 41
Abbildung 11 Ein Flussnetzwerk mit maximalem Fluss 2k. 42
Abbildung 12 Der Algorithmus von Dinic. 46
Abbildung 13 Die Forward-Backward-Propagation. 50
Abbildung 14 Flussnetzwerk, bei dem der Forward-Backward-Propagation Algorithmus vorteilhaft ist. 51
Abbildung 15 Ein Flussnetzwerk, bei dem ein Preflow-Push Algorithmus vorteilhaft ist. 55
Abbildung 16 Der Preflow-Push Algorithmus. 57
Abbildung 17 Der Preflow-Push Algorithmus in einem ungünstigen Beispiel. 58
Abbildung 18 Die Laufzeiten von drei VipTool-Plugins zu Lösung des Fluss-Maximierungs-Problems in Abhängigkeit von der Anzahl der Kanten. 66

Abbildung 19 Ein markiertes S/T-Netz. 70
Abbildung 20 Eine Szenario-Spezifikation. 74
Abbildung 21 Links: Ein markiertes S/T-Netz. Rechts: Ein Prozess. 80
Abbildung 22 Links: Zwei Prozesse des S/T-Netzes aus Abbildung 21. Rechts: Die zwei zu den Prozessen gehörenden Prozess-Szenarien. 81
Abbildung 23 Die Szenario-Verifikation über Prozesse. 84
Abbildung 24 Die Prozesse der Szenario-Verifikation über Prozesse. 85
Abbildung 25 Links: Ein S/T-Netz. Rechts: Ein Szenario mit einem für die Stelle s_3 gültigem Markenfluss. 89
Abbildung 26 Die Modifikation $\mathfrak{M}_1$. 93
Abbildung 27 Die Modifikation $\mathfrak{M}_2$. 93
Abbildung 28 Die Modifikation $\mathfrak{M}_3$. 93
Abbildung 29 Ein Markenfluss x in einem Szenario. 95
Abbildung 30 Ein Flussnetzwerk der Modifikationsfolgen. Die Menge F_m ist nicht vollständig dargestellt. 96
Abbildung 31 Ein markiertes S/T-Netz und ein Szenario. 100
Abbildung 32 Ein assoziiertes Flussnetzwerk. 100
Abbildung 33 Ein maximaler Fluss in einem assoziierten Flussnetzwerk. 101

Abbildung 34 Ein gültiger Markenfluss. 102

Abbildung 35 Links: Eine Szenario-Spezifikation. Mitte: Ein Szenario. Rechts: Ein kompaktes Szenario. 108
Abbildung 36 Links: Ein markiertes S/T-Netz. Mitte: Ein kompaktes Szenario. Rechts: Ein Prozess. 109
Abbildung 37 Links: Ein Szenario mit Markenfluss. Mitte: Die Abbildung *weg*. Rechts: Ein passender kompakter Markenfluss. 111
Abbildung 38 Das assoziierte kompakte Flussnetzwerk zu dem Szenario aus Abbildung 35. 116
Abbildung 39 Links: Innere Kanten eines assoziierten kompakten Flussnetzwerks. Rechts: Inneren Kanten eines assoziierten einfachen Flussnetzwerks. 120

Abbildung 40 Das S/T-Netz eines Geschäftsprozesses. 126
Abbildung 41 Eine Spezifikation des Geschäftsprozesses. 126
Abbildung 42 Eine dichte iterierte Spezifikation. 127
Abbildung 43 Eine lichte iterierte Spezifikation. 131
Abbildung 44 Sechs essende Philosophen. 135
Abbildung 45 Screenshot des Flussnetzwerk-Plugins. 139
Abbildung 46 Screenshot des Verifikations-Plugins. 140

1

EINLEITUNG

In der Informatik gehört es zum guten Ton, sich ausgiebig dem Kaffee hinzugeben. Damit ist nicht nur das Kaffeetrinken gemeint, sondern vielmehr müssen Kaffee und Kaffeemaschinen oft als einführendes Beispiel für neue Theorien und Ideen herhalten. Die erste Webcam, die „Trojan Room Coffee Pot Camera“, zeigt seit dem Jahr 1991 die Kaffeemaschine im Computerlabor der Universität in Cambridge. Die neue Entwicklung diente einem gutem Zweck: Jeder Mitarbeiter konnte sich über den aktuellen Kaffeepegel in der Maschine informieren. Häufig wird die Meinung vertreten, dass regelmäßige Teambesprechungen zur Verbesserung der äußeren Arbeitsbedingungen in der Softwareentwicklung führen, und dabei schafft es der Punkt „die Kaffeemaschine ist kaputt“ leicht auf den ersten Platz jeder Tagesordnung. „Ohne Mampf keinen Kampf“ habe ich damals bei der Bundeswehr gelernt. In der Softwareentwicklung gilt das Gleiche für Kaffee. In der theoretischen Informatik ist der erste Automat vieler Lehrbücher oftmals das Modell eines Kaffeeautomaten, und so schließen wir uns dieser liebgewonnenen Tradition an und beginnen diese Arbeit mit Kaffee.

1.1 EINFÜHRENDES BEISPIEL

Abbildung 1 zeigt die Kaffee-Druckbrühmaschine unseres Lehrgebietes. Wie bei einem Espresso-Kocher wird Wasser in einem Behälter, der sich im unteren Teil der Maschine befindet, zum Kochen gebracht, steigt über ein Rohr nach oben, wird von dort unter dem entstandenen Druck wieder nach unten, erst durch das Kaffeepulver, dann durch ein Sieb in die Kaffeekanne gedrückt. Diese Prozedur wird betrieben, um dem Kaffeepulver ein Höchstmaß an Geschmack zu entlocken. Die Kaffeebohnen mahlen wir für jede Kanne frisch mit unserer elektrischen Kaffeemühle.

Abbildung 1: Unsere Kaffee-Druckbrühmaschine.

Den Prozess des Kaffeekochens kann man wie folgt beschreiben: Der Kaffee muss gemahlen, das Sieb mit Pulver gefüllt und der eingefüllte Kaffee im Sieb verdichtet werden. Dazu muss die Kaffeemaschine entriegelt werden, damit man die Kanne und das Sieb entnehmen kann. Das Sieb wird vom alten Pulver befreit und die Kanne ausgespült. Neues Wasser wird aus der Küche geholt und in die Maschine eingefüllt. Setzt man die Kaffeekanne mit Sieb wieder in die Maschine ein, kann die Kaffeemaschine verriegelt und angeschaltet werden.
Geht man zur Beschreibung des Prozesses methodisch vor, betrachtet man die Arbeitsschritte des Prozesses zunächst isoliert:

- Kaffeebohnen mahlen,
- Kaffeemaschine entriegeln,
- Filter leeren,
- Kaffeekanne auswaschen,
- Wasser mit Kaffeekanne holen,
- Wasser einfüllen,
- Filter füllen und Pulver verdichten,
- Kaffeemaschine zusammensetzen, verriegeln und einschalten.

Bei der Beschreibung von Prozessen unterscheidet man zwischen den Aufgaben eines Prozesses und deren Ausführung. Die Aufgaben nennt man Aktivitäten, die Ausführung einer Aktivität nennt man Ereignis. Die Notwendigkeit, diese beiden Begriffe voneinander zu trennen, wird besonders deutlich, wenn man überlegt, dass eine Aktivität des Prozesses in einem Ablauf mehrfach durchgeführt werden kann. In diesem Fall existiert eine Menge von verschiedenen Ereignissen, welche der gleichen Aktivität zugeordnet sind.
Durch die Ausführung der genannten Arbeitsschritte in der aufgeführten Reihenfolge wird ein möglicher Ablauf unseres Prozesses beschrieben. Bei einem solchen Ablauf spricht man von einem sequentiellen Ablauf. Jedes Ereignis beginnt erst, wenn das vorherige Ereignis abgeschlossen wurde, und die Ereignisse sind damit streng geordnet. Es ist leicht einzusehen, dass es für unseren Prozess neben diesem Ablauf zahlreiche andere Möglichkeiten gibt, die Arbeitsschritte nacheinander auszuführen. So könnte man zum Beispiel mit *Kaffeemaschine entriegeln* beginnen und erst danach die *Kaffeebohnen mahlen* oder *Filter füllen und Pulver verdichten* vor *Wasser einfüllen* ausführen. Diese Ereignisse sind voneinander unabhängig, und es ist gleichgültig, in welcher Reihenfolge sie in einem sequentiellen Ablauf ausgeführt werden. Bei anderen Ereignissen ist es allerdings notwendig, eine Reihenfolge einzuhalten. Erst nach *Kaffeepulver mahlen* kann *Filter füllen* und erst nach *Kaffeemaschine entriegeln* kann *Kaffeekanne auswaschen* ausgeführt werden. Diese Arbeitsschritte sind voneinander abhängig.
Jeder Prozess gibt Bedingungen an die Ordnung der Ereignisse vor. Jede Sequenz von Ereignissen, die die durch den Prozess gegebenen Abhängigkeiten respektiert, ist ein sequentieller Ablauf des Prozesses. Eine Sequenz, die diese Abhängigkeiten verletzt, ist kein Ablauf des Prozesses. Jeder Ablauf des Prozesses beschreibt eine von möglicherweise mehreren Alternativen, den Prozess auszuführen. Die Menge aller Abläufe ist die Sprache des Prozesses. Beschreibt man jeden Ablauf durch

eine Sequenz von Ereignissen, so spricht man von der sequentiellen Sprache des Prozesses.
Für viele Anwendungen ist die Betrachtung der sequentiellen Sprache eines Prozesses zu ungenau. Stellt man sich die Frage, was passiert, wenn unser Prozess des Kaffeekochens von mehreren statt nur von einer Person ausgeführt wird, erkennt man, dass voneinander unabhängige Ereignisse nicht nur in beliebiger Reihenfolge, sondern nebenläufig ausgeführt werden können. Es ist leicht zu sehen, dass jemand schon *Kaffeebohnen mahlen* kann, während ein Anderer *Kaffeekanne auswaschen* und danach *Wasser mit Kaffeekanne holen* durchführen kann. *Kaffeekanne auswaschen* und auch *Wasser mit Kaffeekanne holen* sind jeweils unabhängig vom Arbeitsschritt *Kaffeebohnen mahlen*. Es ist unmöglich, nebenläufiges Verhalten durch einen sequentiellen Ablauf zu beschreiben. Ist man an Nebenläufigkeit interessiert, ordnet man Ereignisse nur dann, wenn zwischen ihnen tatsächlich eine Ordnung bestehen muss. In diesem Fall sagt man, dass die beiden Ereignisse in einer „später als“-Beziehung stehen. Das Eine kann erst „später als“ das Andere ausgeführt werden. Ereignisse, die nicht in einer solchen Beziehung stehen, sind ungeordnet. Eine Ordnung, bei der manche Elemente geordnet und manche Elemente nicht geordnet sind, nennt man treffend Halbordnung. Eine halbgeordnete Menge von Ereignissen nennen wir Szenario.

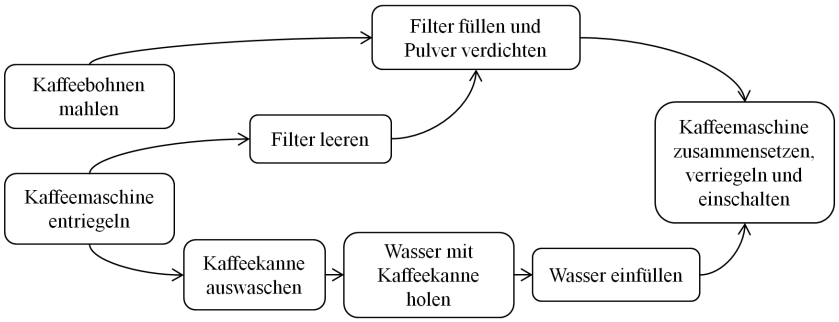

Abbildung 2: Ein Szenario der Kaffeemaschine.

Abbildung 2 zeigt ein Szenario, das Verhalten unseres Prozesses graphisch darstellt. Dazu entsprechen die Ereignisse des Szenarios genau den Arbeitsschritten unserer Aufzählung. Die Ereignisse werden durch abgerundete Vierecke dargestellt und mit den Namen der Aktivitäten beschriftet. Die „später als“-Relation zwischen den Ereignissen wird durch Pfeile dargestellt. Um die Darstellung übersichtlicher zu gestalten, sind hierbei nur die direkten Abhängigkeiten abgebildet. Direkt ist eine Abhängigkeit, wenn sie sich nicht aus zwei anderen im transitiven Abschluss ergibt. In unserem Beispiel ist das Ereignis *Kaffeemaschine entriegeln* eine Voraussetzung für das Ereignis *Filter füllen und Pulver verdichten*, welche aber nicht durch einen zusätzlichen Pfeil ausgedrückt wird, da sich diese „später als“-Beziehung aus den Beziehungen der beiden Ereignisse zu dem Ereignis *Filter leeren* ergibt. Die Menge der direkten Abhängigkeit einer Halbordnung heißt das Skelett der Halbordnung. Das Skelett ist für jede Halbordnung eindeutig. Ein Szenario, welches nur die direkten Abhängigkeiten enthält, nennen wir ein kompaktes Szenario.
Ein Szenario beschreibt sehr anschaulich nebenläufiges Verhalten. Es enthält Ereignisse und ordnet diese nur dort, wo Abhängigkeiten be-

stehen. Szenarien scheinen sich damit hervorragend dafür zu eignen, Verhalten von nebenläufigen Prozessen zu beschreiben. Erstaunlicherweise ist aber die Frage, ob ein Szenario Verhalten eines gegebenen Prozesses beschreibt, recht schwer zu beantworten. Nach unserer Vorstellung muss das Szenario mit seinen Abhängigkeiten die Abhängigkeiten des Prozesses respektieren. Erfüllt es die Abhängigkeiten, so kann man es ausführen. Wie aber überprüfen wir, ob ein Szenario die Abhängigkeiten eines Prozesses erfüllt?
Eine naheliegende Idee ist, das Szenario im Prozess durchzuspielen. Man nimmt die Ereignisse des Szenarios, hält die spezifizierte Halbordnung ein und versucht, die Ereignisse im Prozess auszuführen. Das Problem ist, dass man dabei automatisch eine zeitliche Ordnung ergänzt. Sind zwei Ereignisse im Szenario unabhängig, so treten sie während eines Durchspielens entweder hintereinander oder gleichzeitig ein. Man kann diese durch das Durchspielen entstandene Ordnung aber nicht mit den im Szenario spezifizierten Abhängigkeiten gleichsetzen. Ein Szenario, das die Abhängigkeiten eines Prozesses nicht respektiert, kann durch die beim Durchspielen entstehende zeitliche Ordnung im Prozess ausführbar werden. Wenn man ein Szenario durchspielt, muss man also von der zeitlichen Ordnung des Durchspielens abstrahieren. Dazu notiert man während eines Durchspielens alle Abhängigkeiten, die durch den Prozess entstehen und gleicht diese anschließend mit den im Szenario spezifizierten Abhängigkeiten ab. In einigen Fällen ist dies ein recht ordentliches Verfahren, in anderen ist diese Methode kaum durchführbar. Wir betrachten dazu die Kaffeekasse unseres Lehrstuhls und die vier Mitarbeiter Otto, Alex, Tall und Robin. Otto will ein neues Paket Kaffeebohnen für 12 Euro kaufen. Auf der Kaffeeliste steht, dass Tall noch 5 Euro und Robin noch 10 Euro Kaffeegeld zu zahlen haben. Otto kann sich also von beiden zusammen 12 Euro geben lassen und dann Kaffeebohnen kaufen. Dieser Ablauf ordnet das Kaffeekaufen hinter die beiden Geldübergaben. Wir notieren diese beiden Abhängigkeiten, da diese wirklich im Prozess bestehen. Zu diesem Zeitpunkt macht es keinen Unterschied, wie viel Geld Otto von Robin und wie viel Geld Otto von Tall genau bekommt. Erst wenn Robin das Büro verlässt und Alex danach 3 Euro aus der Kaffeekasse benötigt, um Milch zu kaufen, ist die Frage, ob Tall diese noch schuldig ist, von Bedeutung. Bei einem Durchspielen, bei dem man die Abhängigkeiten notiert, muss man also alle verschiedenen Möglichkeiten, die vorgegebenen Abhängigkeiten des Prozesses zu respektieren, konstruieren. Da es sehr viele solche Möglichkeiten geben kann, führt das einfache Durchspielen nicht zu einem effizienten Verfahren.
Dass Szenarien trotz dieser Schwierigkeiten das geeignete Mittel sind, Verhalten von nebenläufigen Prozessen zu beschreiben, wie man formal definiert, wann ein Szenario Verhalten eines nebenläufigen Systems darstellt, und wie man effizient entscheiden kann, ob ein gegebener Ablauf zu der Sprache eines gegebenen Systems gehört, ist Thema dieser Arbeit.
In der Praxis gewinnt diese Fragestellung und die explizite Beschreibung nebenläufigen Verhaltens durch Szenarien zunehmend an Bedeutung. Aufgaben, Arbeitsschritte und Teilprozesse werden zunehmend verteilt bearbeitet. Dies liegt unter anderem daran, dass es immer einfacher wird, verteilt auf Informationen zuzugreifen. Arbeitsabläufe verteilen sich damit immer stärker mit dem Vorteil, dass sie flexibler

und effizienter werden. So ist es notwendig, dass sich die Informatik mehr und mehr dem Studium der Nebenläufigkeit widmet.
In einer Kooperation zwischen der Katholischen Universität Eichstätt und der Einkaufsabteilung der AUDI AG haben wir die Erfahrung gemacht, dass die AUDI AG vermehrt auf eine Projektorganisation setzt. Anstatt einen Arbeitsablauf in einer Abteilung anzusiedeln, wird ein Projektteam gebildet, das einen dezentralen Arbeitsablauf über verschiedene Abteilungen hinweg koordiniert. So kommt es, dass Teile eines solchen Ablaufes nebenläufig zueinander an verschiedenen Stellen des Unternehmens vorangetrieben werden. Um diese Art von Geschäftsprozessen sinnvoll zu beschreiben, sind Formalismen, die Nebenläufigkeit ausdrücken können, unablässig.
Nicht nur im Kontext der Geschäftsprozessmodellierung, sondern auch in den technischen Bereichen der Informatik ist Nebenläufigkeit entscheidend. So ist es kaum noch möglich, die Taktung von Prozessoren noch weiter zu erhöhen und die Rechengeschwindigkeit von Computern zu verbessern, indem man die Prozessoren schneller arbeiten lässt. Um die Leistung von Rechnern trotzdem noch weiter steigern zu können, ist man dazu übergegangen, die Anzahl der Prozessoren in Computern zu erhöhen. Der einzelne Prozessor wird kaum noch schneller, aber durch die Möglichkeit, sich die Arbeitslast zu teilen, sind Multicore-Prozessoren erheblich schneller als ihre Vorgänger mit nur einem Kern. Auch hier wird ausgenutzt, dass einige Rechenschritte oder ganze Prozesse innerhalb des Computers unabhängig voneinander ablaufen können. Je besser man in der Lage ist, ausschließlich die notwendigen Abhängigkeiten innerhalb eines Programms, eines Protokolls oder eines ganzen Systems zu erfassen und zu beschreiben, desto vorteilhafter kann man es so implementieren, dass es effizient auf den Prozessoren verteilt ablaufen kann.
Bisher wurde durch Abbildung 2 nur ein möglicher Ablauf des Prozesses des Kaffeekochens beschrieben. Typischerweise hat jeder Prozess nicht nur einen, sondern mehrere mögliche Abläufe. Ein Prozess kann damit auf verschiedene Anforderungen, verschiedene Eingaben oder Ressourcenverteilungen reagieren. Ein Prozess enthält dafür an verschiedenen Stellen Alternativen, und ein Szenario beschreibt genau eine Möglichkeit, wie der Prozess durchgeführt werden kann. Ein Szenario enthält keine Alternativen.
Vergleicht man verschiedene Szenarien eines Prozesses, können sich diese in der Menge der Ereignisse, aber auch in der Ordnung dieser untereinander unterscheiden. Betrachtet man dazu, was passiert, wenn man im Prozess des Kaffeekochens das Wasser auch mit einer bereitgestellten Glaskanne holen kann. Als erstes wird der neue Arbeitsschritt *Wasser mit Glaskanne holen* in die Menge der Aktivitäten aufgenommen. Dieser kann nun in einem zweiten Szenario den Arbeitsschritt *Wasser mit Kaffeekanne holen* ersetzten. Abbildung 3 zeigt ein zweites Szenario unseres Prozesses.
Man erkennt, dass es nicht mehr nötig ist, *Kaffeekanne auswaschen* durchzuführen, bevor man *Wasser mit Glaskanne holen* oder *Wasser einfüllen* durchführt. Das bedeutet, dass sich nicht nur die Menge der Ereignisse, sondern auch die Beziehungen der Ereignisse untereinander geändert haben. Jedes Mal, wenn wir an unserem Lehrstuhl Kaffee kochen, haben wir die Möglichkeit zu entscheiden, ob wir die Kaffeekanne oder die Glaskanne zum Holen des Wassers benutzen, und der gesamte Prozess lässt sich durch die zwei genannten alternativen Szenarien beschreiben.

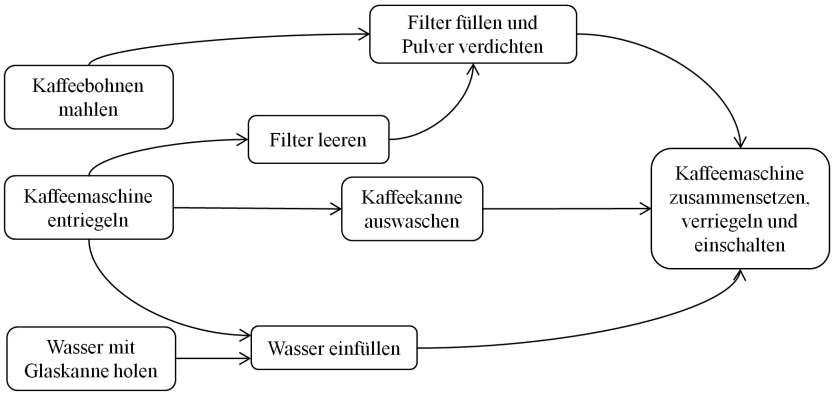

Abbildung 3: Ein zweites Szenario der Kaffeemaschine.

Sprachen heutiger Prozesse sind oft sehr groß. Je flexibler ein Prozess ist, desto unübersichtlicher wird die Menge seiner Szenarien. Zwar gibt es Ansätze, nach denen man ein System nur durch diese Menge beschreibt, üblicherweise ist man allerdings an einer integrierten Darstellung interessiert. Eine integrierte Darstellung enthält bedeutend weniger Redundanz als die Menge aller Szenarien. Dies liegt daran, dass man in diesen Darstellungen gleiche Teile verschiedener Szenarien nur einmal abbildet. Dies erhöht die Übersichtlichkeit, ist leichter zu dokumentieren und vereinfacht damit die Erstellung, Validierung und Wartbarkeit eines Modells. Zudem verbessert eine kompakte Darstellung die Laufzeit von Algorithmen auf dem Modell und erleichtert die Simulation.
Petrinetze sind der bekannteste Formalismus, um Systeme mit nebenläufigem Verhalten zu modellieren. Sie sind die formale Basis für alle heutigen Modellierungssprachen zur Beschreibung von Systemen und Prozessen. Die wohl bekanntesten Modellierungssprachen, die sich direkt auf Petrinetze zurückführen lassen, sind:

- Die von der Objekt Management Group (OMG) standardisierten Aktivitätsdiagramme als Teil der Unified Modeling Language (UML),
- die von August-Wilhelm Scheer entwickelten Ereignisgesteuerten Prozessketten (EPKs) des ARIS-Konzeptes,
- die von der OMG und der Business Process Modeling Initiative (BPMI) gepflegte Sprache Business Process Modeling Notation (BPMN).

Das Konzept der Petrinetze entwickelte Carl Adam Petri in den sechziger Jahren in seiner Dissertation [78]. Ziel war es, nebenläufige Systeme intuitiv und exakt darzustellen. Seit dieser Arbeit wurde diese Urform der Petrinetze in vielen wissenschaftlichen Arbeiten weiterentwickelt. Heute existiert eine Vielzahl von Petrinetzdialekten und eine breite Basis an Algorithmen und Methoden, um Petrinetze zu analysieren, zu simulieren und zu bewerten. Einer der Gründe für den Erfolg der Petrinetze ist die Kombination aus einer intuitiven, graphischen Darstellung und der Möglichkeit einer exakten mathematischen Beschreibung, die die Abhängigkeiten und Alternativen zwischen Ereignissen

in nebenläufigen Systemen abbilden kann. Die Funktionsweise eines Petrinetzes beschreibt man am besten zusammen mit seiner graphischen Darstellung. Abbildung 4 zeigt das Petrinetzmodell des Prozesses des Kaffeekochens.

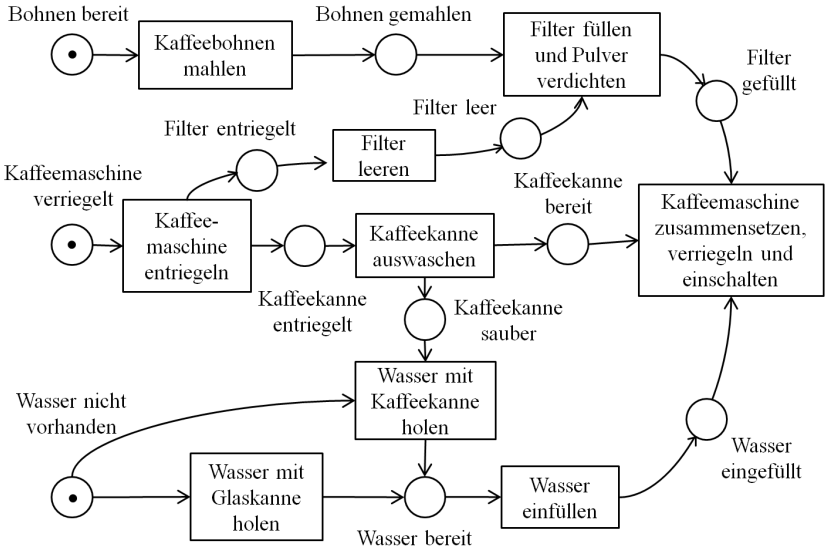

Abbildung 4: Das Petrinetzmodell der Kaffeemaschine.

Ein Petrinetz besteht aus zwei Arten von beschrifteten Knoten, die durch Pfeile verbunden sind. Die erste Art von Knoten heißt Stellen und wird durch Kreise dargestellt. Stellen können Marken tragen und, falls sie dieses tun, wird für jede Marke ein schwarzer Punkt in die entsprechende Stelle gezeichnet. Jede Stelle mit ihren Marken repräsentiert einen lokalen Zustand des Petrinetzes. Die Verteilung der Marken über alle Stellen ist der globale Zustand des Petrinetzes, den man Markierung des Petrinetzes nennt. Man gibt Stellen sprechende Namen, diese können entweder den Teilzustand beschreiben oder sich aus einer Ressource des Systems ergeben. Obwohl die Namensgebung der Stellen beim Ausführen des Petrinetzes keine Rolle spielt, erhöht eine sprechende Namensgebung die Lesbarkeit eines Petrinetzes.
In Abbildung 4 sind die Stellen mit den entsprechenden Teilzuständen beschrieben. Dabei ist jeder Teilzustand des Systems durch eine Ressource des Prozesses zusammen mit dem jeweiligen Zustand dieser Ressource beschrieben (*Kaffeebohnen bereit*, *Bohnen gemahlen*, *Filter gefüllt* usw.). Hierbei soll nun die Belegung einer Stelle mit einer Marke bedeuten, dass sich der Prozess in dem durch die Beschriftung beschriebenen Teilzustand befindet. In Abbildung 4 sind die *Bohnen bereit*, die *Kaffeemaschine entriegelt* und das *Wasser nicht vorhanden*. Da dies den Zustand beschreibt, in dem sich der Prozess vor Beginn der Durchführung befindet, nennt man diese Markierung die Anfangsmarkierung des Petrinetzes.
Die zweite Art von Knoten sind Transitionen, sie werden durch Vierecke dargestellt. Jede Transition repräsentiert eine Aktivität im Petrinetz. Die Beschriftung der Transition ist der Name der Aktivität, und so sind die Transitionen in Abbildung 4 mit den Aktivitäten unseres Prozesses des Kaffeekochens *Kaffeebohnen mahlen*, *Filter leeren*, *Kaffeekanne auswaschen* usw. beschrieben. Transitionen können unter gewissen Umständen ausgeführt werden. Wir sagen, dass eine ausführbare Transition

im Petrinetz aktiviert ist. Das Ausführen einer Transition nennt man Schalten. Durch das Schalten einer Transition wird der Zustand des Petrinetzes geändert. Die Zustandsänderung wird durch gerichtete, gewichtete Kanten beschrieben, die Transitionen und Stellen verbinden. Kanten werden durch Pfeile dargestellt. Falls das Kantengewicht größer Eins ist, wird es am Pfeil notiert. Kanten verbinden immer nur Konten unterschiedlichen Typs, so gehen Pfeile nie von Stelle zu Stelle oder von Transition zu Transition.
Betrachtet man einen Knoten des Petrinetzes, so heißt die Menge der Knoten, von denen aus Kanten zu diesem Knoten führen, der Vorbereich des Knotens. Die Menge der Knoten, zu denen von diesem Knoten aus Kanten führen, heißt der Nachbereich des Knotens. So ist in Abbildung 4 der Vorbereich der Transition *Kaffeemaschine entriegeln* die Stelle *Kaffeemaschine verriegelt*. Der Nachbereich der Transition *Kaffeemaschine entriegeln* besteht aus den beiden Stellen *Filter entriegelt* und *Kaffeekanne entriegelt*. Die sogenannte Schaltregel sagt nun, dass eine Transition schalten kann, wenn jede Stellen ihres Vorbereichs mindestens die Anzahl von Marken trägt, die durch das Kantengewicht der Kante angegeben ist, welche die beiden Knoten verbindet. Beim Schalten konsumiert die Transition aus jeder Stelle ihres Vorbereichs jeweils die durch das Kantengewicht angegebene Anzahl an Marken und erzeugt wieder entsprechend der Kantengewichte neue Marken in den Stellen in ihrem Nachbereich. In unserem Beispiel ist die Transition *Kaffeemaschine entriegeln* aktiviert, da sich in der Stelle *Kaffeemaschine verriegelt* eine Marke befindet. Schaltet die Transition, so wird diese Marke entfernt und zwei neue werden produziert, eine neue Marke in der Stelle *Filter entriegelt* und eine neue Marke in der Stelle *Kaffeekanne entriegelt*. Auf diese Weise ändert eine Transition den Zustand des Petrinetzes und repräsentiert ein Ereignis des Prozesses.
Betrachten wir nun die aktivierten Transitionen des in Abbildung 4 dargestellten Petrinetzes, so kann man die beiden Transitionen *Kaffeebohnen mahlen* und *Kaffeemaschine entriegeln* in der Anfangsmarkierung schalten. Dadurch verschwinden die Marken auf den Stellen *Bohnen bereit* und *Kaffeemaschine verriegelt*. Dafür entsteht je eine neue Marke auf den drei Stellen *Bohnen gemahlen*, *Filter entriegelt* und *Kaffeekanne entriegelt*. Denken wir weiter und überlegen uns, dass in dieser Markierung die Transitionen *Filter leeren* und *Kaffeekanne auswaschen* aktiviert sind und schalten, existieren danach fünf Marken im Petrinetz. Je eine Marke auf den Stellen *Bohnen gemahlen*, *Filter leer*, *Kaffeekanne bereit*, *Kaffeekanne sauber* und die noch aus der Anfangsmarkierung stammende Marke in der Stelle *Wasser benötigt*. Das Schalten der Transition *Filter füllen und Pulver verdichten* konsumiert zwei dieser Marken und eine Marke in der Stelle *Filter gefüllt* entsteht. Das Schalten der Transition *Wasser mit Kaffeekanne holen* konsumiert die zwei Marken aus den Stellen *Kaffeekanne sauber* und *Wasser nicht vorhanden*, produziert eine Marke in *Wasser bereit*. Nun kann die Transition *Wasser einfüllen* und danach zu guter Letzt die Transition *Kaffeemaschine zusammensetzen, verriegeln und einschalten* schalten. Nach diesem Ereignis ist keine der Transitionen mehr aktiviert und ein mögliches Durchspielen des Petrinetzes endet. Man spricht von einem möglichen Szenario des Petrinetzes und die Menge aller dieser Szenarien ist die Szenario-Sprache des Petrinetzes. Ein zweites Szenario dieser Szenario-Sprache entdeckt man, wenn man die Stelle *Wasser bereit* betrachtet. Diese Stelle kann entweder durch ein Schalten der Transition *Wasser mit Kaffeekanne holen* oder durch ein

Schalten der Transition *Wasser mit Glaskanne holen* mit einer Marke belegt werden. Setzen wir unser Petrinetz gedanklich wieder zurück auf die abgebildete Anfangsmarkierung, sehen wir, dass *Wasser mit Glaskanne holen* schon in dieser Markierung schalten kann. Dieses Schalten führt einerseits dazu, dass die Transition *Wasser einfüllen* im nächsten Schritt schalten könnte und dass andererseits der Transition *Wasser mit Kaffeekanne holen* die Marke in der Stelle *Kaffeekanne bereit* weggenommen wird. Diese Transition wird in diesem zweiten Szenario nicht mehr schalten können.
Spielen wir das Petrinetz durch, besteht zwischen dem Schalten mancher Transitionen keine Abhängigkeit und diese Transitionen sind nebenläufig. Das bedeutet, dass sie sich beim Schalten nicht um Marken streiten. Entweder sind dazu die Vorbereiche der Transitionen disjunkt oder die Anzahl der Marken ist in den geteilten Stellen groß genug, dass jede Transition genügend Marken entsprechend der Kantengewichte konsumieren kann. In Abbildung 4 können zum Beispiel die drei in der Anfangsmarkierung aktivierten Transitionen *Bohnen mahlen*, *Kaffeemaschine entriegeln* und *Wasser mit Glaskanne holen* nebenläufig schalten.
Das Durchspielen und Ausprobieren der möglichen Szenarien in einem Petrinetz nennt man das Markenspiel. Beginnend bei einer Markierung wählt man eine aktivierte Menge von Transitionen aus und schaltet diese. So verändert sich die Markierung des Netzes und somit auch die Menge der im nächsten Schritt aktivierten Transitionen. Durch das Markenspiel lassen sich Zusammenhänge zwischen den Schaltvorgängen in einem Petrinetz begreifen, und es ergibt sich ein erster Hinweis darauf, ob man bei der Modellierung eines Prozesses die gewünschten Abhängigkeiten im Petrinetz abbilden konnte. Doch selbst in kleinen und nicht sehr komplexen Petrinetzen ist es nicht trivial, die Menge der mögliche Szenarien und Alternativen auf den ersten Blick oder mit Hilfe des Markenspiels zu erkennen. Es bleibt die wichtige Frage, ob das Petrinetz-Modell die Wirklichkeit korrekt und vollständig abbildet. Im Idealfall ist eine Menge von Szenarien des zu modellierenden Systems bekannt, bei kleinen Systemen meist implizit im Kopf des Modellierenden, bei größeren Systemen explizit. In beiden Fällen lässt sich die Frage nach der Validität des Modells in zwei Fragen aufteilen: Sind alle gewollten Szenarien in dem Modell enthalten und sind keine zusätzlichen Szenarien im Modell möglich? Zur Beantwortung beider Fragen kommt der Entscheidung, ob ein spezifiziertes Szenario in einem gegebenen Modell durchführbar ist, zentrale Bedeutung zu. In unserem Beispiel ist mit den Abbildungen 2 und 3 eine formale Spezifikation des gewünschten Prozesses gegeben. Die Frage ist nun, ob die beiden beschriebenen Szenarien im Petrinetz aus Abbildung 4 durchführbar sind.
Die Aufgabe zu entscheiden, ob ein gegebenes Szenario in einem gegebenen Petrinetz durchführbar ist, nennt man das Szenario-Verifikations-Problem. Zu jedem Petrinetz existiert eine Menge von in diesem Petrinetz durchführbaren Szenarien, die Szenario-Sprache des Petrinetzes. Damit ist das Szenario-Verifikations-Problem die Frage danach, ob ein gegebenes Szenario zu der Szenario-Sprache eines gegebenen Petrinetzes gehört. Die Szenario-Sprache ist in der Literatur auf drei verschiedene Arten charakterisiert. Jede dieser Charakterisierungen induziert einen möglichen Ansatz zur Lösung des Szenario-Verifikations-

Problems. Die Entwicklung einer effizienten Lösung des Szenario-Verifikations-Problems ist Aufgabe dieser Arbeit.
Die erste Charakterisierung der Szenario-Sprache eines Petrinetzes ergibt sich aus der Definition eines Prozessnetzes [55, 54]. Ein Prozessnetz beschreibt genau wie ein Szenario eine Durchführung eines Petrinetzes. Dazu existiert im Prozessnetz für jedes Ereignis eine Transition und für jede Marke, die während der Durchführung des Petrinetzes entsteht, existiert eine Stelle. Diese Transitionen und Stellen im Prozessnetz sind entsprechend der Schaltregel des Petrinetzes durch Kanten verbunden. Da das Prozessnetz auf diese Weise die Schaltregel respektiert, beschreibt jedes Prozessnetz eine Durchführung des Petrinetzes. Jedes Petrinetz lässt sich durch Entfaltungsalgorithmen in eine endliche Repräsentation der Menge seiner Prozessnetze zerlegen. Jedes Prozessnetz beschreibt eine mögliche Ordnung der Ereignisse des Prozesses. Aus dieser Ordnung lässt sich die Szenario-Sprache des Petrinetzes ablesen. Die Anzahl der Prozessnetze wächst allerdings oft exponentiell in der Größe des zu entfaltenden Petrinetzes, wodurch die Laufzeit auf diesem Ansatz basierender Algorithmen exponentiell mit der Eingabe wächst.
Die zweite Charakterisierung der Szenario-Sprache eines Petrinetzes ergibt sich aus der Definition aktivierter Schnitte eines Szenarios (siehe [56]). Ein Schnitt ist eine maximale durch die zu Grunde liegende Halbordnung nicht geordnete Menge von Ereignissen. Die Menge aller Ereignisse, die vor einem Schnitt liegen, nennen wir das durch den Schnitt erzeugte Präfix. Ein Szenario beschreibt damit eine Durchführung eines Petrinetzes, wenn die Ereignisse jedes Schnittes in der Markierung, die durch das Schalten ihres Präfixes entsteht, nebenläufig zueinander aktiviert sind. In diesem Fall nennt man einen Schnitt aktiviert, und ein Szenario, dessen Schnitte aktiviert sind, ist in der Szenario-Sprache des Petrinetzes enthalten. Auch die Anzahl der Schnitte eines Szenarios wächst exponentiell in der Größe des gegebenen Szenarios, wodurch auch auf diesem Ansatz basierende Algorithmen exponentielle Laufzeit besitzen.
Die dritte Charakterisierung der Szenario-Sprache eines Petrinetzes ergibt sich aus der Definition gültiger Markenflüsse (siehe [60]). Ein gültiger Markenfluss ist eine Verteilung von Marken auf der „später als“-Relation, so dass die Anzahl an Marken, die zu und von Ereignissen fließen, konform zur Schaltregel des Petrinetzes ist. Konform bedeutet in diesem Fall, dass jedes Ereignis nur die Anzahl an Marken weitergibt, die die zu dem Ereignis gehörende Transition produziert, und dass jedes Ereignis die Anzahl an Marken erhält, die die zu dem Ereignis gehörende Transition konsumiert. Existiert eine solche Verteilung von Marken auf der „später als“-Relation eines Szenarios, so werden alle Marken, die für die Durchführung des Szenarios benötigt werden, produziert, bevor sie konsumiert werden und das Szenario ist in der Szenario-Sprache des Petrinetzes enthalten. Auch die Anzahl der gültigen Markenflüsse wächst exponentiell in der Größe des gegebenen Szenarios, doch gibt es eine Möglichkeit, in Polynomialzeit zu entscheiden, ob ein gültiger Markenfluss für ein Szenario und zu einem Petrinetz existiert. Ein entsprechender Algorithmus wurde in [14, 11] beschrieben.
Das Szenario-Verifikations-Problem ist also mit Hilfe der Definition der gültigen Markenflüsse in polynomieller Laufzeit entscheidbar. Es hat sich jedoch gezeigt, dass die Laufzeit und die Speicherplatz-Komplexität des entsprechenden Algorithmus in vielen Fällen groß

sind und in einigen sogar größer als die eines Algorithmus, der die Definition der aktivierten Schnitte verwendet. Besonders kritisch wird die Laufzeit dann, wenn ein Szenario viel Ordnung enthält. Ein wichtiger Schritt zur Verbesserung der Laufzeit gelang Robert Lorenz in [66]. Er beschreibt einen Algorithmus, der um den Faktor der Anzahl der in dem Szenario enthaltenen Ereignisse schneller läuft als der Algorithmus in [14]. Dieser Algorithmus wurde für die vorliegende Arbeit erstmalig implementiert.
Eine weitere Verbesserung der Laufzeit und ein damit für jede Eingabe effizienter Algorithmus ist Gegenstand dieser Arbeit. Für diesen Algorithmus ist es notwendig, eine vierte Charakterisierung der Szenario-Sprache zu entwickeln. Diese Charakterisierung wird auf der Definition der kompakten Markenflüsse beruhen, die im Gegensatz zu den einfachen Markenflüssen keine konkrete Verteilung der Marken mehr beschreibt. Durch diese Abstraktion ist es möglich, einen kompakten Markenfluss nur auf dem Skelett der Ordnung eines Szenarios zu definieren und bei der Konstruktion kompakter gültiger Markenflüsse weitere Laufzeit und Speicherplatz zu sparen. Der große Vorteil dieser neuen Definition ist, dass die Laufzeit eines entsprechenden Algorithmus kaum noch von der Struktur des zu testenden Szenarios abhängig ist. Der Makel der Markenflüsse, nur im Fall lichter, also wenig Ordnung enthaltender, Szenarien effizient zu sein, wird ausgeräumt.
Das einfache Beispiel unserer Kaffeemaschine hat uns nicht nur für die Vorteile der Betrachtung halbgeordneten Verhaltens sensibilisiert und eine erste Einführung in die Modellierung mit Petrinetzen geliefert, sondern zugleich zur Fragestellung dieser Arbeit hingeführt: Wie entscheidet man das Szenario-Verifikations-Problem effizient?

1.2 ANWENDUNG

In diesem Abschnitt wollen wir ergänzend zu dem beschriebenen „Toy Example“ einer Kaffeemaschine zwei Projekte unseres Lehrstuhls betrachten, die die Fragestellung dieser Arbeit zusätzlich motivieren. Beim ersten Projekt handelt es sich um ein Industrieprojekt unseres damaligen Lehrstuhls an der Katholischen Universität Eichstätt und der Einkaufsabteilung der AUDI AG in Ingolstadt. Das zweite Projekt ist eine Kooperation zwischen unserem Lehrgebiet an der Fernuniversität Hagen und dem Therapiezentrum Medifit in Hagen. In beiden Projekten sollte ein geeignetes Vorgehen ausgewählt oder entwickelt werden, um in dem jeweiligen Anwendungskontext ein valides Modell der Arbeitsabläufe des Unternehmens erstellen zu können. In beiden Fällen erwies es sich als vorteilhaft, die Aufmerksamkeit zunächst auf einzelne Arbeitsabläufe des Unternehmens und nicht etwa auf das gesamte System zu legen. Die Beschreibung der Projekte und die Darstellung der entwickelten Vorgehensmodelle verdeutlicht die praktische Anwendbarkeit dieser Arbeit.
„Allgemein ist ein Geschäftsprozess eine zusammengehörende Abfolge von Unternehmensverrichtungen zum Zweck einer Leistungserstellung.“ [83]. Die Modellierung einer solchen Abfolge heißt treffend Geschäftsprozessmodellierung. Sie ist ein wichtiger Teil der modernen Softwareentwicklung [72, 75]. Software wird zum größten Teil für Unternehmen entwickelt und auf das jeweilige Unternehmen maßgeschneidert, da sie auf die Arbeitsabläufe abgestimmt sein muss. Da mittlerweile die Zahl der Modellierungssprachen, der Anwendungs-

gebiete und der Einsatzmöglichkeiten der Geschäftsprozessmodelle stetig steigt, hat sich die Geschäftsprozessmodellierung über die Softwareentwicklung hinaus zu einer eigenständigen Disziplin entwickelt [1, 90, 46, 83, 2]. Ziel der Geschäftsprozessmodellierung ist in erster Linie, die Organisation komplexer Unternehmen zu unterstützen, indem sie Dokumentation, Reorganisation, Optimierung, Zertifizierung oder Kostenrechnung ermöglicht. Zudem werden Geschäftsprozesse zunehmend als Eingabe für geschäftsprozesssteuernde Systeme genutzt, um z.B. die korrekte Ausführung von Arbeitsschritten zu überwachen.
Um valide Geschäftsprozessmodelle zu erstellen, wurden in der Geschäftsprozessmodellierung spezielle Vorgehensmodelle entwickelt. Diese Vorgehensmodelle orientieren sich stark an den Vorgehensmodellen der Softwareentwicklung, ohne dabei die spezifischen Anforderungen in der Geschäftsprozessmodellierung zu vernachlässigen [90, 1]. In den Projekten unseres Lehrstuhls stand besonders die Phase der Anforderungserhebung im Fokus. Diese Phase besitzt einen besonderen Reiz, da sich die Arbeitsabläufe in einem Unternehmen stark von dem Begriff eines Ablaufs im Bereich der Softwareentwicklung unterscheiden. Oft beinhalten die Arbeitsabläufe die Interaktionen zwischen Menschen und existieren unabhängig von einem System, wobei die einzelnen Arbeitsschritte eines solchen Ablaufes in der Regel durch reale Ressourcen verknüpft sind. Auch für dieses Umfeld wurden Ansätze beschrieben, um die Anforderungen an einen Geschäftsprozess systematisch erheben zu können [72, 82]. Bei modernen Ansätzen werden zunächst einzelne Abläufe des Geschäftsprozesses erhoben und nicht direkt Verhalten des gesamten Systems. Ist die Menge der Abläufe formalisiert, so lässt sich das Geschäftsprozessmodell aus diesen generieren. Entsprechende Ansätze sind in [35, 37, 42, 39, 85, 25] beschrieben. Gerade im Bereich der Geschäftsprozesse ist ein einzelnes Szenario einfach zu validieren, da es sich auf intuitive Weise erheben, darstellen und kommunizieren lässt.

1.2.1 *Geschäftsprozesse der AUDI AG*

Unser erstes Projekt war ein Industrieprojekt, das zwischen unserem Lehrstuhl an der Katholischen Universität Eichstätt und der Einkaufsabteilung der AUDI AG bestand. Ziel war die Entwicklung eines Vorgehensmodells, mit dem die Einkaufsabteilung ein valides und korrektes Modell ihrer Geschäftsprozesse erstellen konnte. Solche Modelle sind Teil der Dokumentation der Geschäftsprozesse der gesamten AUDI AG, die verpflichtet ist, diese im Rahmen der TÜV-Zertifizierung für deutsche Automobilhersteller zu erstellen.
Die Hauptschwierigkeit bestand darin, dass die Geschäftsprozesse der Einkaufsabteilung eng mit den anderen Bereichen der AUDI AG verwoben sind. Ein Grund dafür ist, dass viele Geschäftsprozesse der Einkaufsabteilung erst angestoßen werden, wenn es in anderen Abteilungen der AUDI AG zu Engpässen kommt, sei es aufgrund von Fehlplanung oder unerwartet hoher Nachfrage. Dadurch ist es selbstverständlich, dass im Verlauf dieser Geschäftsprozesse mit der zu beliefernden Abteilung und mit den Lieferanten kommuniziert und interagiert werden muss. Aus den gleichen Gründen sind die Geschäftsprozesse der Einkaufsabteilung von hoher Priorität und müssen extrem flexibel sein.

Die Erhebung der Geschäftsprozesse und deren Integration in ein Modell sollte von den Mitarbeitern der Einkaufsabteilung durchgeführt werden. Das entstandene Modell sollte dann von der Abteilung Qualitätsmanagement kontrolliert und in das Gesamtmodell der AUDI AG eingepasst werden. Das Gesamtmodell der AUDI AG ist in der Modellierungssprache der Ereignisgesteuerten Prozessketten (EPKs) des ARIS-Konzeptes abgebildet [85, 83] und besteht aus fünf Schichten, was die Übersichtlichkeit erhöht. Wie bei einem klassischen Top-Down-Entwurf zeigt die oberste Schicht einen stark abstrahierten Entwurf aller Geschäftsprozesse der AUDI AG, und jede weitere Schicht verfeinert die auf der über ihr liegenden Schicht befindlichen Aktivitäten der Geschäftsprozesse. Die Aufgabe, die eigenen Prozesse aufzunehmen, war in allen Abteilungen, die wir in unserer Zeit bei der AUDI AG kennengelernt haben, unbeliebt. Dabei sollte die Erhebung der Geschäftsprozesse neben dem eigentlichen Kerngeschäft durchgeführt werden. Zu diesem Zweck wurde aus jeder Abteilung ein Mitarbeiter auf eine zweitägige ARIS-Schulung geschickt, um danach die Prozesse seiner Abteilung beschreiben zu können.
Für unseren Lehrstuhl ergab sich die Möglichkeit, an dieser Schulung teilzunehmen und wir erkannten, dass wohl niemand, der nicht über umfangreiches Vorwissen verfügt, nach einer zweitägigen Schulung in der Lage ist, die Geschäftsprozesse eines Konzerns wie der AUDI AG korrekt abzubilden. Man ging deshalb dazu über, die Geschäftsprozesse stattdessen in Workshops zu erstellen. Dazu trafen sich die an dem Prozess beteiligten Personen in einem Seminarraum und das Modell wurde „on the fly“ meist auf einer großen Tafel erstellt. Der Vorteil eines solchen Workshops ist, dass sich alle am Geschäftsprozess beteiligten Personen Zeit für die Erhebung nehmen, und damit das Modell im direkten Kontakt und somit unter der Kontrolle aller Beteiligten erhoben werden kann. Der große Nachteil an diesem Verfahren ist, dass unterschiedliche Meinungen über die reale Ausführung der Geschäftsprozesse nur durch eilig geschlossene Kompromisse ausgeräumt werden, da der Rahmen des Workshops keinen Raum für eine ausführliche Erhebung, Verhandlung oder detaillierte Analyse des erstellten Modells lässt. Dazu kommt, dass bei einem solchen Vorgehen keinerlei Dokumentation über den Entstehungsprozess des Modells produziert wird. Entdeckt man bei der Integration Fehler im erstellten Geschäftsprozessmodell, ist die Wartung des Modells ohne eine derartige Dokumentation eine sehr schwierige Aufgabe. Erstens gibt es keine Möglichkeit, die Entstehung des Fehlers nachzuvollziehen, zweitens gibt es auch keine Spezifikation, gegen die man eine korrigierte Version des Modells erneut prüfen kann. Damit wird ein Ausbessern des gefundenen Fehlers zum Ratespiel, in dem meist weitere Fehler im Modell eingebaut werden. Die einzige Lösung ist es, den Workshop zu wiederholen, was natürlich sehr aufwendig ist.
In unserem gemeinsamen Projekt wollten wir nun ein Vorgehensmodell entwickeln, mit dessen Hilfe die Mitarbeiter der Einkaufsabteilung ihre Geschäftsprozesse systematisch in ein korrektes Modell überführen können. Im Laufe des Projektes stellte sich heraus, dass in diesem Umfeld ein szenariobasierter Modellierungsansatz am besten geeignet ist, um die Geschäftsprozesse zu erheben und zu formalisieren. Szenariobasiert bedeutet hierbei, dass das Modell nicht direkt erstellt wird, sondern zunächst die Abläufe des Geschäftsprozesses erhoben, in Szenarien formalisiert und validiert werden. Diese Szenarien werden dann in

einen zweiten Schritt zu einem Modell integriert. Einen entsprechenden Modellierungsansatz haben wir im Rahmen des angesprochenen Industrieprojektes im Detail entwickelt und ausgearbeitet. Er ist in [13] beschrieben. Im Folgenden werden einige Erkenntnisse aus dem Projekt zusammengefasst.
In Gesprächen, die wir im Rahmen der Prozesserhebung mit Mitarbeitern der AUDI AG führten, fragten wir ganz bewusst zunächst nach Beispielen einzelner Geschäftsprozesse. Gerne zeigten uns die Befragten alte Ordner aus ihren Regalen und stolz wurde berichtet, wie man die für die Produktion notwendigen, aber fehlenden Teile noch beschaffen konnte. Nicht selten erhielten wir Antworten, die mit „Damals beim A6 quattro . . . " begannen. Es fällt oft leichter einen einzigen und wirklich erlebten Ablauf als Beispiel zu beschreiben als einen abstrakten Prozess. So erinnerten sich die Befragten an alle Details, Arbeitsschritte und Abhängigkeiten. Zudem konnten sie fast jeden Schritt mit Dokumenten, Formularen und alten Notizen belegen. Erst nachdem eine Geschichte und damit ein konkreter Ablauf vollständig beschrieben worden war, gingen wir dazu über, diese in eine allgemeine Form zu gießen. Wir abstrahierten vom konkreten Beispiel und stellten das entstandene Szenario formal und graphisch dar. Ein Szenario kommt dabei ganz ohne Alternativen aus, und es ist für die Informationslieferanten einfach, das formale Szenario mit dem berichteten Beispielablauf im Hinterkopf in einem nächsten Schritt zu validieren. Auf diese Weise erhielten wir allmählich eine Menge formaler und validierter Szenarien der Geschäftsprozesse und damit eine valide Spezifikation für unser gesuchtes Geschäftsprozessmodell. Im letzten Schritt konnten wir die erhobenen Abläufe zu einem Modell integrieren.
Das Formalisieren der einzelnen Beispielabläufe kann durch eine einfache Extraktion der Aktivitäten und deren Abhängigkeiten aus den Beschreibungen erfolgen. Da das Gesamtmodell der AUDI AG in seinen fünf Schichten in der Modellierungssprache der EPKs formuliert ist, haben wir uns damals für den passenden Formalismus der Instanz EPKs [85] entschieden, um die Abläufe zu formalisieren. Dabei ist ein Instanz EPK ein EPK, das keine Alternativen enthält und somit stark einem Szenario ähnelt. Der einzige Unterschied ist, dass Instanz EPKs neben den Ereignissen noch eine zweite Art von Knoten erlauben, die zu Objekten oder Zuständen des Ablaufes assoziiert sind.
Im letzten Schritt wird die Menge der formalen Szenarien in einem Modell integriert. Die Instanz EPKs werden gefaltet, d.h. gleiche Ereignisse und Objekte verschmolzen, wodurch ein kompaktes Modell entsteht. Dieses Modell enthält dann Alternativen, die im Laufe des Verschmelzungsprozesses eingeführt werden müssen, um unterschiedliche Abläufe im integrierten Modell zu trennen. Ein solches Vorgehen gewährleistet, dass alle spezifizierten Instanz EPKs im entstehenden Modell enthalten sind. Es bleibt sicherzustellen, dass dabei keine ungewollten zusätzlichen Abläufe entstehen. Eine ausführliche Diskussion dieser Fragestellung und den Algorithmus, der die Instanz EPKs zu einem EPK integriert, findet sich im Workshopbeitrag [25].

1.2.2 *Geschäftsprozesse im Therapaedicum Medifit*

Unser zweites Projekt war eine Kooperation unseres Lehrstuhls mit einem Hagener Therapiezentrum, dem Therapaedicum Medifit Hagen. Das Therapiezentrum ist über drei verschiedene Standorte verteilt,

wobei jeder Standort direkt an ein Krankenhaus angeschlossen ist. Zur Zeit sind insgesamt 58 Mitarbeiter aus den Berufsgruppen Physiotherapie, Ergotherapie, Logopädie und Tanztherapie sowie zwei Verwaltungsangestellte in der Rezeption für das Therapiezentrum tätig. In der Kooperation setzten wir uns das Ziel, einen geeigneten Formalismus und ein geeignetes Vorgehen zu finden, mit dem das Therapiezentrum seine Geschäftsprozesse dokumentieren kann. Die Geschäftsprozesse des Therapiezentrums sind einerseits Prozesse, die sich mit der Verwaltung von Patienten befassen, andererseits beschreiben sie die Behandlung der Patienten. Das Therapiezentrum ist an der Dokumentation dieser Geschäftsprozesse interessiert, da diese im Rahmen der Zertifizierung durch das Institut für Qualitätssicherung in der Heilmittelbranche (IQH) beschrieben werden müssen.
Der Kontakt mit dem Therapiezentrum ist wohl dem körperlichen Zustand der Mitarbeiter unseres Lehrgebiets zu verdanken. Die Kooperation kam zustande, da das Therapiezentrum die Chance nutzte, sich bei der Erstellung der Dokumentation unterstützen zu lassen, während wir die Möglichkeit sahen, das im AUDI Projekt erstellte Vorgehensmodell in einer weiteren Anwendung zu testen.
Bei der Erhebung der Geschäftsprozesse sollte der Fokus wieder auf einer Erhebung der einzelnen Szenarien liegen. Diese grundlegende Idee hatte sich bereits im AUDI Projekt bewährt und schien uns in der Anwendung bei Medifit zwingend. Dafür gab es verschiedene Gründe: Erstens gilt für alle Mitarbeiter des Therapiezentrums, dass sie aufgrund ihres beruflichen Hintergrunds noch nie mit Geschäftsprozessmodellierung oder einer formalen Darstellung von Abläufen oder Systemen in Berührung gekommen sind. Meist hilft schon die Kenntnis einer Programmiersprache dabei, sich auch in der Geschäftsprozessmodellierung schneller zurecht zu finden, da man mit den Konstrukten Sequenz, Schleife und Verzweigung bereits vertraut ist. Bei den Mitarbeitern des Therapiezentrums konnten wir das nicht voraussetzen. Zweitens existiert im Therapiezentrum nahezu jeder Ablauf in drei verschiedenen Versionen, da sich die Schwerpunkte der drei Standorte in Bezug auf die Erkrankungen der zu behandelnden Patienten unterscheiden. Die unterschiedlichen Schwerpunkte sind: Erkrankungen des Bewegungsapparates, Patienten mit neurologischen, internistischen und psychischen Erkrankungen sowie Patienten der Geriatrie, Onkologie und Hals-Nasen-Ohren-Kunde. Dazu unterscheiden sich die Abläufe zusätzlich durch die Art der Behandlung der Patienten. Ambulante Patienten werden meist von ihrem Hausarzt überwiesen, während stationäre vom Stationsarzt geschickt oder direkt in ihrem Krankenzimmer behandelt werden. Ein integriertes Modell, welches diese Besonderheiten und Alternativen berücksichtigt, ist zu komplex und unübersichtlich, um es „ad hoc“ zu erstellen, ohne vorher die einzelnen Abläufe in den unterschiedlichen Ausprägungen erfasst zu haben.
Das Institut für Qualitätssicherung macht über die zu erstellende Dokumentation und die zu verwendende Modellierungssprache keine strikten Vorgaben. Anders als bei der AUDI AG existiert auch kein übergeordnetes Modell, in das sich die Geschäftsprozesse einfügen sollten. So ergab sich die Aufgabe, gleichzeitig mit der Erhebung der Abläufe auch die für diesen Kontext geeignete Modellierungssprache festzulegen. Die Darstellung der Geschäftsprozesse sollte einfach und klar sein, damit die Mitarbeiter sie leicht verstehen können. Ande-

rerseits sollte sie in der Lage sein, die tatsächlichen Abläufe korrekt und ausreichend präzise zu beschreiben, um einer Prüfung der Zertifizierungsstelle zu genügen. Aus diesem Grund haben wir in einer frühen Phase der Kooperation zwei beispielhafte Abläufe in verschiedenen Modellierungssprachen dargestellt und in Gesprächen mit den Therapeuten die beste Darstellungsform für unseren Kontext gewählt. Zur Auswahl standen die üblichen Verdächtigen:

- Eine textuelle Beschreibung von Anwendungsfällen,
- Anwendungsfalldiagramme,
- Sequenzdiagramme,
- Ereignisgesteuerte Prozessketten (EPKs) und die
- Business Process Modeling Language (BPMN).

Die textuellen Beschreibungen wurden von den Mitarbeitern des Theraphiezentrums als sehr einfach zu verstehen aber umständlich eingestuft. Die anderen, graphischen Modellierungsformen seien schneller zu überblicken und besser strukturiert.
Die zweite Möglichkeit, die Anwendungsfalldiagramme, besitzen eine intuitive, graphische Repräsentation, wodurch sie einen guten Überblick über die zu beschreibenden Abläufe bieten. Allerdings liegt der Fokus bei Anwendungsfalldiagrammen nicht auf der zeitlichen Abfolge beschriebener Arbeitsschritte, sondern auf anderen Beziehungenstypen zwischen den Arbeitsschritten und verwendeten Ressourcen. So kann z.B. ein Arbeitsschritt einen anderen enthalten oder ein Arbeitsschritt kann eine Spezialisierung eines anderen sein. Dieser Fokus der Anwendungsfalldiagramme führte in den Gesprächen mit den Therapeuten oft zu Missverständnissen.
Die Sequenzdiagramme wurden als zu technisch eingestuft. Zwar lobte man auch hier die intuitive graphische Repräsentation, in der für jedes Objekt eine Lebenslinie von oben nach unten verläuft und Interaktionen zwischen Objekten durch die Lebenslinien verbindenden Pfeile dargestellt sind, aber die Vielzahl dieser Querverbindungen wurde von Vielen als zu unübersichtlich und als dem Kontext unangemessen angesehen.
Damit blieben nur noch die zwei Petrinetz-Dialekte, die Ereignisgesteuerten Prozessketten (EPKs) und die Business Process Modelling Language (BPML). Die Mitarbeiter des Therapiezentrums entschieden sich zugunsten der Business Process Modeling Language. Da beide Sprachen dieselben Netzelemente und Strukturen besitzen, war diese Entscheidung reine Geschmackssache. Beide Sprachen stellen Arbeitsschritte als Knoten dar, welche durch Pfeile entsprechend der zeitlichen Abhängigkeiten verbunden sind. Beide Sprachen verfügen zudem über die gleichen den Kontrollfluss steuernden Bausteine wie Alternativen, Nebenläufigkeit und Synchronisation. Beide Sprachen erlauben zudem das Annotieren von Aktivitäten mit zur Ausführung benötigten Dokumenten und Akteuren. Zudem wurden beide Sprachen als leicht verständlich und schnell zu überblicken eingestuft. Sie sind ausdrucksmächtig genug, um als Modellierungssprache für die Geschäftsprozesse des Therapiezentrums zu dienen.
Im Laufe der Kooperation wurden insgesamt 21 Arbeitsabläufe ermittelt. Diese wurden dem im AUDI Projekt entwickelten Vorgehensmodell

folgend zunächst in Beispielabläufen gesammelt, zu Szenarien abstrahiert und in der BPML formalisiert.

1.3 SZENARIOBASIERTE MODELLIERUNG

Die beiden Projekte mit der AUDI AG und dem Therapiezentrum Medifit beschreiben völlig unterschiedliche, aber typische Betätigungsfelder der Geschäftsprozessmodellierung. In beiden Projekten war es sinnvoll, sich bei der Modellierung der Systeme zunächst intensiv mit der Menge der Abläufe des Systems auseinanderzusetzen. Einerseits um aus den gewonnenen Szenarien das Modell zu generieren, andererseits um in einem weiteren Schritt die Szenarien des erstellten Modells mit den spezifizierten Abläufen zu vergleichen. Im Bereich der Softwareentwicklung hat sich dieser Gedanke bereits stärker etabliert als im Bereich der Geschäftsprozessmodellierung. Da sich diese aber stark an den Vorgehensmodellen der Softwareentwicklung orientiert, wächst auch hier die Anzahl an Methoden, die eine szenariobasierte Erstellung und Validierung von Geschäftsprozessen unterstützen [42, 39, 85, 25, 71]. Generell lassen sich die Vorgehensmodelle der Softwareentwicklung und der Geschäftsprozessmodellierung in drei Kategorien unterteilen: Die klassischen Ansätze, die szenariobasierten Ansätze und die streng szenariobasierten Ansätze. Wir wollen die wesentlichen Unterschiede und Annahmen kurz skizzieren.
Bei den klassischen Ansätzen steht das Verhalten des Systems im Vordergrund. Die Szenarien des Systems werden erst am Ende der Erstellung eines Modells betrachtet, wenn man diese mit den Szenarien des generierten Modells vergleicht. Modellierungsexperten generieren zunächst das Systemmodell, indem sie Informationen sammeln, strukturieren und, sobald sie das System als Ganzes überblicken, in ein formales Modell übersetzen. Aus diesem Modell werden dann im Zuge einer Testphase formale Szenarien generiert, welche mit den Anforderungen an das System verglichen werden können. Finden sich Unstimmigkeiten, wird das Modell verbessert und die neue Menge der im Modell enthaltenen Abläufe berechnet, bevor sich die Testphase wiederholt. Die Anforderungen an die Kompetenz der an der Modellierung Beteiligten sind bei einem solchen Vorgehen natürlich enorm. Sie müssen erstens großes Fachwissen über das zu modellierende System besitzen oder sich dieses erarbeiten und zweitens über gute Fähigkeiten in der Modellierung verfügen, um die gesammelten Informationen integrieren und gleichzeitig formalisieren zu können.
Szenariobasierte Vorgehensmodelle kommen zum Einsatz, wenn es schwierig ist, alle Informationen über ein System in einem Schritt zu überblicken und zu integrieren, sei es auf Grund der Komplexität des Systems oder der Qualifikation der an der Modellierung beteiligten Personen. Dies ist häufig der Fall, wenn Wissen über das zu modellierende System weit über ein Unternehmen verteilt ist. Dadurch, dass Unternehmen ihre Abläufe immer häufiger in abteilungsübergreifenden Projekten organisieren, wird eine solche Verteilung des Wissens zum Standard [83]. Bei dieser Prozessorientierung liegt der Fokus nicht länger auf einzelnen Arbeitsschritten, sondern vielmehr auf den Teilprozessen eines Unternehmens. Durch die damit einhergehende Flexibilität wird eine direkte Integration der komplexen, verteilten Geschäftsprozesse in ein Modell im höchsten Maße fehleranfällig.

Um eine Hilfestellung auf dem Weg zum integrierten Modell zu liefern, haben sich szenariobasierte Ansätze entwickelt. Bei den szenariobasierten Ansätzen werden zunächst die Abläufe des Systems isoliert betrachtet. Alle Informationen werden in Bezug auf die verschiedenen Abläufe gesammelt und dokumentiert. Dadurch gelingt es, die schwierige Phase der Anforderungsermittlung zu strukturieren und zu entzerren. Sobald die Szenarien beschrieben sind, wird mit ihrer Hilfe das integrierte Modell erstellt. Aus diesem Modell werden dann wieder Szenarien generiert, welche genau wie im klassischen Ansatz mit den realen Abläufen verglichen werden können. Ein weiterer Vorteil ist, dass diese Beschreibung der realen Abläufe bereits aus der Anforderungsermittlung vorliegt.
Es ist charakteristisch für szenariobasierte Ansätze, dass die Abläufe meist in natürlicher oder semiformaler Sprache gesammelt und beschrieben werden. Solche Beschreibungen bieten zu Beginn des Erhebungsprozesses die notwendig Flexibilität und sind außerdem für die beteiligten Personen leicht zu verstehen. Die grundlegende Idee im szenariobasierten Ansatz ist, den im klassischen Ansatz aufwendigen Schritt der Integration zu entzerren. Dies gelingt, indem man in der Phase der Erhebung den Fokus ganz bewusst auf einzelne Szenarien des Geschäftsprozesses legt. Dabei wird vom Kontrollfluss des gesamten Systems zunächst abstrahiert. Bei der Erhebung einzelner Szenarien kann es durchaus sinnvoll sein, ein laufendes System zu beobachten oder dokumentierte Beispielabläufe auszuwerten. Natürlich ist es genauso möglich, Soll-Prozesse oder bereits laufende Prozesse abstrakt zu beschreiben und dadurch zu erfassen.
Als konsequente Weiterentwicklung der szenariobasierten Vorgehensmodelle entstanden die sogenannten streng szenariobasierten Ansätze. Genau wie bei einem szenariobasierten Ansatz werden zunächst einzelne Abläufe des Systems erhoben. Der nächste Schritt ist aber nicht die Integration zu einem Modell, sondern eine Übersetzung der informalen Abläufe in eine streng formale Form. Diese Szenarien werden nach der Formalisierung erneut mit den Informationslieferanten oder Anwendern validiert. Danach kann die Integration semiautomatisch oder gar automatisch durch Synthese- oder Faltungsalgorithmen [43, 25, 18] durchgeführt werden. Als abschließender Schritt wird wieder das gesamte Verhalten des automatisch generierten Modells berechnet. Dies dient aber nicht der Validierung der im Erhebungsprozess gesammelten Informationen, sondern ausschließlich der Kontrolle des Ergebnisses des Integrationsalgorithmus. Die Menge der aus dem Modell berechneten Szenarien kann dazu automatisch mit der Menge der erhobenen Szenarien verglichen werden.
Für die Integration kann man Synthese- oder Faltungsalgorithmen verwenden. Synthesealgorithmen haben den großen Vorteil, dass sie ein Modell generieren, dessen Verhalten so nah wie möglich an das durch die vorgegebenen Szenarien spezifizierte Verhalten herankommt, wobei das entstehende Modell alle vorgegebenen Szenarien beinhaltet. Für diese Eigenschaft muss man allerdings einen hohen Preis zahlen. Synthesealgorithmen nutzen Methoden der linearen Optimierung und sind dadurch für größere Eingaben auf Grund der Komplexität der entstehenden Optimierungsprobleme nicht mehr anwendbar. Synthesealgorithmen eignen sich hervorragend zur Integration von Abläufen komplexer Systeme mit einer moderaten Mächtigkeit der Menge der zu integrierenden Szenarien. Eine ausführliche Darstellung der Mög-

lichkeiten Synthesealgorithmen für die Geschäftsprozessmodellierung einzusetzen findet sich in [16] und in der Dissertationsschrift von Sebastian Mauser [71].
Die Faltungsalgorithmen sind in der Lage, auch große Probleminstanzen in akzeptabler Laufzeit zu bewältigen. Bei ihnen liegt das Problem darin, dass sie Modelle erzeugen, die über das spezifizierte Verhalten hinaus weitere Szenarien enthalten können. Je größer die Ausgangsspezifikation ist, desto mehr zusätzliches Verhalten ist im Modell enthalten. Dies liegt darin begründet, dass ein Faltungsalgorithmus oft alle Kombinationen aller Alternativen als Abläufe in dem Modell erzeugt. So kann es sein, dass das entstehende Modell nachgebessert werden muss. Eine Darstellung eines möglichen Faltungsalgorithmus für die Geschäftsprozessmodellierung findet sich in [25] oder [44].
Bei streng szenariobasierten Ansätzen wird die ausführliche Validierungsphase und der Übergang zu einer streng formalen Beschreibung vor die eigentliche Integration des Modells gestellt. Ein Vorteil ist dabei, dass Informationslieferanten die Menge einzelner Abläufe besser verstehen und somit besser validieren können als das bereits integrierte Modell. Ein weiterer Vorteil ist, dass durch einen solchen Ansatz eine sehr detaillierte Dokumentation der Phase, in der Informationen zum System erhoben werden, entsteht. Jeder Ablauf liegt zunächst als Beschreibung, dann in formaler Darstellung und schließlich im integrierten Modell vor. Sollte sich das System ändern, so sind Teile der dokumentierten Szenarien wiederverwendbar. Es ist nicht notwendig, das gesamte Modell neu zu erstellen, sondern man kann eines der formalen Szenarien korrigieren und dann die automatische Integration wiederholen. Auf diese Weise kann man sich nicht nur die Modellerstellung sondern auch die Wartung, Optimierung und Refaktorisierung eines Modells erheblich vereinfachen.
Neben allen Vorteilen ist ein Nachteil des streng szenariobasierten Vorgehens die fehlende Kontrolle über die Struktur des automatisch erzeugten Modells. Diese hängt ausschließlich von dem gewählten Integrationsalgorithmus ab und ist damit unflexibel. Es ergeben sich zwei Nachteile: Erstens kann man keine Designentscheidungen während der Integration treffen. Gibt es zur Abbildung eines Systems mehrere Möglichkeiten, so wird eine dieser Möglichkeiten beliebig entsprechend der Funktionsweise des Integrationsalgorithmus gewählt. Zweitens ist es für die Modellierenden schwierig, den Bezug zu dem automatisch generierten Modell nicht zu verlieren. Der Modellierende kennt zwar die Eingabe des Integrationsalgorithmus, kann aber, da er an der Integration selbst nicht beteiligt ist, leicht den Überblick über das erzeugte Modell verlieren, was besonders kritisch ist, wenn das Modell später zur Optimierung oder zur Dokumentation des Systems verwendet werden soll. In beiden Fällen sollte das erzeugte Modell dem Modellierenden sehr vertraut und so intuitiv wie möglich gestaltet sein.
Ein weiterer Nachteil streng szenariobasierter Verfahren ist, dass ein hoher Arbeitsaufwand nötig ist, um die Kompatibilität der einzelnen Szenarien vor dem Schritt der Integration sicher zu stellen. Es kann leicht passieren, dass zwar jeder Ablauf für sich valide und korrekt dargestellt ist, sich aber erst im Vergleich von Szenarien herausstellt, dass z.B. die Schnittstellen der Abläufe oder der Grad der Abstraktion unterschiedlich gewählt wurde. Integriert man eine Menge von solchen Szenarien, ohne das Zusammenspiel und das Zusammenpassen der

Szenarien im Vorhinein genau zu überprüfen, ist das Ergebnis eines automatischen Integrationsalgorithmus nicht zu gebrauchen.
Trotz dieser Nachteile haben wir uns in beiden beschriebenen Projekten, dem AUDI Projekt und der Kooperation mit dem Therapiezentrum Medifit, dazu entschieden, einen streng szenariobasierten Ansatz zu wählen, da die Vorteile gegenüber einem klassischen oder einem szenariobasierten Ansatzes in diesen Kontexten klar überwiegen. Im AUDI Projekt gab es durch das bereits vorgegebene Modell der AUDI AG, in das sich die Szenarien einfügen sollten, eine Vorlage, an der sich die einzelnen Szenarien orientieren mussten. So war die Konformität der einzelnen Szenarien sichergestellt. In der Kooperation mit dem Therapiezentrum Medifit wurde gänzlich auf die Integration verzichtet und somit wurden fast alle Phasen eines streng szenariobasierten Vorgehensmodells durchlaufen. Hier waren die Qualifikation der Mitarbeiter aber auch die verschiedenen Versionen einzelner, an sich nicht komplexer Szenarien Gründe, ein solches Vorgehen zu wählen.
Im nächsten Abschnitt wollen wir die Beobachtungen und Erfahrungen aus den beschrieben Projekten nutzen, um ein weiteres, agiles Vorgehensmodell zu entwickeln.

1.4 AGILE GESCHÄFTSPROZESSMODELLIERUNG

An dieser Stelle schlagen wir ein neues Vorgehensmodell vor, um in Kontexten ähnlich den beiden beschrieben Projekten einen optimalen Kompromiss zwischen Flexibilität, Fehlerkontrolle, Lesbarkeit und Wartbarkeit des erzeugten Modells zu erlangen. In der Softwareentwicklung erfreuen sich die agilen Vorgehensmodelle immer größerer Beliebtheit. Der erste und wohl bekannteste agile Vorgehensmodelle ist das Extreme Programming [8]. Extreme Programming wurde während der Durchführung eines Projektes zur Erstellung einer Lohnabrechnungssoftware bei Chrysler im Rahmen einer Case Study von Kent Beck, Ward Cunningham und Ron Jeffries entwickelt. Innerhalb eines Extreme Programming Projektes gilt es, gewisse Praktiken und Prinzipien einzuhalten. Die wohl bekanntesten Praktiken sind dabei Pair-Programming, Permanente Integration, Permanentes Testen, Kundeneinbeziehung und einfaches Design, deren Zusammenspiel zu einem effizientem, flexiblen, aber fehlerresistenten Vorgehensmodell der Softwareentwicklung führen soll.
Beim Extreme Programming geht man davon aus, dass ein Kunde zu Beginn eines Projektes die Anforderungen noch nicht komplett kennt oder zumindest nicht in der Lage ist, diese hinreichend strukturiert darzustellen. Die Anforderungen an die zu erstellende Software werden dazu Stück für Stück und in mehreren Iterationen erhoben. Zu Beginn jeder Iteration werden neue Anforderungen gesammelt. Zu diesen Anforderungen werden im nächsten Schritt Testfälle erzeugt. Die neuen Anforderungen werden in einer neuen Version der Software implementiert, welche dann mit Hilfe der Menge aller Testfälle sofort getestet und mit den Informationslieferanten validiert werden. Damit wächst die Software mit jeder durchgeführten Iteration. Durch diesen Ansatz steht die Software in frühen Versionen mit den bis dahin erhobenen Funktionalitäten schnell zur Verfügung. Man erlangt frühzeitig und regelmäßig Eindrücke darüber, in welche Richtung sich das Softwareprojekt entwickelt, und kann gegebenenfalls gegensteuern. Damit ist es möglich, Kunden stark an der Entwicklung der Software zu beteiligen.

Wir beschreiben in diesem Abschnitt nur die Praktiken des Extrem Programming, die wir auf die Geschäftsprozessmodellierung übertragen wollen. Weitere Praktiken wie z.B. Pair-Programming, Einhalten der Arbeitszeiten, gemeinsames Code-Eigentum usw. können direkt aus den agilen Methoden der Softwareentwicklung übernommen werden. Alle Praktiken des Extrem Programming werden in [8, 9, 10] ausführlich dargestellt und diskutiert.
Eine Praktik des Extreme Programming ist die permanente Integration. Anders als bei den szenariobasierten Ansätzen werden nicht erst alle Szenarien gesammelt und anschließend integriert, sondern neu erhobene Funktionalitäten werden direkt in die aktuelle Version des Modells integriert. Dabei akzeptiert man, dass nicht immer nur neuer, zusätzlicher Code erzeugt wird, sondern auch die bereits bestehende Software wieder geändert und angepasst werden muss. Man nimmt diesen Aufwand auf sich, um auf eine ausgedehnte Planungsphase vor der Integration verzichten und um besonders flexibel auf neue Anforderungen reagieren zu können. Daher nennt man diese Vorgehensmodelle agil. Durch die permanente Integration stehen nicht nur die Prototypen der Software frühzeitig zum Zwecke des Tests und der Validierung zur Verfügung. Auch die Kompatibilität der neuen Anforderungen zum gesamten Programm wird ständig überprüft. Die Integration wird durch die regelmäßige Durchführung dieser Tätigkeit zur Routine, verbraucht dadurch wenig Ressourcen und verschafft den Modellierenden einen guten Überblick über das erstellte System.
Die gesamte Integration und Refaktorisierung soll beim Extreme Programming testgetrieben stattfinden. Das bedeutet, dass bereits vor der Integration und für alle neue Funktionalitäten Testfälle erstellt werden, die nach der Integration automatisch ablaufen. Eine neue Version wird erst erstellt, wenn sie alle bis zu diesem Zeitpunkt erstellten Testfälle erfolgreich besteht. Damit werden nicht nur die neuen, sondern die Lauffähigkeit aller, eventuell bereits vor Beginn der letzten Iteration existenten Funktionalitäten sichergestellt. Insgesamt erzeugt man damit stetige Rückmeldungen über die Qualität des erzeugten Programms. Für diesen Ansatz ist es wichtig, dass die Testfälle automatisiert ablaufen, da der Aufwand, alle Testfälle am Ende jeder Iteration per Hand auszuführen, zu groß wäre. Man nennt eine solche Praktik testgetriebene Entwicklung, da einerseits die Testfälle einen beträchtlichen Teil der Spezifikation ausmachen und somit am Anfang jeder Iteration stehen und andererseits das Bestehen der Menge aller Testfälle eine Iteration abschließt. Das Programm wird damit durch die Tests und Testfälle vorangetrieben.
Die dritte hier betrachtete Praktik ist die starke Einbindung des Auftraggebers in das Software-Projekt. Diese geschieht, indem der Auftraggeber das Ziel der jeweils nächsten Iteration mitbestimmt und indem er jede Version der Software ausprobieren und validieren kann. Das Spezifizieren neuer Anforderungen geschieht in sogenannten User-Stories, in denen der Auftraggeber meist in einem kurzen Satz eine gewünschte Funktionalität beschreibt. User-Stories beschreiben also eine gewünschte, neue Funktionalität der Software aus der Sicht des zukünftigen Anwenders und vermitteln somit zwischen Auftraggeber und Programmierer. Sind die neuen Anforderungen als User-Stories gesammelt, wird entsprechend des geschätzten Aufwands ein Teil davon ausgewählt und in der nächsten Iteration umgesetzt. Diese User-Stories werden zunächst in Testfälle übersetzt, und anschließend wird die beschriebene

Funktionalität ins Programm integriert. Der Auftraggeber hat somit Einfluss auf die Reihenfolge, in der Funktionalitäten entwickelt und integriert werden. Somit kann er den Verlauf des Projektes aktiv steuern. Durch die regelmäßig erstellten Versionen des Programm ist der Auftraggeber jederzeit über den Stand des Software-Projekts informiert, kann entsprechend neue Anforderungen formulieren und das Projekt mit gestalten. Die Akzeptanz und die Bindung zwischen Auftraggeber und Programm kann damit schon bei der Entwicklung der Software gefördert und überprüft werden.
Diese Arbeit schlägt vor ein agiles Verfahren für die Geschäftsprozessmodellierung zu verwenden. Dazu werden wir die eben beschriebenen Praktiken aus der agilen Softwareentwicklung in die Geschäftsprozessmodellierung übertragen und die Sinnhaftigkeit eines solchen Vorgehens mit den Erfahrungen aus den beschriebenen Projekten belegen.
Zu Beginn führen wir die Erstellung des Geschäftsprozessmodells ebenfalls iterativ durch. Zu Beginn jeder Iteration werden zunächst neue Anforderungen in Form von neuen Abläufe erhoben und diese in Testfälle, übersetzt. Dann werden die neuen Anforderungen zu der aktuellen Version des Geschäftsprozessmodells hinzugefügt und alle bisher gesammelten Testfälle in der neuen Version getestet. Besteht die neue Version des Modells alle Tests, wird sie mit dem Auftraggeber validiert und es beginnt eine nächste Iteration.
Agile Vorgehensmodelle der Softwareentwicklung benutzen einfache User-Stories, um neue Anforderungen zu erheben. Im Kontext der agilen Geschäftsprozessmodellierung verwenden wir dazu einfache dokumentierte Beispielabläufe oder Ablaufbeschreibungen analog zur Vorgehensweise im AUDI Projekt. Ein solcher Ablauf ist genau wie eine User-Storie der für den Auftraggeber natürlichste Weg, den zu modellierenden Ablauf zu beschreiben.
Im Unterschied zu den in Abschnitt 1.3 dargestellten Ansätzen wird in der agilen Geschäftsprozessmodellierung zu Beginn jeder Iteration nur ein Teil der Menge aller Abläufe betrachtet. Dabei kann es sinnvoll sein, zunächst mit gut dokumentierten Abläufen zu beginnen, um in späteren Iterationen mit mehr Vorwissen schlechter dokumentierte Abläufe behandeln zu können. Man könnte auch mit den Kern-Abläufen des Geschäftsprozesses beginnen und in späteren Iterationen die weniger oft genutzten Abläufe ergänzen. In den beschriebenen Projekten fielen beide Eigenschaften zusammen. Insgesamt liegt, im Einklang mit den szenariobasierten Methoden, der Fokus der agilen Geschäftsprozessmodellierung auf den Szenarien des Systems.
Im nächsten Schritt werden die Ablaufbeschreibungen formalisiert. Dafür sollte man eine Darstellungsform wählen, die im späteren Modell automatisch auf Ausführbarkeit getestet werden kann. In dieser Form können die formalen Ablaufbeschreibungen, also Szenarien, als Testfälle für die zu modellierenden Abläufe dienen. Während beim Extreme Programming ein Test eine Funktionalität abbildet, indem Eingaben gewünschten Ausgaben oder gewünschtem Verhalten des Systems gegenübergestellt werden, ist ein Test in der Geschäftsprozessmodellierung die Beschreibung einer Menge auszuführender Ereignisse in einer möglichen Ordnung. Diese Menge der Ereignisse muss später in der beschriebenen Ordnung im Modell ausführbar sein. Ein Testfall in der Softwareentwicklung übersetzt sich somit intuitiv in ein Szenario im Anwendungsfeld der Geschäftsprozessmodellierung.

Sind die neuen Abläufe als Szenarien spezifiziert, wird die aktuelle Version des Modells um die neuen Abläufe erweitert. Hierbei wird bewusst auf eine automatische Integration verzichtet, und so wird die Kompatibilität der aktuellen Version des Modells mit den neu spezifizierten Abläufen überprüft. Außerdem sorgt ein solches Vorgehen dafür, dass der Modellierende Designentscheidungen aktiv treffen und damit die Lesbarkeit und Wartbarkeit des entstehenden Modells deutlich erhöhen kann. Natürlich muss man diese Vorteile mit einem gegenüber der automatische Integration erhöhten Arbeitsaufwand bezahlen. Im Gegensatz zu einem klassischen Ansatz ist der kognitive Aufwand in dieser Integrationsphase jedoch deutlich geringer, da bei jeder Iteration nur wenige neue Abläufe dem bereits existierenden Geschäftsprozessmodell hinzugefügt werden müssen. Damit ist die Integration leicht zu überschauen und wird bei einer hohen Anzahl an Iterationen schnell zur gewohnten Tätigkeit.
Nach der Integration werden alle Szenarien in dem Modell auf Ausführbarkeit getestet. Natürlich wächst mit jeder Iteration auch die Menge der erhobenen Szenarien. Genau wie beim Extrem Programming ist es daher erforderlich, dass der Test auf Ausführbarkeit automatisch und schnell durchführbar ist. Dies gilt um so mehr, je komplexer und zahlreicher die zu modellierenden Abläufe sind. Die Entwicklung des nötigen Algorithmus ist Kern dieser Arbeit. Sind die Szenarien getestet, wird die neue Version des Modells gemeinsam mit dem Auftraggeber validiert, bevor die nächste Iteration beginnt. Das Einbinden des Auftraggebers führt dazu, dass die Validität des Modells und das Verständnis des Auftraggebers für das Modell erhöht wird.
So gelingt es, die zentralen Praktiken permanente Integration, testgetriebene Entwicklung und Kundeneinbeziehung des Extreme Programming auf den Kontext der Geschäftsprozessmodellierung zu übertragen. Durch die hohe Flexibilität in der Integrationsphase und die explizite Einbeziehung des Auftraggebers eignet sich ein solches agiles Vorgehensmodell perfekt für die in den Abschnitten 1.2.1 und 1.2.2 beschriebenen Einsatzgebiete. Das Team der Auftraggeber hatte jeweils wenig Erfahrung mit der Geschäftsprozessmodellierung, sollte aber bei beiden Projekten ausgiebig an der Modellerstellung beteiligt werden. Weiter sorgt ein agiles Vorgehen dafür, dass die erzeugten Modelle lesbar und damit für den Zweck der Dokumentation, die Hauptaufgabe in beiden Projekten, besonders tauglich sind. Die Menge der Szenarien bietet wiederum eine solide Grundlage für die kontinuierliche Integration und dokumentiert gleichzeitig das gesamte Vorgehen. Die anderen Praktiken und Prinzipien des Extreme Programming können eins zu eins für die agile Geschäftsprozessmodellierung übernommen werden.

1.5 PROBLEMSTELLUNG UND LITERATURÜBERSICHT

Die vorliegende Arbeit behandelt ausführlich das Thema der Verifikation von Szenarien in Petrinetzen und entwickelt dazu eine für diesen Kontext effiziente Charakterisierung der Szenario-Sprache. Bevor wir einen Überblick über die bis zu diesem Zeitpunkt existierende Literatur geben, fassen wir die Gründe, aus denen wir uns mit dieser Thematik beschäftigen, noch einmal zusammen. Dazu betrachten wir die in den Abschnitten 1.2, 1.3 und 1.4 beschriebenen Vorgehensmodelle und fragen uns, an welchen Stellen die Verifikation formal spezifizierter

Abläufe eines realen Systems in einem formalen Modell von zentraler Bedeutung ist.
In den klassischen und den szenariobasierten Vorgehensmodellen sind Szenarien in der Phase der Integration und in der Testphase von großer Bedeutung. So kann zum Beispiel in der Testphase eine endliche Repräsentation aller in einem Modell enthaltenen Szenarien berechnet werden. Die dazu nötigen Entfaltungsalgorithmen konstruieren diese Repräsentation, die man mit dem spezifizierten Verhalten vergleichen kann. Im Allgemeinen laufen die Entfaltungsalgorithmen um so schneller, je enger der Ausführbarkeitsbegriff für Szenarios definiert ist. Basiert ein Entfaltungsalgorithmus auf Prozessnetzen, entsteht eine Vielzahl von isomorphen Szenarien (siehe z.B. [74] oder [49]). Basiert ein Entfaltungsalgorithmus auf Markenflüssen, wird die Anzahl der konstruierten Szenarien reduziert [23]. Eine kompakte Charakterisierung der Szenario-Sprache ist Grundlage für einen effizienten Entfaltungsalgorithmus.
Bei den streng szenariobasierten Vorgehensmodellen ist das Szenario-Verifikations-Problem ein wesentlicher Bestandteil jeder Phase. Zur Integration der Szenarien werden Synthese- oder Faltungsalgorithmen eingesetzt. Neuere Synthesealgorithmen basieren auf der Regionentheorie [3, 45, 86, 34, 16]. Der spezielle Begriff einer Region ergibt sich wieder aus der Wahl der Charakterisierung der Szenario-Sprache und die Laufzeit der Synthesealgorithmen hängt stark von dieser ab. Da die Laufzeit regionenbasierter Synthesealgorithmen für große Eingaben sehr hoch ist [15], ist es von Vorteil eine kompaktere Charakterisierung der Szenario-Sprache zu entwickeln, falls sich dadurch ein Regionenbegriff definieren lässt, mit dem schnellere Synthese-Algorithmen entwickelt werden können. Greift man in der Phase der Integration auf Faltungsalgorithmen zurück, ergibt sich eine sehr direkte Anwendung des Szenario-Verifikations-Problems. Ein durch Faltung erzeugtes Modell enthält oft zusätzliches Verhalten. Eine elegante Möglichkeit nicht gewolltes, aber im Modell ausführbares Verhalten zu entdecken ist es, auszuschließende Szenarien in einer Negativ-Spezifikation festzuhalten. Nach der Integration des Modells, werden die in der Negativ-Spezifikation enthaltenen Szenarien im Modell auf Ausführbarkeit getestet, wobei der Test bestanden ist, wenn keines der spezifizierten Szenarien im Modell ausführbar ist [26].
In agilen Vorgehensmodellen wird die Validität des erzeugten Modells, durch das kontinuierliche Entscheiden des Szenario-Verifikations-Problems sichergestellt. Dafür muss die Menge der spezifizierten Testfälle effizient in der aktuellen Version des Modells auf Ausführbarkeit zu testen sein. Aus dem Bericht über das erste Extreme Programming Projekt stammt das Zitat: „Our complete suite of unit tests, exercising the entire domain, runs in less than ten minutes, so we can afford to run all the tests every time anyone releases any code. And we do: We only release code when all the unit tests in the entire system run at 100 %“ [5]. Damit ist die Möglichkeit, das Szenario-Verifikations-Problem für jede noch so große Eigabe schnell zu entscheiden, die Grudlage jedes agilen Vorgehensmodells.
Man kann das Szenario-Verifikations-Problem für verschiedene Klassen von Petrinetzen formulieren. Diese Arbeit betrachtet das Szenario-Verifikations-Problem für Petrinetze mit Kantengewichten. Diese ausdrucksmächtige Form der Petrinetze nennt man S/T-Netze. Für eingeschränktere Klassen von Petrinetzen, zum Beispiel für die Menge

der Petrinetze, bei denen die Anzahl an Marken auf jedem Platz maximal eins beträgt, ist das Szenario-Verifikations-Problem durch Algorithmen entscheidbar, deren Laufzeit mit der Größe der Eingabe nur linear wächst. Die Verwendung von S/T-Netzen ermöglicht allerdings das intuitive Modellieren von Ressourcen, kollaborativ ausgeführten Aufgaben und komplexen Ablaufstrukturen. Viele der in der Praxis eingesetzten Modellierungssprachen lassen sich leicht in S/T-Netze übersetzen. Anwendungsfelder der S/T-Netze finden sich in der Geschäftsprozessmodellierung, bei Kommunikations-Protokollen [6, 34, 28], Webdiensten [84], der Steuerung von Fertigungssystemen [31, 92] und im Hardware-Design [33, 58, 38].
Obwohl das Szenario-Verifikations-Problem das fundamentale Problem im Zusammenhang von S/T-Netzen und Szenarien ist, ist es in diesem speziellen Fall aufwendig zu entscheiden und hängt stark von der gewählten Charakterisierung der Szenario-Sprache ab. Die Szenario-Sprache eines S/T-Netzes ist in der Literatur auf drei verschiedene Arten charakterisiert.

- Über die Menge der Prozessenetze eines S/T-Netzes [55, 54],
- über die Menge aller möglichen Schritt-Sequentialisierungen eines S/T-Netzes [56] oder
- als die Menge der Szenarien mit gültiger Markenflussfunktion [60].

Aus jeder dieser Charakterisierungen lässt sich die Menge der im S/T-Netz ausführbaren Szenarien bestimmen, und glücklicherweise führen diese drei Ansätze zu einer äquivalenten Definition der Szenario-Sprache des S/T-Netzes. Die Äquivalenz der ersten beiden wurde in [64] und [87] gezeigt, was einen langen und nicht trivialen Beweis erfordert. Die Äquivalenz der ersten beiden Charakterisierungen zur Dritten wurde in [60] gezeigt.
Im Gegensatz zu den ersten beiden Charakterisierungen liefert die Charakterisierung über Markenflüsse einen in Bezug auf die Größe der Eingabe in Polynomialzeit laufenden Algorithmus zur Lösung des Szenario-Verifikations-Problems für S/T-Netze. Wie in [60] beschrieben wird dazu das Szenario-Verifikations-Problem in mehrere Fluss-Maximierungs-Probleme übersetzt, für deren Lösung in Polynomialzeit laufende Verfahren existieren. Die Anzahl der zu lösenden Fluss-Maximierungs-Probleme ist dabei im schlechtesten Fall das Produkt der Anzahl der Stellen des Netzes und der Anzahl der Ereignisse des zu verifizierenden Szenarios. In [11] wurde dieses iterative Verfahren für das VipTool [14] implementiert und es wurden Laufzeittests durchgeführt. In [66] wurde ein zweites Verfahren vorgeschlagen, das mit wenigeren Übersetzungen in Fluss-Maximierungs-Probleme auskommt. Die Anzahl der Übersetzungen ist hier die Anzahl der Stellen des Petrinetzes, ohne dass diese mit der Anzahl der Ereignisse des Szenarios multipliziert werden muss. Dieses direkte Verfahren wurde bisher jedoch noch nicht implementiert. Beide in Polynomialzeit laufenden Algorithmen werden in dieser Arbeit aufgegriffen. In beiden Fällen kommt der Wahl eines geeigneten Fluss-Maximierungs-Algorithmus eine entscheidende Rolle zu. Die Laufzeit der Algorithmen ergibt sich unter Anderem aus der Laufzeit der verwendeten Fluss-Maximierungs-Algorithmen. Dabei ist zu beachten, dass die nötige Transformation ausschließlich Flussnetzwerke mit besonderen Charakteristika erzeugt.

Diese sind für die Laufzeit der Fluss-Maximierungs-Algorithmen entscheidend. Für das iterative Verfahren wurde in [11] gezeigt, dass die Verwendung des Urvaters der Fluss-Maximierungs-Algorithmen, der Algorithmus von Ford und Fulkerson [52], die beste durchschnittliche Laufzeit ergibt. Für die direkte Variante führen wir in Kapitel 6 analoge Überlegungen durch.
Unsere Erfahrungen zeigen, dass die Laufzeiten der auf der Charakterisierung der Markenflüsse beruhenden Algorithmen in einigen Probleminstanzen hinter der Laufzeit von Algorithmen zurückbleiben, die eine auf Schrittsequenzen beruhende Charakterisierung der Szenario-Sprache verwenden [19]. Markenflüsse sind auf der gesamten transitiven Hülle der Ordnung eines Szenarios definiert und alle Kanten dieser Hülle finden sich nach der Übersetzung in das zugehörige Flussnetzwerk in diesem wieder. Dadurch werden die zu lösenden Probleminstanzen für das Fluss-Maximierungs-Problem groß, und der Algorithmus, der das Szenario-Verifikations-Problem entscheidet, hat eine schlechte Laufzeit, sobald das zu testende Szenario viel Ordnung enthält.
In der vorliegenden Arbeit wird aus diesem Grund eine vierte Charakterisierung der Szenario-Sprache entwickelt. Diese beruht auf der Idee der Markenflüsse, abstrahiert jedoch, wie die Charakterisierung über Schrittsequenzen, von einer konkreten Verteilung der Marken. So wird ein Markenfluss nicht auf der transitiven Hülle, sondern auf dem Skelett der Ordnung des Szenarios definiert. Durch diese Darstellung sind die bei der Lösung des Szenario-Verifikations-Problems entstehenden Flussnetzwerke schlanker, und man kann das Szenario-Verifikations-Problem unabhängig von der Form des Szenarios effizient entscheiden. Wir nennen diese vierte Charakterisierung kompakte Markenflüsse.
Zusätzlich zu der Entwicklung dieser neuen Charakterisierung wird in der vorliegenden Arbeit die Äquivalenz dieser zu den übrigen Charakterisierungen bewiesen, ein entsprechender Szenario-Verifikations-Algorithmus abgeleitet, implementiert und ausgiebige Laufzeittests aller vorgestellten Verfahren werden dargestellt.

1.6 GLIEDERUNG

In Kapitel 1 haben wir das Thema dieser Arbeit, die Verifikation von Szenarien in S/T-Netzen, vorgestellt, ausführlich motiviert und einen Einblick in die praktische Anwendbarkeit dieses Themas gegeben. In Kapitel 2 werden die für die folgenden theoretischen Überlegungen notwendigen formalen Grundlagen geklärt. Kapitel 3 beschreibt die formalen Grundlagen der Flussnetzwerk und anschließend Fluss-Maximierungs-Algorithmen, die wir bezüglich der durchschnittlichen Laufzeiten miteinander vergleichen. Kapitel 4 beginnt mit den formalen Grundlagen von S/T-Netzen und deren Semantik. Im Anschluss werden drei völlig verschiedene aber äquivalente Charakterisierungen der Szenario-Sprache eines S/T-Netzes, die Aktiviertheit aller Schnitte des Szenarios [56], die Ausführbarkeit bezüglich der Prozessnetze [55, 54] und die Existenz gültiger Markenflüsse [60], vorgestellt. Für jedes Verfahren beschreiben wir den zugehörigen Algorithmus zur Entscheidung des Szenario-Verifikations-Problems. In Kapitel 5 entwickeln wir eine neue und kompakte Charakterisierung der Szenario-Sprache eines S/T-Netzes, beweisen deren Äquivalenz zu den bisher bekannten Charakterisierungen und leiten einen Algorithmus ab, der

das Szenario-Verifikations-Problem schnell und unabhängig von der Form der Eingabe entscheidet. In Kapitel 6 werden die Laufzeiten aller beschriebenen und entwickelten Verfahren verglichen und diskutiert. Dazu werden alle Verfahren implementiert und ausgiebige Laufzeitexperimente durchgeführt. Nach der Bewertung aller Möglichkeiten, das Szenario-Verifikations-Problem zu entscheiden, zeigt dieses Kapitel die Integration aller Verifikations-Algorithmen in das Viptool [37]. In Kapitel 7 werden zuerst die wichtigsten Aspekte dieser Arbeit noch einmal zusammengefasst, bevor die Arbeit im letzten Abschnitt mit einem Ausblick auf interessante weiterführende Themen schließt.

2

FORMALE GRUNDLAGEN

Dieses Kapitel beschreibt die formalen Grundlagen dieser Arbeit. Zunächst werden mathematische Notationen und die Begriffe Graph und Halbordnung eingeführt. Die zweite Hälfte des Kapitels beschäftigt sich mit den Grundlagen der Komplexitätsbetrachtung von Algorithmen. Weitere Grundlagen von Flussnetzwerken, Szenarien und Petrinetzen werden zu Beginn der Kapitel 3 und 4 beschrieben.

2.1 NOTATIONEN

Die Menge der natürlichen Zahlen bezeichnen wir mit $\mathbb{N}$. Die Menge der nicht-negativen ganzen Zahlen bezeichnen wir mit $\mathbb{N}_0$. Das Symbol ∞ steht für eine unendlich große Zahl und für die Menge $\mathbb{N} \cup \{\infty\}$ schreiben wir kurz $\mathbb{N}_\infty$. Die Menge der ganzen Zahlen beschreiben wir mit $\mathbb{Z}$ und für die Menge $\mathbb{Z} \cup \{\infty\}$ schreiben wir kurz $\mathbb{Z}_\infty$.
Ist eine endliche Menge A gegeben, so bezeichnet das Symbol $|A|$ die Kardinalität von A. Die Menge aller Folgen von Elementen aus A bezeichnen wir mit A^*.
Eine Multimenge über A ist eine Funktion $m : A \to \mathbb{N}_0$, und mit $\mathbb{N}_0^A$ beschreiben wir die Menge aller Multimengen über A. Für ein Element $a \in A$ bezeichnet $m(a)$ die Häufigkeit des Vorkommens von a in m. Wir schreiben eine Multimenge m als Summe der Elemente in A mit deren Häufigkeiten als Koeffizienten. In dieser Schreibweise ist $m = \sum_{a \in A} m(a) \cdot a$.
Den Vergleich, die Addition und die Subtraktion von Multimengen definieren wir elementweise, und so gilt für zwei Multimengen m und m':

$$m \leqslant m' \iff \forall a \in A : m(a) \leqslant m'(a),$$

$$m < m' \iff m \leqslant m' \wedge m \neq m',$$

$$m + m' = \sum_{a \in A} (m(a) + m'(a)) \cdot a,$$

$$m - m' = \sum_{a \in A} \max(m(a) - m'(a), 0) \cdot a.$$

Neben Mengen und Multimengen benötigen wir den Begriff einer Relation auf einer Menge.

Definition 2.1.1 (Relation)

Definition 2.1.1 (Relation)
Eine Relation $\to$ auf der Menge A ist eine Teilmenge $\to \subseteq A \times A$. Für $a, b \in A$ sagen wir a steht in Relation $\to$ zu b, falls $(a, b) \in \to$ gilt. In diesem Fall schreiben wir auch $a \to b$.

Sind zwei Relationen gegeben, so kann man diese vergleichen.

Definition 2.1.2 (Sequentialisierung)

Definition 2.1.2 (Sequentialisierung)
Seien $\to_1, \to_2$ zwei Relationen. Genau dann wenn $\to_1 \subseteq \to_2$ gilt, ist $\to_2$ eine Sequentialisierung von $\to_1$.

Eine gegebene Relation kann gewisse Eigenschaften erfüllen. Im Kontext dieser Arbeit sind die folgenden Eigenschaften von besonderem Interesse. Eine Relation $\to$ über einer Menge A heißt

reflexiv, falls $\forall\, a \in A : (a, a) \in \to$,

irreflexiv, falls $\forall\, a \in A : (a, a) \notin \to$,

transitiv, falls $\forall\, a, b, c \in A : (a \to b \text{ und } b \to c \Rightarrow a \to c)$.

Jede Relation kann man so erweitern, dass eine transitive Relation entsteht. Der Schnitt über alle transitiven Relationen, die die ursprüngliche Relation enthalten, heißt transitive Hülle.

Definition 2.1.3 (Transitive Hülle)

Definition 2.1.3 (Transitive Hülle)
Sei $\to$ eine Relation. Sei $R^* = \{\to' \supseteq \to \mid \to' \text{ ist transitiv}\}$, so heißt $\to^* = \bigcap_{\to' \in R^*} \to'$ die transitive Hülle der Relation $\to$.
Man bezeichnet $\to^*$ auch als den transitiven Abschluss von $\to$.

Ist eine Relation transitiv, so kann man Relationen betrachten, deren transitiver Abschluss die ursprüngliche Relation ergibt. Der Schnitt dieser Relationen heißt das Skelett der Relation.

Definition 2.1.4 (Skelett)

Definition 2.1.4 (Skelett)
Sei $\to$ eine transitive Relation. Sei $R^\diamond = \{\to' \subseteq \to \mid (\to')^* = \to\}$, so heißt $\to^\diamond = \bigcap_{\to' \in R^\diamond} \to'$ das Skelett der Relation $\to$.
Für eine nicht transitive Relation $\to$ definieren wir das Skelett von $\to$ als $(\to^*)^\diamond$.

Es ist üblich eine Relation als Graph darzustellen. Ein Graph ist eine Relation zusammen mit der Menge, auf der sie definiert ist.

Definition 2.1.5 (Gerichteter Graph)

Definition 2.1.5 (Gerichteter Graph)
Ein gerichteter Graph ist ein Paar $G = (V, \to)$, wobei V eine endliche Menge von Knoten und $\to \subseteq V \times V$ eine Relation über V ist. Wir nennen die Elemente von $\to$ die Kanten des Graphen. Für eine Kante $(a, b) \in \to$ nennen wir den Knoten a den Anfangsknoten und den Knoten b den Endknoten der Kante.

Knoten werden durch Kreise oder Vierecke dargestellt und gemäß der Kanten des Graphen durch Pfeile verbunden. Wir nennen zwei Knoten benachbart, falls zwischen ihnen ein Pfeil existiert. Für einen Knoten wird die Menge der Nachbarn in Vor- und Nachbereich unterteilt.

Definition 2.1.6 (Vorbereich und Nachbereich)
Sei $G = (V, \rightarrow)$ ein Graph und $a \in V$ ein Knoten. Die Menge ${}^\bullet a := \{b \in V \mid b \rightarrow a\}$ heißt Vorbereich und die Menge $a^\bullet := \{b \in V \mid a \rightarrow b\}$ heißt Nachbereich von a.
Sei $G = (V, \rightarrow)$ ein Graph und $A \subseteq V$ eine Menge von Knoten. Die Menge ${}^\bullet A := \bigcup_{a \in A} {}^\bullet a$ heißt Vorbereich und die Menge $A^\bullet := \bigcup_{a \in A} a^\bullet$ heißt Nachbereich von A.

Definition 2.1.6 (Vorbereich und Nachbereich)

Für einen Graphen können wir nicht nur benachbarte Knoten sondern auch Folgen von Knoten betrachten, die durch Kanten verbunden sind.

Definition 2.1.7 (Weg)
Sei $G = (V, \rightarrow)$ ein Graph und $v_1 \dots v_n \in V^*$ eine nicht-leere Folge von Knoten. Gilt für alle $i \in \{1, \dots, n-1\}$: $v_i \rightarrow v_{i+1}$, so ist die Folge $(v_1, v_2) \dots (v_{n-1}, v_n)$ ein Weg im Graph. Für einen Weg heißen $v_1, \dots, v_n$ Knoten des Weges. Die Länge eines Weges ist die Anzahl der Kanten des Weges.

Definition 2.1.7 (Weg)

Eine Teilmenge von Knoten eines Graphen zusammen mit Kanten innerhalb der Teilmenge heißt Teilgraph.

Definition 2.1.8 (Teilgraph)
Seien $G_1 = (V_1, \rightarrow_1)$ und $G_2 = (V_2, \rightarrow_2)$ zwei Graphen. Gilt $V_2 \subseteq V_1$ und $\rightarrow_2 \subseteq \rightarrow_1$, so heißt G_2 Teilgraph von G_1.
Ist $G_1 = (V_1, \rightarrow_1)$ ein Graph und $V_2 \subseteq V_1$ eine Menge von Knoten, so ist der Teilgraph $G_2 = (V_2, \rightarrow_2)$ mit $\rightarrow_2 = \rightarrow_1 \cap (V_2 \times V_2)$ der von V_2 in G_1 induzierte Teilgraph. Für den von V_2 in G_1 induzierten Teilgraph schreiben wir $G_1[V_2]$.

Definition 2.1.8 (Teilgraph)

Diese Arbeit verwendet meist transitive und irreflexive Relationen auf einer endlichen Mengen von Knoten. Wir nennen diese spezielle Klasse von Graphen Halbordnung.

Definition 2.1.9 (Halbordnung)
Ein gerichteter Graph $po = (V, <)$ heißt Halbordnung, falls die Relation $<$ irreflexiv und transitiv ist.

Definition 2.1.9 (Halbordnung)

Auch für Halbordnungen lässt sich deren Skelett betrachten. Das Skelett einer Halbordnung $(V, <)$ ist der Graph $(V, <^\diamond)$. Nach der Definition eines Skelettes existiert für jede Kante (a, b) einer Halbordnung im Skelett mindestens ein Weg von a nach b.
Knotenpaare einer Halbordnung, die nicht über Wege verbunden sind nennen wir nicht geordnet. Die sogenannte co-Relation enthält alle nicht geordneten Knotenpaare.

Definition 2.1.10 (co-Menge und Schnitt)
Sei $po = (V, <)$ eine Halbordnung und $a, b \in V$ zwei Knoten. Wir schreiben $a \,co\, b$, falls weder $a < b$ noch $b < a$ gilt.
Sei $S \subset V$ eine Menge von Knoten. Die Menge S ist eine co-Menge, falls $\forall a, b \in S : a \,co\, b$ gilt. Ist eine co-Menge S in keiner anderen co-Menge enthalten, so nennen wir S einen Schnitt.

Definition 2.1.10 (co-Menge und Schnitt)

In einer Halbordnung ist eine Menge von Knoten, die ihren gesamten Vorbereich beinhaltet, eine Art Anfangsstück. Eine solche Menge zusammen mit allen Kanten zwischen Knoten dieser Menge heißt Präfix. Da eine Halbordnung transitiv ist, existiert kein Knoten, der vor ein Präfix geordnet und nicht in diesem enthalten ist.

Definition 2.1.11 (Präfix)

Definition 2.1.11 (Präfix)
Sei $po = (V, <)$ *eine Halbordnung und* P *eine Menge von Knoten für die* $P = P \cup {}^\bullet P$ *gilt. Der induzierte Graph* $po[P]$ *heißt Präfix von* po. *Jede Menge von Knoten* $S \subseteq V$ *definiert durch* $po[{}^\bullet S]$ *ein Präfix.*

2.2 KOMPLEXITÄTSBETRACHTUNGEN

Diese Arbeit stellt Algorithmen vor und diskutiert deren Laufzeit. In diesem Abschnitt führen wir Grundlagen ein, um die Komplexität eines Algorithmus unabhängig von einer konkreten Eingabe und unabhängig von der zur Verfügung stehenden Hardware beschreiben zu können (siehe [76] und [62] für eine ausführliche Darstellung).
Um Laufzeiten von Algorithmen unabhängig von einem Rechner betrachten zu können, setzt man die Anzahl der Rechenschritte eines Algorithmus in Bezug zur Größe der Eingabe. Gelingt dies, erhält man für eine Eingabe der Länge n eine Anzahl an Rechenschritten $f(n)$, die der Algorithmus für die Bearbeitung der Eingabe benötigt. Oft ist es notwendig die möglichen Eingaben in Klassen einzuteilen. Man spricht bei einer Eingabe, die unter allen möglichen Eingaben derselben Länge die meisten Rechenschritte verursacht, von einem schlechtesten Fall. Der Zusammenhang f zwischen der Länge der Eingaben und der Anzahl der Rechenschritte für die schlechtesten Fälle ist die Schlechteste-Fall-Komplexität. Man spricht von Durchschnittlicher-Fall-Komplexität, wenn man die begründete Annahme hat, dass die Funktion f die Anzahl der Rechenschritte für einem Großteil aller möglichen Eingaben beschreibt.
Da die Komplexität eines Algorithmus bei Eingaben gleicher Länge stark variieren kann, betrachtet man das asymptotische Wachstum der Funktion f bei großen Eingaben. Damit lassen sich Komplexitätsklassen definieren, in die man die entsprechenden Laufzeiten der Algorithmen einteilt. Um diese Klassen von Funktionen zu beschreiben verwendet man die O-Notation.

Definition 2.2.1 (O-Notation)

Definition 2.2.1 (O-Notation)
Seien f *und* g *Funktionen. Wir schreiben* $f \in O(g)$, *falls ein* $n_0 \in \mathbb{N}$ *und ein* $c > 0$ *existieren, so dass für alle* $n \geqslant n_0$ *die Gleichung* $f(n) \leqslant c \cdot g(n)$ *gilt.*

In Bezug auf die Laufzeit-Komplexität kann man die Klassen Polynomialzeit und Exponentialzeit unterscheiden. Ein Algorithmus läuft in Polynomialzeit, falls der Zusammenhang zwischen der Länge der Eingabe und der Anzahl der Arbeitsschritte durch eine Funktion $f \in O(g)$ beschrieben werden kann und g ein Polynom ist. Ist g eine Exponentialfunktion, so liegt die Laufzeit des Algorithmus in Exponentialzeit. Die selben Komplexitätsklassen lassen sich für die Speicherplatz-Komplexität beschreiben.
Die Laufzeiten von Algorithmen können mit Hilfe der O-Notation nur ihrem Wachstum nach und für sehr große Eingaben treffend beschrieben werden. Aus diesem Grund führen wir in dieser Arbeit Laufzeitexperimente durch. Auf diese Weise können die Laufzeiten verschiedener Algorithmen bei identischer Eingabe verglichen werden. Die in Abschnitt 3.6 und Abschnitt 6.1 dargestellten Experimente wurden mit einem Dell Precision 4500, Dual Core Prozessor mit je 1.7 GHz und 4GB Hauptspeicher durchgeführt.
Alle in dieser Arbeit beschriebenen Algorithmen werden in der Sprache Java als Plugin für das VipTool [37] implementiert. In der Arbeit selbst

sind die Algorithmen in Pseudocode dargestellt, um deren Lesbarkeit zu erhöhen. Im Pseudocode beschreiben wir Variablen, die Integer-Werte enthalten, als kleine Buchstaben oder klein geschriebene Wörter. Listen oder Arrays werden durch groß geschriebene, kurze, aussagekräftige Wörter, wie zum Beispiel *Weg* oder `Liste`, benannt. Matrizen oder Listen von Listen beschreiben wir durch einzelne Großbuchstaben.

3

FLÜSSE IN NETZWERKEN

Dieses Kapitel beschreibt Flüsse in Flussnetzwerken. Wir benötigen maximale Flüsse in Flussnetzwerken, um in Kapitel 4, 5 und 6 das Szenario-Verifikations-Problem für Petrinetze effizient zu entscheiden. Einen guten Überblick über die Thematik der Flussnetzwerke geben z.B. [61] und [4]. In diesem Kapitel wollen wir ausgewählte Flussnetzwerk-Algorithmen vorstellen, da wir ihre Strategien in späteren Kapiteln aufgreifen und für die Entscheidung des Szenario-Verifikations-Problems nutzen werden.

3.1 FLUSSNETZWERKE

Ein Flussnetzwerk kann man sich als Kanalsystem vorstellen, durch das stetig Wasser fließt. Ein Beispiel ist der Abschnitt des Mississippis in Abbildung 5. Das Flussnetzwerk ist hierbei das Flussbett. Die Menge des Wassers, welches das Flussbett durchläuft, ist der Fluss und durch die Kapazität der einzelnen Arme und deren Anordnung ist die Menge Wasser festgelegt, die dieser Abschnitt des Mississippis aufnehmen kann. Das Fluss-Maximierungs-Problem ist nun die Frage danach, wie viel Wasser durch das gesamte System fließen kann, ohne dass die Kapazität eines Arms überschritten wird bzw. der Mississippi über seine Ufer tritt.

Abbildung 5: Ein Foto des Mississippi.

Abbildung 6 zeigt den in Abbildung 5 dargestellten Abschnitt in einer formalen Darstellung als Flussnetzwerk. Ein Flussnetzwerk ist ein Graph, in dessen Knotenmenge zwei Knoten ausgezeichnet sind. Der eine ist die Quelle, der andere ist die Senke des Flussnetzwerks. Für unsere Beispiele nummerieren wir alle Knoten durch. Die Quelle erhält dabei immer die Nummer Null, die Senke erhält immer die größte Nummer. Die gerichteten Kanten sind zusätzlich mit einem Gewicht versehen und beschreiben die Arme des Mississippi mit ihrer Fließrichtung. Die Gewichte der Kanten beschreiben die Kapazitäten der

Arme. Kanten bilden Wege von der Quelle zur Senke, die an Knoten verzweigen und wieder zusammenlaufen können. Die Quelle besitzt keine eingehenden und die Senke keine ausgehenden Kanten.

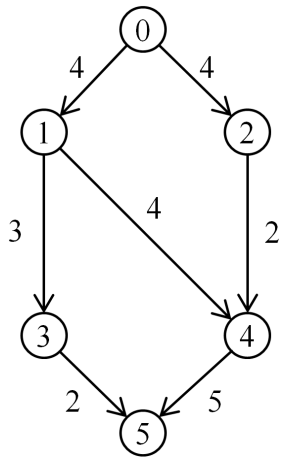

Abbildung 6: Ein Flussnetzwerk.

Definition 3.1.1 (Flussnetzwerk)

Definition 3.1.1 (Flussnetzwerk)
Ein Flussnetzwerk ist ein Tupel $G = (E, F, c, q, s)$. *Dabei ist* (E, F) *ein Graph und* $c : F \to \mathbb{N}$ *eine Funktion, die jeder Kante eine Kapazität zuordnet. Die Quelle* $q \in E$ *und die Senke* $s \in E$ *sind zwei verschiedene, ausgezeichnete Knoten, wobei die Quelle* q *keine eingehenden und die Senke* s *keine ausgehenden Kanten besitzt.*

Für die Darstellung eines Flussnetzwerks in einem Rechner gibt es zwei Möglichkeiten: In objektorientierten Programmiersprachen liegt es nahe, Knoten und Kanten durch zwei verschiedene Klassen darzustellen. Jeder Knoten besitzt dazu Referenzen auf Kanten, die zu dem Knoten hin führen, und Referenzen auf Kanten, die von ihm ausgehen. Kanten besitzen das Attribut Kapazität und je eine Referenz auf den Knoten, bei dem sie beginnen, und eine Referenz auf den Knoten, bei dem sie enden. Die zweite Möglichkeit der Repräsentation eines Flussnetzwerks ist die sogenannte Matrix-Darstellung. Ein Flussnetzwerk mit n Knoten kann durch eine $(n \times n)$-Matrix repräsentiert werden. Dazu ordnet man jedem Knoten eine Zahl zwischen 1 und n und damit eine Zeile und eine Spalte der Matrix zu. In die Matrix trägt man die Kapazitäten der Kanten ein, indem man in die Zeile des Knotens u an die Stelle des Knotens v die Kapazität der Kante (u, v) einträgt. Man trägt den Wert 0 ein, falls zwischen u und v keine Kante existiert.
Für Flussnetzwerke, bei denen die Anzahl der Kanten im Verhältnis zur der Anzahl der Knoten gering ist, ist die Repräsentation durch entsprechende Objekte speicherplatzeffizienter als die Darstellung durch eine Matrix. Allerdings ist die Zugriffszeit auf konkrete Kanten in der Matrix-Darstellung wesentlich kürzer als die Zugriffszeit auf Kanten in der objektorientierten Darstellung, da in einer objektorientierten Darstellung die Kanten und Knoten erst in entsprechenden Sets von Referenzen gesucht werden müssen. Die Matrix-Darstellung unseres Mississippi-Abschnitts findet sich in Abbildung 7.
Ein Fluss in einem Flussnetzwerk ist eine Abbildung, die jeder Kante eine nicht-negative Zahl, den sogenannten Wert des Flusses auf dieser

$$\begin{pmatrix} 0 & 4 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Abbildung 7: Die Matrix-Darstellung des Flussnetzwerks.

Kante zuordnet. Der Wert des Flusses auf einer Kante darf dabei deren Kapazität nicht überschreiten. Die Summe der Werte des Flusses auf Kanten, die zu einem Knoten führen, ist der Zufluss des Knotens. Analog definiert man den Abfluss eines Knotens als Summe der Werte des Flusses aller ausgehenden Kanten. Für einen Fluss fordern wir die Flusserhaltung, was bedeutet, dass für alle Knoten außer der Quelle und der Senke der Zufluss genau so groß ist wie der Abfluss. Das bedeutet, dass innerhalb des Netzwerks kein Fluss entsteht oder verloren geht.

Definition 3.1.2 (Fluss)

Definition 3.1.2 (Fluss)
Sei $G = (E, F, c, q, s)$ *ein Flussnetzwerk. Ein Fluss in* G *ist eine Funktion* $f : F \to \mathbb{N}_0$ *für die zwei Eigenschaften gelten:*

(Flusserhaltung) *Für alle Knoten* $u \in E \backslash \{q, s\}$ *gilt*

$$\sum_{(v,u)\in F} f(v,u) = \sum_{(u,v)\in F} f(u,v).$$

(Kapazitätsbeschränkung) *Für alle Kanten* $e \in F$ *gilt* $f(e) \leqslant c(e)$.

Aus der Flusserhaltung folgt: $\sum_{(q,v)\in F} f(q,v) = \sum_{(v,s)\in F} f(v,s)$. *Wir definieren diese Summe als Wert des Flusses* f *und bezeichnen sie mit* $w(f)$.

Abbildung 8 zeigt zwei verschiedene Flüsse im Flussnetzwerk aus Abbildung 6. An jeder Kante ist zuerst die Kapazität, dann der Wert des Flusses auf dieser Kante notiert. Der Wert des linken Flusses ist 5, der Wert des rechten Flusses ist 6.
Das Fluss-Maximierungs-Problem beschreibt die Aufgabe, in einem gegebenen Flussnetzwerk einen Fluss mit maximalem Wert zu konstruieren.

Definition 3.1.3 (Fluss-Maximierungs-Problem)

Definition 3.1.3 (Fluss-Maximierungs-Problem)

Gegeben: *Flussnetzwerk* $G = (E, F, c, q, s)$.
Gesucht: *Ein Fluss* f *in* G, *so dass für alle* f' *in* $G : w(f) \geqslant w(f')$ *gilt.*

Zur Lösung des Fluss-Maximierungs-Problems existieren verschiedene Strategien, die wir in den folgenden Abschnitten kurz beschreiben. Der Fokus unserer Betrachtungen liegt dabei auf der Laufzeit-Komplexität.

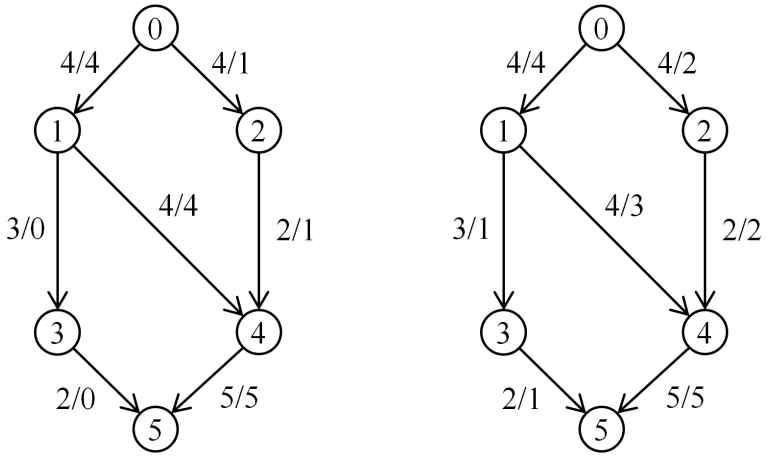

Abbildung 8: Zwei Flüsse in einem Flussnetzwerk.

3.2 DER ALGORITHMUS VON FORD UND FULKERSON

In diesem Abschnitt betrachten wir den Algorithmus von Ford und Fulkerson [52]. Dieser Algorithmus war der erste, der das Fluss-Maximierungs-Problem entscheiden konnte. Obwohl der Algorithmus von Ford und Fulkerson nicht in Polynomialzeit läuft, ist er die Grundlage für eine ganze Klasse von Fluss-Maximierungs-Algorithmen, die einen maximalen Fluss in Polynomialzeit konstruieren. Einer dieser Algorithmen ist der Algorithmus von Edmonds und Karp [47], welcher am Ende dieses Abschnittes beschrieben wird.
In einem Flussnetzwerk ist der Fluss, der auf jeder Kante den Wert 0 besitzt, der Nullfluss. Dieser Nullfluss erfüllt trivialerweise die Flusserhaltung und die Kapazitätsbeschränkung. Der Nullfluss hat den Wert 0. Um einen maximalen Fluss zu konstruieren, beginnt der Algorithmus von Ford und Fulkerson mit diesem Nullfluss, den er sukzessive vergrößert. In jeder Iteration sucht der Algorithmus einen Weg von der Quelle zur Senke, auf dem die Kapazität jeder Kante durch den aktuellen Fluss noch nicht ausgeschöpft ist. Findet der Algorithmus einen solchen Weg, kann er den Fluss auf den Kanten dieses Wegs um 1 erhöhen.
Dieses Vorgehen alleine schafft es nicht, einen maximalen Fluss zu konstruieren. Auf der linken Seite in Abbildung 8 ist ein Fluss abgebildet, dessen Wert noch nicht maximal ist, obwohl kein Weg von der Quelle zur Senke existiert, der noch Fluss aufnehmen könnte. Die Lösung von Ford und Fulkerson besteht darin, dass der Algorithmus bereits erzeugten Fluss im Flussnetzwerk umleitet, falls er dadurch einen Fluss mit größerem Wert konstruieren kann.
Abbildung 9 zeigt auf der linken Seite einen Fluss mit Wert 5 in unserem Flussnetzwerk. Die hervorgehobenen Kanten bilden einen speziellen Weg von der Quelle zur Senke. Dabei dürfen Kanten auf zwei Arten durchlaufen werden. Man darf Kanten vorwärts durchlaufen, falls deren Kapazität noch nicht ausgeschöpft ist, zudem darf man Kanten rückwärts durchlaufen, falls sie bereits Fluss enthalten. Diese Wege heißen flussvergrößernde Wege und in jeder Iteration sucht der Algorithmus von Ford und Fulkerson einen davon. Findet er einen

flussvergrößernden Weg, erhöht er den Fluss auf jeder Vorwärts-Kante um Eins bzw. verringert den Fluss auf jeder Rückwärts-Kante um Eins.

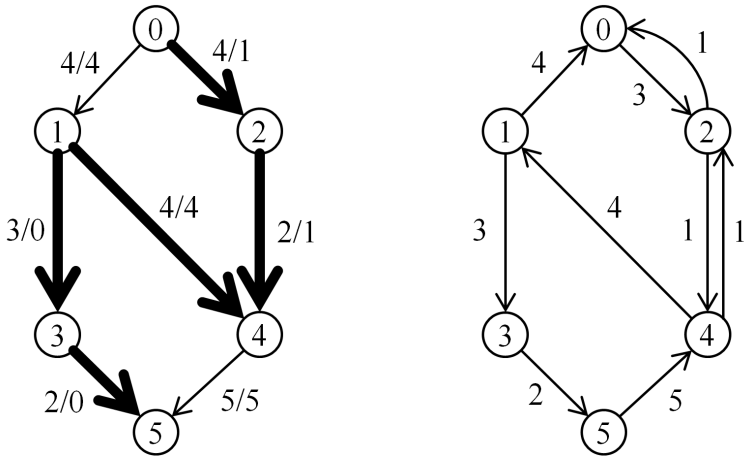

Abbildung 9: Links: Ein flussvergrößernder Weg. Rechts: Das zugehörige Restnetzwerk.

Auf der linken Seite in Abbildung 9 sehen wir, wie mit Hilfe dieses Vorgehens ein maximaler Fluss konstruiert wird. Die Kante $(0,1)$ ist in ihrer Kapazität bereits ausgeschöpft. Die einzige Alternative, die Senke zu erreichen, führt über Knoten 2 zu Knoten 4. Ohne die Möglichkeit, Kanten rückwärts zu durchlaufen, wäre der Algorithmus an dieser Stelle beendet und könnte den Wert des Fluss nicht weiter erhöhen. Da die Kante $(1,4)$ bereits Fluss trägt, hat der Algorithmus in einer früheren Iteration einen Weg von der Quelle zu Knoten 1, über Knoten 4 und dann zur Senke gefunden. Der Algorithmus verknüpft an dieser Stelle implizit das neue Anfangsstück [0 2 4] mit dem Endstück des alten Wegs [4 5], indem er den Fluss auf der Kante $(1,4)$ zurücknimmt und danach versucht, das alte Anfangsstück [0 1] auf einem anderen Weg zur Senke zu führen. Nach dieser Modifikation entsteht der auf der rechten Seite von Abbildung 8 abgebildete maximale Fluss mit Wert 6.
Ford und Fulkerson beweisen, dass ein Fluss mit maximalem Wert konstruiert ist, sobald kein flussvergrößernder Weg mehr existiert [52]. Um flussvergrößernde Wege zu finden, konstruieren Ford und Fulkerson das sogenannte Restnetzwerk, welches zu einem Fluss in einem Flussnetzwerk alle flussvergrößernden Wege enthält.

Definition 3.2.1 (Restnetzwerk)

Definition 3.2.1 (Restnetzwerk)
Sei $G = (E, F, c, q, s)$ *ein Flussnetzwerk und* f *ein Fluss in* G. *Wir definieren die Funktion* $r_f : (E \times E) \to \mathbb{N}_0$ *durch:*

$$r_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \textit{falls } (u,v) \in F, \\ f(v,u), & \textit{falls } (v,u) \in F, \\ 0, & \textit{sonst.} \end{cases}$$

Das Tupel $G_f = (E, F_f, r_f|_{F_f}, q, s)$ *mit* $F_f = \{(u,v) \in (E \times E) \mid r_f(u,v) > 0\}$ *heißt Restnetzwerk zu* G *und* f.

Das Restnetzwerk ist in der Regel kein Flussnetzwerk, da die Quelle eingehende und die Senke ausgehende Kanten besitzen kann. Abbil-

dung 9 zeigt auf der rechten Seite das Restnetzwerk zu dem auf der linken Seite abgebildeten Fluss in unserem Flussnetzwerk.

Definition 3.2.2 (flussvergrößernder Weg)

Definition 3.2.2 (flussvergrössernder Weg)
Ein flussvergrößernder Weg in einem Flussnetzwerk G zu einem Fluss f ist ein Weg von q nach s im Restnetzwerk G_f, der jede Kante in G_f höchstens einmal enthält.

Wir modifizieren einen Fluss f durch einen flussvergrößernden Weg ω in einem Flussnetzwerk G zu einem Fluss f', indem wir den Wert des neuen Flusses f' auf allen Kanten $(u, v) \in F$ definieren.

$$f'(u,v) = \begin{cases} f(u,v) + 1, & \text{falls } (u,v) \in \omega, \\ f(u,v) - 1, & \text{falls } (v,u) \in \omega, \\ f(u,v), & \text{sonst.} \end{cases}$$

Modifizieren wir einen Fluss f durch einen flussvergrößernden Weg, so besitzt der neue Fluss f' auf keiner Kante einen negativen Wert und erfüllt die Kapazitätsbeschränkung. Um zu zeigen, dass f' auch die Flusserhaltung erfüllt, betrachten wir einen beliebigen Knoten auf dem flussvergrößernden Weg von der Quelle zur Senke und unterscheiden vier Fälle. Wird der Knoten durch eine Vorwärts-Kante erreicht und durch eine Vorwärts-Kante verlassen (z.B. Knoten 2 in Abbildung 9), so erhöhen sich der Zufluss und der Abfluss dieses Knotens um je Eins. Sind beides Rückwärts-Kanten, verringern sich Zufluss und Abfluss dieses Knotens um Eins. Wird der Knoten durch eine Vorwärts-Kante erreicht und durch ein Rückwärts-Kante verlassen (z.B. Knoten 4 in Abbildung 9), wird der Zufluss des Knotens durch die Vorwärts-Kante um Eins erhöht und durch die Rückwärtskante wieder um Eins verringert. Insgesamt bleibt in diesem Fall der Zufluss und der Abfluss des Knoten konstant. Analog gleicht sich im letzten Fall der Wert des Abflusses eines Knotens aus, wenn der Knoten über eine Rückwärts-Kante erreicht und über eine Vorwärts-Kante wieder verlassen wird (z.B. Knoten 1 in Abbildung 9). In jedem der vier Fälle bleibt der Zufluss des Knoten gleich seinem Abfluss, somit erfüllt f' die Flusserhaltung, und wir haben gezeigt, dass f' wiederum ein Fluss in G ist. Jeder flussvergrößernde Weg beginnt bei der Quelle q. Sie wird durch eine Vorwärts-Kante verlassen und der Abfluss der Quelle erhöht sich damit um eins. Daraus folgt, dass für den Wert des Flusses $w(f') = w(f) + 1$ gilt.
Durch Iteration der Konstruktion des Restnetzwerks und der Suche nach einem flussvergrößernden Weg können Flüsse mit immer größeren Werten im Flussnetzwerk konstruiert werden. Ford und Fulkerson zeigen, dass dieses Verfahren terminiert, da eine obere Grenze für den Wert aller Flüsse existiert. Damit ist ein Erhöhen des Flusses nur endlich oft möglich. Bricht das Verfahren ab, so hat man einen maximalen Fluss konstruiert.
Um zu zeigen, dass der maximale Wert eines Flusses beschränkt ist, betrachten Ford und Fulkerson Engstellen eines Flussnetzwerks. Engstellen sind Mengen von Kanten, die das Flussnetzwerk in zwei Teile teilen. Jede Partition der Knotenmenge in zwei Mengen $Oben$ und $Unten$ legt eine solche Menge von Kanten fest und heißt Schnitt eines Flussnetzwerks.

DEFINITION 3.2.3 (SCHNITT)
Definition 3.2.3 (Schnitt)

Ein Schnitt (Q, S) *in einem Flussnetzwerk* $G = (E, F, c, q, s)$ *ist eine Partition der Knotenmenge* $E = Q \cup S$*, falls* $Q \cap S = \varnothing$ *und zusätzlich* $q \in Q$ *und* $s \in S$ *gilt. Für einen Schnitt* (Q, S) *definieren wir*

$$\textit{Kapazität} \qquad c(Q,S) = \sum_{(u,v)\in(F\cap(Q\times S))} c(u,v),$$

$$\textit{Fluss} \qquad f(Q,S) = \sum_{(u,v)\in(F\cap(Q\times S))} f(u,v) - \sum_{(v,u)\in(F\cap(S\times Q))} f(v,u).$$

Abbildung 10 zeigt ein Beispiel eines Schnittes. Die weißen Knoten bilden die Menge Q, die schwarzen Knoten die Menge S. Die Kapazität des Schnittes ist 9, der Wert des Flusses über den Schnitt ist 6.

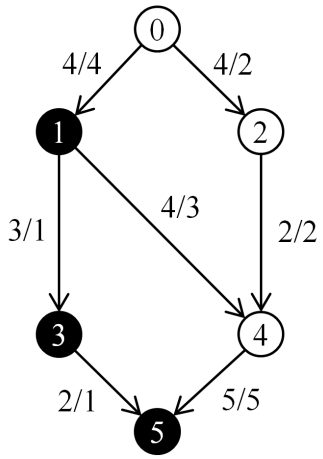

Abbildung 10: Ein Schnitt im Flussnetzwerk.

Für einen Fluss ist der Wert des Flusses über jeden Schnitt gleich. Dies ist leicht einzusehen, indem wir zunächst den Schnitt betrachten, dessen Menge S nur die Senke enthält. Der Wert dieses Schnittes ist der Zufluss der Senke. Jeder weitere Knoten, den man zu der Menge S hinzunimmt, besitzt aufgrund der Flusserhaltung den gleichen Zufluss und Abfluss. Der Wert des Schnittes ändert sich durch Hinzufügen weiterer Knoten nicht. Daraus folgt, dass der Wert des Flusses über jeden Schnitt dem Wert des Flusses entspricht. Da der Wert jedes Schnittes maximal so groß seinen kann, wie die Kapazität des Schnittes, ist der Wert eines maximalen Flusses genau so groß, wie die minimale Kapazität aller Schnitte eines Flussnetzwerks. Insbesondere ist damit der Wert eines Flusses immer beschränkt und es ergibt sich der Satz von Ford und Fulkerson.

SATZ 3.2.1 (SATZ VON FORD UND FULKERSON [52])
Satz 3.2.1 (Satz von Ford und Fulkerson [52])

Folgende drei Aussagen sind äquivalent:

1. f *ist ein maximaler Fluss im Flussnetzwerk* G.
2. *Das Restnetzwerk* G_f *enthält keinen flussvergrößernden Weg.*
3. *Es gibt einen Schnitt* (Q, S) *mit* $w(f) = c(Q, S)$.

Ein Korollar aus Satz 3.2.1 ist, dass es immer einen maximalen Fluss mit ganzzahligen Werten auf allen Kanten gibt. Dies ist der Fall, da unsere Kapazitäten eines Flussnetzwerks ganzzahlig sind und der Algorithmus mit dem Nullfluss beginnt. Jeder flussvergrößernde Weg ändert die Werte des Flusses auf Kanten nicht oder um Eins. Somit ist auch der maximale Fluss auf allen Kanten ganzzahlig.
Betrachten wir abschließend die Komplexität des Algorithmus von Ford und Fulkerson. Sei dafür eine Flussnetzwerk $G = (E, F, c, q, s)$ gegeben. Ein Algorithmus, der einen Weg durch einen Graphen findet, betrachtet dazu im ungünstigsten Fall alle Knoten und Kanten. Damit ist die Laufzeit dieser Suche in $O(|E| + |F|) = O(|F|)$. Damit ist die Komplexität des Ford und Fulkerson Algorithmus durch $O(|F| \cdot w(f_{max}))$ Schritte beschränkt, wobei $w(f_{max})$ der Wert eines maximalen Flusses ist. Da die Kapazitäten von Kanten, die den Wert eines maximalen Flusses beeinflussen, durch Binär-Schreibweise oder Dezimal-Schreibweise von Zahlen eingegeben werden können, deren Werte exponentiell in ihrer Länge wachsen, läuft der Algorithmus nicht in Polynomialzeit.
Um einen Algorithmus zu erhalten, der einen maximalen Fluss in einem Flussnetzwerk in Polynomialzeit konstruiert, erweitern Edmonds und Karp den Algorithmus von Ford und Fulkerson [47]. Sie verbesserten den Algorithmus im Wesentlichen durch zwei Ideen. Die erste Idee ist, den Wert eines Flusses auf einem flussvergrößernden Weg ω nicht nur um 1 zu erhöhen, sondern um den Wert der kleinsten Kapazität der Kanten von ω im Restnetzwerk, den wir mit Δ_ω bezeichnen. Diese Idee führt zu einer wesentlichen Verbesserung der Laufzeit, kann aber Polynomialzeit noch nicht garantieren. Als Beispiel betrachten wir Abbildung 11, die ein Flussnetzwerk zeigt, in dem man 2k durch 1 beschränkte flussvergrößernde Wege finden kann, wenn jeder Weg die mittlere Kante, deren Kapazität den Wert Δ_ω auf 1 beschränkt, beinhaltet.

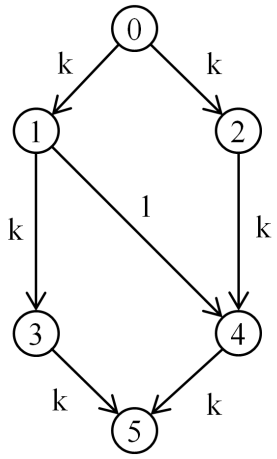

Abbildung 11: Ein Flussnetzwerk mit maximalem Fluss 2k.

Die entscheidende zweite Idee ist, anstatt eines beliebigen flussvergrößernden Wegs in jeder Iteration einen kürzesten flussvergrößernden Weg zu wählen. Kürzeste Wege können leicht mit Hilfe von Breitensuche in $O(|E| + |F|) = O(|F|)$ Schritten gefunden werden. Wählen wir einen kürzesten Weg ω und vergrößern einen Fluss um Δ_ω, so füllen wir mindestens eine Kante des Flussnetzwerks in jeder Iteration. Damit verschwinden die kürzesten flussvergrößernde Wege sukzessive

aus dem Restnetzwerk, und Wege zwischen Quelle und Senke werden immer länger. Dass bei dieser Prozedur keine neuen kürzeren Wege entstehen, zeigen Edmonds und Karp. Da die Länge der Wege durch die Anzahl der Knoten begrenzt ist, terminiert dieser Algorithmus. Auf den Beweis wollen wir verzichten, halten aber fest:

SATZ 3.2.2 (SATZ VON EDMONDS UND KARP [47])
Sei $G = (E, F, c, q, s)$ *ein Flussnetzwerk. Der Algorithmus von Ford und Fulkerson entscheidet das Fluss-Maximierungs-Problem in* $O(|E| \cdot |F|^2)$ *Schritten, falls jeder flussvergrößernde Weg durch Breitensuche konstruiert wird.*

Satz 3.2.2 (Satz von Edmonds und Karp [47])

Algorithmus 3.2.1 beschreibt unsere Implementierung des Algorithmus von Ford und Fulkerson nach Edmonds und Karp in Pseudocode.

ALGORITHMUS 3.2.1 (ALGORITHMUS VON EDMONDS UND KARP)

Eingabe: *Flussnetzwerk* G
Ausgabe: *Wert w eines maximalen Flusses*

Algorithmus 3.2.1 (Algorithmus von Edmonds und Karp)

```
1   n ← Anzahl der Knoten
2   N ← Nachbarschaften in G
3   w ← 0
4   WHILE (Weg ← Weg im Restnetzwerk(G,F,N))
5   {
6       Δ ← ∞, u ← n − 1
7       WHILE ((v ← Weg[u]) ⩾ 0)
8       {
9           Δ ← Minimum(Δ, G[v][u] − F[v][u])
10          u ← v
11      }
12      u ← n − 1
13      WHILE (Weg[u] ⩾ 0)
14      {
15          F[Weg[u]][u] + ← Δ
16          F[u][Weg[u]] − ← Δ
17          u ← Weg[u]
18      }
19      w + ← Δ
20  }
21  RETURN w
```

In der Implementierung des Algorithmus von Edmonds und Karp 3.2.1 ist die Eingabe eine Matrix G mit ganzzahligen Werten, die das Flussnetzwerk beschreibt. Dabei hat die Quelle immer den Index 0 und die Senke den größten existierenden Index. Ausgabe ist ein Wert w, der Wert eines maximalen Flusses. Um einen effizienten Algorithmus zu erhalten, berechnen wir im ersten Schritt für jeden Knoten des Flussnetzwerks eine Liste mit Indizes von Knoten, die zu diesem benachbart sind. Nur zwischen diesen Knoten können im Verlauf des Algorithmus Kanten im Restnetzwerk entstehen. Die Listen der Nachbarschaften der einzelnen Knoten werden in der Liste N von Listen zusammengefasst (Zeile 2). Der Algorithmus muss ab diesem Schritt nicht mehr die gesamte Matrix G, sondern immer nur noch die in N gespeicherten Indizes der Matrix G untersuchen. In Zeile 4 liefert die Funktion *Weg im Restnetzwerk* (Algorithmus 3.2.2) einen kürzesten Weg *Weg* durch das aktuelle Restnetzwerk. Das Restnetzwerk wird in dieser Implementierung nicht explizit gespeichert, sondern ergibt sich als $G - F$. Die

Matrix F besitzt genau die gleiche Anzahl von Spalten und Zeilen wie die Matrix G, ist aber zu Beginn des Algorithmus leer. Im Laufe des Algorithmus wird in F neben dem Wert des Flusses auf einer Kante (u, v) auch immer der negative Wert des Flusses auf der Kante (v, u) eingetragen. Solange ein Weg *Weg* im Restnetzwerk existiert, werden folgende Schritte ausgeführt: Zunächst wird in Zeile 6-11 der Wert Δ bestimmt, welcher den minimalen Wert der Kapazitäten der Kanten auf dem Weg *Weg* im Restnetzwerk $G - F$ angibt. Danach wird in Zeile 12-18 der Fluss auf den Kanten des Wegs um Δ erhöht. Dies geschieht, indem sich der Algorithmus bei der Senke beginnend von Vorgänger zu Vorgänger hangelt, bis er die Quelle erreicht. Dieses Erhöhen des Flusses über Kanten geschieht dabei so, dass für jede Kante des Wegs in Zeile 15 der Wert Δ an der entsprechenden Stelle der Matrix F addiert und in Zeile 16 der Wert Δ an der gespiegelten Stelle subtrahiert wird, so dass $G - F$ immer das aktuelle Restnetzwerk ergibt. Diese indirekte Berechnung des Restnetzwerks ist wesentlich effizienter als die erneute Berechnung des Restnetzwerks vor jedem Durchlauf. Zudem erspart man sich durch dieses Vorgehen eine aufwändige Unterscheidung der Kanten in Vorwärts- und Rückwärts-Kanten. Der Wert w des konstruierten Flusses wird in Zeile 19 aktualisiert und schließlich in Zeile 21 zurückgegeben.

Algorithmus 3.2.2 (Weg im Restnetzwerk)

ALGORITHMUS 3.2.2 (WEG IM RESTNETZWERK)

Eingabe: *Restnetzwerk* $G - F$, *Nachbarschaften* N
Ausgabe: *kürzester Weg* Weg *durchs Restnetzwerk*

```
1   kopf ← 0, ende ← 1
2   Liste[0] ← 0
3   Farbe[0] ← 1
4   Weg[0] ← −1
5   WHILE (kopf ≠ ende)
6   {
7       u ← Liste[kopf]
8       FOR i ← 0 TO |N[u]| − 1
9       {
10          v ← N[u][i]
11          IF (Farbe[v] = 0 AND G[u][v] − F[u][v] > 0)
12          {
13              Liste[ende] ← v
14              ende++
15              Farbe[v] ← 1
16              Weg[v] ← u
17          }
18      }
19      Farbe[u] ← 2
20      kopf++
21  }
22  IF (Farbe[n − 1] = 2) RETURN Weg
23  ELSE RETURN null
```

Algorithmus 3.2.2 zeigt die Funktion Weg im Restnetzwerk, die eine Breitensuche im Restnetzwerk $G - F$ mit Hilfe der Liste von Nachbarschafen N implementiert. Die Ausgabe ist null oder, falls dieser existiert, ein Weg von der Quelle zur Senke im Restnetzwerk. Die Breitensuche beginnt bei der Quelle. Die Quelle hat den Index 0. Dieser

wird in Zeile 2 dem Array Liste hinzugefügt. Das Array Liste speichert die bereits erreichten und noch nicht betrachteten Knoten. Dazu existieren zwei Zähler: kopf und ende. kopf markiert den aktuellen Knoten im Array, ende markiert die Position des Endes der Liste. Holt der Zähler kopf den Zähler ende ein, bricht die Suche ab (Zeile 5). In den Zeilen 8-18 werden für jeden Knoten der Liste Liste jeweils die benachbarten Knoten gesucht. Indizes möglicher Nachbarn sind in der Liste N gespeichert (Zeile 10). Falls zu einem der möglichen Nachbarn eine Kante im Restnetzwerk $G - F$ existiert (Zeile 11) und dieser in einer früheren Iteration noch nicht gefunden wurde, wird dieser der Liste Liste hinzugefügt (Zeile 13), durch die Liste Farbe als gefunden markiert (Zeile 15) und der Weg vom aktuellen Knoten zu dem gefundenen in *Weg* gespeichert (Zeile 16). Damit beschreibt die Liste *Weg* einen Weg, indem sie für jeden Knoten den Index des Knotens enthält, über den der Knoten bei der Breitensuche erreicht wurde. Wurde nach Ablauf des Algorithmus die Senke markiert, wird der so gefundene Weg als Liste *Weg* zurückgegeben (Zeile 22).

3.3 DER ALGORITHMUS VON DINIC

In diesem Abschnitt betrachten wir den Fluss-Maximierungs-Algorithmus von Dinic [41]. Wie der Algorithmus von Edmonds und Karp erhöht auch der Algorithmus von Dinic einen gegebenen Fluss in einem Flussnetzwerk über die Betrachtung kürzester Wege im Restnetzwerk. Doch Dinic konstruiert zunächst aus dem Restnetzwerk ein sogenanntes Niveaunetzwerk, welches man erhält, indem man alle Kanten streicht, die nicht zu kürzesten Wegen von der Quelle zur Senke gehören. Ein Niveaunetzwerk kann durch Breitensuche aus dem Restnetzwerk berechnet werden. Anstatt in jedem Durchlauf nur einen kürzesten Weg aus dem Restnetzwerk zu entfernen, wird bei dem Algorithmus von Dinic ein sogenannter Sperrfluss im Niveaunetzwerk konstruiert, der alle kürzesten Wege blockiert. Dafür muss ein Sperrfluss auf jedem Weg im Niveaunetzwerk mindestens eine Kante füllen. Ist ein Sperrfluss berechnet, so ist die Länge kürzester Wege im Restnetzwerk länger geworden. Für den nächsten Durchlauf wird ein neues Niveaunetzwerk berechnet und ein weiterer Sperrfluss konstruiert. Die Länge kürzester Wege erhöht sich damit in jeder Iteration um mindestens 1, bis ein maximaler Fluss berechnet ist. Formal definieren wir ein Niveaunetzwerk wie folgt.

Definition 3.3.1 (Niveaunetzwerk)

Definition 3.3.1 (Niveaunetzwerk)
Sei $G = (E, F, c, q, s)$ *ein Flussnetzwerk und* $G_f = (E, F_f, c_f, q, s)$ *das Restnetzwerk zu* G *und einem Fluss* f. *Für ein* $i \in \mathbb{N}_0$ *betrachten wir die Menge* $V_i = \{v \in E \mid$ *die Länge eines kürzesten Wegs von* q *nach* v *in* G_f *ist* $i\}$.
Das Flussnetzwerk G_f^N *mit der Knotenmenge* E, *Quelle* q *und Senke* s, *der Kantenmenge* $F_f^N = \{(u, v) \in F_f \mid \exists\, i \in \mathbb{N}_0$, *so dass* $(u, v) \in (V_i \times V_{i+1})\}$ *und Kapazitätsfunktion* $c_f|_{F_f^N}$ *heißt Niveaunetzwerk zu* G *und* f.

Ein Weg von der Quelle zur Senke im Niveaunetzwerk ist immer ein kürzester flussvergrößernder Weg und damit ergibt sich der Algorithmus von Dinic. Beginne mit dem Nullfluss in einem Flussnetzwerk G und, solange Wege von der Quelle zur Senke im Restnetzwerk G_f existieren, berechne das Niveaunetzwerk G_f^N, finde einen Sperrfluss f_s in G_f^N und erhöhe den Fluss in G um f_s.

Dinic beschreibt eine einfach Möglichkeit einen Sperrfluss in einem Niveaunetzwerk zu berechnen. Mit Hilfe von Tiefensuche werden Wege von der Quelle zur Senke im Niveaunetzwerk gesucht und diese gefüllt, bis ihre Kapazitäten ausgeschöpft sind.

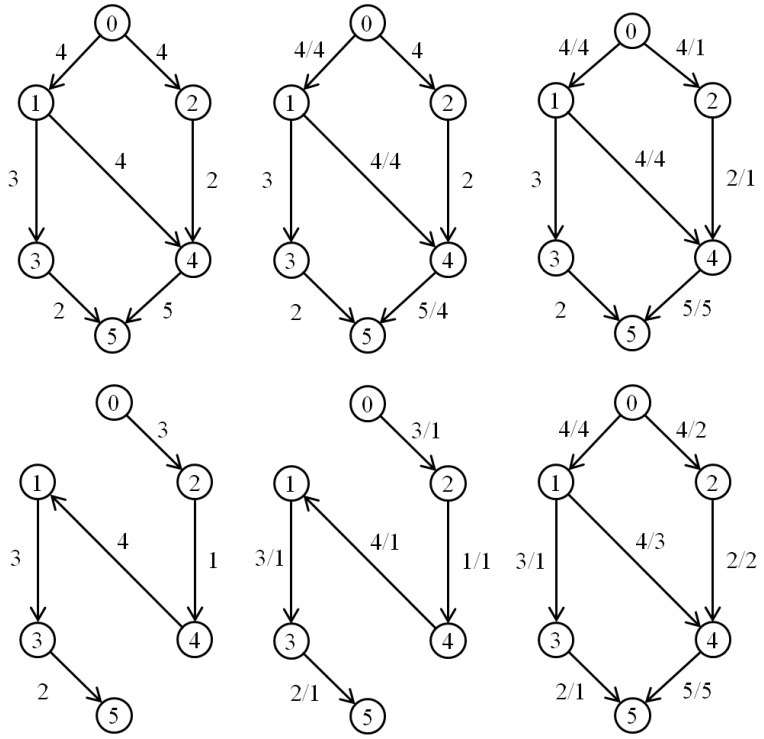

Abbildung 12: Der Algorithmus von Dinic.

Abbildung 12 zeigt links oben das Flussnetzwerk unseres Ausschnitts des Mississippi. Der Algorithmus von Dinic beginnt mit dem Nullfluss und somit ist das Flussnetzwerk zugleich das erste Restnetzwerk. In diesem Beispiel haben dort alle Wege von der Quelle zur Senke die Länge drei. Aus diesem Grund ist das erste Restnetzwerk auch das erste Niveaunetzwerk. In diesem wird nun ein Sperrfluss konstruiert. Über Tiefensuche findet der Algorithmus einen Weg über die Knoten 1 und 4 zur Senke. Der Wert des Flusses wird auf diesem Weg maximal erhöht (Abbildung 12 oben Mitte). Damit ist die Kapazität der Kante $(0, 1)$ erschöpft und es existiert nur noch ein Weg von Quelle zur Senke im Niveaunetzwerk. Abbildung 12 zeigt oben rechts den Zustand, nachdem auch dieser Weg mit dem maximal möglichen Fluss gefüllt ist. In dieser Situation ist ein Sperrfluss im Niveaunetzwerk konstruiert. Im nächsten Schritt wird das Niveaunetzwerk neu berechnet. Dazu wird das Restnetzwerk konstruiert und alle Kanten entfernt, die nicht auf einem kürzesten Weg zur Senke liegen. Das Ergebnis ist in Abbildung 12 links unten abgebildet. Dieses Niveaunetzwerk enthält nur noch einen Weg, der in Abbildung 12 unten Mitte ausgeschöpft wird. An dieser Stelle ist der zweite Sperrfluss berechnet. Abbildung 12 zeigt rechts unten den berechneten Fluss. Bei dem Versuch, ein weiteres Niveaunetzwerk zu berechnen, stellt man fest, dass das Restnetzwerk keinen Weg von der Quelle zur Senke mehr enthält, und der Algorithmus von Dinic terminiert.

Dinic konnte beweisen, dass dieser Algorithmus nach spätestens $|E| - 1$ Sperrfluss-Berechnungen terminiert und die Länge eines kürzesten Wegs im Restnetzwerk durch jede Sperrfluss-Berechnung um mindestens Eins wächst [41]. In jedem dieser Durchläufe wird zuerst das Niveaunetzwerk in $O(|F|)$ Schritten durch Breitensuche und danach ein Sperrfluss in $O(|E| \cdot |F|)$ Schritten berechnet. Insgesamt ist die Laufzeitkomplexität des Algorithmus von Dinic damit in $O((|E| - 1) \cdot (|F| + |E| \cdot |F|)) = O(|E|^2 \cdot |F|)$.
Algorithmus 3.3.1 beschreibt unsere konkrete Implementierung des Algorithmus von Dinic in Pseudocode.

ALGORITHMUS 3.3.1 (ALGORITHMUS VON DINIC)

Algorithmus 3.3.1 (Algorithmus von Dinic)

Eingabe: *Flussnetzwerk* G

Ausgabe: *Wert w eines maximalen Flusses*

```
1   n ← Anzahl der Knoten
2   N ← Nachbarschaften in G
3   w ← 0
4   WHILE (Niveau ← Niveau im Restnetzwerk(G,F,N))
5   {
6       WHILE (Weg ← Weg im Niveaunetzwerk
              (G,F,N,Niveau))
7     {
8           Δ ← ∞, u ← n − 1
9           WHILE (v ← Weg[u] ⩾ 0)
10          {
11              Δ ← Minimum(Δ, G[v][u] − F[v][u])
12              u ← v
13          }
14          u ← n − 1
15          WHILE (Weg[u] ⩾ 0)
16          {
17              F[Weg[u]][u] +← Δ
18              F[u][Weg[u]] −← Δ
19              u ← Weg[u]
20          }
21          w +← Δ
22      }
23  }
24  RETURN w
```

In Algorithmus 3.3.1 ist die Eingabe erneut ein Flussnetzwerk G und die Ausgabe der Wert w eines maximalen Flusses in G. In Zeile 4 gibt die Funktion Niveau im Restnetzwerk entweder null oder, falls ein Weg von der Quelle zur Senke existiert, eine Liste Niveau mit den Entfernungen aller Knoten zur Quelle im Restnetzwerk zurück. Diese Funktion ist eine Breitensuche im Restnetzwerk $G - F$ und damit ähnlich zu Algorithmus 3.2.2. In dieser Implementierung des Algorithmus von Dinic verzichten wir auf eine explizite Konstruktion des Restnetzwerks und des Niveaunetzwerks. Das Restnetzwerk ergibt sich wieder als $G - F$ und das Niveaunetzwerk enthält alle Kanten von $G - F$, bei denen der Wert der Liste Niveau an der Position des Startknotens der Kante einen um eins kleineren Wert hat als der Wert in der Liste Niveau an der Position des Endknotens der Kante. Für jedes Niveaunetzwerk muss der Algorithmus nun einen Sperrfluss konstruieren

(Zeile 6-21). Dazu gibt die Funktion *Weg im Niveaunetzwerk* 3.3.2 entweder null oder, falls dieser existiert, einen Weg von Quelle zur Senke im Niveaunetzwerk zurück. Da für diesen Weg *Weg* das Niveaunetzwerk durchsucht wird, ist *Weg* automatisch ein kürzester Weg im Restnetzwerk. In Zeile 8-21 wird analog zu Algorithmus 3.2.1 der Fluss entlang *Weg* um Δ erhöht. Die Suche nach Wegen im Niveaunetzwerk wird so lange fortgesetzt, bis kein solcher Weg mehr existiert und ein Sperrfluss berechnet ist. In diesem Fall werden die neuen Entfernungen zur Quelle für das nächste Niveaunetzwerk berechnet (Zeile 4). Ist die Senke von der Quelle aus im Restnetzwerk nicht mehr erreichbar, wird in Zeile 24 der Wert w des maximalen Flusses zurückgegeben.

Algorithmus 3.3.2 (Weg im Niveaunetzwerk)

ALGORITHMUS 3.3.2 (WEG IM NIVEAUNETZWERK)

Eingabe: *Restnetzwerk* $G - F$, Niveau *aller Knoten in* $G - F$, *Nachbarschaften* N

Ausgabe: *Weg* Weg *durchs Niveaunetzwerk*

```
1   Weg[0] ← −1
2   WHILE (true)
3   {
4       suchen ← true
5       WHILE (suchen AND Nummer[u] < |N[u]|)
6       {
7           v ← N[u][Nummer[u]];
8           IF (Niveau[v] = Niveau[u] + 1 AND
              G[u][v] − F[u][v] > 0)
9           {
10              Weg[v] ← u
11              suchen ← false
12              Nummer[u]++
13              u ← v
14              IF (v = n − 1) RETURN Weg
15          }
16          ELSE Nummer[u]++
17      }
18      IF (suchen)
19      {
20          Niveau[u] ← ∞
21          u ← Weg[u]
22          IF (u = −1) RETURN null
23      }
24  }
```

Der Algorithmus *Weg im Niveaunetzwerk* 3.3.2 implementiert eine Tiefensuche im Niveaunetzwerk. In Zeile 4 beginnt die eigentliche Tiefensuche. Die Liste Nummer beschreibt die Anzahl der bereits untersuchten Nachbarn eines Knotens u. Solange noch nicht alle Nachbarn betrachtet wurden (Zeile 5), wird in Zeile 8 untersucht, ob die entsprechende Kante eine Kante des Niveaunetzwerks ist. Ist dies der Fall, wird der Weg *Weg* um diese Kante verlängert (Zeile 10) und der Endknoten der Kante zum neuen aktuellen Knoten (Zeile 13). In jedem Fall wird die Anzahl der durchsuchten Möglichkeiten für den betrachteten Knoten um Eins erhöht (Zeile 12, Zeile 16). Wenn der Algorithmus einen Schritt zum nächsten Knoten machen kann, wird die Variable suchen zunächst auf false gesetzt. Ist der neue, aktuelle Knoten die

Senke, wird der konstruierte Weg *Weg* zurückgegeben (Zeile 14). Ist die Senke noch nicht erreicht, wird der neue Knoten u auf die gleiche Weise bearbeitet. Ist es nicht möglich, den Weg von einem Knoten u aus zu verlängern, wird der letzte Schritt des Wegs rückgängig gemacht (Zeile 18-23) und der Knoten aus dem Niveaunetzwerk entfernt (Zeile 20). Dies sorgt für eine deutlich bessere Effizienz, da dieser Knoten nicht bei jeder weiteren Iteration der Tiefensuche in diesem Niveaunetzwerk erneut durchsucht werden muss. Wird die Quelle aus dem Niveaunetzwerk entfernt, gibt die Funktion `null` zurück, da kein Weg von der Quelle zur Senke mehr existiert.

3.4 DER FORWARD-BACKWARD-PROPAGATION ALGORITHMUS

In diesem Abschnitt betrachten wir den Fluss-Maximierungs-Algorithmus von Malhotra, Kumar, Maheshwari [70]. Bekannt ist dieser Algorithmus unter dem Namen Forward-Backward-Propagation Algorithmus. Der Algorithmus ähnelt im Aufbau dem Algorithmus von Dinic, wobei er eine elegantere Methode verwendet, um einen Sperrfluss in einem Niveaunetzwerk zu berechnen.
Die Idee der Forward-Backward-Propagation ist es, einen Sperrfluss in einem Niveaunetzwerk zu konstruieren, indem man Fluss an einem Knoten im Niveaunetzwerk entstehen lässt und diesen anschließend zur Senke und zur Quelle hin weiterführt. Das Weiterführen des Flusses in Richtung Senke nennt man die Forward-Phase des Algorithmus und das Weiterführen des Flusses in Richtung Quelle nennt man die Backward-Phase. Dadurch, dass man den Fluss an einem Knoten im Niveaunetzwerk entstehen lässt, kann man diesen Knoten im Niveaunetzwerk saturieren. Saturieren bedeutet, dass entweder der mögliche Zufluss zu einem Knoten oder der mögliche Abfluss aus einem Knoten durch einen Fluss ausgeschöpft ist und somit kein flussvergrößernder Weg über diesen Knoten führen kann. Saturierte Knoten können aus dem Niveaunetzwerk entfernt werden. Sind alle Knoten in einem Niveaunetzwerk saturiert, kann es keinen flussvergrößernden Weg mehr geben und ein Sperrfluss ist berechnet.
Bei dieser Konstruktion eines Sperrflusses ist es entscheidend, den Fluss nur an Knoten entstehen zu lassen, von denen sich der zusätzliche Fluss auch bis zur Senke und Quelle hin weiterführen lässt. Dazu betrachtet man das Potential eines Knotens. Das Potential eines Knotens im Niveaunetzwerk ist das Minimum der Summe der Restkapazitäten der in den Knoten eingehenden Kanten und der Summe der Restkapazitäten der aus dem Knoten ausgehenden Kanten. Erhöht man Fluss an einem Knoten mit minimalem Potential, so kann dieser Fluss immer zur Senke und zur Quelle hin weitergeführt werden, da jeder andere Knoten mindestens genau so viel Potential besitzt und damit den Fluss aufnehmen und auch weiterleiten kann.
Abbildung 13 zeigt links oben erneut das Flussnetzwerk unseres Abschnittes des Mississippi. Wieder entspricht es in der ersten Iteration dem Restnetzwerk und dem Niveaunetzwerk. Im ersten Schritt wird das Potential jedes Knotens bestimmt. Ein minimales Potential mit dem Wert 2 besitzt zum Beispiel der Knoten 2. In der Forward-Phase wird also vom Knoten 2 aus ein Fluss mit Wert 2 in Richtung Senke konstruiert. Dazu erreicht man zuerst Knoten 4 und dann die Senke. In der Backward-Phase wird die Quelle direkt erreicht und ein Fluss

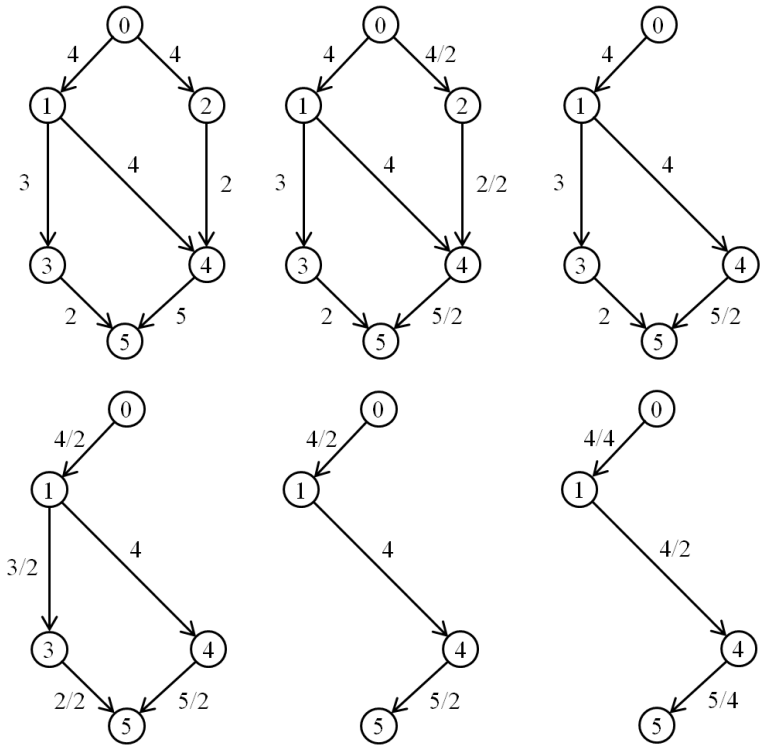

Abbildung 13: Die Forward-Backward-Propagation.

mit Wert 2 ist berechnet. Da dieser Fluss so gewählt ist, dass er den Knoten 2 saturiert, kann dieser, wie in Abbildung 13 rechts oben abgebildet, aus dem Niveaunetzwerk entfernt werden. Erneut wird ein Knoten mit minimalem Potential bestimmt, hier Knoten 3. Abbildung 13 links unten zeigt die Situation nach einer erneuten Forward-Phase und Backward-Phase. Unten in der Mitte von Abbildung 13 wurde Knoten 3 aus dem Niveaunetzwerk entfernt. Nach einer dritten Iteration der Forward- und Backward-Phase von der Quelle aus ist die Endsituation des Algorithmus erreicht (Abbildung 13 unten rechts). Insgesamt wurden drei Iterationen benötigt, um einen Sperrfluss mit Wert 6 zu konstruieren. Es wird ein neues Niveaunetzwerk konstruiert, wobei der Algorithmus terminiert, da in dieser Situation kein Weg von der Quelle zur Senke mehr existiert.

Ein Beispiel eines Flussnetzwerks, bei dem der Forward-Backward-Propagation Algorithmus Vorteile gegenüber dem Algorithmus von Dinic besitzt, ist in Abbildung 14 abgebildet. Von der Quelle zur Senke existieren vier verschiedene Wege, die der Algorithmus von Dinic nacheinander aus dem Niveaunetzwerk entfernen würde. Der Forward-Backward-Propagation Algorithmus beginnt in diesem Beispiel mit dem Knoten 1, einem Knoten mit minimalem Potential. In der Forward-Phase wird damit ein Fluss mit dem Wert 2 über den Knoten 2 und danach über die beiden Knoten 3 und 4 zur Senke weitergeführt. Damit werden in einer Iteration beide Wege über den Knoten 2 aus dem Restnetzwerk entfernt. Danach werden in zwei weiteren Iterationen die beiden verbleibenden Wege über die Knoten 3 und 4 aus dem Niveaunetzwerk entfernt. Der Algorithmus von Dinic würde die Kanten $(0,1)$ und $(1,2)$ zweimal betrachten, der Forward-Backward-Propagation Algorithmus betrachtet dagegen jede dieser Kanten nur einmal wäh-

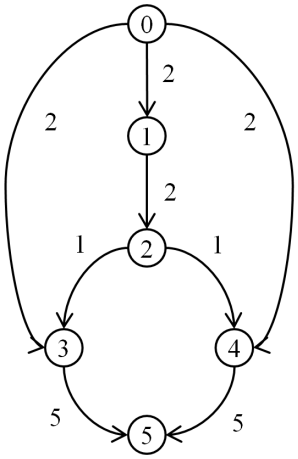

Abbildung 14: Flussnetzwerk, bei dem der Forward-Backward-Propagation Algorithmus vorteilhaft ist.

rend der ersten Iteration. Obwohl der Unterschied an dieser Stelle nur gering ist, kann man mit genau diesem Argument eine für den Forward-Backward-Propagation Algorithmus bessere Schlechtester-Fall Komplexität als für den Algorithmus von Dinic beweisen.
Um einen Sperrfluss im Niveaunetzwerk zu berechnen, wird Fluss in Höhe des Potentials eines Knotens zuerst nach vorne, dann nach hinten durch das Niveaunetzwerk geleitet. Eine Forward- mit zugehöriger Backward-Phase nennt man Propagations-Phase. Man kann die Knoten im Niveaunetzwerk leicht nach ihrer Entfernung zur Senke hin ordnen und betrachtet dann in jeder Propagations-Phase jeden Knoten nur einmal. Betrachten wir exemplarisch die Forward-Phase. Für jeden Knoten führt der Algorithmus den Fluss über ausgehende Kanten des Knotens weiter. Dafür betrachtet man die ausgehenden Kanten der Reihe nach, bis der weiterzuführende Fluss verteilt ist. Dabei entstehen drei Mengen von Kanten. Die ersten Kanten werden durch das Weiterführen des Flusses in ihrer Kapazität ausgeschöpft. Diese Kanten werden sofort aus dem Niveaunetzwerk entfernt und müssen in einer späteren Iteration nicht mehr betrachtet werden. Die folgende Kante nimmt den restlichen Fluss auf, wird aber nicht entfernt, da ihre Kapazität noch nicht ausgeschöpft ist. Da an dieser Stelle kein Fluss mehr zu verteilen ist, werden die restlichen Kanten erst in einer späteren Iteration betrachtet, verursachen also in dieser Iteration keine Laufzeit. Um einen Sperrfluss zu berechnen, wird also jede Kante einmal betrachtet während sie saturiert wird. Addieren muss man die Zahl der Kanten, die in jeder Propagations-Phase betrachtet, aber dabei nicht saturiert werden. Dies gilt für maximal $|E|$ Kanten, da dies für jeden Knoten nur einmal in jeder Propagations-Phase geschehen kann. Da der Algorithmus maximal $|E|$ Propagations-Phasen benötigt, bis alle Knoten saturiert sind, ist die Laufzeit der Konstruktion eines Sperrflusses in $O(|F| + |E|^2) = O(|E|^2)$. Mit dem gleichen Argument wie bei der Betrachtung der Laufzeit des Algorithmus von Dinic, benötigt der Forward-Backward-Propagation Algorithmus maximal $|E| - 1$ Konstruktionen eines Sperrflusses. Insgesamt ergibt sich damit für den Algorithmus eine Laufzeit in $O(|E|^3)$.
Algorithmus 3.4.1 beschreibt unsere Implementierung des Forward-Backward-Propagation Algorithmus in Pseudocode.

Algorithmus 3.4.1 (Forward-Backward-Propagation Algorithmus)

ALGORITHMUS 3.4.1 (FORWARD-BACKWARD-PROPAGATION ALGORITHMUS)

Eingabe: *Flussnetzwerk* G

Ausgabe: *Wert w eines maximalen Flusses*

```
1   n ← Anzahl der Knoten
2   N ← Nachbarschaften in G
3   w ← 0
4   WHILE (A, E, P, Knoten, a ← Niveaunetzwerk(G, F, N))
5   {
6       WHILE (P[1][0] > 0 AND P[0][n − 1] > 0)
7       {
8           p, u, a ← Minimales Potential(P, a, Knoten)
9           w + ← p
10          Forward(G, F, A, p, u)
11          Backward(G, F, E, p, u)
12          Update Potential(P, E, A, G, F, u)
13      }
14  }
15  RETURN w
```

In Algorithmus 3.4.1 ist die Eingabe erneut ein Flussnetzwerk G und die Ausgabe der Wert w eines maximalen Flusses in G. In Zeile 4 gibt die Funktion Niveaunetzwerk entweder den Wert null oder, falls ein Weg von der Quelle zur Senke im Restnetzwerk existiert, das Niveaunetzwerk zurück. In dieser Implementierung wird das Niveaunetzwerk anders als beim Algorithmus von Dinic explizit konstruiert, um der Überlegung zur Schlechtester-Fall-Komplexität folgend Kanten im Niveaunetzwerk so selten wie möglich betrachten zu müssen. Das Niveaunetzwerk wird dazu durch Breitensuche konstruiert und setzt sich aus zwei Listen von Listen A und E für die einem Knoten ausgehenden bzw. eingehenden Kanten zusammen. Die Matrix P ist eine $(3 \times |E|)$-Matrix, sie beschreibt für jeden Knoten zuerst den noch möglichen Zufluss, dann den noch möglichen Abfluss und an dritter Stelle das Potential. Die Liste Knoten enthält alle Knoten des Niveaunetzwerks. Da die Knoten im Verlauf des Algorithmus aus dieser Liste gelöscht werden, markiert der Index a einen aktuell letzten Knoten in diesem Array. Werden im Laufe der Sperrflussberechnung Knoten saturiert, tauscht der letzte Knoten in der Liste den Platz mit dem saturierten Knoten und der Index a wird um eins verringert. Nach der Konstruktion des Niveaunetzwerks folgt in Zeile 6-13 die Sperrflussberechnung. Ein Sperrfluss ist berechnet, wenn entweder der mögliche Abfluss der Quelle oder der mögliche Zufluss der Senke den Wert 0 besitzt (Zeile 6). Bis dahin wird in Zeile 8 durch die Funktion Minimales Potential ein Knoten mit minimalem Potential bestimmt. Die Funktion gibt den Wert dieses Potentials p, den Index dieses Knotens u in der Matrix G und die neue Länge a der Liste der Knoten im Niveaunetzwerk zurück. Das Potential p wird in Zeile 7 zu dem maximalen Fluss addiert. Danach beginnt in Zeile 8 die Forward-Phase des Algorithmus. Die Funktion Forward (Algorithmus 3.4.2) setzt den Fluss mit dem Wert p von dem Knoten u aus zur Senke hin fort. Dazu benutzt die Funktion die Liste A, die für jeden Knoten die Menge der ihm ausgehenden Kanten beinhaltet. Analog setzt in Zeile 9 die Funktion Backward den Fluss mit dem Wert p vom Knoten u aus rückwärts mit Hilfe der Matrix E in Richtung Quelle fort. Während der Ausführung der Funktionen Forward und

Backward werden die Potentiale der Knoten auf den Wegen zur Senke und zur Quelle aktuell gehalten, sobald sich der Wert des Flusses auf Kanten ändert. Durch das Entfernen des saturierten Knotens u kann sich das Potential benachbarter Knoten zusätzlich ändern. Diese lokale Veränderung wird in Zeile 10 durch die Funktion Update Potential umgesetzt. Erreicht der Algorithmus Zeile 13, ist ein maximaler Fluss berechnet und dessen Wert w wird zurückgegeben.

ALGORITHMUS 3.4.2 (FORWARD)

Algorithmus 3.4.2 (Forward)

Eingabe: *Niveaunetzwerk* $G - F$ *und* A, *minimales Potential* p *und Knoten* u

Ausgabe:

```
1   Über[u] ← p
2   List[0] ← u
3   kopf ← 0, ende ← 0
4   IF (minP ≠ n − 1)ende++
5   WHILE (kopf ≠ ende)
6   {
7     u ← List[kopf]
8     WHILE (Über[u] > 0)
9     {
10        v ← A[u][PosA[u]]
11        IF (Farbe[v] = 0 AND G[u][v] − F[u][v] > 0)
12        {
13            IF (G[u][v] − F[u][v] > Über[u]) Δ ← Über[u]
14            ELSE Δ ← G[u][v] − F[u][v]
15            F[u][v] + ← Δ
16            Über[v] + ← Δ
17            F[v][u] − ← Δ
18            P[1][u] − ← Δ
19            P[0][v] − ← Δ
20            Über[u] − ← Δ
21            IF (v ≠ n − 1)
22            {
23                List[ende] ← v
24                ende++
25            }
26        }
27        ELSE PosA[u]++
28    }
29    kopf++
30  }
```

Die Funktion Forward 3.4.2 bekommt als Eingabeparameter das Niveaunetzwerk übergeben. Dieses ergibt sich aus dem Restnetzwerk $G - F$ zusammen mit einer Liste aller Kanten des Niveaunetzwerks, die in der Matrix A den Knoten zugeordnet sind, von denen sie ausgehen. Die Liste Über speichert für jeden Knoten einen sogenannten Überschuss. Ein Knoten besitzt Überschuss, wenn in diesen mehr Fluss ein- als ausgeht. Der Algorithmus Forward ordnet zuerst dem Knoten u sein Potential als Überschuss zu (Zeile 1). Das Weiterführen des Überschusses wird über Breitensuche implementiert. Der Startknoten dieser Breitensuche ist der Knoten u (Zeile 2). Die Liste List sorgt dafür, dass die Knoten in einer First-In-First-Out Reihenfolge abgearbeitet werden. Dadurch wird jeder Knoten in der Forward-Phase nur einmal

betrachtet. Dass jede Kante nur ein einziges Mal betrachtet wird, stellt die Liste $PosA$ sicher. Diese speichert für jeden Knoten den Index der ersten nicht saturierten Kante in der Matrix A.

3.5 DER PREFLOW-PUSH ALGORITHMUS

Die bis zu diesem Abschnitt vorgestellten Fluss-Maximierungs-Algorithmen basieren alle auf der Idee, iterativ flussvergrößernde Wege zu suchen und einen Fluss mit Hilfe dieser zu erhöhen, bis ein maximaler Fluss konstruiert ist. Dieser Abschnitt beschreibt nun eine zweite Klasse von Fluss-Maximierungs-Algorithmen, die sogenannten Preflow-Push Algorithmen [63].
Wir stellen uns ein Flussnetzwerk wieder als Kanalsystem vor, das zu Beginn platt auf der Erde liegt. In einem ersten Schritt füllen wir alle Arme, die von der Quelle ausgehen, randvoll mit Wasser und heben das Flussnetzwerk an der Quelle an. Durch die Schräglage beginnt das Wasser nach unten und damit in Richtung Senke zu fließen. Auf dem Weg zur Senke kann es passieren, dass die Kapazität von Kanten zu gering ist, um das nach unten schießende Wasser aufzunehmen. In diesem Fall staut sich das Wasser an Knoten und bildet dort eine Welle. Diese Welle türmt sich auf und versucht, über alternative Wege zur Senke zu fließen. Dabei kann es passieren, dass sich Wasser auf dem Weg nach unten in Sackgassen sammelt. Um das Wasser aus diesen Sackgassen wieder heraus zu bekommen, heben wir die entsprechenden Arme des Flussnetzwerks so lange an, bis das Wasser aus den Sackgassen heraus und wieder Richtung Senke fließt. Durch dieses Verfahren wird die Kapazität des gesamten Flussnetzwerks erschöpft und, sollten trotzdem noch Wellen übrig bleiben, wird das Flussnetzwerk so lange gekippt, bis die übrig gebliebenen Wellen zurück zur Quelle und dort aus dem Flussnetzwerk laufen können. An diesem Punkt haben sich die Wogen geglättet und die Kapazität des Netzwerks ist ausgeschöpft, was bedeutet, dass ein maximaler Fluss konstruiert ist.
Ein Beispiel eines Flussnetzwerks, bei dem ein solcher Preflow-Push-Algorithmus Vorteile gegenüber einem Algorithmus besitzt, der flussvergrößernde Wege benutzt, ist in Abbildung 15 abgebildet. Von der Quelle zur Senke existieren fünf verschiedene Wege, die von einem Algorithmus, der flussvergrößernde Wege benutzt, nacheinander gefüllt werden. Der Algorithmus von Edmonds und Karp und der Algorithmus von Dinic würden diese jeweils durch Tiefensuche von der Quelle, der Algorithmus Forward-Backward-Propagation würde dieselben Wege von den Knoten mit minimalem Potential (Knoten 3, 4, 5, 6 und 7) aus finden. Ein Preflow-Push Algorithmus füllt zunächst die Kante $(0,1)$ mit ihrer maximalen Kapazität 5. Der Zufluss am Knoten 1 ist zu diesem Zeitpunkt 5, während sein Abfluss 0 beträgt. Die dadurch am Knoten 1 vorhandene Welle wird über Kante $(1,2)$ an Knoten 2 weitergegeben, wodurch auch Kante $(1,2)$ gesättigt wird. Die Welle ist nun bei Knoten 2. Dann verteilt sich das Wasser von Knoten 2 aus über die Knoten 3, 4, 5, 6 und 7, bis es von diesen aus die Senke erreicht. Das gemeinsame Anfangsstück der fünf Wege wird in diesem Beispiel nur einmal, anstatt fünfmal betrachtet.
Die Preflow-Push Algorithmen arbeiten nicht mit einem Fluss, sondern zunächst mit einem Quasi-Fluss (im Englischen Preflow). Der Quasi-Fluss erfüllt zwar die Kapazitätsbeschränkung auf jeder Kante des Flussnetzwerks, aber nicht die Flusserhaltung. Der Zufluss eines Kno-

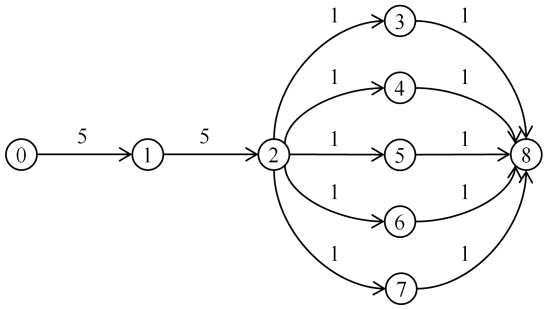

Abbildung 15: Ein Flussnetzwerk, bei dem ein Preflow-Push Algorithmus vorteilhaft ist.

tens kann größer sein als sein Abfluss. Die Differenz der beiden Werte, die Welle in unserer informalen Beschreibung, wird der Überschuss des Knotens genannt.

DEFINITION 3.5.1 (QUASI-FLUSS) *Definition 3.5.1 (Quasi-Fluss)*
Ein Quasi-Fluss in einem Flussnetzwerk $G = (E, F, c, q, s)$ *ist eine Funktion* $f : F \to \mathbb{N}_0$ *für die zwei Eigenschaften gelten:*

(Flussüberschuss) *Für alle Knoten* $u \in E \setminus \{q, s\}$ *gilt*

$$\sum_{(v,u)\in F} f(v,u) \geqslant \sum_{(u,v)\in F} f(u,v).$$

(Kapazitätsbeschränkung) *Für alle Kanten* $e \in F$ *gilt* $f(e) \leqslant c(e)$.

Für ein Flussnetzwerk G*, einen Quasi-Fluss* f *und einen Knoten* u *heißt der Wert* $\ddot{u}(u) := \sum_{(v,u)\in F} f(v,u) - \sum_{(u,v)\in F} f(u,v)$ *der Überschuss des Knotens* u*.*

Die Preflow-Push Algorithmen beginnen mit einem Quasi-Fluss, dessen Wert größer oder gleich dem Wert eines maximalen Flusses ist, und bauen den Überschuss aller Knoten ab, bis kein Knoten mehr positiven Überschuss besitzt. In diesem Fall erfüllt der Quasi-Fluss auch die Flusserhaltung und ist zu einem Fluss geschrumpft. Maximal ist dieser Fluss, da nach Ablauf eines Preflow-Push Algorithmus kein Weg von der Quelle zur Senke im Restnetzwerk mehr existiert. Um dieses sicher zu stellen, wird dafür gesorgt, dass die sogenannte Höhe jedes Knotens im Flussnetzwerk immer kleiner als der kürzeste Weg von diesem Knoten zur Senke im Restnetzwerk lang ist. Dazu definieren wir eine Höhenfunktion auf einem Restnetzwerk.

DEFINITION 3.5.2 (HÖHENFUNKTION) *Definition 3.5.2 (Höhenfunktion)*
Sei $G = (E, F, c, q, s)$ *ein Flussnetzwerk. Eine Funktion* $h : E \to \mathbb{N}_0$ *ist eine Höhenfunktion, falls folgende zwei Eigenschaften gelten:*

$h(s) = 0$ *und*

$h(u) \leqslant h(v) + 1$*, falls eine Kante* (u, v) *im Restnetzwerk existiert.*

Durch diese Definition ist die Höhe eines Knotens immer kleiner oder gleich seiner Entfernung zur Senke im Restnetzwerk. Die Höhe aller Knoten steuert die Richtung, in der sich später Überschuss verteilt.
Jeder Preflow-Push Algorithmus beginnt mit einer Initialisierungsphase. Zuerst werden alle Kanten, die von der Quelle ausgehen, ihrer Kapazität entsprechend mit Fluss gefüllt. Damit entsteht Überschuss an den der Quelle benachbarten Knoten. Dann wird eine Höhenfunktion berechnet, wobei die einfachste Möglichkeit hierfür ist, jedem Knoten die Höhe 0 zu geben. Nachdem alle der Quelle ausgehenden Kanten gefüllt sind, existiert im Restnetzwerk kein Weg mehr von der Quelle zur Senke. Aus diesem Grund darf man als letzten Schritt der Initialisierungsphase die Höhe der Quelle auf den Wert der Anzahl aller Knoten setzen.
Nach der Initialisierung beginnt der Hauptteil des Algorithmus. Solange ein Knoten u mit positivem Überschuss existiert, werden die Funktionen `Push` und `Relable` ausgeführt. Dazu wird der Überschuss von u auf Kanten (u, v) im Restnetzwerk verteilt, falls die Höhe von v kleiner ist als die Höhe von u. Dieses Verteilen nennt man `Push`. Ist es nicht möglich, den Überschuss eines Knotens zu verteilen, wird die Höhe des Knotens um Eins erhöht. Das Anheben der Knoten nennt man `Relable`. Dies geschieht so lange, bis der Überschuss des Knotens wieder zurück zur Quelle und somit aus dem Netzwerk fließt. Ein Abbauen des Überschusses ist also immer möglich. Die Höhe eines Knotens wird in jedem Schritt zunächst auf den Wert der Höhe des tiefsten Nachbarn plus Eins gesetzt. Durch dieses Vorgehen wird sicher gestellt, dass die Höhe aller Knoten immer die Eigenschaften einer Höhenfunktion erfüllt.
Existiert an keinem Knoten des Flussnetzwerks mehr Überschuss, ist der Quasi-Fluss ein Fluss. Die Höhen aller Knoten ergeben eine Höhenfunktion und sind damit kleiner als die Länge des kürzesten Wegs zwischen sich und der Senke. Da die Höhe der Quelle nach Ablauf des Algorithmus immer noch gleich der Anzahl aller Knoten ist, hat das Restnetzwerk nie einen Weg von der Quelle zur Senke enthalten. Der konstruierte Fluss ist damit ein maximaler Fluss.
In Abbildung 16 sehen wir die Arbeitsweise eines Preflow-Push Algorithmus an unserem Beispiel. An den Kanten steht zuerst deren Kapazität, dann der Wert des Flusses über die Kante. Die Höhe der Knoten ist in den abgerundeten Rechtecken an den Knoten dargestellt. Das Flussnetzwerk links oben zeigt die Situation nach der Initialisierungsphase. Die Höhe der Quelle ist 6, die Kanten $(0, 1)$ und $(0, 2)$ sind gesättigt. Überschuss existiert an den Knoten 1 und 2. Das Flussnetzwerk rechts oben zeigt die Situation, nachdem die Knoten 1 und 2 angehoben wurden und Fluss von ihnen auf die Knoten 3 und 4 gepusht wurde. Während Knoten 1 keinen positiven Überschuss mehr besitzt, kann der Überschuss an Knoten 2 auf 2 verringert werden. Das Flussnetzwerk links unten zeigt die Situation, nachdem auch Knoten 3 und 4 angehoben wurden. Besitzt die Höhe des Knotens 3 den Wert 1, wird die Kante $(3, 5)$ gesättigt. Wird die Höhe weiter auf den Wert 2 erhöht, wird der restliche Überschuss von 1 rückwärts wieder an Knoten 1 gepusht. Der gesamte Überschuss an Knoten 4 kann zur Senke abfließen. Positiver Überschuss befindet sich noch an den Knoten 1 und 2. Das Flussnetzwerk rechts unten zeigt die Situation, nachdem die Knoten 1 und 2 weiter erhöht wurden. Der Überschuss an Knoten 1 kann danach zu Knoten 4 und weiter zur Senke abfließen. Da von Knoten 2 aus nun im Restnetzwerk nur noch die Quelle erreichbar ist, muss

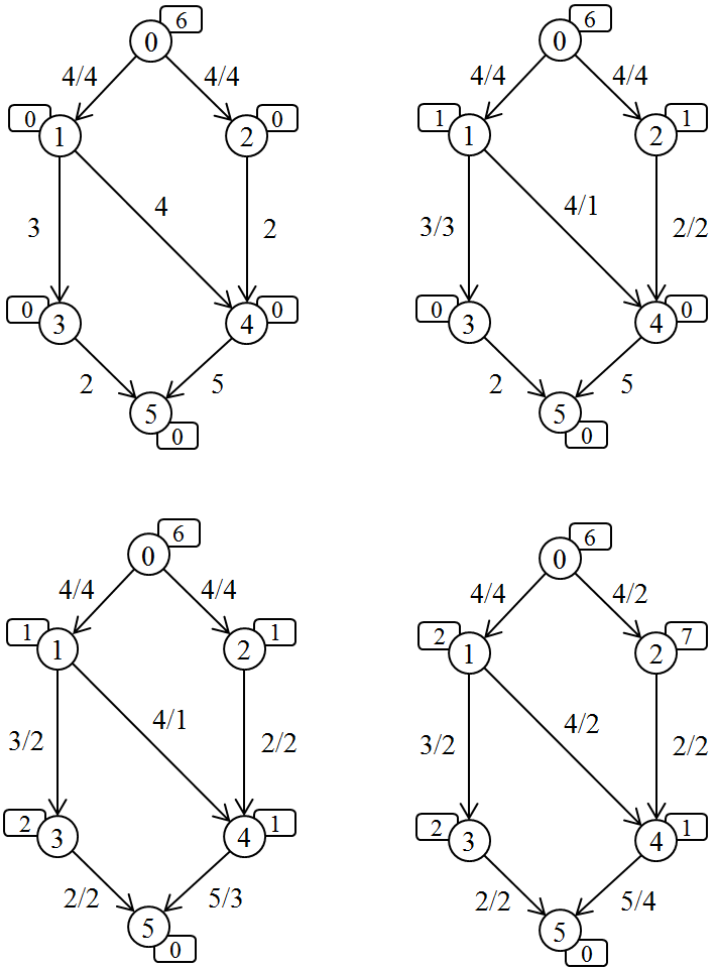

Abbildung 16: Der Preflow-Push Algorithmus.

die Höhe des Knotens 2 auf 7 erhöht werden, bis der Überschuss zur Quelle und damit aus dem Flussnetzwerk abfließen kann. Letztendlich ist ein maximaler Fluss konstruiert.
Die Laufzeit des eben beschriebenen generischen Preflow-Push Algorithmus ist in $O(|E|^2 \cdot |F|)$, falls man die Knoten, an denen ein Überschuss entsteht, in beliebiger Reihenfolge abarbeitet. Bearbeitet man die Knoten mit positivem Überschuss in der Reihenfolge, in der an ihnen Überschuss entstanden ist, verbessert sich die Schlechteste-Fall Komplexität sogar auf $O(|E|^3)$. Die Begründung der Laufzeiten bedarf einer etwas längeren Argumentation, auf die wir in dieser Arbeit verzichten wollen. Sie ist z.B. in [4] nachzulesen.
Wichtiger ist es an dieser Stelle, ein Beispiel zu betrachten, bei dem ein Preflow-Push Algorithmus in der beschrieben Form erhebliche Nachteile aufweist, die wir anschließend mit einer sogenannten Gap-Heuristik lösen wollen.
In Abbildung 17 besteht das Flussnetzwerk aus einer einfachen Sequenz von Knoten. Die erste Zeile zeigt den Zustand nach der Initialisierungsphase. Die zweite Zeile zeigt den Zustand, nachdem der Überschuss mit dem Wert 5 an Knoten 4 angekommen ist. Nur ein Teil des Überschusses kann an der Senke abfließen. Da damit die Kante $(4, 5)$ aus dem Restnetzwerk verschwindet, wird die Höhe von Knoten 4 auf 2

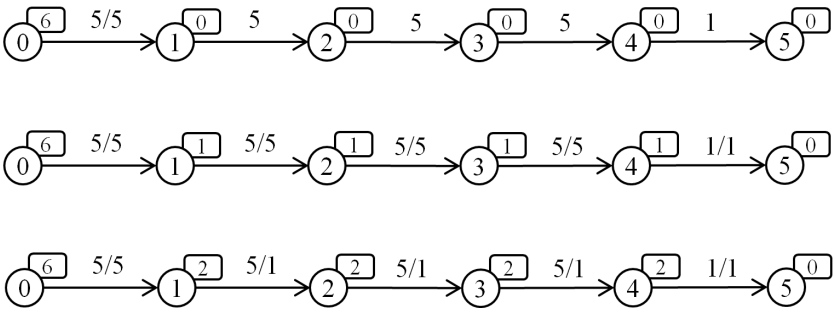

Abbildung 17: Der Preflow-Push Algorithmus in einem ungünstigen Beispiel.

erhöht und der Überschuss begibt sich wieder auf den Weg in Richtung Quelle. Die dritte Zeile des Bildes zeigt die Situation, in der der Überschuss wieder an Knoten 1 angekommen ist. Aufgrund der Höhe der Quelle kann jetzt aber der Überschuss das Flussnetzwerk noch nicht verlassen. Die Höhe des Knotens 1 wird auf 3 erhöht und der Überschuss wird in dieser Situation wieder Richtung Knoten 4 geschoben. Bildlich gesprochen schwappt der Überschuss so lange hin und her, bis die Knoten 1 bis 4 eine Höhe von 7 erreicht haben und der Überschuss das Flussnetzwerk verlassen kann.
Die Lösung dieses Laufzeit fressenden Problems kann man gut in der dritten Zeile von Abbildung 17 erkennen. Die Quelle hat dort die Höhe 6, die Senke Höhe 0 und alle anderen Knoten eine Höhe von 2. Da die Höhe die Eigenschaften einer Höhenfunktion erfüllt, kann kein Knoten mit der Höhe zwei im Restnetzwerk direkt mit der Senke verbunden sein. Existiert ein Weg zur Senke, so existiert auch ein Knoten mit Höhe 1. Da kein solcher Knoten existiert, kann man den Überschuss von allen Knoten, die eine Höhe größer als 1 haben, löschen. Dieser Überschuss wird die Senke nie mehr erreichen. Das ist die Grundidee der Gap-Heuristik. Als Gap, englisch für Lücke, wird ein Sprung in der Menge aller Werte der Höhen von Knoten bezeichnet. Existiert ein solches Gap, so kann der Überschuss auf Knoten mit einer Höhe größer als dieser Wert entfernt werden. Ein Preflow-Push Algorithmus mit Gap-Heuristik würde dadurch in der dritten Zeile von Abbildung 17 enden. In dieser Situation ist zwar noch kein Fluss konstruiert, aber man kann bereits den Wert eines maximalen Flusses an dem Zufluss der Senke ablesen.
Algorithmus 3.5.1 beschreibt unsere konkrete Implementierung des Preflow-Push Algorithmus in Pseudocode.

ALGORITHMUS 3.5.1 (PREFLOW-PUSH ALGORITHMUS)

Algorithmus 3.5.1 (Preflow-Push Algorithmus)

Eingabe: *Flussnetzwerk* G

Ausgabe: *Wert w eines maximalen Flusses*

```
1   n ← Anzahl der Knoten
2   N ← Nachbarschaften in G
3   w ← 0
4   Höhe, Liste, kopf, ende ← Höhenfunktion(G, N)
5   Höhe[0] ← n
6   FOR (i ← 0, i < |N[u]|, i++)
7   {
8       v ← N[u][i]
9       IF (G[u][v] > 0)
10      {
11          F[u][v] ← G[u][v]
12          F[v][u] ← −G[u][v]
13          Über[v] ← G[u][v]
14      }
15  }
16  WHILE (kopf ≠ ende)
17  {
18  u ← Liste[kopf]
19  IF (!Gap[u])
20  {
21      i ← 0
22      WHILE (Über[u] > 0 AND i < |N[u]|)
23      {
24      v ← N[u][i]
25      IF (Höhe[u] > Höhe[v] AND G[u][v] − F[u][v] > 0)
26      {
27          IF (Farbe[v] = 0 AND v ≠ n − 1 AND v ≠ 0)
28          {
29              Liste[ende] ← v
30              Farbe[v] ← 2
31              IF (ende = n − 1) ende ← 0
32              ELSE ende++
33          }
34          Δ ← Minimum(G[u][v] − F[u][v], Uber[u])
35          F[u][v]+ ← Δ
36          F[v][u]− ← Δ
37          Über[u]− ← Δ
38          Über[v]+ ← Δ
39      }
40      i++
41      }
42      IF (Über[u] > 0)
43      {
44          Anzahl[Höhe[u]]−−
45          IF (Anzahl[Höhe[u]] = 0 AND !Gap[u])
46          {
47              Gap, Höhe ← Gap(Anzahl, Gap, Höhe)
48          }
49          Höhe[u] ← Berechne Nächste Höhe(N, Höhe, u)
50          Anzahl[Höhe[u]]++
51      }
52  }
53  IF (Über[u] = 0 OR Gap[u])
54  {
55      Farbe[u] ← 0
```

```
56          IF (kopf = n − 1) kopf ← 0
57          ELSE kopf++
58      }
59      }
60      w ← Über[n − 1]
61      RETURN w
```

In Algorithmus 3.5.1 ist die Eingabe ein Flussnetzwerk G und die Ausgabe der Wert w eines maximalen Flusses in G. In Zeile 4 berechnet die Funktion Höhenfunktion die maximale Anfangshöhe jedes Knotens. Die Funktion ist als rückwärtsgerichtete Breitensuche implementiert, wobei die dabei besuchten Knoten in umgekehrter Reihenfolge in der Liste Liste gespeichert werden. Das führt dazu, dass der erste Durchlauf des Preflow-Push Algorithmus die Knoten in einer sehr günstigen Reihenfolge betrachtet. Zeile 5-15 implementiert den initialen Schritt eines Preflow-Push Algorithmus. Die Höhe der Quelle wird auf n erhöht und jede der Quelle ausgehende Kante wird gefüllt. Dadurch entsteht der initiale Überschuss an den der Quelle benachbarten Knoten. Anders als beim generischen Preflow-Push Algorithmus werden diese Knoten hier nicht mehr der Liste Liste zugefügt, da sie durch die Funktion Höhenfunktion bereits in dieser enthalten sind. In Zeile 16 beginnt der Hauptteil des Algorithmus, der die Liste Liste von Knoten abarbeitet. Wurde ein Knoten u noch nicht durch die Gap-Heuristik ausgeschlossen (Zeile 19) und besitzt er positiven Überschuss (Zeile 22), so versucht der Algorithmus den Überschuss an benachbarte Knoten mit einer kleineren Höhe weiterzugegeben (Zeile 25). Ist ein solcher Knoten v noch nicht oder nicht mehr in der Liste Liste der noch zu betrachteten Knoten gespeichert, wird v in diese Liste eingefügt (Zeile 27-33). Gelingt es auf diese Weise nicht, den Überschuss des Knotens u zu verteilen (Zeile 42), wird die Höhe des Knotens u so weit erhöht, bis mindestens ein Nachbar mit kleinerem Index existiert (Zeile 49). Die Liste Anzahl enthält für jede mögliche Höhe die Anzahl der Knoten mit dieser Höhe. In Zeile 44 und 50 wird diese Liste aktuell gehalten. Ist in Zeile 45 eine Höhe nicht mehr vertreten, markiert die Funktion Gap die Menge aller Knoten, die eine größere Höhe besitzen, und setzt deren Höhe auf n. Knoten, die durch die Liste Gap markiert sind, werden nach diesem Schritt nicht mehr betrachtet, da ihr Fluss die Senke nicht mehr erreichen kann. Wurde die Höhe eines Knotens verändert, wird der Knoten erneut betrachtet. Ist der Überschuss eines Knotens verteilt (Zeile 53), wird er aus der Liste Liste entfernt, indem der Index kopf erhöht (Zeilen 56, 57) und die Farbe des Knotens wieder auf den Wert 0 gesetzt wird. Nachdem die Liste Liste vollständig abgearbeitet ist, wird in Zeile 60 der maximale Fluss, der dem Potential der Senke entspricht, zurückgegeben.
Wir wollen an dieser Stelle bemerken, dass Algorithmus 3.5.1 anders als die anderen Algorithmen durch die Anwendung der Gap-Heuristik nur den Wert des maximalen Flusses berechnet. Die konstruierte Matrix F ist ein Quasi-Fluss, da der Überschuss auf Knoten, die durch die Gap-Heuristik markiert worden sind, liegen bleibt. Der gleiche Algorithmus ohne Gap-Heuristik würde zwar diesen überschüssigen Fluss zur Quelle zurück und aus dem Flussnetzwerk schieben, aber entsprechend länger dafür brauchen. In Kapitel 4 und 5 werden wir feststellen, dass uns der Wert eines maximalen Flusses genügt, um das Szenario-Verifikations-Problem zu entscheiden.

3.6 VERGLEICH DER FLUSS-MAXIMIERUNGS-ALGORITHMEN

Wir haben verschiedene Fluss-Maximierungs-Algorithmen betrachtet, welche sich in zwei Kategorien einteilen lassen. Die Algorithmen der ersten Kategorie lösen das Fluss-Maximierungs-Problem, indem sie schrittweise kürzeste Wege im Restnetzwerk finden und den Fluss im Flussnetzwerk auf diesen Wegen erhöhen. Die Algorithmen der zweiten Gruppe erzeugen Fluss auf Kanten, die von der Quelle ausgehen und versuchen, diesen bis zur Senke hin fortzuführen oder, wenn dies nicht mehr möglich ist, den überschüssigen Fluss zurückzunehmen. Zu Beginn dieses Abschnittes wollen wir die Schlechteste-Fall-Komplexitäten der in den Kapiteln 3.2 bis 3.5 betrachteten Algorithmen noch einmal zusammenfassen. Sei dazu $G = (E, F, c, q, s)$ ein Flussnetzwerk.

- Die Erweiterung des Algorithmus von Ford und Fulkerson durch Edmonds und Karp ist in $O(|E| \cdot |F|^2)$,
- die Methode der iterierten Sperrflussberechnung durch Tiefensuche von Dinic ist in $O(|E|^2 \cdot |F|)$,
- die Methode der iterierten Sperrflussberechnung durch Forward-Backward-Propagation ist in $O(|E|^3)$,
- die einfache Implementierung eines Preflow-Push-Algorithmus ist in $O(|E|^2 \cdot |F|)$ und
- die Implementierung des Preflow-Push-Algorithmus, bei dem die Knoten in einer First-In-First-Out Reihenfolge durchlaufen werden, ist in $O(|E|^3)$.

Neben dieser Auswahl gibt es noch weitere, ausgefallenere Algorithmen. Besonders in der Kategorie der Preflow-Push Algorithmen existieren viele Varianten. Ein neuerer Algorithmus mit einer für lichte Flussnetzwerke guten Schlechtesten-Fall-Komplexität ist der Algorithmus von King, Rao und Tarjan [65]. Er besitzt eine Komplexität in $O(|E| \cdot |F| \cdot \log_{|F|/(|E| \cdot \ln(|E|))}(|E|))$. Die Reihenfolge der Knoten, von denen der Algorithmus Fluss fortführt, richtet sich bei dieser Methode nach der Möglichkeit, eine Kante des Restnetzwerks zu saturieren. Vereinfacht dargestellt muss man dazu die ausgehenden Kanten eines Knotens durchlaufen und für eine Push-Operation eine Kante auswählen, deren Restkapazität klein genug ist. Diese Methode senkt die Schlechteste-Fall-Komplexität einiger Beispiele, erhöht allerdings die durchschnittliche Rechenzeit.
Wir steigen an dieser Stelle aus dem Rennen um die Schlechteste-Fall-Komplexität aus und konzentrieren uns auf den durchschnittlichen Rechenaufwand. Um ein Gefühl für die durchschnittlichen Laufzeiten zu entwickeln, implementieren wir die bis jetzt vorgestellten Algorithmen je als Plug-In für das VipTool. Später haben wir so die Möglichkeit, die Algorithmen direkt in der Problemklasse gegeneinander antreten zu lassen, die bei der Lösung des Szenario-Verifikations-Problems entstehen.

Aus unseren bisherigen Betrachtungen ergeben sich die zu untersuchenden Fluss-Maximierungs-Algorithmen wie folgt:

Alg i	Edmonds und Karp (Algorithmus 3.2.1),
Alg ii	Dinic (Algorithmus 3.3.1),
Alg iii	Dinic (Algorithmus 3.3.1, mit Konstruktion des Restnetzwerks),
Alg iv	Forward-Backward-Propagation (Algorithmus 3.4.1, ohne Konstruktion des Niveaunetzwerks),
Alg v	Forward-Backward-Propagation (Algorithmus 3.4.1),
Alg vi	Preflow-Push (Algorithmus 3.5.1, ohne Gap-Heuristik und mit initialer Höhenfunktion 0 für jeden Knoten),
Alg vii	Preflow-Push (Algorithmus 3.5.1).

Algorithmus i ist eine Implementierung des Algorithmus von Edmonds und Karp. Er ist der wohl am einfachsten zu implementierende Fluss-Maximierungs-Algorithmus. Die iterierte Breitensuche im Restnetzwerk ist schlank zu implementieren und so sollte der Algorithmus für kleine Probleminstanzen eine gute Wahl darstellen.
Algorithmus ii und Algorithmus iii sind Implementierungen des Algorithmus von Dinic. Der Algorithmus von Dinic spart Laufzeit, indem er Durchläufe, in denen der Algorithmus von Edmonds und Karp Breitensuche verwendet, durch Durchläufe mit Tiefensuche ersetzt. Zuerst wird das Niveaunetzwerk durch Breitensuche berechnet. In diesem Niveaunetzwerk genügt dann iterierte Tiefensuche, um einen Sperrfluss zu konstruieren. Algorithmus ii berechnet das Niveaunetzwerk, indem er jedem Knoten des Restnetzwerks seine Entfernung zur Senke zuordnet. Algorithmus iii speichert während der Bestimmung dieser Entfernungen zusätzlich die dabei konstruierte Menge der im Niveaunetzwerk gefundenen Kanten. Dadurch benötigt Algorithmus iii fast doppelt so viel Speicherplatz wie Algorithmus ii. Für große Probleminstanzen könnte sich der zusätzliche Aufwand allerdings lohnen, da bei der Sperrflussberechnung nicht auf der Menge der Nachbarschaften, sondern auf der Menge der im Niveaunetzwerk enthaltenen Kanten gesucht werden kann.
Algorithmus iv und v sind Implementierungen des Forward-Backward-Propagation Algorithmus. Beide sind wie der Algorithmus von Dinic aufgebaut, sollen aber Laufzeit bei der Konstruktion des Sperrflusses sparen. Die Anzahl der benötigten Tiefensuchen bei der Sperrflusskonstruktion wird verringert, indem diese mehrere Wege gleichzeitig abarbeiten. Dafür beginnen die Algorithmen die Suche nach flussvergrößernden Wegen in der Mitte des Niveaunetzwerks und setzen dann Wege zur Quelle und zur Senke hin fort. Wie Algorithmus ii konstruiert Algorithmus iv das Niveaunetzwerk nur über die Berechnung der Entfernung der Knoten im Restnetzwerk. Dagegen speichert Algorithmus v (genau wie Algorithmus iii) neben den verschiedenen Entfernungen der Knoten gleichzeitig auch die Menge der im Niveaunetzwerk enthaltenen Kanten. Da diese Kanten vorwärts und rückwärts durchlaufen werden müssen, verwenden wir zwei separate Matrizen.
Algorithmus vi und vii sind Implementierungen des Preflow-Push Algorithmus. Beide Algorithmen betrachten die Knoten in der First-In-First-Out Reihenfolge. Algorithmus vi ist ebenso schlank zu implementieren wie Algorithmus i und beginnt mit einer Höhenfunktion, die

den Wert 0 auf allen Knoten außer der Quelle besitzt. Algorithmus vii begegnet den in Abschnitt 3.5 diskutierten Risiken der Preflow-Push Algorithmen und verwendet neben einer etwas aufwändigeren Berechnung der Anfangshöhe jedes Knotens zusätzlich die Gap-Heuristik. Die maximal mögliche Höhe jedes Knotens ergibt sich durch rückwärts gerichtete Breitensuche und die Gap-Heuristik verhindert das Hin- und Herschwappen des überzähligen Flusses im Flussnetzwerk.
Die Algorithmen i bis vii stellen für uns die besten Kandidaten für eine Lösung des Fluss-Maximierungs-Problems dar. Um einen ersten Eindruck zu gewinnen, testen wir alle sieben Implementierungen an vier Serien von je 100 zufällig erstellten Flussnetzwerken und vergleichen die mittlere Laufzeit.

EXPERIMENT 3.6.1 *Experiment 3.6.1*

Wir betrachten vier Serien von je 100 zufällig erstellten Flussnetzwerken. Flussnetzwerke der ersten Serie besitzen 1000 Knoten und wir verdoppeln die Anzahl der Knoten in jeder weiteren Serie. Um Zyklen zu vermeiden, erlauben wir nur Kanten, deren Anfangsknoten einen kleineren Index besitzen als ihre Endknoten. Insgesamt ergeben sich damit $|E| \cdot (|E| - 1)/2$ *mögliche Kanten, von denen wir für jedes Beispiel zufällig 20% auswählen und mit einem zufälligen Kantengewicht von 1-10 versehen. Je Serie betrachten wir die sich ergebenden Mittelwerte der Laufzeiten der sieben Fluss-Maximierungs-Algorithmen.*

Darstellung der Laufzeiten des Experimentes 6.1.1

	Serie 1	*Serie 2*	*Serie 3*	*Serie 4*
Knoten	*1000*	*2000*	*4000*	*8000*
Kanten	*10000*	*400000*	*1600000*	*6400000*
	Laufzeit in ms			
Edmonds Karp (Alg i)	*234*	*1978*	*14953*	*114414*
Dinic (Alg ii)	*40*	*170*	*775*	*3311*
Dinic (Alg iii)	*34*	*152*	*660*	*2920*
Propagation (Alg iv)	*293*	*1718*	*10353*	*52739*
Propagation (Alg v)	*101*	*670*	*4564*	*23467*
Preflow-Push (Alg vi)	*1076*	*8258*	*68093*	*581873*
Preflow-Push (Alg vii)	*18*	*87*	*433*	*1901*
	Quotient aufeinander folgender Laufzeiten			
Edmonds Karp (Alg i)		*8,5*	*7,6*	*7,7*
Dinic (Alg ii)		*4,3*	*4,6*	*4,3*
Dinic (Alg iii)		*4,5*	*4,3*	*4,4*

Propagation (Alg iv)	*5,9*	*6,0*	*5,1*
Propagation (Alg v)	*6,6*	*6,8*	*5,1*
Preflow-Push (Alg vi)	*7,7*	*8,2*	*8,5*
Preflow-Push (Alg vii)	*4,8*	*5,0*	*4,4*

Im oberen Teil der Tabelle sind die Anzahlen der Knoten und Kanten der zufällig generierten Flussnetzwerke zusammengefasst. Im mittleren Teil befinden sich die durchschnittlichen Laufzeiten der sieben Algorithmen. Abschließend sind die gerundeten Wachstumsfaktoren der Laufzeiten der Algorithmen dargestellt. Diese ergeben sich als Quotient der durchschnittlichen Laufzeit der entsprechenden Serie und der durchschnittlichen Laufzeit der vorangegangenen.
Die durchschnittliche Laufzeit von Algorithmus i verachtfacht sich bei jeder Verdopplung der Knoten. Dieses Wachstum entspricht ziemlich genau unseren Erwartungen. Für jede Breitensuche besuchen wir jede Kante des Restnetzwerks, und die Anzahl der flussvergrößernden Wege in unseren Netzwerken entspricht in etwa der Anzahl aller Knoten. Etwas überraschend ist, dass sich selbst bei Flussnetzwerken mit nur 1000 Knoten die schlanke Implementierung des Algorithmus von Edmonds und Karp kaum auszahlt. Der Algorithmus ist über alle vier Serien betrachtet der zweitlangsamste im Test.
Die durchschnittlichen Laufzeiten der Algorithmen ii und iii sind bereits in der ersten Serie erheblich besser als die Laufzeit des Algorithmus von Edmonds und Karp. Der Vorsprung der beiden Algorithmen vergrößert sich mit wachsender Anzahl der Knoten, da sich bei einer Verdopplung der Knoten die Laufzeit circa vervierfacht. Wieder entspricht dieses Wachstum unseren Erwartungen, da eine Tiefensuche eher mit der Anzahl der Knoten als mit der Anzahl der Kanten wächst. Das Spendieren von zusätzlichem Speicherplatz, um die Kanten des Niveaunetzwerks festzuhalten, scheint sich bereits in der ersten Serie zu rentieren. Insgesamt sind beide Algorithmen nur knapp langsamer als der schnellste Algorithmus im Tests.
Enttäuschend schneiden die beiden Algorithmen iv und v ab. Würde man sich an deren Schlechtester-Fall-Komlexität orientieren, sollte sich der Trend der Laufzeitverbesserung von Algorithmus i zu Algorithmus ii und iii eigentlich zu Algorithmus iv und v fortsetzten, aber die durchschnittlichen Laufzeiten der Forward-Backward-Propagation Algorithmen fallen deutlich hinter die durchschnittlichen Laufzeiten des Algorithmus von Dinic zurück. Eine erste Erklärung hierfür ist, dass die Berechnung des Potentials aller Knoten laufzeitintensiv ist. Dies ist daran zu erkennen, dass bereits bei Serie 1 die Laufzeiten beider Algorithmen weit über den Laufzeiten von Algorithmus ii und iii liegen. Darüber hinaus scheint das Betrachten der Wege im Niveaunetzwerk in einer Reihenfolge, die durch das Potential vorgeben ist, in diesem Versuch nicht zu einer besonders effizienten Sperrflussberechnung zu führen, was wir an den Wachstumsfaktoren der Laufzeiten beider Algorithmen erkennen. Während beim Algorithmus von Dinic die Laufzeit für die Sperrflussberechnung ungefähr mit der Anzahl der Knoten wächst, wächst die Laufzeit von Algorithmen iii und iv stärker

als die Anzahl der Knoten und etwas weniger stark als die Anzahl der Kanten. Der Unterschied zwischen den beiden Algorithmen iv und v fällt extremer aus als der Unterschied zwischen den Algorithmen ii und iii. Das Speichern aller Kanten des Niveaunetzwerks kann im Fall der Forward-Backward-Propagation die an sich schlechte Laufzeit immerhin noch halbieren.
Der größte Unterschied zwischen zwei implementierten Versionen eines Algorithmus besteht allerdings bei den Implementierungen der Preflow-Push Algorithmen, den Algorithmen vi und vii. Nach ihrer Schlechtester-Fall-Komplexität sollten beide Algorithmen zu den schnellsten der Testserie gehören, doch Algorithmus vi besitzt in allen vier Serien die schlechteste aller Laufzeiten. Dies ist wohl darin begründet, dass bei unseren zufälligen Testfällen der Flussnetzwerk-Knoten mit dem größtmöglichen ausgehenden Fluss die Quelle ist, da sie den kleinsten Index besitzt. So ist die Wahrscheinlichkeit groß, dass zu viel Fluss im Netzwerk erzeugt und wie in Abschnitt 3.5 beschrieben der überschüssige Fluss kreuz und quer durch das Flussnetzwerk geschoben wird. Die hohe Abhängigkeit des Preflow-Push Algorithmus vom speziellen Beispiel bestätigte sich außerdem während der Durchführung der Laufzeittests. Neben der schlechtesten durchschnittlichen Laufzeit besitzt dieser Algorithmus auch die höchste Varianz in den gemessenen Laufzeiten. Zum Beispiel lagen die 100 Werte der Serie 3 mit 4000 Knoten und einer durchschnittlichen Laufzeit von ungefähr einer Minute zwischen wenigen Sekunde und drei Minuten. Der Faktor des Wachstums der Laufzeiten des Algorithmus vii ist ähnlich dem Faktor bei dem Algorithmus von Edmonds und Karp. Dies ist leicht einzusehen, wenn man bedenkt, dass der überschüssige Fluss durch das gesamte Flussnetzwerk und damit über alle Kanten geschoben werden muss. Damit gleicht er eher einer Breitensuche als einer Tiefensuche. Diese Breitensuche wird iteriert, da der überschüssige Fluss hin und her durchs Flussnetzwerk schwappt, bevor er das Flussnetzwerk verlassen kann.
Genau dieses Hin- und Herschwappen wird in Algorithmus vii durch die Anwendung der Gap-Heuristik verhindert. Durch diese Heuristik halbiert sich der Faktor des Wachstums und sinkt bis auf das Niveau des Algorithmus von Dinic. Die Gap-Heuristik lässt sich leicht implementieren und auch der Aufwand zur Berechnung einer initialen Höhenfunktion, die jedem Knoten eine maximale Starthöhe zuordnet, ist einfach durch Breitensuche zu realisieren. So kommt es, dass der Algorithmus vii in allen Serien die beste Laufzeit aufweist und der Faktor des Wachstums nur sehr knapp über dem des Algorithmus von Dinic liegt. Die Varianz der einzelnen Laufzeiten ist zwar deutlich höher als die beim Algorithmus von Dinic, fällt aber bei weitem nicht mehr so extrem aus wie bei Algorithmus vi.
Mit Ausnahme von Algorithmus vi befinden sich die Wachstumsraten aller Algorithmen weit unter dem theoretisch bestimmten Wachstum der Schlechtester-Fall-Komplexitäten, was allerdings nur im Ausmaß der Ergebnisse verwunderlich ist. Algorithmus i und vi sind mit einem kubischen Wachstumsfaktor die klaren Verlierer der Laufzeittests. Die Algorithmen iv und v fallen deutlich hinter den Gewinnern der Laufzeittests Algorithmus ii, iii und vii zurück. Die quadratischen Wachstumsfaktoren der Algorithmen ii, iii und vii liegen weit unter dem theoretisch bestimmten Wert und auch in den gemessenen Laufzeiten schlagen sie die anderen Algorithmen so deutlich, dass wir für die

zweite Serie von Laufzeittests nur noch diese Algorithmen betrachten wollen.
Nachdem wir in Experiment 3.6.1 das Verhalten der Laufzeiten bei steigender Knotenanzahl und konstantem Verhältnis von Knoten zu Kanten betrachtet haben, betrachten wir in Experiment 3.6.2 die durchschnittlichen Laufzeiten der drei Algorithmen ii, iii und vii in Abhängigkeit von der Anzahl der Kanten in einem Flussnetzwerk.

Experiment 3.6.2

EXPERIMENT 3.6.2
Wir betrachten acht Serien von je 100 zufällig erstellten Flussnetzwerken. Alle Flussnetzwerke besitzen 4000 Knoten, die Serien unterscheiden sich in den Anzahlen der im Flussnetzwerk enthaltenen Kanten. Kanten besitzen Kantengewichte mit zufälligen Werten von 1-10 und insgesamt ergeben sich wieder $|E| \cdot (|E|-1)/2$ *mögliche Kanten für jedes Flussnetzwerk. Je nach Serie wählen wir zufällig 0.5%, 1%, 2,5%, 5%, 10%, 20%, 50% und 90% dieser möglichen Kanten aus und erzeugen damit unsere Flussnetzwerke. Abbildung 18 zeigt drei Kurven, die die durchschnittlichen Laufzeiten der Algorithmen ii, iii und vii interpolieren.*

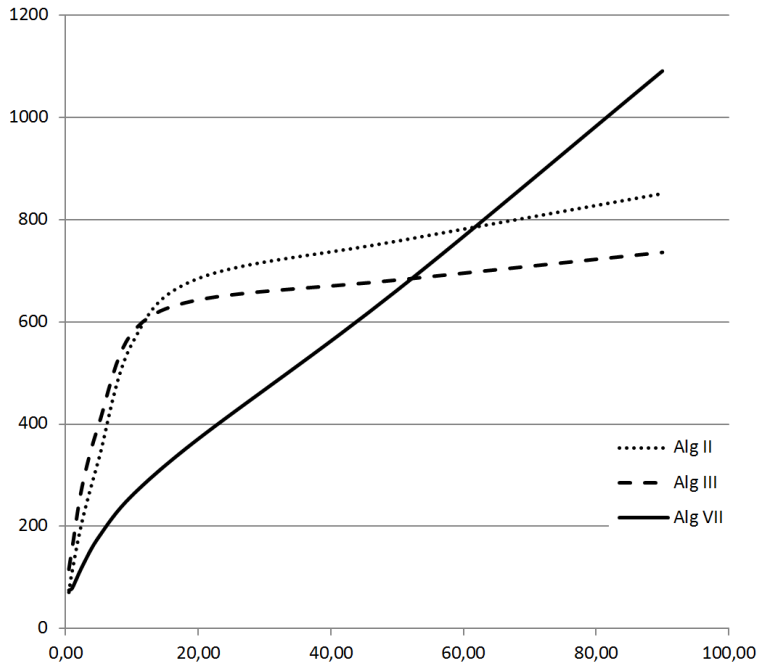

Abbildung 18: Die Laufzeiten von drei VipTool-Plugins zu Lösung des Fluss-Maximierungs-Problems in Abhängigkeit von der Anzahl der Kanten.

Experiment 3.6.2 zeigt, dass der Preflow-Push Algorithmus fast linear mit der Anzahl der Kanten wächst. Der Algorithmus versucht über jede zusätzliche Kante den am Anfang konstruierten Fluss in Richtung Senke fortzusetzen. Jede weitere Kante verursacht in gleichem Maß zusätzliche Laufzeit unabhängig von dem bisherigen Anteil an Kanten im Flussnetzwerk. Insgesamt ist die Steigung der Kurve des Preflow-Push Algorithmus relativ gleichmäßig über alle Testfälle.
Bei den beiden Versionen des Algorithmus von Dinic sind zwei Auffälligkeiten zu betrachten. Falls ein Flussnetzwerk relativ wenige Kanten besitzt, führt jede weitere Kante dazu, dass die Laufzeit des Algorithmus stärker steigt als die Laufzeit des Preflow-Push Algorithmus. In

diesem Fall besteht eine große Wahrscheinlichkeit, dass eine zusätzliche Kante das Flussnetzwerk verbreitert und zu einen neuen Weg im Flussnetzwerk führt, der eine weitere Iteration der Tiefensuche verursacht. Die Steigerung der durchschnittlichen Laufzeit ist in diesem Fall groß. Die zweite Auffälligkeit ist, dass, wenn das Flussnetzwerk bereits viele Kanten besitzt und damit relativ breit ist, die Wahrscheinlichkeit, dass die zusätzliche Kante nach einigen Iterationen im Restnetzwerk überhaupt noch zu erreichen ist, stark sinkt. Der Wachstumsfaktor der durchschnittlichen Laufzeiten ist in diesem Fall relativ klein, insbesondere kleiner als beim Preflow-Push Algorithmus.
Durch diese Effekte ergibt sich ein interessantes Bild. Vergleicht man die Algorithmen ii und iii, lohnt sich das explizite Speichern von Kanten im Niveaunetzwerk ab ca. 10% aller möglichen Kanten im Flussnetzwerk. Dies ist auch der Wert, an dem zusätzliche Kanten zu einem viel geringeren Anstieg der Laufzeit dieser beiden Algorithmen führen. Zu Beginn ist die Laufzeit von Algorithmus vii robuster gegen zusätzliche Kanten, so dass er bis zu einem Wert von 50% aller Kanten deutlich effizienter ist als die Algorithmen ii und iii. Ab einer Dichte von 50% besitzt Algorithmus iii die beste Laufzeit der Testserie.
Insgesamt halten wir unsere Ergebnisse der zwei Experimente 3.6.1 und 3.6.2 fest:

- Für lichte Flussnetzwerke (weniger als 50% aller möglichen Kanten) scheint der Preflow-Push Algorithmus mit Gap-Heuristik und maximaler Höhenfunktion der für uns geeignete Algorithmus zur Lösung des Fluss-Maximierungs-Problems zu sein.
- Für dichte Flussnetzwerke (mehr als 50% aller möglichen Kanten) scheint der Algorithmus von Dinic, der die Kanten des Niveaunetzwerks bei seiner Konstruktion explizit abspeichert, der für uns geeignete Algorithmus zur Lösung des Fluss-Maximierungs-Problems zu sein.

In Kapitel 6 werden wir diese Ergebnisse an den Instanzen von Flussnetzwerken betrachten, die bei der Entscheidung des Szenario-Verifikations-Problems von Petrinetzen auftreten.

4

ABLÄUFE UND PETRINETZE

Dieses Kapitel behandelt Petrinetze, Szenarien und das Szenario-Verifikations-Problem. Es beschreibt und vergleicht erste Ansätze, das Szenario-Verifikations-Problem zu entscheiden. Dazu enthalten die Abschnitte 4.1 und 4.2 grundlegende Definitionen, bevor in den Abschnitten 4.3, 4.4 und 4.5 drei verschiedene Charakterisierungen der Szenario-Sprache eines Petrinetzes vorgestellt werden. In jedem Abschnitt wird aus der entsprechenden Charakterisierung ein Algorithmus zur Lösung des Szenario-Verifikations-Problems abgeleitet und implementiert.

4.1 STELLEN/TRANSITIONS-NETZE

Petrinetze modellieren Systeme, die nebenläufiges Verhalten aufweisen [78, 40, 80, 7, 77, 36]. Sie besitzen eine exakte mathematische Beschreibung und eine leicht verständliche graphische Darstellung. Aus diesem Grund haben sich Petrinetze und Petrinetz-Dialekte in der Praxis sowie in der Theoretischen Informatik weit verbreitet. Aus der Urform der Petrinetze haben sich vielfältige Erweiterungen entwickelt. Beispiele hierfür sind Petrinetze mit Kapazitätsbeschränkungen für Stellen, Kontextkanten oder farbigen Marken. Prominente Beispiele für Petrinetz-Dialekte sind die in der Geschäftsprozessmodellierung eingesetzten UML-Aktivitätsdiagramme und Ereignisgesteuerte Prozessketten. Erweiterungen und Petrinetz-Dialekte verbessern oft die Modellierungstauglichkeit für konkrete Anwendungen, stützen sich aber alle auf die klassischen Petrinetze, um auf deren vielfältige Analyseverfahren, Methoden und Algorithmen zurückgreifen zu können. In dieser Arbeit betrachten wir Petrinetze in einer klassischen Form, die endlichen Stellen/Transitions-Netze [78, 80].

Definition 4.1.1 (Stellen/Transitions-Netz)

Definition 4.1.1 (Stellen/Transitions-Netz)
Ein Stellen/Transitions-Netz (kurz S/T-Netz) N *ist ein Tripel* (P, T, W). *Dabei ist*

P *eine endliche Menge von Stellen,*

T *eine endliche Menge von Transitionen mit* $P \cap T = \emptyset$ *und*

$W : (P \times T) \cup (T \times P) \to \mathbb{N}_0$ *eine Multimenge von Kanten.*

Ein markiertes S/T-Netz ist ein Paar (N, m_0), *wobei* $N = (P, T, W)$ *ein S/T-Netz und* $m_0 : P \to \mathbb{N}_0$ *eine Markierung von* N *ist. Eine Markierung ordnet jeder Stelle eine Anzahl von Marken zu. Die Markierung* m_0 *heißt Anfangsmarkierung von* (N, m_0).

Die Stellen und Transitionen nennt man auch die Knoten eines S/T-Netzes. Transitionen modellieren Aufgaben und Aktionen, sie sind die aktiven Knoten. Stellen modellieren Teilzustände und Ressourcen, indem sie durch eine Anzahl an Marken belegt werden. Stellen sind die passiven Knoten. Die Häufigkeit jeder Kante in der Multimenge W ist das Gewicht der Kante.
In einem S/T-Netz können Transitionen schalten und damit das Ausführen einer Aktion modellieren. Das Schalten einer Transition ändert die Anzahl der Marken auf den Stellen, mit denen die Transition über Kanten verbunden ist und damit die Markierung des S/T-Netzes. Damit eine Transition in einem S/T-Netz in einer Markierung schalten kann, müssen alle Stellen Marken entsprechend der Kantengewichte der Kanten zwischen den Stellen und der Transition tragen. Um dies zu formalisieren, betrachten wir den gewichteten Vorbereich und Nachbereich einer Transition.

Definition 4.1.2 (Gewichteter Vorbereich und Nachbereich)

Definition 4.1.2 (Gewichteter Vorbereich und Nachbereich)
Sei $N = (P, T, W)$ *ein S/T-Netz und* $t \in T$ *eine Transition. Wir definieren*

$$^\circ t = \sum_{p \in P} W(p,t) \cdot p, \quad \textit{den gewichteten Vorbereich von } t \textit{ und}$$

$$t^\circ = \sum_{p \in P} W(t,p) \cdot p, \quad \textit{den gewichteten Nachbereich von } t.$$

Abbildung 19 zeigt ein Beispiel eines markierten S/T-Netzes. Wie bereits in der Einleitung beschrieben, werden Stellen durch Kreise, Transitionen durch Vierecke und Marken durch schwarze Punkte in den entsprechenden Stellen dargestellt. Die Multimenge der Kanten wird als Menge von Pfeilen dargestellt, wobei eine annotierte Zahl die Häufigkeit einer Kante in der Multimenge der Kanten angibt, wenn diese größer als Eins ist. Kanten, die nicht in der Multimenge enthalten sind, werden nicht gezeichnet.

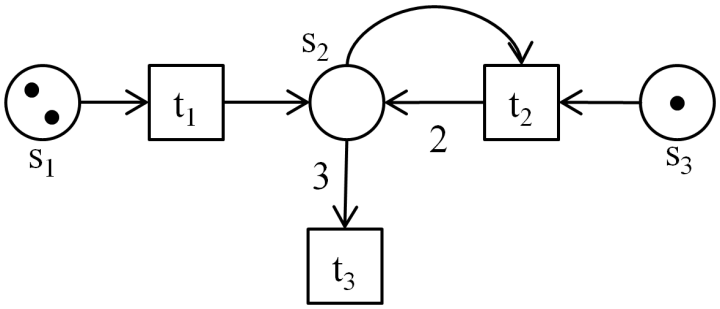

Abbildung 19: Ein markiertes S/T-Netz.

Das markierte S/T-Netz aus Abbildung 19 besitzt drei Transitionen t_1, t_2 und t_3, drei Stellen s_1, s_2 und s_3 und befindet sich in der Anfangsmarkierung $2 \cdot s_1 + s_3$. Die Kante (s_2, t_3) besitzt das Gewicht drei, die Kante (t_2, s_2) besitzt das Gewicht zwei. Kanten ohne annotierte Zahl sind einmal, nicht dargestellte Kanten sind nicht in der Multimenge der Kanten enthalten.
Die Markierung eines S/T-Netzes stellt seinen Zustand dar. Eine Änderung der Markierung findet durch das Schalten von Transitionen statt.

Eine Schaltregel legt Bedingungen für das Schalten einer Transition und die durch einen Schaltvorgang bewirkte Markierungsänderung fest. Eine Transition t kann in einer Markierung schalten, wenn ihr gewichteter Vorbereich kleiner oder gleich der Markierung ist. In diesem Fall sagen wir, dass eine Transition in einer Markierung aktiviert ist. Wenn eine Transition schaltet, konsumiert sie Marken aus den Stellen in ihrem Vorbereich und produziert Marken in den Stellen in ihrem Nachbereich. Die Anzahl der konsumierten und produzierten Marken ergibt sich aus dem gewichteten Vor- bzw. Nachbereich der Transition.

Definition 4.1.3 (Schaltregel für Transitionen)

Definition 4.1.3 (Schaltregel für Transitionen)
Sei $N = (P, T, W)$ ein S/T-Netz und m eine Markierung. Eine Transition $t \in T$ ist in m aktiviert, falls $m \geqslant {}^\circ t$ gilt.
Ist eine Transition t in einer Markierung m aktiviert, kann t in m schalten. Das Schalten überführt m in die Folgemarkierung $m' = m - {}^\circ t + t^\circ$ und wir schreiben $m \xrightarrow{t} m'$.

Ändert sich die Markierung eines S/T-Netzes, ändert sich auch die Menge der aktivierten Transitionen. Durch sequentielles Schalten aktivierter Transitionen entstehen Schaltfolgen. Die Menge aller aus der Anfangsmarkierung möglichen Schaltfolgen ist die sequentielle Sprache eines markierten S/T-Netzes.

Definition 4.1.4 (Sequentielle Sprache)

Definition 4.1.4 (Sequentielle Sprache)
Sei (N, m_0) mit $N = (P, T, W)$ ein markiertes S/T-Netz. Eine endliche Folge von Transitionen $\sigma = t_1 \dots t_n \in T^*$ heißt Schaltfolge.
Eine Schaltfolge σ heißt in m_0 aktiviert, falls eine Folge von Markierungen $m_1 \dots m_n$ existiert, so dass

$$m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n \text{ gilt.}$$

Die Menge $L^{Seq}(N, m_0)$ aller in m_0 aktivierten Schaltfolgen ist die sequentielle Sprache von (N, m_0).

In Abbildung 19 ist die Stelle s_2 nicht in der Anfangsmarkierung, aber im gewichteten Vorbereich der Transitionen t_2 und t_3 enthalten. Es folgt, dass weder t_2 noch t_3 aktiviert ist. Der gewichtete Vorbereich der Transition t_1 ist kleiner als die Anfangsmarkierung und daher ist t_1 aktiviert. Schaltet t_1, wird die Anfangsmarkierung in eine neue Markierung $s_1 + s_2 + s_3$ überführt, da entsprechend dem gewichteten Vorbereich eine Marke aus der Stelle s_1 entfernt und eine Marke der Stelle s_2 hinzugefügt wird. In dieser neuen Markierung ist wieder die Transition t_1, aber nun auch die Transition t_2 aktiviert. Durch ein weiteres Schalten von t_1 ändert sich die Markierung zu $2 \cdot s_2 + s_3$, und die Transition t_2 ist aktiviert. Nach einem Schalten der Transition t_2 entsteht die neue Markierung $3 \cdot s_2$. Diese Markierung aktiviert Transition t_3 und wir erhalten eine mögliche Schaltfolge $t_1t_1t_2t_3$. Nach dieser Schaltfolge ist keine Transition im S/T-Netz mehr aktiviert. Alternativ existiert zu der Schaltfolge $t_1t_1t_2t_3$ eine weitere aktivierte Schaltfolge $t_1t_2t_1t_3$. Die sequentielle Sprache $L^{Seq}(N, m_0)$ des markierten S/T-Netzes (N, m_0) aus Abbildung 19 können wir durch das Auflisten der Elemente $\{t_1t_1t_2t_3, t_1t_2t_1t_3\}$ beschreiben, denn aus der Definition 4.1.4 folgt, dass jede sequentielle Sprache gegenüber Präfixbildung abgeschlossen ist.
Neben der sequentiellen Sprache lassen sich für Petrinetze verschiedene andere Semantiken betrachten. Die Schaltregel für Transitionen lässt

sich leicht auf eine Schaltregel für Multimengen von Transitionen, sogenannte Schritte, erweitern. Dazu betrachten wir zunächst den gewichteten Vor- und Nachbereich eines Schrittes.

Definition 4.1.5 (Gewichteter Vorbereich und Nachbereich eines Schrittes)

DEFINITION 4.1.5 (GEWICHTETER VORBEREICH UND NACHBEREICH EINES SCHRITTES)
Sei $N = (P, T, W)$ *ein S/T-Netz und* $\tau : T \to \mathbb{N}_0$ *ein Schritt. Wir definieren*

$$^\circ\tau = \sum_{t \in T} \tau(t) \cdot {}^\circ t, \quad \textit{den gewichteten Vorbereich von } \tau \textit{ und}$$

$$\tau^\circ = \sum_{t \in T} \tau(t) \cdot t^\circ, \quad \textit{den gewichteten Nachbereich von } \tau.$$

Ein Schritt ist in einer Markierung aktiviert, wenn sein gewichteter Vorbereich kleiner oder gleich der Markierung ist. Anschaulich bedeutet das, dass die Marken der Markierung so auf die Transitionen des Schrittes aufgeteilt werden können, dass jede Transition mit den ihr zugeteilten Marken aktiviert ist. Schaltet ein Schritt, so ist die Folgemarkierung identisch der Markierung, die sich durch das sequentielle Schalten der Transitionen des Schrittes ergibt.

Definition 4.1.6 (Schaltregel für Schritte)

DEFINITION 4.1.6 (SCHALTREGEL FÜR SCHRITTE)
Sei $N = (P, T, W)$ *ein S/T-Netz und* m *eine Markierung von* N. *Ein nicht leerer Schritt* τ *ist in* m *aktiviert, falls* $m \geqslant {}^\circ\tau$ *gilt.*
Ist τ *in einer Markierung* m *aktiviert, kann* τ *in* m *schalten. Das Schalten überführt* m *in die Folgemarkierung* $m' = m - {}^\circ\tau + \tau^\circ$, *und wir schreiben* $m \xrightarrow{\tau} m'$.

Durch sequentielles Schalten aktivierter Schritte entstehen Schrittfolgen. Die Menge aller möglichen Schrittfolgen ist die Schritt-Sprache eines markierten S/T-Netzes.

Definition 4.1.7 (Schritt-Sprache)

DEFINITION 4.1.7 (SCHRITT-SPRACHE)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz. Eine endliche Folge von nicht leeren Schritten* $\rho = \tau_1 \dots \tau_n$ *heißt Schrittfolge.*
Eine Schrittfolge ρ *heißt in* m_0 *aktiviert, falls eine Folge von Markierungen* $m_1, \dots, m_n$ *existiert, so dass*

$$m_0 \xrightarrow{\tau_1} m_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} m_n \textit{ gilt.}$$

Die Menge $L^{Step}(N, m_0)$ *aller in* m_0 *aktivierten Schrittfolgen ist die Schritt-Sprache von* (N, m_0).

In Abbildung 19 ist die Transition t_1 aktiviert und kann in der Anfangsmarkierung nebenläufig zu sich selbst schalten. Der gewichtete Vorbereich des Schrittes $2 \cdot t_1$ ist $2 \cdot s_1$, und dieser ist kleiner als die Anfangsmarkierung $2 \cdot s_1 + s_3$. Schaltet dieser Schritt, so ergibt sich die Markierung $2 \cdot s_2 + s_3$, wodurch zunächst die Transition t_2 und danach die Transition t_3 schalten kann. Nun ist keine Transition mehr aktiviert und wir erhalten eine maximale Schrittfolge $(2 \cdot t_1)\ t_2\ t_3$. Eine weitere aktivierte Schrittfolge ergibt sich, wenn man in der Anfangsmarkierung zunächst die Transition t_1 schaltet. In der folgenden Markierung $s_1 + s_2 + s_3$ ist nun der Schritt $t_1 + t_2$ aktiviert. Das Schalten dieses Schrittes führt zu der Markierung $3 \cdot s_3$, in welcher die

Transition t_3 schalten kann. Die Schritt-Sprache $L^{Step}(N, m_0)$ des markierten S/T-Netzes (N, m_0) aus Abbildung 19 können wir damit durch das Auflisten der Elemente $\{(2 \cdot t_1)\ t_2\ t_3, t_1\ (t_1 + t_2)\ t_3\}$ beschreiben, denn aus der Definition 4.1.7 folgt, dass jede Schritt-Sprache gegenüber Präfixbildung und Sequentialisierung abgeschlossen ist.
Schaltregeln beschreiben, wie Stellen das Verhalten eines S/T-Netzes einschränken. Jede Stelle stellt Bedingungen an das Schalten der Transitionen. Genau dann, wenn eine Transition diese Schaltbedingungen für jede Stelle eines S/T-Netzes erfüllt, ist die Transition im gesamten S/T-Netz aktiviert. Auf diese Weise kann man die Aktiviertheit einer Transition überprüfen, indem man die Schaltregel isoliert für jede Stelle betrachtet. Diesen Umstand werden wir in späteren Abschnitten dieser Arbeit ausnutzen.
Im nächsten Abschnitt werden wir Szenarien und danach die Szenario-Sprache eines S/T-Netzes betrachten. Diese Semantik von S/T-Netzen kann nebenläufiges Verhalten viel intuitiver beschreiben als die Schritt-Sprache.

4.2 SZENARIEN

Der vorangegangene Abschnitt beschreibt die sequentielle Sprache und die Schritt-Sprache eines S/T-Netzes. Jedes Element dieser Sprachen ist entweder eine Sequenz von Ereignissen (Schaltfolge) oder eine Sequenz von Schritten (Schrittfolge). In diesem Abschnitt betrachten wir nun eine Sprache, deren Elemente halbgeordnete Mengen von Ereignissen sind. Ereignisse können damit zueinander geordnet oder ungeordnet sein.
Anders als bei Schrittsequenzen, bei denen zwar Nebenläufigkeit auftritt, aber nach jedem Schritt das gesamte Verhalten wieder synchronisiert wird, ist die Relation der paarweisen Unabhängigkeit von Ereignissen in einer Halbordnung nicht transitiv. Auf diese Weise lässt sich beschreiben, dass zwei Mengen von Ereignissen nur unter sich, nicht aber zueinander geordnet sind. Aus diesem Grund sind solche Sprachen die bedeutendsten Formalismen zur Beschreibung des Verhaltens von Petrinetzen und verteilter oder nebenläufiger Systeme [79, 57, 27, 91, 59, 81, 50]. In der Einleitung dieser Arbeit haben wir die Vorteile dieser halbgeordneten Mengen von Ereignissen zur Verhaltensbeschreibung kennengelernt.
Formal ist eine halbgeordnete Menge von Ereignissen eine Halbordnung, bei der jeder Knoten zusätzlich mit einer Beschriftung versehen ist, die ihm den Namen einer Aktion zuordnet. Da wir Abläufe von S/T-Netzen betrachten, beschreibt ein Ereignis das Schalten einer Transition. Also wird jedes Ereignis mit dem Namen einer Transition beschriftet. Wir nennen diese beschrifteten Halbordungen Szenarien. Szenarien werden in anderen Arbeiten auch beschriftete partielle Ordnungen (labeled partial orders) [67, 60] oder Pomsets [50] genannt.

Definition 4.2.1 (Szenario)

Definition 4.2.1 (Szenario)
Ein Szenario über einer Menge von Beschriftungen T *ist ein Tripel* $\mathsf{bpo} = (\mathsf{E}, <, \mathsf{l})$, *wobei* $(\mathsf{E}, <)$ *eine Halbordnung und* $\mathsf{l} : \mathsf{E} \to \mathsf{T}$ *eine Abbildung ist, die jedem Knoten* $\mathsf{e} \in \mathsf{E}$ *eine Beschriftung aus* T *zuordnet.*
Ein Knoten $\mathsf{e} \in \mathsf{E}$ *zusammen mit seiner Beschriftung nennen wir ein Ereignis des Szenarios.*

Szenarien beschreiben Verhalten, dabei ist ein Ablauf dann Verhalten gemäß eines Szenarios, wenn die Ordnung der Ereignisse im Ablauf die Ordnung des Szenarios respektiert. Sind zwei Ereignisse e_1 und e_2 in einem Szenario durch eine Kante (e_1, e_2) geordnet, so muss das Ereignis e_2 immer später als Ereignis e_1 eintreten. Aus diesem Grund bezeichnen wir die Menge der Kanten auch als „später als"-Relation des Szenarios. Sind zwei Ereignisse e_1 und e_2 nicht geordnet, so können die Ereignisse unabhängig voneinander eintreten.
Die „später als"-Relation jedes Szenarios ist transitiv. So kann ein Szenario Verhalten in dem oben beschriebenen Sinn darstellen. Sind e_1, e_2 und e_3 Ereignisse und muss e_2 später als e_1 und e_3 später als e_2 eintreten, so folgt sofort, dass auch Ereignis e_3 erst später als e_1 eintreten muss.
Es ist unüblich, in Spezifikationen oder Abbildungen die vollständige Menge der „später als"-Relation darzustellen. Eine Menge von Ereignissen mit einer zyklenfreien und irreflexiven Ordnung nennen wir Szenario-Spezifikation und jede Szenario-Spezifikation definiert durch ihren transitiven Abschluss $(\mathsf{E}, <^*, \mathsf{l})$ eindeutig ein Szenario.

Definition 5.1.1 (Szenario-Spezifikation)

Definition 4.2.2 (Szenario-Spezifikation)
Eine Szenario-Spezifikation über einer Menge von Beschriftungen T *ist ein Tripel* $\mathsf{bpo} = (\mathsf{E}, <, \mathsf{l})$, *wobei* $(\mathsf{E}, <)$ *ein zyklenfreier und irreflexiver Graph und* $\mathsf{l} : \mathsf{E} \to \mathsf{T}$ *eine Abbildung ist, die jedem Knoten* $\mathsf{e} \in \mathsf{E}$ *eine Beschriftung aus* T *zuordnet.*
Jede Szenario-Spezifikation beschreibt durch $(\mathsf{E}, <^*, \mathsf{l})$ *eindeutig das zu der Szenario-Spezifikation gehörende Szenario.*

Das von einer Szenario-Spezifikation beschriebene Verhalten ist immer das Verhalten, das von dem zugehörigen Szenario beschrieben wird. Aus diesem Grund verwenden wir die Begriffe Szenario und Szenario-Spezifikation synonym, wenn die Unterscheidung nicht von Belang ist. Ein Beispiel einer Szenario-Spezifikation findet sich in Abbildung 20.

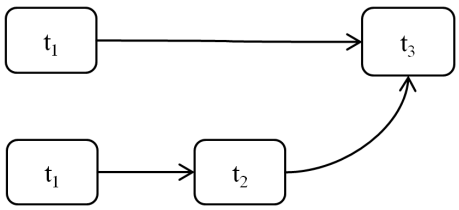

Abbildung 20: Eine Szenario-Spezifikation.

Wie für eine Halbordnung kann man auch für ein Szenario aus der „später als"-Relation die co-Relation auf der Menge der Ereignisse ableiten. Die co-Relation beschreibt voneinander unabhängige Ereignispaare. Maximale Mengen voneinander unabhängiger Ereignisse nennt man Schnitte. Wie ein Schritt beschreibt auch ein Schnitt eine Menge von Ereignissen, die gleichzeitig eintreten können. Bei einer Schrittsequenz

folgt allerdings Schritt auf Schritt, so dass das Verhalten nach jedem Schritt synchronisiert wird, bevor der nächste Schritt beginnt. Bei einem Szenario sind Schnitte nicht notwendigerweise disjunkt, und so können Sequenzen von Ereignissen innerhalb eines Szenarios unabhängig voneinander ablaufen.
Die Breite eines Szenarios ist die maximale Mächtigkeit seiner Schnitte und die Länge eines Szenarios ist die Länge eines längsten Weges im Szenario. Abbildung 20 zeigt ein Szenario mit der Länge drei und der Breite zwei.
Besitzen zwei Szenarien die gleiche Ereignismenge, so kann man diese bezüglich ihrer Ordnung vergleichen. Enthält ein Szenario alle Abhängigkeiten eines anderen Szenarios und darüber hinaus noch weitere, so nennt man es, analog zum Begriff der Sequentialisierung der zu Grunde liegenden Relation, eine Sequentialisierung des anderen Szenarios.
Das durch ein Szenario beschriebene Verhalten beinhaltet auch das durch seine Sequentialisierungen beschriebene Verhalten. Sind zwei Ereignisse unabhängig voneinander, so können sie insbesondere nacheinander eintreten. Außerdem beinhaltet Verhalten gemäß eines Szenarios das gesamte Verhalten aller Präfixe des Szenarios. Jedes Präfix kann man sich als eine noch nicht abgeschlossene Ausführung des Szenarios vorstellen. Aus diesen Gründen ist es üblich, eine Semantik zu definieren, die eine sequentialisierungs- und präfixabgeschlossene Sprache ergibt.

Definition 4.2.3 (Szenario-Sprache)

Definition 4.2.3 (Szenario-Sprache)
Eine sequentialisierungs- und präfixabgeschlossene Menge von Szenarien heißt Szenario-Sprache.
Für jede Menge von Szenarien BPO *nennen wir deren Präfix- und Sequentialisierungsabschluss* $\mathsf{L(BPO)}$ *die Szenario-Sprache von* BPO.

Für eine Menge von Szenario-Spezifikationen BPO ist $\mathsf{L(BPO)}$ die Szenario-Sprache, die durch die transitiven Abschlüsse der Szenario-Spezifikationen gegeben ist. Auf diese Weise lässt sich eine Szenario-Sprache komfortabel durch ihre Menge der bzgl. Präfixbildung maximalen Szenario-Spezifikationen mit minimaler Ordnung beschreiben. Ab dieser Stelle der Arbeit wollen wir halbgeordnetes Verhalten, die Szenario-Sprache, eines S/T-Netzes betrachten.
Um die Szenario-Sprache eines S/T-Netzes zu definieren, existieren in der Literatur drei verschiedene, zum Glück äquivalente, Charakterisierungen der Szenario-Sprache. Die erste Charakterisierung verwendet Schnitte und deren Präfixe. Die zweite Charakterisierung basiert auf einer dem Szenario-Begriff ähnlichen Ablaufbeschreibung, den sogenannten Prozessnetzen. Die dritte Charakterisierung beschreibt Verteilungen von Marken auf den Abhängigkeiten eines Szenarios, die sogenannten Markenflüsse. Streng genommen unterscheidet man den Begriff eines in einem S/T-Netz ausführbaren Szenarios je nach der zugrundeliegenden Definition. Man spricht von ausführbaren Szenarien, aktivierten Szenarien oder Szenarien, die die Markenflusseigenschaft erfüllen. Dass die ersten beiden Charakterisierungen äquivalent sind, zeigten Astrid Kiehn [64] und Walter Vogler [87]. Dass die dritte Charakterisierung äquivalent zu den ersten beiden ist, zeigten Gabriel Juhas, Robert Lorenz und Jörg Desel [60]. Dieser Beweis und die Definition der Charakterisierung entstanden deutlich später als die ersten beiden Charakterisierungen. Sie erlaubt es erstmals, das Szenario-Verifikations-Problem für S/T-Netze in Polynomialzeit zu entscheiden. Wir konkre-

tisieren an dieser Stelle das Szenario-Verifikations-Problem mit den eingeführten Begriffen.

Definition 4.2.4 (Das Szenario-Verifikations-Problem)

Definition 4.2.4 (Das Szenario-Verifikations-Problem)

Gegeben: *Ein Szenario oder eine Szenario-Spezifikation* bpo *und ein markiertes S/T-Netz* (N, m_0).

Gefragt: *Ist* bpo *in der Szenario-Sprache des S/T-Netzes enthalten?*

Die Definition der Szenario-Sprache eines S/T-Netzes bleiben wir jetzt noch schuldig, diese wird in den nächsten Abschnitten nachgeholt. Wir betrachten in den nächsten Abschnitten kurz die drei Charakterisierungen der Szenario-Sprache, da wir aus jeder einen Algorithmus ableiten können, der das Szenario-Verifikations-Problem entscheidet.

4.3 DER AKTIVIERTE SCHNITTE ALGORITHMUS

In diesem Abschnitt wird zunächst die Charakterisierung der Szenario-Sprache über Schnitte eines S/T-Netzes [56] und danach ein Algorithmus, der das Szenario-Verifikations-Problem mit Hilfe dieser entscheidet, beschrieben.
Ein Szenario ist in der Szenario-Sprache eines S/T-Netzes, wenn die Ereignisse in der im Szenario spezifizierten Ordnung im S/T-Netz eintreten können, wenn es also möglich ist, dass die zu den Ereignissen gehörenden Transitionen unter den beschriebenen Abhängigkeiten im S/T-Netz schalten können. Dabei wird nicht verlangt, dass alle im Szenario spezifizierten Abhängigkeiten auch im S/T-Netz bestehen, es genügt, wenn jede Ordnung, die das Szenario respektiert, im S/T-Netz ausführbar ist. Sind Ereignisse im Szenario ungeordnet, so müssen die entsprechenden Transitionen im S/T-Netz nebenläufig zueinander schalten können. Sind Ereignisse im Szenario geordnet, so kann zunächst die eine, dann die andere zugehörige Transition im S/T-Netz schalten. Die zweite Transition kann damit vom Schalten der ersten Transition abhängig sein. Nach dieser Überlegung muss für ein Szenario und ein S/T-Netz jeder Schritt von Transitionen, der durch eine Menge von zueinander ungeordneten Ereignissen C beschrieben wird, gleichzeitig in der Markierung ausführbar sein, die sich durch ein Schalten der Transitionen ergibt, die zu den Ereignissen gehören, die durch das Szenario vor C geordnet sind [88]. Es reicht, dies für die maximalen ungeordneten Mengen (die Schnitte) des Szenarios zu fordern, da sich daraus dieselbe Aussage für alle co-Mengen ergibt. Das Szenario heißt in diesem Fall aktiviert im S/T-Netz.

Definition 4.3.1 (Szenario-Sprache eines S/T-Netzes)

Definition 4.3.1 (Szenario-Sprache eines S/T-Netzes)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz. Ein Szenario* $\mathrm{bpo} = (E, <, l)$ *mit* $l : E \to T$ *ist genau dann in* (N, m_0) *aktiviert, wenn für jeden Schnitt* C *von* bpo *und jede Stelle* $p \in P$ *die Ungleichung*

$$m_0(p) + \sum_{e \in {}^\bullet C} (W(l(e), p) - W(p, l(e))) \geqslant \sum_{e \in C} W(p, l(e)) \text{ gilt.}$$

Die Menge $L(N, m_0)$ *aller in* m_0 *aktivierten Szenarien heißt Szenario-Sprache von* (N, m_0).

Betrachten wir nun ein Verfahren zur Lösung des Szenario-Verifikations-Problems mit Hilfe der Definition 4.3.1. Sei dazu ein markiertes S/T-Netz (N, m_0) mit $N = (P, T, W)$ und ein Szenario $bpo = (E, <, l)$ gegeben. Als erstes wird das Szenario topologisch sortiert. Das bedeutet, dass jedem Ereignis ein Index zugeordnet wird, so dass die entstehende Ordnung die Halbordnung des Szenarios respektiert. Dadurch kann man ein Szenario anschließend als obere Dreiecks-Matrix darstellen, was die Laufzeiten der folgenden Algorithmen verringert.
Um ein Szenario algorithmisch topologisch zu sortieren, berechnet man zuerst die Menge aller Knoten mit leerem Vorbereich. Danach beginnt eine iterative Prozedur. Ein beliebiger Knoten mit leerem Vorbereich bekommt den ersten noch freien Index. Danach werden alle diesem Knoten ausgehende Kanten aus der Halbordnung gelöscht und die Menge der Knoten ohne Vorbereich aktualisiert. In einer effizienten Implementierung betrachtet man jede Kante des Szenarios zweimal. Das erste Mal, während man im initialen Schritt die Anzahl der Kanten bestimmt, die jedem Knoten eingehen, und ein zweites Mal, wenn im Verlauf des Algorithmus jede Kante gelöscht wird. Mit der Größe der Eingabe wächst die Laufzeit dieses Algorithmus wie die Funktion $2\,|<|$ und hat damit eine Laufzeitkomplexität in $O(|<|)$.
Ist ein Szenario mittels einer Szenario-Spezifikation gegeben, so wird nach der topologischen Sortierung zunächst die transitive Hülle der Spezifikation berechnet. Da die Laufzeit des topologischen Sortierens von der Anzahl der Kanten abhängt, sollte die transitive Hülle erst nach dem topologischen Sortieren berechnet werden.
Der Algorithmus von Robert W. Floyd und Stephen Warshall [51, 89] konstruiert die transitive Hülle eines Graphen. Der Algorithmus betrachtet nacheinander jeden Knoten eines Graphen. Für jeden Knoten wird der Vorbereich des Knotens durchlaufen und mit dem Nachbereich des Knotens verbunden. Dadurch entstehen nach und nach der Vor- und Nachbereich jedes Knotens, bis die transitive Hülle gebildet ist. Der Algorithmus besteht aus drei ineinander liegenden Schleifen. Die erste durchläuft alle Knoten, die zweite für jeden Knoten dessen aktuellen Vorbereich und die dritte dazu den aktuellen Nachbereich. Damit besitzt der Algorithmus eine Laufzeitkomplexität in $O(|E|^3)$ und je weniger Ordnung die Spezifikation besitzt, um so schneller ist der Algorithmus.
Ist ein topologisch sortiertes Szenario konstruiert, wird die co-Relation des Szenarios berechnet. Zwei Ereignisse e und e' sind geordnet, wenn entweder eine Kante (e, e') oder eine Kante (e', e) existiert. Da unser Szenario topologisch sortiert ist, können wir eine dieser Möglichkeiten direkt ausschließen. Damit ergibt sich der Algorithmus zur Berechnung der co-Relation: Durchlaufe alle Paare von Knoten e und e', wobei der Index der topologischen Ordnung von e echt kleiner ist als der Index von e'. Existiert keine Kante (e, e') in dem Szenario, so ist (e, e') und auch (e', e) in der co-Relation enthalten. Die Laufzeit der Konstruktion der co-Relation hat damit eine Laufzeitkomplexität in $O(|E|^2)$.
Aus der co-Relation des Szenarios wird nun die Menge der Schnitte des Szenarios berechnet. Ein Schnitt ergibt sich dabei als eine maximale Menge zueinander geordneter Ereignisse des ungerichteten Graphen (E, co). In der Graphentheorie bezeichnet man eine solche maximale Menge als maximale Clique.

Definition 4.3.2 (Clique)

Definition 4.3.2 (Clique)
Sei $G = (E, \sim)$ *ein ungerichteter Graph. Eine Menge* $C \subseteq E$ *heißt Clique des Graphen* G*, falls für alle* $e, e' \in C : e \sim e'$ *gilt.*
Eine Clique C *heißt maximale Clique des Graphen* G*, falls keine andere Clique* C' *in* G *existiert, für die* $C \subset C'$ *gilt.*

Der bekannteste Algorithmus, der die Menge der maximalen Cliquen eines Graphen berechnet, ist der Algorithmus von Coenraad Bron und Joep Kerbosch [30]. Obwohl es Algorithmen mit einer theoretisch besseren Schlechtesten-Fall-Komplexität gibt, gilt der Algorithmus von Bron und Kerbosch als der im Durchschnitt effektivste Algorithmus [32]. Das Hauptproblem bei der Bestimmung aller maximalen Cliquen ist, dass die Anzahl der Cliquen exponentiell mit der Anzahl der Knoten des Graphen wachsen kann und damit auch die Laufzeitkomplexität jedes Algorithmus, der die Menge aller Cliquen konstruiert, in Exponentialzeit liegt.
Der Algorithmus von Bron und Kerbosch ist ein klassischer, rekursiver Backtracking-Algorithmus. Er unterscheidet in jedem Aufruf drei Mengen von Knoten. Die erste Menge C enthält die bis zu diesem Zeitpunkt konstruierte (evtl. noch nicht maximale) Clique, die zweite Menge K ist die Menge der Knoten, die eventuell der bis zu diesem Zeitpunkt konstruierten Clique hinzugefügt werden können, und die dritte Menge X beschreibt Knoten, die dieser Clique nicht mehr hinzugefügt werden, da sie in einem anderen Aufruf bereits betrachtet wurden. Der Algorithmus wird mit den leeren Mengen C und X und der Menge aller Knoten K aufgerufen. Der Algorithmus betrachtet alle Knoten in K. Für jeden dieser Knoten k verschiebt er k aus der Menge K in die Menge C und löscht danach aus den Mengen K und X alle anderen Knoten des Graphen, die nicht mit k verbunden sind, da diese nicht mehr zur Clique C gehören können. Mit diesen neuen Mengen C, K und X ruft sich der Algorithmus erneut auf, bevor er k in die Menge X aufnimmt und den nächsten Knoten der Menge K betrachtet. Sind bei einem Aufruf die Mengen K und X leer, ist in der Menge C eine maximale Clique konstruiert und kann ausgegeben werden.
Die Laufzeit dieses Algorithmus ist zwar in Exponentialzeit, orientiert sich aber stark an der Anzahl der Elemente der co-Relation. Für die meisten Fälle gilt: Um so mehr Nebenläufigkeit in einem Szenario spezifiziert ist, um so länger benötigt dieser Algorithmus.
Zu jedem Schnitt C des Szenarios wird nun der Vorbereich ${}^\bullet C$ bestimmt. In jeder Stelle des S/T-Netzes muss das Schalten der zu den Ereignissen in ${}^\bullet C$ gehörenden Transitionen zusammen mit der Anfangsmarkierung genügend Marken ergeben, dass die zu den Ereignissen im Schnitt C gehörenden Transitionen gleichzeitig aktiviert sind (vgl. Definition 4.3.1). Die Laufzeit dieses Vergleiches ist wieder von der Anzahl der Schnitte abhängig.
Wir fassen die Schritte, mit deren Hilfe wir das Szenario-Verifikations-Problem nach der Definition 4.3.1 entscheiden, in Algorithmus 4.3.1 zusammen.

ALGORITHMUS 4.3.1 (SZENARIO-VERIFIKATION ÜBER SCHNITTE)

Algorithmus 4.3.1 (Szenario-Verifikation über Schnitte)

Eingabe: *Szenario oder Szenario-Spezifikation* $\mathrm{bpo} = (E, <, l)$ *und ein markiertes S/T-Netz* $N = (P, T, W, m_0)$.

Ausgabe: *Entscheidung ob* $\mathrm{bpo} \in L(N, m_0)$ *gilt.*

```
1 Berechne eine topologische Ordnung des Szenarios
2 Berechne die transitive Ordnung mit dem Algorithmus
    von Floyd und Warshall
3 Berechne die co—Relation
4 Berechne die maximalen Schnitte mit dem Algorithmus
    von Bron und Kerbosch
5 Überprüfe für jede Stelle und jeden Schnitt die
    Ungleichung aktivierter Szenarien
```

Algorithmus 4.3.1 ist eine „straight forward"-Lösung des Szenario-Verifikations-Problems. Die ersten drei Zeilen haben eine Laufzeitkomplexität in $O(|E|^3)$. Dabei ist Zeile 1 optional, verbessert aber die durchschnittliche Laufzeit und Zeile 2 wird nur ausgeführt, wenn die Eingabe eine Szenario-Spezifikation ist. Problematisch wird die Komplexität ab Zeile 4, da die Anzahl der maximalen Schnitte exponentiell mit der Größe des eingegebenen Szenarios wachsen kann. Damit sind die maximalen Schnitte durch Zeile 4 nicht in Polynomialzeit berechenbar und in Zeile 5 müssen exponentiell viele Ungleichungen überprüft werden.
Trotz einer theoretisch großen Schlechtester-Fall-Komplexität bietet Algorithmus 4.3.1 einen Vorteil: Ist das Szenario im S/T-Netz nicht ausführbar, existiert eine Menge von Paaren eines Schnittes des Szenarios zusammen mit einer Stelle des S/T-Netzes, die die geforderte Ungleichung nicht erfüllen. Der Algorithmus kann entweder abbrechen, nachdem er das erste solche Paar gefunden hat, oder die gesamte Menge dieser Paare berechnen. Diese Menge beschreibt die Zustände, in denen es im S/T-Netz zu einem Engpass an Ressourcen kommt. Diese Zeitpunkte erlauben eine einfache Fehleranalyse.
Die durchschnittlichen Laufzeiten des Algorithmus 4.3.1 werden in Kapitel 6 diskutiert und mit anderen Möglichkeiten, das Szenario-Verifikations-Problem zu entscheiden, verglichen. Ein entsprechendes Plugin wurde für diese Arbeit für das VipTool implementiert.

4.4 DER PROZESSNETZ ALGORITHMUS

Dieser Abschnitt behandelt eine alternative Charakterisierung der Szenario-Sprache und danach einen entsprechenden Algorithmus, der das Szenario-Verifikations-Problem mit Hilfe dieser entscheidet.
Das klassische Konzept, um halbgeordnetes Verhalten von S/T-Netzen zu beschreiben, sind Prozessnetze [55, 54]. Ein Prozessnetz ist ein Petrinetz, das einen Ablauf eines S/T-Netzes beschreibt. Jede Transition des Prozessnetzes beschreibt das Schalten einer Transition und jede Stelle des Prozessnetzes beschreibt eine Marke des S/T-Netzes. Aus diesem Grund heißen in einem Prozessnetz die Transitionen Ereignisse und die Stellen Bedingungen. Jedes Ereignis ist mit dem Namen der zugehörigen Transition und jede Bedingung ist mit dem Namen der Stelle, auf der die zugehörige Marke liegt, beschriftet. Die Kanten zwischen Ereignissen und Bedingungen sind ungewichtet und beschreiben

Schaltvorgänge. Dazu entspricht die Multimenge der Beschriftungen der Bedingungen vor jedem Ereignis dem Multivorbereich der zugehörigen Transition im S/T-Netz. Außerdem entspricht die Multimenge der Beschriftungen von Bedingungen hinter jedem Ereignis genau dem Multinachbereich der zugehörigen Transition. Existiert eine Kante von einer Bedingung zu einem Ereignis, so bedeutet dies, dass das Schalten der zugehörigen Transition die zugehörige Marke konsumiert. Existiert eine Kante von einem Ereignis zu einer Bedingung, so bedeutet dies, dass das Schalten der zugehörigen Transition die zugehörige Marke produziert. Da eine Marke im S/T-Netz nur einmal produziert und einmal konsumiert werden kann, ist vor und hinter jeder Bedingung maximal ein Ereignis. Ein Prozessnetz beschreibt dadurch eine Ausführung des S/T-Netzes und enthält keine Alternativen mehr.

Definition 4.4.1 (Prozess)

Definition 4.4.1 (Prozess)
Ein Prozessnetz ist ein Tripel $O = (B, E, F)$*, dabei ist* B *eine Menge von Bedingungen,* E *eine Menge von Ereignissen und* $F \subset (B \times E) \cup (E \times B)$ *eine Menge von Kanten. Der Graph* $(B \cup E, F)$ *ist azyklisch und die Bedingungen sind unverzweigt, d.h. für alle* $b \in B$ *gilt* $|{}^\bullet b|, |b^\bullet| \leqslant 1$.
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz. Ein Prozess von* (N, m_0) *ist ein Paar* $K = (O, \rho)$ *bestehend aus einem Prozessnetz* $O = (B, E, F)$ *und einer Beschriftungsfunktion* $\rho : (B \cup E) \to (S \cup T)$*, welches folgende Eigenschaften erfüllt:*

$$\rho(B) \subseteq P \text{ und } \rho(E) \subseteq T,$$

$$\forall e \in E\ \forall p \in P: \quad |\{b \in {}^\bullet e \mid \rho(b) = p\}| = W(p, \rho(e)) \wedge$$

$$|\{b \in e^\bullet \mid \rho(b) = p\}| = W(\rho(e), p),$$

$$\forall p \in P: \quad |\{b \in B \mid {}^\bullet b = \emptyset, \rho(b) = p\}| = m_0(p).$$

Abbildung 21 zeigt links ein S/T-Netz und rechts einen Prozess dieses Netzes. Im Prozess sind die Beschriftungen der Ereignisse und Bedingungen in den Knoten notiert. Die beiden mit s_1 und s_2 beschrifteten Bedingungen haben einen leeren Vorbereich und repräsentieren die Anfangsmarkierung des S/T-Netzes. Das mit t_1 beschriftete Ereignis konsumiert die eine Bedingung und das mit t_2 beschriftete Ereignis konsumiert die andere. Beide Ereignisse produzieren je zwei Marken in der Stelle s_3 im S/T-Netz und damit vier mit s_3 beschriftete Bedingungen im Prozess. Transition t_3 schaltet und konsumiert drei dieser Marken, während Transition t_4 die letzte Marke konsumiert. Die Vor- und Nachbereiche der Ereignisse stimmen mit der Schaltregel der zugehörigen Transitionen überein.

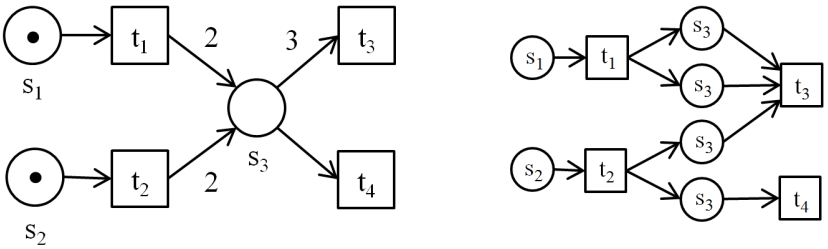

Abbildung 21: Links: Ein markiertes S/T-Netz. Rechts: Ein Prozess.

Das Beispiel verdeutlicht, dass ein Prozess Bedingungen bezüglich ihrer Vergangenheit und ihrer Identität unterscheidet. Aus diesem Grund spricht man von einer „individual token semantics“ [53]. Ein Prozess enthält die komplette Information über eine Ausführung eines S/T-Netzes. Er beschreibt alle Ereignisse und neben den Abhängigkeiten zwischen den Ereignissen die zugehörigen Ressourcen, die die Abhängigkeiten erzeugen. Ein Prozess beinhaltet damit auch alle möglichen Zustände, die während einer Ausführung des S/T-Netzes durchlaufen werden, sowie die Verteilung der Ressourcen von Ereignis zu Ereignis. Durch diese hohe Informationsdichte ist die Menge der zu einem S/T-Netz gehörenden Prozesse groß. Die Anzahl der Prozesse kann exponentiell mit der Anzahl der Knoten des S/T-Netzes wachsen.
Um die Szenario-Sprache eines S/T-Netzes über Prozesse zu charakterisieren, abstrahieren wir für einen gegeben Prozess von den Bedingungen und betrachten die transitiven Abhängigkeiten zwischen den Ereignissen. Auf diese Weise entstehen Szenarien, die Prozess-Szenarien genannt werden. Die Menge der Sequentialisierungen der Prozess-Szenarien ergibt die Szenario-Sprache eines S/T-Netzes [64, 87].

DEFINITION 4.4.2 (PROZESS-SZENARIO)
Sei $K = (O, \rho)$ *mit* $O = (B, E, F)$ *ein Prozess eines markierten S/T-Netzes* (N, m_0). *Das Szenario* $bpo = (E, F^*|_{(E\times E)}, \rho|_E)$ *heißt Prozess-Szenario von* (N, m_0).

Definition 4.4.2 (Prozess-Szenario)

SATZ 4.4.1 (ÜBER PROZESS-SZENARIEN)
Die Menge $L(N, m_0)$ *aller Sequentialisierungen von Prozess-Szenarien von* (N, m_0) *ist die Szenario-Sprache von* (N, m_0).

Satz 4.4.1 (über Prozess-Szenarien)

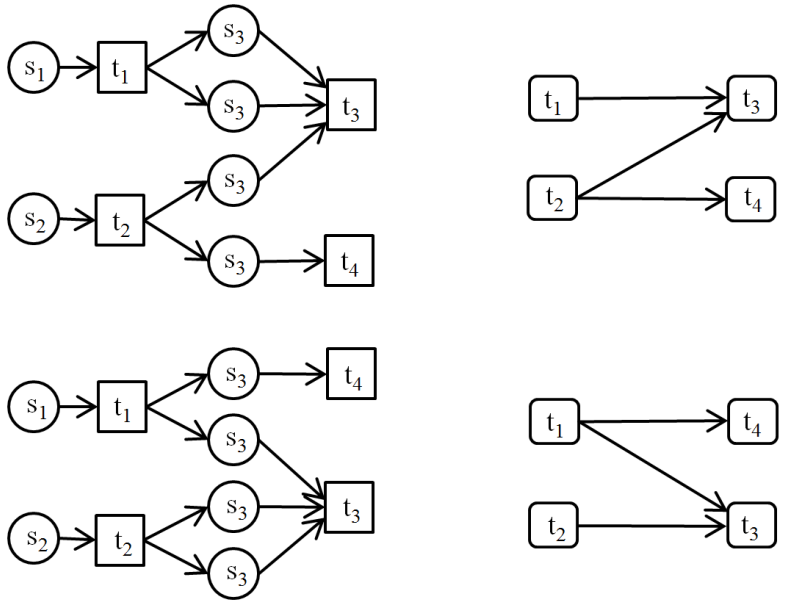

Abbildung 22: Links: Zwei Prozesse des S/T-Netzes aus Abbildung 21. Rechts: Die zwei zu den Prozessen gehörenden Prozess-Szenarien.

Abbildung 22 zeigt links zwei Prozesse des S/T-Netzes aus Abbildung 21 und rechts die zugehörigen Prozess-Szenarien. Abstrahiert man von den Bedingungen, ist der Begriff eines Szenarios äquivalent zum klassischen Konzept der Prozesse.

Mit Hilfe von Satz 4.4.1 können wir einen alternativen Test beschreiben, der das Szenario-Verifikations-Problem entscheidet. Sei dazu ein markiertes S/T-Netz (N, m_0) mit $N = (P, T, W)$ und ein Szenario $bpo = (E, <, l)$ gegeben. Zu dem Szenario bpo versucht man ein Prozess-Szenario des S/T-Netzes zu konstruieren, so dass das Szenario dem Prozess-Szenario entspricht oder dieses sequentialisiert.
Konstruiert man einen Prozess zu einem S/T-Netz, so beginnt man damit, für jede Marke der Anfangsmarkierung eine Bedingung im Prozessnetz zu erzeugen. Findet man unter diesen Bedingungen eine Menge, deren Multimenge von Beschriftungen mit einem Multivorbereich einer Transition t übereinstimmt, konstruiert man ein neues, mit t beschriftetes Ereignis e und fügt im Prozess Kanten von den Bedingungen zu e ein. Danach erzeugt man entsprechend dem Multinachbereich von t eine neue Menge von Bedingungen und ordnet diese hinter e. Dieses Anhängen von neuen Ereignissen wird iteriert, und in jedem neuen Schritt kommen alle Bedingungen mit leerem Nachbereich als Vorbereich des neuen Ereignisses in Frage. Auf diese Weise bleiben die Bedingungen unverzweigt. Die Reihenfolge, in der man die Transitionen betrachtet, und die Auswahl der Bedingungen, an denen der Prozess erweitert wird, bestimmt den konstruierten Prozess.
Es existieren Entfaltungs-Algorithmen, die zu einem S/T-Netz die Menge seiner Prozesse berechnen [73, 48, 21]. Ist diese Prozess-Sprache unendlich groß, so kann jeder Prozess nur bis zu einer festgelegten Anzahl von Ereignissen konstruiert werden. Da die Menge der Transitionen endlich ist, ist diese eingeschränkte Prozess-Sprache endlich. Bei diesen Algorithmen wird nicht nur eine, sondern es werden in jeder Iteration alle Möglichkeiten betrachtet, einen Prozess um Ereignisse zu erweitern. Ist die Prozess-Sprache berechnet, kann man diese mit dem gegeben Szenario vergleichen. Leider eignet sich diese Methode nur bedingt, da die Anzahl der Prozesse oft exponentiell mit der Größe des S/T-Netzes wächst. Die Berechnung der kompletten Prozess-Sprache ist damit viel zu aufwändig.
Um das Szenario-Verifikations-Problem mit Hilfe von Satz 4.4.1 zu entscheiden, müssen wir darauf achten, nicht die gesamte Prozess-Sprache des S/T-Netzes zu konstruieren. Für diese Arbeit beschreiben wir ein Verfahren, eine möglichst kleine Menge von Prozess-Szenarien zu konstruieren, die zur Entscheidung des Szenario-Verifikations-Problems genügt. Dazu stützen wir uns auf die Arbeitsweise der Entfaltungs-Algorithmen, schränken die Menge der produzierten Prozesse jedoch durch folgende Ideen ein:

- Es genügt, die Prozesse zu betrachten, deren Ereignismenge mit den Ereignissen des Szenarios übereinstimmt.
- Ein Prozess, dessen Ordnung das gegebene Szenario respektiert, kann durch Konstruktion seiner Ereignisse in einer beliebigen, das Szenario respektierenden Ordnung, entstehen.
- Das Unterscheiden von Bedingungen mit „gleicher“ Vergangenheit führt zu isomorphen Prozessen.
- Um die Szenario-Sprache zu berechnen, kann von den Bedingungen so weit wie möglich abstrahiert werden.

Algorithmus 4.4.1 entscheidet das Szenario-Verifikations-Problem mit Hilfe der Definition 4.4.2. Er realisiert die aufgezählten Ideen auf der Basis eines Entfaltungs-Algorithmus.

ALGORITHMUS 4.4.1 (SZENARIO-VERIFIKATION ÜBER PROZESSE)

Algorithmus 4.4.1 (Szenario-Verifikation über Prozesse)

Eingabe: *Szenario oder Szenario-Spezifikation* $\mathtt{bpo} = (E, <, l)$
ein markiertes Petrinetz $N = (P, T, W, m_0)$
Ausgabe: *Entscheidung ob* $\mathtt{bpo} \in L(N, m_0)$ *gilt*

```
1   index ← topologische Ordnung auf E
2   B[0] ← m0
3   O ← neue (|E| × |E|)–Matrix
4   Sprache ADD (B, O)
5   FOREACH e ∈ index
6   {
7       FOREACH (B, O) ∈ Sprache
8       {
9           Liste ← Verteilungen von •l(e) auf B
10          FOREACH X ∈ Liste
11          {
12              B' ← B − X
13              B'[index(e)] ← l(e)•
14              O' ← erzeuge Abhängigkeiten(O, X)
15              Sprache ADD (B', O')
16          }
17      }
18      Bereinige Sprache
19  }
20  FOREACH (B, O) ∈ Sprache
21  {
22      IF (O ⩽ <) RETURN true
23  }
24  RETURN false
```

In Zeile 1 berechnet Algorithmus 4.4.1 eine topologische Ordnung index des Szenarios. Dieser Algorithmus wurde bereits in Abschnitt 4.3 beschrieben. In der durch index gegebenen Reihenfolge werden die Ereignisse konstruiert und zu Prozessen zusammengesetzt. Der Startpunkt ist der initiale Prozess. Dieser besteht aus der Menge der Bedingungen, die sich aus der Anfangsmarkierung ergeben, zusammen mit der leeren Ordnung auf den zu konstruierenden Ereignissen.
Da im initialen Prozess alle Bedingungen die gleiche Vergangenheit haben, nämlich keine, unterscheidet der Algorithmus gleich beschriftete Bedingungen nicht. Es genügt daher, die Menge der Bedingungen als Multimenge von Stellen zu beschreiben, die durch die Anfangsmarkierung gegeben ist. Das entsprechende Array wird in Zeile 2, die leere Matrix der Abhängigkeiten in Zeile 3 konstruiert. Beide werden in Zeile 4 an die Liste Sprache angehängt, welche nach Ablauf des Algorithmus die konstruierte Menge an Prozessen enthält. Ab dieser Stelle wiederholt sich eine iterative Prozedur.
In der Reihenfolge der topologischen Ordnung index wird jeder Prozess um das nächste Ereignis erweitert (Zeile 5-19). Dazu durchläuft der Algorithmus die Menge der bis zu diesem Zeitpunkt konstruierten Prozesse (Zeile 7). Die Matrix B beschreibt für einen Prozess die hinter jedem Ereignis hängende Menge von Bedingungen mit leerem Nachbereich durch Multimengen von Stellen. Damit werden gleichbeschriftete Bedingungen mit identischem Vorbereich nicht unterschieden. Die Matrix O beschreibt die bis zu diesem Zeitpunkt konstruierten Abhängigkeiten auf der Menge der Ereignisse.

Soll ein neues Ereignis e an einen Prozess angehängt werden, kann man aus der Matrix B des Prozesses und dem Multivorbereich der Transition $l(e)$ alle verschiedenen Möglichkeiten berechnen, das Prozessnetz um e zu erweitern. Alle diese Möglichkeiten werden in der Liste *Liste* gespeichert (Zeile 9). Alle nicht isomorphen Möglichkeiten zu berechnen ist ein kombinatorisches Problem. Wir werden es später in diesem Abschnitt kurz betrachten, um einen Eindruck von der Komplexität des Algorithmus zu gewinnen.
Jede Möglichkeit, ein Ereignis an die Bedingungen zu hängen, wird als Matrix dargestellt. Die Matrix beschreibt, in welcher Anzahl das neue Ereignis Bedingungen von anderen Ereignissen konsumiert. Für jede dieser Matrizen wird das aktuelle Prozessnetz kopiert und das Ereignis entsprechend der Matrix konstruiert (Zeile 10-16). In einem ersten Schritt verringert sich die Anzahl der Bedingungen mit leerem Nachbereich. Im Algorithmus subtrahieren wir deshalb die Matrix X von der Matrix B (Zeile 12). Das Eintreten des Ereignisses erzeugt eine neue Multimenge von Bedingungen, die dem Multinachbereich der zu dem Ereignis gehörenden Transition entspricht. Diese Multimenge ergibt eine neue Zeile an der Position des aktuellen Ereignisses e in der Matrix B (Zeile 13). Die Matrix O wird entsprechend der in diesem Schritt neu eingeführten Abhängigkeiten aktualisiert. Immer dann, wenn ein Ereignis eine Bedingung eines vorangegangenen Ereignisses konsumiert, wird in der Matrix O eine Abhängigkeit notiert (Zeile 14). Das auf diese Weise konstruierte Prozessnetz (B', O') wird nun der Liste *Sprache* angehängt (Zeile 15).
In Zeile 18 hat der Algorithmus die Menge aller Prozesse um das aktuelle Ereignis auf alle möglichen Arten erweitert und in der Liste *Sprache* gespeichert. Aus dieser Liste werden die nicht maximalen Prozesse gelöscht und es wird mit dem nächsten Ereignis fortgefahren. Nachdem der Algorithmus die gesuchte Teilmenge der Prozess-Sprache konstruiert hat, kann er in Zeile 20-24 diese mit dem eingegebenen Szenario vergleichen und das Szenario-Verifikations-Problem entscheiden.

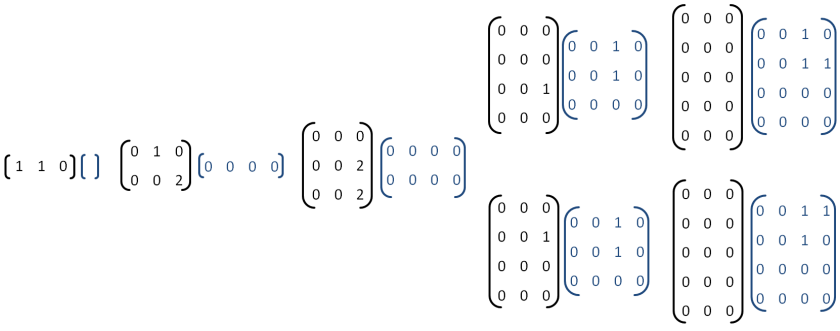

Abbildung 23: Die Szenario-Verifikation über Prozesse.

Abbildung 23 zeigt die Arbeitsweise des Algorithmus 4.4.1 und den Verlauf (von links nach rechts) der Matrizen-Paare (B, O). Der Algorithmus beginnt mit den zwei initialen Matrizen und hängt danach vier Ereignisse an, wodurch neue Matrizen-Paare entstehen. Ab dem vierten Matrizen-Paar besteht die Liste *Sprache* aus zwei Matrizen-Paaren, diese sind untereinander dargestellt. Die Matrizen B (die jeweils linken jedes Paares) beschreiben die noch freien Bedingungen nach jedem Ereignis. Die Matrizen O (die jeweils rechten jedes Paares) beschreiben

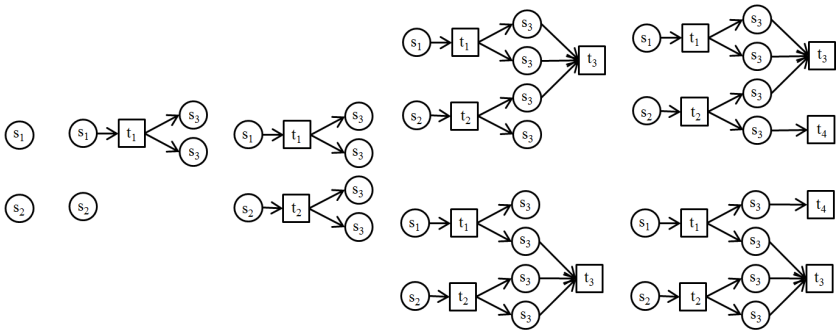

Abbildung 24: Die Prozesse der Szenario-Verifikation über Prozesse.

die bis zu diesem Zeitpunkt konstruierte Ordnung. Vergleicht man die Matrizen-Paare mit ihrer graphischen Darstellung als Prozess aus Abbildung 24, wird noch einmal deutlich, auf welche Weise der Algorithmus die oben beschriebenen Ideen zur Reduktion der Laufzeit umsetzt. Wir fassen unsere Reduktionen zusammen:

- Es werden nur die im zu prüfenden Szenario vorhandenen Ereignisse konstruiert.
- Die Reihenfolge der Ereignisse respektiert eine topologische Ordnung des Szenarios.
- Gleichbeschriftete Bedingungen mit identischem Vorbereich werden nicht unterschieden.
- Informationen über mögliche Zwischenzustände, die Prozesse enthalten, gehen während der Konstruktion verloren.

Trotz dieser Verbesserungen ist die Laufzeit von Algorithmus 4.4.1 kritisch. Die hohe Komplexität wird deutlich, wenn man die Anzahl der verschiedenen Möglichkeiten betrachtet, ein neues Ereignis an eine Menge von Bedingungen anzuhängen. Die Aufgabe alle diese Möglichkeiten aufzuzählen, lässt sich zunächst für jede Stelle einzeln lösen. So betrachtet man für eine Stelle eine Spalte der Matrix B und konstruiert alle Möglichkeiten, Bedingungen so auszuwählen, dass die Summe der Bedingungen der Anzahl der benötigten Bedingungen entspricht. Dabei unterscheiden wir Bedingungen nicht, die von demselben Ereignis produziert werden. Es ergibt sich ein kombinatorisches Problem, welches das Herzstück unseres Algorithmus bildet und seine hohe Komplexität erklärt. Wie für kombinatorische Fragestellungen üblich, übersetzten wir die Fragestellung in ein Urnenmodell.
Für eine Stelle und damit eine Spalte der Matrix B besitzen wir n unterscheidbare und in einer Reihe angeordnete Urnen. Jede Urne steht für ein Ereignis. Jede Urne hat eine nicht-negative Kapazität. Die Kapazität steht für die Anzahl der zur Stelle gehörenden Bedingungen mit leerem Nachbereich hinter dem Ereignis. Dazu haben wir eine Anzahl a von nicht-unterscheidbaren Kugeln und wollen alle verschiedenen Möglichkeiten aufzählen, die Kugeln auf die Urnen zu verteilen. Dieses Verfahren wird durch Algorithmus 4.4.2 implementiert.

Algorithmus 4.4.2 (Verteilungen von a Kugeln auf n Urnen)

ALGORITHMUS 4.4.2 (VERTEILUNGEN VON a KUGELN AUF n URNEN)

Eingabe: K *ein Array der Kapazitäten der Urnen*
a *eine Anzahl von Kugeln*
Ausgabe: Liste *eine Liste von möglichen Verteilungen*

```
1   fertig ← false
2   s ← 0, i ← 0
3   x ← neues |K|-Array
4   WHILE (s < a AND i < |K|)
5   {
6       x[i] ← Minimum(a − s, K[i])
7       s ← s+ Minimum(a − s, K[i])
8       i++
9   }
10  IF (s < a) fertig ← true
11  WHILE (!fertig)
12  {
13      Liste ADD x
14      popped ← false, pushed ← false
15      i ← 0
16      WHILE (!pushed AND !fertig)
17      {
18          IF (i = |K|) fertig ← true
19          ELSEIF (popped AND x[i] < K[i])
20          {
21              x[i]++
22              i−−
23              s ← 0, p ← 0
24              WHILE (i > 0)
25              {
26                  s ← s + x[i]
27                  x[i] ← 0
28                  i−−
29              }
30              WHILE (p < s)
31              {
32                  x[i] ← x[i]+ Minimum(s − p, K[i] − x[i])
33                  p ← p+ Minimum(s − p, K[i] − x[i])
34                  i++
35              }
36              pushed ← true
37          }
38          ELSEIF (!popped AND x[i] > 0)
39          {
40              x[i]−−
41              popped ← true
42          }
43          i++
44      }
45  }
46  RETURN Liste
```

Wir betrachten eine Ordnung auf der Menge aller verschiedenen Verteilungen der Kugeln. Eine Verteilung x ist größer als eine andere Verteilung x', falls x in der Urne mit dem höchsten Index, in dem sich die beiden Verteilungen unterscheiden, mehr Kugeln besitzt als x'. Algorithmus 4.4.2 konstruiert in den Zeilen 1 bis 10 das nach dieser

Ordnung minimale Element der möglichen Verteilungen. Dazu werden die Urnen der Reihe nach gefüllt, bis alle Kugeln verteilt sind. Gelingt es nicht, alle Kugel zu verteilen, existiert keine mögliche Verteilung und der Algorithmus bricht ab (vgl. Zeile 10). Zeilen 11-45 beschreiben den iterativen Teil des Algorithmus. Um aus einer gegebenen Verteilung die nächstgrößere Verteilung zu konstruieren, sucht man zunächst eine nicht-leere Urne u mit möglichst kleiner Ordnung und bewegt eine Kugel dieser Urne in die nächstgrößere Urne u', deren Kapazität noch nicht erschöpft ist. In einem zweiten Schritt werden alle Kugeln aus den Urnen mit einer Ordnung kleiner als u' wieder auf die ersten Urnen verteilt, um eine Verteilung möglichst kleiner Ordnung zu konstruieren. Auf diese Weise werden alle Möglichkeiten, die Kugeln auf die Urnen zu verteilen, aufgezählt.
Algorithmus 4.4.2 wird für jede Spalte der Matrix B durchgeführt. Alle Kombinationen der verschiedenen Möglichkeiten der einzelnen Spalten ergeben die Matrizen X in Algorithmus 4.4.1.
Die Schlechtester-Fall-Komplexität des Algorithmus 4.4.1 ist in Exponentialzeit. Die durchschnittlichen Laufzeiten werden in Kapitel 6 diskutiert und mit anderen Möglichkeiten, das Szenario-Verifikations-Problem zu entscheiden, verglichen. Ein entsprechendes Plugin wurde für diese Arbeit für das VipTool implementiert.
Ein kurzer Vergleich des Algorithmus 4.4.1 mit einem im VipTool bereits existierenden Plugin, das die Prozess-Sprache eines gegebenen S/T-Netzes berechnet [23], zeigt, dass Algorithmus 4.4.1 für viele S/T-Netze das Szenario-Verifikations-Problem entscheiden kann, während der Entfaltungs-Algorithmus [23] nicht in der Lage ist, die Prozess-Sprache des S/T-Netzes zu berechnen. Dieses Resultat ist nicht überraschend und legitimiert unsere Bemühungen, die Menge der zu berechnenden Prozesse so weit wie möglich einzuschränken.

4.5 DER MARKENFLUSS ALGORITHMUS

In diesem Abschnitt wird eine dritte Charakterisierung, die sogenannten Markenflüsse, der Szenario-Sprache eines S/T-Netzes beschrieben und zwei Algorithmen vorgestellt, die das Szenario-Verifikations-Problem mit Hilfe dieser Charakterisierung entscheiden.
Ein Markenfluss ist eine Abbildung aus der Menge der Abhängigkeiten eines Szenarios in die nicht negativen ganzen Zahlen. Ein Markenfluss ist gültig, wenn er bezüglich des S/T-Netzes gewisse Eigenschaften erfüllt. Der Begriff eines gültigen Markenflusses sowie der entscheidende Satz, dass diese Charakterisierung äquivalent zu den Begriffen der Aktiviertheit und der Ausführbarkeit ist, geht auf Gabriel Juhas, Robert Lorenz und Jörg Desel zurück [60]. Die erste Implementierung eines Algorithmus, der das Szenario-Verifikations-Algorithmus mit Hilfe dieser Definition entscheidet, wurde in [14, 11] vorgestellt.
Diese Charakterisierung bezieht sich direkt auf die Menge der Abhängigkeiten eines Szenarios. Dadurch ergibt sich eine entscheidende Verbesserung, da die Anzahl der Abhängigkeiten, anders als die Anzahl der Schnitte eines Szenarios oder die Anzahl der Prozesse eines S/T-Netzes, nicht exponentiell mit der Größe des Szenarios oder des S/T-Netzes wächst. Dies ist entscheidend, um das Szenario-Verifikations-Problem effizient zu lösen.
Ein Markenfluss beschreibt, wie Marken während einer Ausführung des Szenarios zwischen den Ereignissen verteilt werden. Betrachten wir

ein Szenario mit zwei Ereignissen e_1 und e_2. Ist das Ereignis e_2 hinter das Ereignis e_1 geordnet, kann das Ereignis e_1 in einer Stelle Marken produzieren, die e_2 bei seinem Eintreten konsumiert. Ist e_2 in einem Szenario nicht hinter das Ereignis e_1 geordnet, aber auf Marken von e_1 angewiesen, beschreibt dieses Szenario nicht Verhalten des S/T-Netzes. Aus dieser Überlegung ergibt sich folgender Sachverhalt: In ausführbaren Szenarien gibt es die Möglichkeit entlang der Abhängigkeiten Marken von Ereignis zu Ereignis weiterzugeben, so dass jedes Ereignis genügend Marken erhält, um eintreten zu können. Wie viele Marken ein Ereignis benötigt und wie viele Marken jedes Ereignis weitergeben darf, wird durch den Multivorbereich und den Multinachbereich der zu dem Ereignis gehörenden Transition im S/T-Netz beschrieben.
Neben den im Verlauf des Szenarios produzierten Marken existieren zusätzlich die Marken der Anfangsmarkierung des S/T-Netzes. Diese Marken stehen jedem Ereignis zur Verfügung. Sie können frei auf die Ereignisse des Szenarios verteilt werden. Insgesamt können Marken entweder aus der Anfangsmarkierung oder entlang der Abhängigkeiten des Szenarios zu Ereignissen fließen.

Definition 4.5.1 (Markenfluss)

Definition 4.5.1 (Markenfluss)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario. Eine Funktion* $x : (< \cup E) \to \mathbb{N}_0$ *ist ein Markenfluss.* [1]
Für einen Knoten $e \in E$ *und einen Markenfluss* x *ist*

$$zu(e) = x(e) + \sum_{e' < e} x(e', e) \quad \textit{der Markenzufluss von } e \textit{ und}$$

$$ab(e) = \sum_{e < e'} x(e, e') \quad \textit{der Markenabfluss von } e.$$

Eine Menge von Markenflüssen, die auf einem Szenario definiert sind, beschreibt Verhalten eines S/T-Netzes, falls für jede Stelle des S/T-Netzes ein Markenfluss existiert, der die folgenden drei Bedingungen erfüllt.

- Der Markenzufluss zu jedem Ereignis ist so groß wie die Anzahl der Marken, die die zugehörige Transition aus der Stelle konsumiert.
- Der Markenabfluss jedes Ereignisses ist maximal so groß wie die Anzahl der Marken, die die zugehörige Transition in der Stelle produziert.
- Die Summe der Markenflüsse zu Ereignissen aus der Anfangsmarkierung ist maximal so groß wie die Anfangsmarkierung in der Stelle.

Existiert für jede Stelle ein solcher Markenfluss, so beschreibt das Szenario Verhalten des S/T-Netzes. Die Abhängigkeiten des Szenarios respektierend bekommt jedes Ereignis genügend Marken und gibt nicht zu viele Marken weiter. Für eine Stelle nennen wir einen solchen Markenfluss einen gültigen Markenfluss, und wir nennen ein Szenario gültig für ein S/T-Netz (N, m_0), wenn ein gültiger Markenfluss für jede Stelle von (N, m_0) existiert.

1 Diese Definition unterscheidet sich leicht von der ursprünglichen Definition in [60]. Dort wird eine Anfangsmarkierung nicht durch Werte der Funktion auf Knoten, sondern durch ein zusätzliches initiales Ereignis berücksichtigt.

DEFINITION 4.5.2 (GÜLTIGER MARKENFLUSS)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und* $bpo = (E, <, l)$ *ein Szenario mit* $l(E) \subseteq T$. *Für eine Stelle* $p \in P$ *ist ein Markenfluss* x *gültig, falls folgende Eigenschaften gelten:*

Definition 4.5.2 (Gültiger Markenfluss)

(I) *Für jeden Knoten* $e \in E$ *ist* $zu(e) = W(p, l(e))$,

(II) *für jeden Knoten* $e \in E$ *ist* $ab(e) \leqslant W(l(e), p)$ *und*

(III) $\sum_{e \in E} x(e) \leqslant m_0(p)$.

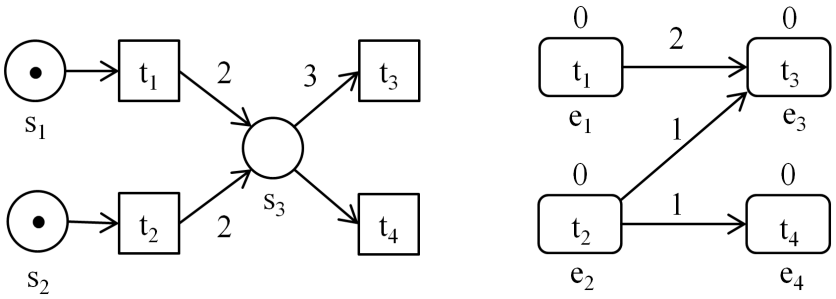

Abbildung 25: Links: Ein S/T-Netz. Rechts: Ein Szenario mit einem für die Stelle s_3 gültigem Markenfluss.

Abbildung 25 zeigt auf der rechten Seite ein Szenario mit einem für die Stelle s_3 gültigen Markenfluss zu dem S/T-Netz auf der linken Seite. Der Markenfluss beschreibt Verhalten des S/T-Netzes bezüglich s_3 in folgender Weise: Die Anzahl der Marken, die ein Ereignis aus der Anfangsmarkierung konsumiert, ist der Wert des Markenflusses auf dem Ereignis und über jedem Ereignis notiert. An den Kanten steht die Verteilung der Marken zwischen den Ereignissen. Ereignis e_1 produziert zwei Marken in s_3 für das Ereignis e_3 und Ereignis e_2 produziert eine Marke in s_3 für e_3 und eine Marke in s_3 für e_4. Damit erhält das Ereignis e_3 insgesamt drei und das Ereignis e_4 eine Marke. Alle Ereignisse produzieren und konsumieren Marken entsprechend dem Multivor- und Multinachbereich der ihnen annotierten Transitionen. Der abgebildete Markenfluss ist damit gültig für die Stelle s_3 des S/T-Netzes. Existiert in einem Szenario ein gültiger Markenfluss für jede Stelle, beschreibt das Szenario Verhalten des S/T-Netzes. Die Menge aller gültigen Szenarien ist eine weitere Charakterisierung der Szenario-Sprache [60].

SATZ 4.5.1 (ÜBER GÜLTIGE MARKENFLÜSSE)
Die Menge $L(N, m_0)$ *aller Szenarien, für die zu einem markierten S/T-Netz* (N, m_0) *zu jeder Stelle von* N *ein gültiger Markenfluss existiert, ist die Szenario-Sprache von* (N, m_0).

Satz 4.5.1 (über gültige Markenflüsse)

Für den Beweis dieses Satzes wird die Menge der Prozesse eines S/T-Netzes mit der Menge der gültigen Szenarien verglichen. Beschreiben ein Prozess und ein Szenario die gleiche Ordnung zwischen Ereignissen, so lässt sich aus dem Prozess für jede Stelle p ein gültiger

Markenfluss konstruieren. Die Anzahl der mit p beschrifteten Bedingungen zwischen Ereignissen ergibt den Wert des Markenflusses auf der entsprechenden Kante. Bedingungen der Anfangsmarkierung werden dem Ereignis, welches diese konsumiert, direkt zugeordnet. Dass der so erzeugte Markenfluss gültig ist, geht direkt aus den Anforderungen an einen Prozess hervor. Mit den gleichen Überlegungen lässt sich aus der Menge gültiger Markenflüsse ein Prozess des S/T-Netzes konstruieren. Dazu wird für jede Stelle der entsprechende Markenfluss auf den Kanten in Bedingungen übersetzt bzw. der Markenfluss auf Ereignissen des Szenarios ergibt Bedingungen mit leerem Vorbereich vor dem entsprechenden Ereignis im Prozess. Insgesamt halten wir fest, dass Satz 4.5.1 zeigt, dass das Szenario-Verifikations-Problem über eine Konstruktion gültiger Markenflüsse in Szenarien entscheidbar ist.
Die zentrale Idee zur Konstruktion eines gültigen Markenflusses ist die Übersetzung in ein Fluss-Maximierungs-Problem. Sie geht auf Robert Lorenz zurück und wurde erstmalig in [60] und später ausführlich in [66] beschrieben. Dabei beschreibt [60] eine iterative und [66] eine direkte Variante der Konstruktion.
Bei der ersten Variante wird ein Markenfluss so lange umverteilt, bis er entweder gültig, oder ein Umverteilen nicht mehr möglich ist. Dieser Algorithmus ist in [11] implementiert und ausführlich diskutiert worden (siehe [14] für einen Überblick). Die Ideen der iterativen Variante werden im nächsten Unterabschnitt dargestellt.
Bei der zweiten Variante wird ein für eine Stelle gültiger Markenfluss konstruiert, indem man ein einziges Fluss-Maximierungs-Problem löst. Entweder kann man aus dieser Lösung einen gültigen Markenfluss ablesen, oder es existiert keiner. Dieser Algorithmus hat eine theoretisch bessere Laufzeit als die erste Variante, liefert aber für den Fall, dass kein gültiger Markenfluss existiert, keine Informationen darüber, an welcher Stelle die Konstruktion gescheitert ist [66]. Diese Variante wurde bislang allerdings nur theoretisch untersucht und ist noch nicht implementiert und getestet worden. Die Ideen der direkten Variante werden im übernächsten Unterabschnitt dargestellt.

4.5.1 *Der iterative Test*

Um das Szenario-Verifikations-Problem mit Hilfe einer iterativen Konstruktion eines gültigen Markenflusses zu entscheiden, betrachten wir ein S/T-Netz (N, m_0) mit $N = (P, T, W)$, ein Szenario $bpo = (E, <, l)$ und eine konkrete Stelle $p \in P$. Zunächst werden die Ereignisse des Szenarios topologisch sortiert. Dadurch erhält jedes Ereignis einen Index. Dann konstruiert man einen Markenfluss, der die Eigenschaft (I) aus Definition 4.5.2 für alle Ereignisse erfüllt. Danach stellt man in einem zweiten Schritt die Eigenschaft (III) sicher. In der Reihenfolge der topologischen Ordnung wird dann für jedes Ereignis die Eigenschaft (II) erfüllt, ohne die bis zu diesem Zeitpunkt bereits erreichten Eigenschaften zu verletzten. Gelingt dies für alle Ereignisse, ist ein gültiger Markenfluss konstruiert.
Wir definieren den initialen Markenfluss eines Szenarios zu einer Stelle eines S/T-Netzes. Dieser besitzt auf jedem Ereignis e den Wert $W(p, l(e))$ und den Wert 0 auf jeder Kante.

DEFINITION 4.5.3 (INITIALER MARKENFLUSS)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario und* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz. Für eine Stelle* $p \in P$ *heißt die Funktion* $x : (< \cup E) \to \mathbb{N}_0$ *mit*

$$x(a) = \begin{cases} W(p, l(a)), & \text{falls } a \in E, \\ 0, & \text{sonst} \end{cases}$$

der zu (N, m_0) *und* p *gehörende initiale Markenfluss.*

Definition 4.5.3 (Initialer Markenfluss)

BEMERKUNG 4.5.1 (ÜBER INITIALE MARKENFLÜSSE)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und* $p \in P$ *eine Stelle. Der zu* (N, m_0) *und* p *gehörende initiale Markenfluss erfüllt für alle* $e \in E : zu(e) = W(p, l(e))$ *und damit Eigenschaft (I).*

Bemerkung 4.5.1 (über initiale Markenflüsse)

In einem weiteren Schritt wollen wir zusätzlich Eigenschaft (III) sicherstellen. Minimale Ereignisse des Szenarios können ihre Marken nur aus der Anfangsmarkierung erhalten, nicht minimale Ereignisse können ihre Marken auch von anderen Ereignissen erhalten. Für einen initialen Markenfluss x kann man den Markenfluss von jedem nicht minimalen Ereignis e' auf eine Kante (e, e') verschieben und so den Markenabfluss der Anfangsmarkierung verringern. Diese Modifikation erhält die Eigenschaft (I) für das Ereignis e', da sich der Markenzufluss nicht ändert. Wir führen diese Modifikationen so lange durch, bis der Markenfluss Eigenschaft (III) erfüllt oder nur noch auf minimalen Ereignissen einen positiven Wert besitzt. Kann man Eigenschaft (III) nicht erreichen, so ist das Szenario-Verifikations-Problem entschieden: Der minimale Schnitt des Szenarios ist nicht in der Anfangsmarkierung aktiviert und das Szenario nicht in der Szenario-Sprache des S/T-Netzes enthalten. Gilt Eigenschaft (III), erhalten wir einen Markenfluss, der Eigenschaft (I) und (III) erfüllt.

DEFINITION 4.5.4 (BIS i GÜLTIGER MARKENFLUSS)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und* $\mathrm{index} : E \to \mathbb{N}$ *eine topologische Ordnung auf* $(E, <)$*. Ein Markenfluss* x *ist für eine Stelle* $p \in P$ *bis zum Index* $i \in \mathbb{N}_0$ *gültig, falls folgende Eigenschaften gelten:*

Definition 4.5.4 (Bis i gültiger Markenfluss)

(I) *Für jeden Knoten* $e \in E$ *gilt* $zu(e) = W(p, l(e))$,

(II)^i *für jeden Knoten* $e \in E$ *mit* $\mathrm{index}(e) \leqslant i$ *gilt* $ab(e) \leqslant W(l(e), p)$,

(III) $\displaystyle\sum_{e \in E} x(e) \leqslant m_0(p)$.

Ist ein Markenfluss gültig bis zum Index i *und nicht gültig bis zum Index* $i + 1$*, so nennen wir* $i + 1$ *den kritischen Index.*

Wurde der initiale Markenfluss so modifiziert, dass er zusätzlich die Eigenschaft (III) erfüllt, ist ein bis zum Index 0 gültiger Markenfluss für die Stelle p konstruiert. Durch Modifikationen versuchen wir in den nächsten Schritten, einen bis zum Index $|E|$ gültigen Markenfluss zu konstruieren.

Bemerkung 4.5.2 (über bis i gültige Markenflüsse)

BEMERKUNG 4.5.2 (ÜBER BIS I GÜLTIGE MARKENFLÜSSE)
Sei $bpo = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und* $p \in P$ *eine Stelle. Ein für* p *bis zum Index* $|E|$ *gültiger Markenfluss ist gültig für* p.

Zu einem bis i gültigen Markenfluss definieren wir für die Anfangsmarkierung des S/T-Netzes und für jedes Ereignis des Szenarios das Potential, eine Funktion $\pi : (m_0 \cup E) \to \mathbb{Z}_\infty$. Das Potential gibt den Wert an, um den man den Markenabfluss der Ereignisse oder der Anfangsmarkierung erhöhen kann, wenn man einen bis $i+1$ gültigen Markenfluss konstruieren will. Für ein Ereignis e mit einem Index kleiner oder gleich $i+1$ ist dieses Potential $\pi(e) = W(l(e), p) - ab(e)$. Für ein Ereignis e mit einem Index größer als $i+1$ ist dieses Potential beliebig groß und wir definieren $\pi(e) = \infty$. Das Potential der Anfangsmarkierung ergibt sich als $\pi(m_0) = m_0(p) - \sum_{e \in E} x(e)$.
Sei x ein Markenfluss mit kritischem Index $i+1$, so ist das Potential des Ereignisses e_{i+1} negativ. Wir suchen ein Ereignis e mit positivem Potential und versuchen das Potential des Ereignisses e auf das Ereignis e_{i+1} zu verschieben, ohne die Markenzu- und Markenabflüsse anderer Ereignisse zu verändern. Wir wiederholen diesen Schritt, bis das Potential von e_{i+1} nicht-negativ ist und e_{i+1} damit Eigenschaft (II) erfüllt. An dieser Stelle ist ein bis $i+1$ gültiger Markenfluss konstruiert.
Bleibt das Potential von e_{i+1} negativ, beschreibt die Menge der Ereignisse mit einem Index kleiner als $i+1$ ein Präfix des Szenarios, welches Verhalten des S/T-Netzes beschreibt. Nach dem Ausführen dieses Präfixes ist der nächste Schnitt hinter diesem Präfix jedoch nicht aktiviert, da kein Ereignis die benötigten Marken für das Ereignis e_{i+1} zu Verfügung stellen kann (siehe z.B. [11] für eine formale Argumentation).
Es existieren drei Modifikationen $\mathfrak{M}_1, \mathfrak{M}_2, \mathfrak{M}_3$, die Potential zwischen zwei Ereignissen verschieben. Diese Modifikationen verändern die Markenzu- und Markenabflüsse aller anderen Ereignisse oder der Anfangsmarkierung nicht. Sei x ein bis i gültiger Markenfluss und $\delta \in \mathbb{N}$ die Höhe des Potentials, das zwischen zwei Ereignissen oder zwischen einem Ereignis und der Anfangsmarkierung verschoben werden soll.

$\mathfrak{M}_1$ Existieren zwei Kanten (e, e'') und (e', e''), ist $\pi(e') \geqslant \delta$ sowie $x(e, e'') \geqslant \delta$, so kann man den Markenfluss auf der Kante (e, e'') um den Wert δ verringern und den Markenfluss auf der Kante (e', e'') um den Wert δ erhöhen. Das Potential von e erhöht sich und das Potential von e' verringert sich um den Wert δ.

$\mathfrak{M}_2$ Existiert eine Kante (e, e''), ist $\pi(m_0) \geqslant \delta$ sowie $x(e, e'') \geqslant \delta$, so kann man den Markenfluss auf der Kante (e, e'') um den Wert δ verringern und den Markenfluss auf dem Ereignisses e'' um den Wert δ erhöhen. Das Potential von e erhöht sich und das Potential der Anfangsmarkierung verringert sich um den Wert δ.

$\mathfrak{M}_3$ Existiert eine Kante (e', e''), ist $\pi(e') \geqslant \delta$ sowie $x(e'') \geqslant \delta$, so kann man den Markenfluss auf dem Ereignis e'' um den Wert δ verringern und den Markenfluss auf der Kante (e', e'') um den Wert δ erhöhen. Das Potential der Anfangsmarkierung erhöht sich und das Potential des Ereignis e' verringert sich um den Wert δ.

Abbildung 26: Die Modifikation $\mathfrak{M}_1$.

Abbildung 26 zeigt ein Beispiel der Modifikationen $\mathfrak{M}_1$. Das Potential ist in den Kreisen an den Ereignissen dargestellt. Zwei Kanten führen zu dem Ereignis e''. Verschiebt man Markenfluss zwischen diesen Kanten, bleibt der Markenzufluss von e'' erhalten. Potential verschiebt sich von e' zu e.

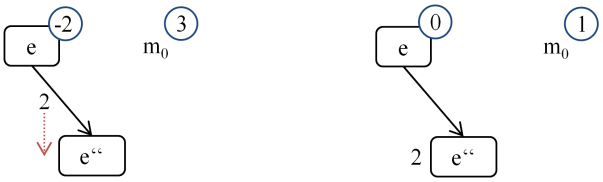

Abbildung 27: Die Modifikation $\mathfrak{M}_2$.

Abbildung 27 zeigt ein Beispiel der Modifikationen $\mathfrak{M}_2$. Eine Kante führt zu dem Ereignis e''. Verschiebt man Markenfluss von dieser Kante auf das Ereignis e'', bleibt der Markenzufluss von e'' erhalten. Das Potential verschiebt sich von der Anfangsmarkierung zu e.

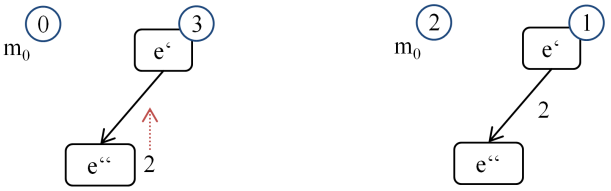

Abbildung 28: Die Modifikation $\mathfrak{M}_3$.

Abbildung 28 zeigt ein Beispiel der Modifikationen $\mathfrak{M}_3$. Eine Kante führt zu dem Ereignis e''. Verschiebt man Markenfluss von Ereignis e'' auf diese Kante, bleibt der Markenzufluss von e'' erhalten. Das Potential verschiebt sich von e' zur Anfangsmarkierung.
Um Potential über das gesamte Szenario hinweg zu verschieben, werden einzelne Modifikationen zu Modifikationsfolgen gekoppelt. Wird auf ein Ereignis e Potential geschoben und wird e direkt zum Ausgangspunkt einer weiteren Modifikation, hebt sich die Veränderung des Potentials von e auf. Auf diese Weise entstehen Zick-Zack-Wege durch das Szenario. Jeder Zick-Zack-Weg setzt sich aus gekoppelten Modifikationen zusammen. Führt man alle diese Modifikationen mit dem gleichen Wert δ durch, so verschiebt sich das Potential vom Anfang der Modifikationsfolge bis zu ihrem Ende. Alle anderen Markenzu- und Markenabflüsse bleiben erhalten.

Definition 4.5.5 (Potential verschiebende Modifikationsfolge)

Definition 4.5.5 (Potential verschiebende Modifikationsfolge)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und eine Folge* $e_1 e_2 \dots e_n \in (E \cup m_0)^*$ *von Ereignissen und der Anfangsmarkierung gegeben.*
Existiert eine Folge $\mu_1 \mu_2 \dots \mu_{n-1}$ *von Modifikationen* $\mathfrak{M}_1$, $\mathfrak{M}_2$ *und* $\mathfrak{M}_3$, *so dass die Modifikation* μ_i *Potential des Elementes* e_i *zu dem Element* e_{i+1} *verschiebt, so nennen wir die Folge der Modifikationen eine Potential verschiebende Modifikationsfolge.*

Um iterativ einen gültigen Markenfluss zu erzeugen, müssen wir die Menge aller Modifikationsfolgen eines Markenflusses mit kritischem Index $i + 1$ in einem Szenario betrachten. Dies geschieht, indem wir das Problem in ein Fluss-Maximierungs-Problem übersetzen. Zu jedem Ereignis werden Knoten im Flussnetzwerk konstruiert und Kanten zwischen den Knoten beschreiben die möglichen Modifikationsfolgen. Ein maximaler Fluss in diesem Flussnetzwerk legt fest, mit welchen Werten die Modifikationen ausgeführt werden, so dass sich genügend Potential zum Ereignis mit dem kritischen Index verschiebt.
Eine einzelne Modifikation hat immer einen Startpunkt, von dem aus Potential geschoben wird. Dieser Startpunkt kann ein Ereignis oder die Anfangsmarkierung sein. Das Potential wird zunächst auf einen Zwischenstop geschoben. Von dort aus erreicht das Potential ein Ziel, ein Ereignis oder die Anfangsmarkierung, das zusätzliches Potential erhält. Wir erstellen für jedes Ereignis zwei Knoten im Flussnetzwerk. Der Erste ist der Start-Knoten, dieser wird als Startpunkt und als Ziel verwendet. Der Zweite ist der Stop-Knoten, über ihn wird Potential verschoben. Später setzen wir die verschiedenen Modifikationen an den zugehörigen Start-Knoten zusammen. Für die Anfangsmarkierung erzeugen wir einen zusätzlichen Start-Knoten im Flussnetzwerk. Diese Knotenmenge ergänzen wir um eine Quelle und eine Senke. Die Quelle ist mit allen Start-Knoten des Flussnetzwerks verbunden, falls das zugehörige Ereignis positives Potential besitzt. Das Kantengewicht dieser Kanten entspricht dem Potential. Der Start-Knoten, der zu dem Ereignis mit dem kritischen Index gehört, ist mit der Senke verbunden. Der Wert dieser Kante entspricht dem Wert des negativen Potentials des Ereignisses. Es beschreibt die Höhe des Potentials, das zu diesem Knoten geschoben werden muss.
Jede Kante des Szenarios und zusätzliche Kanten von der Anfangsmarkierung zu jedem Knoten werden vom entsprechenden Start-Knoten zum entsprechenden Stop-Knoten im Flussnetzwerk eingefügt. Diese Kanten beschreiben den ersten Teil der drei Modifikationen. Für den zweiten Teil fügen wir alle Kanten des Szenarios in umgekehrter Orientierung vom entsprechenden Stop-Knoten zum entsprechenden Start-Knoten ein, falls diese positiven Fluss besitzen. Dazu kommen Kanten vom Stop-Knoten eines Ereignisses zum Start-Knoten der Anfangsmarkierung, falls der Markenfluss auf diesen Ereignissen einen positiven Wert besitzt.
Wir definieren das Flussnetzwerk der Modifikationsfolgen zunächst formal, bevor ein veranschaulichendes Beispiel vorgestellt wird.

DEFINITION 4.5.6 (FLUSSNETZWERK DER MODIFIKATIONSFOLGEN)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario mit* $E = \{e_1, \ldots, e_{|E|}\}$, x *ein bis* i *gültiger Markenfluss in* bpo *und* π *das Potential der Ereignisse in* E. *Wir definieren ein Flussnetzwerk* $G = (K, F, c, q, s)$:

Definition 4.5.6 (Flussnetzwerk der Modifikationsfolgen)

$K = \{k_1, \ldots, k_{2|E|}\} \cup \{k_0, q, s\}$ *und dazu ist* $\mathrm{start} : E \to K$, $\mathrm{stop} : E \to K$ *definiert durch* $\mathrm{start}(e_j) = k_j$ *und* $\mathrm{stop}(e_j) = k_{|E|+j}$.

$$F = F_m \cup F_< \cup F_x \cup F_q \cup F_s \text{ mit}$$

$$F_m = \{(k_0, \mathrm{stop}(e)) \mid e \in E\},$$

$$F_< = \{(\mathrm{start}(e), \mathrm{stop}(e')) \mid e < e'\},$$

$$F_x = \{(\mathrm{stop}(e'), \mathrm{start}(e)) \mid e < e', x(e, e') > 0\}$$

$$\cup \{(\mathrm{stop}(e'), k_0) \mid e' \in E, x(e') > 0\},$$

$$F_q = \{(q, \mathrm{start}(e)) \mid e \in E, \pi(e) > 0\},$$

$$F_s = \{(\mathrm{start}(e_{i+1}), s)\}.$$

$$c(k, k') = \begin{cases} \pi(\mathrm{start}^{-1}(k')), & \text{falls } k = q, \\ x(\mathrm{start}^{-1}(k'), \mathrm{stop}^{-1}(k)), & \text{falls } (k, k') \in F_x, \\ -\pi(\mathrm{start}^{-1}(k)), & \text{falls } k' = s, \\ \infty, & \text{sonst.} \end{cases}$$

Die Kapazität ∞ an Kanten des Flussnetzwerks der Modifikationsfolgen kann man durch eine obere Schranke für den maximalen Fluss ersetzen. Eine mögliche obere Schranke ist der Wert $-\pi(e_{i+1})$.

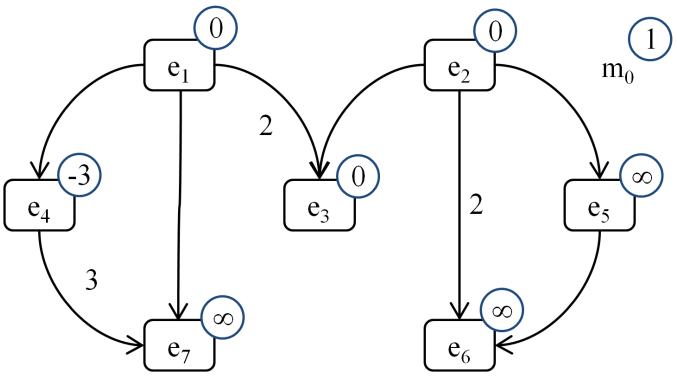

Abbildung 29: Ein Markenfluss x in einem Szenario.

Abbildung 29 zeigt einen Markenfluss in einem Szenario. Der Index j jedes Ereignisses e_j gibt eine topologische Ordnung an. Die Potentiale

sind in den Kreisen an den Ereignissen dargestellt. Der Markenfluss x ist gültig bis zum Index 3, das Potential des Ereignisses e_4 ist -3. Um einen bis zum Index 4 gültigen Markenfluss zu erhalten, müssen wir den Markenabfluss des Ereignisses e_4 um 3 verringern. Die Potentiale der Ereignisse e_5, e_6 und e_7 sind unendlich groß, da ihre Indizes größer als 4 sind. Die Potentiale aller anderen Knoten sind 0 und das Potential der Anfangsmarkierung ist 1.

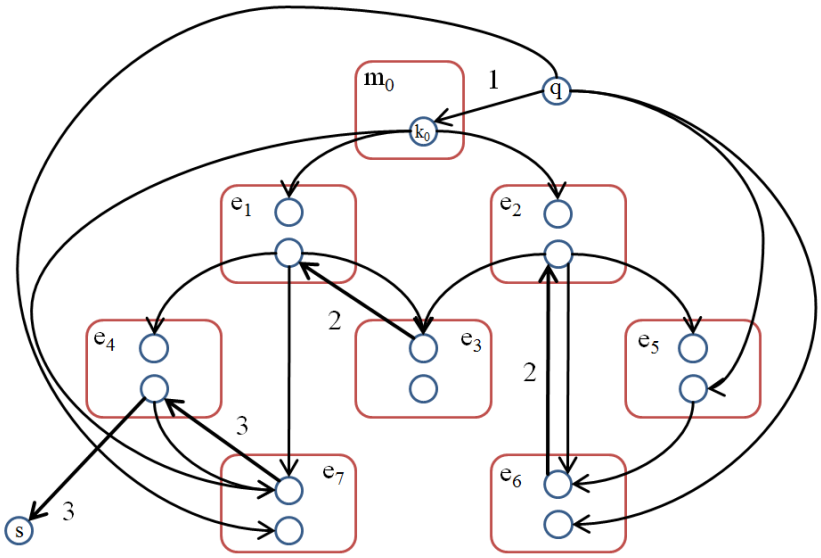

Abbildung 30: Ein Flussnetzwerk der Modifikationsfolgen. Die Menge F_m ist nicht vollständig dargestellt.

Abbildung 30 zeigt das zugehörige Flussnetzwerk der Modifikationsfolgen. Kanten ohne Kantengewicht sollen in dieser Abbildung den Fluss nicht begrenzende Kapazität besitzen. Jedes Ereignis übersetzt sich in ein Paar von Knoten des Flussnetzwerks. In der Abbildung sind die Paare von Knoten in einer mit dem Namen des zugehörigen Ereignisses beschrifteten Kontur gruppiert. Dabei ist der Start-Knoten der untere Knoten jeder Kontur. Der zur Anfangsmarkierung gehörende Knoten k_0 ist mit m_0 beschriftet. Kanten, die die Quelle verlassen, beschreiben das zur Verfügung stehende Potential. Die Kante, die zur Senke führt, beschreibt das von Ereignis e_4 benötigte Potential. Tripel von Knoten bilden die Modifikationen $\mathfrak{M}_1$, $\mathfrak{M}_2$ und $\mathfrak{M}_3$. Diese Modifikationen sind über die Start-Knoten verbunden. Ein Weg von der Quelle zu Senke entspricht somit einer Modifikationsfolge. Ein maximaler Fluss in dem Flussnetzwerk beschreibt das maximale Potential, das im Szenario zu dem Ereignis e_4 verschoben werden kann. Sättigt ein maximaler Fluss die Kante $(\mathtt{start}(e_4), s)$, haben wir eine Menge von Modifikationsfolgen gefunden, durch deren Anwendung ein neuer Markenfluss entsteht, der die Eigenschaft (II) für den Knoten e_4 erfüllt. Da sich die gesamte Modifikation aus den Modifikationen $\mathfrak{M}_1$, $\mathfrak{M}_2$ und $\mathfrak{M}_3$ zusammensetzt, ist der entstehende Markenfluss gültig bis 4.
In Abbildung 30 ist der Wert eines maximalen Flusses 3. Ein Arm des Flusses geht über die Konturen e_5, e_6, e_2, e_3, e_1, e_7, e_4, der andere Arm geht über m_0, e_7, e_4. Daraus ergeben sich in diesem Beispiel zwei Modifikationsfolgen. Die erste verschiebt Potential der Höhe zwei von

Ereignis e_5 über e_2 nach e_1 und zu e_4. Die zweite verschiebt Potential in Höhe eins von der Anfangsmarkierung direkt auf Knoten e_4.

ALGORITHMUS 4.5.1 (SZENARIO-VERIFIKATION ÜBER MARKENFLÜSSE (ITERATIV))

Algorithmus 4.5.1 (Szenario-Verifikation über Markenflüsse (iterativ))

Eingabe: *Szenario oder Szenario-Spezifikation* $\mathrm{bpo} = (E, <, l)$ *und ein markiertes S/T-Netz* $N = (P, T, W, m_0)$
Ausgabe: *Entscheidung ob* $\mathrm{bpo} \in L(N, m_0)$ *gilt*

```
1   Berechne eine topologische Ordnung des Szenarios
2   Berechne die transitive Ordnung des Szenarios
3   FOREACH p ∈ P
4   {
5       x ← Initialer Markenfluss für p
6       x ← Stelle Eigenschaft (III) her
7       j ← kritischer Index
8       WHILE (j ⩽ |E|)
9       {
10          G ← Flussnetzwerk Modifikationsfolgen(p)
11          f ← Maximaler Fluss in G
12          IF (w(f) < −π(e_j)) RETURN false
13          Modifiziere x durch f
14          j ← kritischer Index
15      }
16  }
17  RETURN true
```

Algorithmus 4.5.1 zeigt die Implementierung des Algorithmus, der das Szenario-Verifikations-Problem durch Konstruktion gültiger Markenflüsse entscheidet. In Zeile 1 wird die Menge der Ereignisse topologisch sortiert. In Zeile 2 wird die transitive Hülle des Szenarios berechnet, falls das eingegebene Szenario eine Szenario-Spezifikation ist. In der Schleife, die bei Zeile 3 beginnt, versucht der Algorithmus für jede Stelle p des S/T-Netzes einen gültigen Markenfluss zu konstruieren. Zeile 5 konstruiert den initialen Markenfluss nach Definition 4.5.3 und in Zeile 6 wird Eigenschaft (III) sichergestellt. Dazu wird Markenfluss von nicht minimalen Ereignissen auf Kanten verschoben, die dem Ereignis eingehen. Ist es auf diese Weise nicht möglich, Eigenschaft (III) sicher zu stellen, terminiert der Algorithmus bereits an dieser Stelle. Die Zeilen 7 bis 15 beschreiben die iterative Prozedur. In der Reihenfolge der topologischen Ordnung wird für jedes Ereignis, das Eigenschaft (II) verletzt, das Flussnetzwerk der Modifikationsfolgen berechnet (Zeile 10) und ein maximaler Fluss in diesem konstruiert (Zeile 11). In Zeile 13 wird der Markenfluss dem Fluss entsprechend mit Hilfe der Modifikationen so umverteilt, dass er Eigenschaft $(\mathrm{II})^j$ erfüllt. Ist in einer Iteration der Wert des maximalen Flusses kleiner als das negative Potential des aktuellen Ereignisses, kann der Markenfluss nicht hinreichend modifiziert werden und der Algorithmus terminiert (Zeile 12). Gelingt die Konstruktion für alle Ereignisse und alle Stellen, wird in Zeile 17 eine positive Antwort gegeben.
Betrachten wir im Folgenden die Laufzeit des Algorithmus 4.5.1. In jeder der Iterationen muss das Flussnetzwerk konstruiert, ein maximaler Fluss in diesem berechnet und der Markenfluss modifiziert werden. Die Laufzeit der Konstruktion ergibt sich aus der Größe des Flussnetzwerks. Diese Größe ist bestimmt durch die Anzahl der Knoten

$|K| = 2|E| + 3$ und die Anzahl der Kanten $|F|$. Die Anzahl der Kanten des Flussnetzwerks entspricht in etwa der Anzahl der Kanten des Szenarios und der Menge der Elemente, auf denen der aktuelle Fluss einen Wert größer 0 besitzt. Die Laufzeit der Modifikation ergibt sich direkt aus der Anzahl der Kanten des Szenarios. Die Laufzeiten der Fluss-Maximierungs-Algorithmen wurde bereits ausführlich in Kapitel 3 diskutiert. Verwenden wir den Preflow-Push Algorithmus mit einer Schlechtester-Fall-Komplexität in $O(|K|^3)$, so ist die Schlechtester-Fall-Komplexität des Algorithmus 4.5.1 in $O(|P| \cdot |E|^4)$.
Wir halten an dieser Stelle fest, dass der iterative Test des Szenario-Verifikations-Problems mit Hilfe von Markenflüssen der erste mit einer Laufzeit in Polynomialzeit ist. Obwohl dieser Test sehr aufwändig erscheint, ist er der derzeit schnellste implementierte Algorithmus zur Lösung des Szenario-Verifikations-Problems. In Kapitel 6 werden wir die durchschnittliche Laufzeit dieses Algorithmus testen und mit allen anderen Szenario-Verifikations-Algorithmen vergleichen.

4.5.2 *Der direkte Test*

In diesem Abschnitt entwickeln wir einen Algorithmus, der das Szenario-Verifikations-Problem mit Hilfe von Markenflüssen durch einen direkten Test entscheidet. Dieses Verfahren wurde in [66] vorgestellt und wird für diese Arbeit erstmalig auch implementiert. Die Idee des iterativen Verfahrens ist die schrittweise Konstruktion gültiger Markenflüsse auf einer Folge von größer werdenden Präfixen des Szenarios. Dadurch wird ein gültiger Markenfluss konstruiert. Dieser gültige Markenfluss wird in diesem Abschnitt direkt konstruiert. Für den Fall, dass kein gültiger Markenfluss existiert, verliert man zwar die Information über gültige Markenflüsse auf Präfixen, kann aber die Laufzeit zur Entscheidung des Szenario-Verifikations-Problems deutlich verbessern.
Der direkte Test übersetzt ein gegebenes Szenario $\mathtt{bpo} = (E, <, l)$ zu einem markierten S/T-Netz (N, m_0) mit $N = (P, T, W)$ für eine Stelle $p \in P$ in ein Flussnetzwerk G, so dass sich ein maximaler Fluss im Flussnetzwerk G direkt in einen Markenfluss x in dem Szenario übersetzen lässt. Dazu werden Kanten und Kapazitäten des Flussnetzwerks so konstruiert, dass der Markenfluss x die Eigenschaften (I), (II) und (III) erfüllt, falls für das Szenario ein für p gültiger Markenfluss existiert. Um dieses Flussnetzwerk zu konstruieren, erzeugen wir für jedes Ereignis des Szenarios zwei Knoten im Flussnetzwerk. Der erste Knoten wird dem Ereignis durch eine Abbildung $\mathtt{oben}$, der zweite Knoten wird dem Ereignis durch eine Abbildung $\mathtt{unten}$ zugeordnet. Für ein Ereignis e beschreibt der Fluss durch den Knoten $\mathtt{oben}(e)$ die Anzahl an Marken, die das Ereignis für sein Eintreten erhält. Der Fluss durch den Knoten $\mathtt{unten}(e)$ beschreibt die Anzahl an Marken, die das Ereignis durch sein Eintreten produziert. Weiter konstruieren wir einen Knoten k_0, so dass der Fluss durch diesen Knoten den Markenfluss aus der Anfangsmarkierung beschreibt, eine Quelle q und eine Senke s.
Sind zwei Ereignisse e und e' im Szenario geordnet, so konstruieren wir im Flussnetzwerk eine entsprechende Kante des Knotens $\mathtt{unten}(e)$ zu dem Knoten $\mathtt{oben}(e')$. Besitzt ein Fluss auf dieser Kante einen positiven Wert, steht dieser Wert für Marken, die von dem Ereignis e für das Ereignis e' produziert werden. Die Kapazitäten dieser Kanten sind nicht beschränkt. Insgesamt beschreibt ein Fluss auf diesen Kanten die Verteilung der Marken auf den Abhängigkeiten des Szenarios. Der Knoten

k_0 ist mit allen oben-Knoten verbunden. Fluss auf dieser Kantenmenge beschreibt die Verteilung von Marken aus der Anfangsmarkierung auf alle Ereignisse des Szenarios. Es kommt darauf an, diese Verteilung von Marken so zu begrenzen, dass sie die Eigenschaften (I), (II) und (III) erfüllt. Kein Ereignis e darf zu viele Marken produzieren. Wir begrenzen den Durchfluss jedes Knotens $\mathtt{unten}(e)$, indem wir für jedes Ereignis e eine Kante $(q, \mathtt{unten}(e))$ mit der Kapazität $W(l(e), p)$ konstruieren. Damit jedes Ereignis e genügend Marken bekommt, konstruieren wir für jedes Ereignis e eine Kante $(\mathtt{oben}(e), s)$ mit der Kapazität $W(p, l(e))$ und überprüfen später, ob ein Fluss die Kapazität jeder dieser Kanten ausschöpft. Eigenschaft (III) stellen wir sicher, indem wir den Durchfluss des Knotens k_0 begrenzen. Dies geschieht durch eine weitere Kante (q, k_0) mit der Kapazität $m_0(p)$. Insgesamt ergibt sich das assoziierte Flussnetzwerk. Wenn wir einen Fluss konstruieren können, der alle der Senke eingehenden Kanten füllt, und wir den Fluss auf inneren Kanten des Flussnetzwerks in einen Markenfluss im Szenario übersetzen, respektiert die Verteilung der Marken die Abhängigkeiten des Szenarios. Jedes Ereignis erhält genügend Marken, und weder die Anfangsmarkierung noch die Ereignisse haben einen zu großen Markenabfluss. Ein solcher Markenfluss ist gültig für das Szenario und die Stelle p.

Definition 4.5.7 (Assoziiertes Flussnetzwerk)

Definition 4.5.7 (Assoziiertes Flussnetzwerk)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario mit* $E = \{e_1, \ldots, e_{|E|}\}$, (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz und* $p \in P$ *eine Stelle. Sei* $M_p(\mathrm{bpo}, N) = \sum_{e \in E} W(p, l(e))$ *die Summe der durch die Ereignisse aus der Stelle* p *konsumierten Marken. Wir definieren das assoziierte Flussnetzwerk* $G = (K, F, c, q, s)$*:*

$K = \{k_1, \ldots, k_{2|E|}\} \cup \{k_0, q, s\}$ *und* $\mathtt{oben} : E \to K$, $\mathtt{unten} : E \to K$ *ist definiert durch* $\mathtt{oben}(e_i) = k_i$ *und* $\mathtt{unten}(e_i) = k_{|E|+i}$.

$$
\begin{aligned}
F &= F_m \cup F_< \cup F_q \cup F_0 \cup F_s \text{ mit} \\
F_m &= \{(k_0, \mathtt{oben}(e)) \mid e \in E\}, \\
F_< &= \{(\mathtt{unten}(e), \mathtt{oben}(e')) \mid e < e'\}, \\
F_q &= \{(q, \mathtt{unten}(e)) \mid e \in E\}, \\
F_0 &= \{(q, k_0)\}, \\
F_s &= \{(\mathtt{oben}(e), s) \mid e \in E\}.
\end{aligned}
$$

$$
c(k, k') = \begin{cases} W(l(\mathtt{unten}^{-1}(k')), p), & \text{falls } k = q, k' \neq k_0, \\ m_0(p), & \text{falls } (k, k') = (q, k_0), \\ W(p, l(\mathtt{oben}^{-1}(k))), & \text{falls } k' = s, \\ M_p(\mathrm{bpo}, N), & \text{sonst.} \end{cases}
$$

Abbildung 31 zeigt ein weiteres Beispiel eines markierten S/T-Netzes und ein in dem S/T-Netz ausführbares Szenario. Das markierte S/T-Netz besitzt vier Transitionen und nur eine einzige Stelle. Das Szenario

besteht aus vier Ereignissen. Die Ereignisse sind mit den Namen der Transitionen beschriftet.

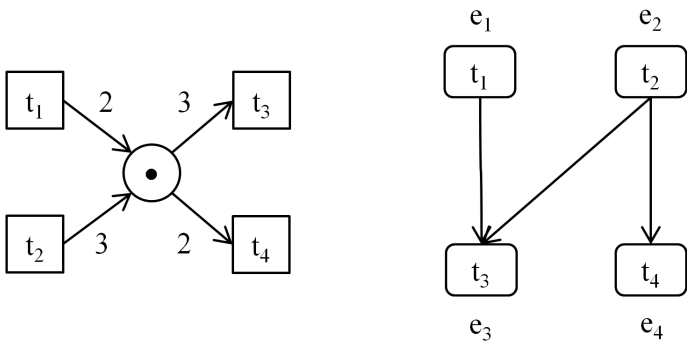

Abbildung 31: Ein markiertes S/T-Netz und ein Szenario.

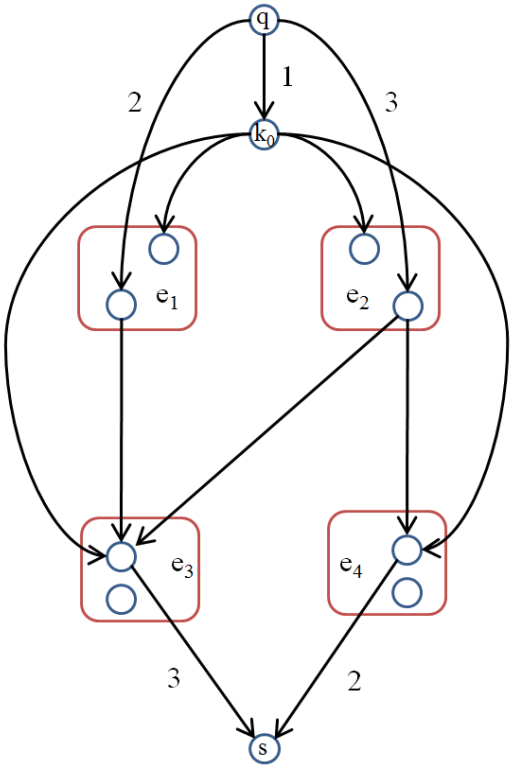

Abbildung 32: Ein assoziiertes Flussnetzwerk.

Abbildung 32 zeigt das zu Abbildung 31 assoziierte Flussnetzwerk. Die Struktur des Flussnetzwerks ergibt sich aus dem Szenario, während das markierte S/T-Netz die Kapazitäten der Kanten bestimmt. Kanten, die nicht mit einer Kapazität beschriftet sind, tragen in dieser Abbildung die Kapazität $M_p(bpo, N) = 5$. Zu jedem Ereignis gehören zwei Knoten des Flussnetzwerks. Um diese Paare ist eine Kontur gezeichnet, die mit dem Namen des zugehörigen Ereignisses beschriftet ist. Der jeweils obere Knoten in der Kontur ist der oben-Knoten, der untere ist der unten-Knoten. Zusätzlich besitzt das Flussnetzwerk eine Quelle q, eine Senke s und einen der Anfangsmarkierung zugeordneten Knoten k_0. Die Kanten, die von der Quelle ausgehen, begrenzen den

Durchfluss der unten-Knoten, die das Produzieren von Marken in der Stelle beschreiben. Die Kanten, die in die Senke eingehen, begrenzen den Durchfluss der oben-Knoten, die das Konsumieren von Marken in der Stelle beschreiben. Die übrigen Kanten ermöglichen eine Verteilung von Marken, die die Abhängigkeiten des Szenarios respektiert. Diese Kanten sollen den Fluss nicht beschränken und sind deshalb mit dem maximal möglichen Markenfluss $M_p(bpo, N)$ in diesem Flussnetzwerk versehen. Dieser ergibt sich aus der Kapazität 5 des Schnittes $(K\backslash\{s\},\{s\})$.
Einen Fluss in dem assoziierten Flussnetzwerk kann man in einen Markenfluss im Szenario zu übersetzten. Wir nennen diesen Markenfluss den dem Fluss assoziierten Markenfluss.

DEFINITION 4.5.8 (ASSOZIIERTER MARKENFLUSS)
Sei $bpo = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $p \in P$ *eine Stelle und* g *ein Fluss im assoziierten Flussnetzwerk* $G = (K, F, c, q, s)$.
Wir definieren den assoziierten Markenfluss $x : (E \cup <) \to \mathbb{N}_0$ *als*

$$x(a) = \begin{cases} g(unten(e), oben(e')), & \text{falls } a = (e, e') \in <, \\ g(k_0, oben(e)), & \text{falls } a = e \in E. \end{cases}$$

Definition 4.5.8 (Assoziierter Markenfluss)

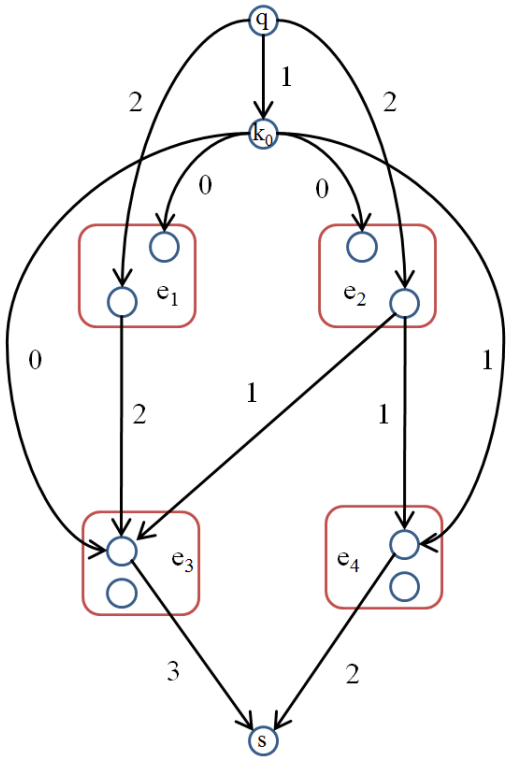

Abbildung 33: Ein maximaler Fluss in einem assoziierten Flussnetzwerk.

Abbildung 33 zeigt einen maximalen Fluss im assoziierten Flussnetzwerk aus Abbildung 32. Die Kanten sind in dieser Abbildung mit dem Fluss über die Kante und nicht mit der Kapazität beschriftet. Der abgebildete Fluss ist ein maximaler Fluss, da er den Schnitt $(K\backslash\{s\},\{s\})$ sättigt. Das bedeutet insbesondere, dass für jedes Ereignis e der Durchfluss an jedem oben-Knoten den Wert $W(p, l(e))$ besitzt. Der der Quelle

ausgehende Fluss ist durch die Gewichte $W(l(e), p)$ und durch $m_0(p)$ beschränkt. Der Fluss auf den restlichen Kanten beschreibt die Verteilung der Marken über die Ereignisse. Ereignis e_1 erzeugt zwei Marken für Ereignis e_3, Ereignis e_2 erzeugt eine Marke für Ereignis e_3 und eine Marke für Ereignis e_4. Eine weitere Marke erhält Ereignis e_4 aus der Anfangsmarkierung. Dieser sich ergebende assoziierte Markenfluss ist in Abbildung 34 dargestellt.

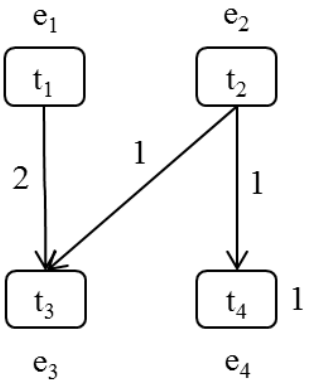

Abbildung 34: Ein gültiger Markenfluss.

In [66] ist gezeigt, dass für ein Szenario, ein markiertes S/T-Netz und für eine Stelle $p \in P$ genau dann ein gültiger Markenfluss x existiert, wenn ein maximaler Fluss im assoziierten Flussnetzwerk den Wert $M_p(bpo, N)$ besitzt . Dieses ist der entscheidende Satz dieses Abschnittes. Für seinen Beweis betrachtet man zunächst eine Richtung der Aussage in Form eines Lemmas.

Lemma 4.5.1 (über den assoziierten Fluss)

LEMMA 4.5.1 (ÜBER DEN ASSOZIIERTEN FLUSS)
Sei $bpo = (E, <, l)$ *ein Szenario,* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $p \in P$ *eine Stelle,* $G = (K, F, c, q, s)$ *das assoziierte Flussnetzwerk und* x *der assoziierte Markenfluss zu einem maximalen Fluss* g *in* G.
Ist der Wert des Flusses g *in* G *gleich* $M_p(bpo, N) = \sum_{e \in E} W(p, l(e))$, *so ist* x *gültig für* (N, m_0) *und* p.

BEWEIS: Sei x der zu g assoziierte Markenfluss. Nach Voraussetzung gilt $w(g) = \sum_{e \in E} W(p, l(e)) = \sum_{e \in E} c(oben(e), s)$. Damit gilt für jedes Ereignis $e \in E$ die Gleichung $g(oben(e), s) = W(p, l(e))$. Die folgenden Umformungen verwenden die Struktur des assoziierten Flussnetzwerks und die Eigenschaft der Flusserhaltung. Es folgt für jedes Ereignis e:

$$\begin{aligned} zu(e) &= x(e) + \sum_{e' < e} x(e', e) \\ &= g(k_0, oben(e)) + \sum_{e' < e} g(unten(e'), oben(e)) \\ &= g(oben(e), s) \\ &= W(p, l(e)). \end{aligned}$$

$$\begin{aligned} ab(e) &= \sum_{e<e'} x(e,e') \\ &= \sum_{e<e'} g(unten(e), oben(e')) \\ &= g(q, unten(e)) \\ &\leqslant c(q, unten(e)) \\ &= W(l(e), p). \end{aligned}$$

$$\begin{aligned} \sum_{e\in E} x(e) &= \sum_{e\in E} g(k_0, oben(e)) \\ &= g(q, k_0) \\ &\leqslant c(q, k_0) \\ &= m_0(p). \end{aligned}$$

□

Der Beweis der entgegengesetzten Aussage zu Lemma 4.5.1 funktioniert auf die gleiche Weise. Man gibt für einen Markenfluss einer Stelle p die Übersetzung in einen Fluss im assoziierten Flussnetzwerk an und zeigt, dass wenn der Markenfluss die Bedingungen (I), (II) und (III) erfüllt, der Fluss ein maximaler Fluss mit Wert $M_p(bpo, N)$ ist. Auch diese zweite Richtung des Beweises ist in [66] dargestellt. Wir halten für diese Arbeit den folgenden Satz fest.

Satz 4.5.2 (über das assoziierte Flussnetzwerk)

Satz 4.5.2 (Über das assoziierte Flussnetzwerk)
Ein Szenario bpo *ist genau dann in der Szenario-Sprache eines markierten S/T-Netzes* (N, m_0)*, wenn für jede Stelle* $p \in P$ *der Wert eines maximalen Flusses im assoziierten Flussnetzwerk* G_p *der Wert* $M_p(bpo, N)$ *ist.*

Satz 4.5.2 beschreibt, wie man das Szenario-Verifikations-Problem mit Hilfe des assoziierten Flussnetzwerks und der Berechnung eines maximalen Flusses in diesem entscheiden kann.

Algorithmus 4.5.2 (Szenario-Verifikation über Markenflüsse (direkt))

ALGORITHMUS 4.5.2 (SZENARIO-VERIFIKATION ÜBER MARKENFLÜSSE (DIREKT))

Eingabe: *Szenario oder Szenario-Spezifikation* $\mathtt{bpo} = (E, <, l)$ *und ein markiertes Petrinetz* $N = (P, T, W, m_0)$

Ausgabe: *Entscheidung ob* $\mathtt{bpo} \in L(N, m_0)$ *gilt*

```
1   Berechne eine topologische Ordnung des Szenarios
2   Berechne die transitive Ordnung des Szenarios
3   FOREACH p ∈ P
4   {
5       G ← Assoziiertes Flussnetzwerk(p)
6       w ← Wert maximaler Fluss(G)
7       IF (w < M_p(bpo, N)) RETURN false
8   }
9   RETURN true
```

Algorithmus 4.5.1 entscheidet das Szenario-Verifikations-Problem. In Zeile 1 wird das Szenario topologisch sortiert und in Zeile 2 wird die transitive Hülle des Szenarios berechnet, falls die Eingabe eine Szenario-Spezifikation ist. Die Zeilen 3-8 beschreiben das Vorgehen für jede Stelle des markierten S/T-Netzes. Zunächst wird das zu der aktuellen Stelle assoziierte Flussnetzwerk berechnet (Zeile 5) und der Wert eines maximalen Flusses in diesem bestimmt (Zeile 6). In Zeile 7 wird dieser Wert mit dem Wert $M_p(\mathtt{bpo}, N)$ verglichen. Ist der Wert des Flusses zu klein, bricht der Algorithmus in Zeile 7 ab. Erreicht der Algorithmus Zeile 9, ist das Szenario-Verifikations-Problem mit positiver Antwort entschieden.
Die Laufzeit von Algorithmus 4.5.2 setzt sich aus zwei Zeiten zusammen: Erstens der Zeit für die Konstruktionen der Flussnetzwerke und zweitens der Zeit zur Lösung der Fluss-Maximierungs-Probleme. Die Laufzeit der Konstruktion eines assoziierten Flussnetzwerks kann man gut durch die Anzahl der Elemente des Flussnetzwerkes beschreiben. Es besteht aus $2|E| + 3$ Knoten und aus ungefähr $|<| + 3|E|$ Kanten. Die Laufzeiten der Fluss-Maximierungs-Algorithmen haben wir ausführlich in Kapitel 3 diskutiert. Dabei hängt die Laufzeit stark von der Anzahl der Kanten des assoziierten Flussnetzwerks ab. Die Menge der Kanten, die die Verteilung der Marken beschreiben, ergibt sich immer aus der Ordnung des Szenarios. Die Kanten, die der Quelle ausgehen oder der Senke eingehen, ergeben sich aus den Kantengewichten des S/T-Netzes. Insgesamt erhalten wir eine Schlechtester-Fall-Komplexität in $O(|P| \cdot |E|^3)$, wenn wir den Preflow-Push Algorithmus verwenden.
Wir halten an dieser Stelle fest, dass der direkte Test des Szenario-Verifikations-Problems mit Hilfe von Markenflüssen in Polynomialzeit läuft. Die theoretischen Laufzeiten sind um den Faktor $|E|$ schneller als die theoretischen Laufzeiten des iterativen Tests. Die erste Implementierung dieses Algorithmus als Plugin für das VipTool war Teil dieser Arbeit. In Kapitel 6 werden wir die Laufzeiten aller Szenario-Verifikations-Algorithmen vergleichen und genauer diskutieren.

5

KOMPAKTE MARKENKENFLÜSSE UND PETRINETZE

Dieses Kapitel behandelt kompakte Markenflüsse in Szenarien. Existiert in einem Szenario zu jeder Stelle eines S/T-Netzes ein kompakter Markenfluss, so ist das Szenario in der Szenario-Sprache des S/T-Netzes enthalten. Auf diese Weise lässt sich durch Konstruktion kompakter Markenflüsse das Szenario-Verifikations-Problem entscheiden. Der Abschnitt 5.1 definiert den Begriff eines kompakten Markenflusses und beweist die Äquivalenz zwischen dieser neuen Charakterisierung der Szenario-Sprache und den Charakterisierungen der Szenario-Sprache über Schnitte, Prozessnetze oder Markenflüsse. In Abschnitt 5.2 entwickeln wir einen Algorithmus, der das Szenario-Verifikations-Problem über die Konstruktion kompakter Markenflüsse effizient entscheidet.

5.1 KOMPAKTE MARKENFLÜSSE

In Kapitel 4 haben wir drei grundsätzlich verschiedene, äquivalente Charakterisierungen der Szenario-Sprache eines markierten S/T-Netzes betrachtet. Aus jeder lässt sich ein Algorithmus ableiten, der das Szenario-Verifikations-Problem entscheidet. Die Laufzeitkomplexität des aktivierte Schnitte Algorithmus und die Laufzeitkomplexität des Prozessnetz Algorithmus sind in Exponentialzeit. Die Laufzeitkomplexitäten der beiden Markenfluss Algorithmen sind stark von der Menge der Kanten des Szenarios abhängig. Die Algorithmen laufen schnell für lichte Szenarien.
Der Begriff eines gültigen Markenflusses und der iterative Test des Szenario-Verifikations-Problems wurden ursprünglich entwickelt, um den Zusammenhang zwischen aktivierten und ausführbaren Szenarien elegant zu beweisen. Die Intention der Definition eines Markenflusses war es nicht, einen besonders effizienten Verifikations-Algorithmus entwickeln zu können, sondern mit Hilfe der Markenflüsse den in [64, 87] geführten und recht komplexen Beweis eleganter zu gestalten und zu vereinfachen [60]. Mit dem Beweis erhielt man zusätzlich einen ersten in Polynomialzeit laufenden Test, den iterativen Test aus Abschnitt 4.5.1, zur Entscheidung des Szenario-Verifikations-Problems. In [11] wurde ein entsprechender Algorithmus als Plugin für das VipTool implementiert, und es stellte sich heraus, dass dieser Algorithmus sehr gut auf Szenarien anzuwenden ist, falls diese stark nebenläufiges Verhalten beschreiben. Enthält ein Szenario viele Abhängigkeiten, ist die Laufzeit allerdings unzureichend. Aus diesem Grund wurde in [66] der direkte Test formal beschrieben. Dies war ein erster Schritt in Richtung eines Verfahrens mit effizienter Laufzeit.
Nach der Beschreibung des direkten Tests verschob sich die Aufmerksamkeit weg vom Szenario-Verifikations-Problem und hin zu ande-

ren Problemen, die mit Hilfe von Markenflüssen betrachtet werden können. Die Markenflüsse wurden zur Synthese von S/T-Netzen aus Szenarien [17, 19, 69, 15], zur Synthese aus unendlichen Mengen von Szenarien [24, 20, 86], zu der Definition des Ausführbarkeitsbegriffes eines Szenarios in allgemeineren Klasse von Petrinetzen [68] sowie für das Berechnen der gesamten halbgeordneten Sprache von S/T-Netzes eingesetzt [23, 21]. Nach der ersten Implementierung des iterativen Algorithmus ist die effiziente Lösung des Szenario-Verifikations-Problems, das grundlegende Problem im Zusammenhang von S/T-Netzen und Szenarien, in den Hintergrund getreten. Die Idee dieser Arbeit ist es, eine auf der Definition der Markenflüsse basierende und auf das Szenario-Verifikations-Problem zugeschnittene Definition der Szenario-Sprache eines S/T-Netzes zu erarbeiten, die eine in jedem Fall effiziente Entscheidung des Szenario-Verifikations-Problems erlaubt.
Ein Markenfluss besitzt große Ähnlichkeit mit einem Prozessnetz. Betrachtet man eine einzige Stelle eines Petrinetzes und einen zugehörigen gültigen Markenfluss, so beschreibt der Markenfluss die genaue Verteilung der Bedingungen zwischen den Ereignissen. Ein Prozessnetz enthält die gleiche Information, nur dass Bedingungen zwischen Ereignissen zusätzlich mit Identitäten versehen sind. Abstrahiert man von diesen und geht man davon aus, dass man einen Markenfluss für jede Stelle des Netzes kennt, so lässt sich diese Menge von Markenflüssen eins zu eins in einen Prozess übersetzen. Die Vielzahl der möglichen und unmöglichen Verteilungen von Bedingungen (oder Marken) auf Paare von geordneten Ereignissen trägt zur hohen Laufzeit der entsprechenden Verfahren bei.
Die Komplexität des Problems, alle möglichen Verteilungen von Marken auf geordneten Paaren von Ereignissen zu untersuchen, zeigt sich im Vergleich einer Szenario-Spezifikation mit dem zugehörigen Szenario. Die transitive Hülle wächst meist quadratisch mit der Anzahl der Ereignisse eines Szenarios, und ein Markenfluss beschreibt einen Wert für jedes Paar transitiv geordneter Ereignisse. Glücklicherweise helfen die Fluss-Maximierungs-Algorithmen dabei, eine gültige Verteilung zu berechnen, ohne alle möglichen Verteilungen aufzuzählen. Auf diese Weise bleibt die Schlechtester-Fall-Komplexität in Polynomialzeit.
Um uns einer effizienteren Charakterisierung der Szenario-Sprache zu nähern, überlegen wir, warum die durchschnittliche Laufzeit des aktivierte Schnitte Algorithmus in vielen Beispielen gut ist. Warum schneidet dieser in Exponetialzeit laufende Algorithmus im Vergleich zu dem Markenfluss Algorithmus oft so gut ab? Ein Grund ist sicherlich, dass die Anzahl der Schnitte eines Szenarios nur in wenigen Fällen wirklich exponentiell mit der Anzahl der Ereignisse wächst. Die Anzahl der Schnitte hängt stark von der Breite eines Szenarios ab. Die Breite eines Szenarios ist aber beschränkt, falls es Abläufe eines Systems mit begrenzten Ressourcen beschreibt. Ressourcen können dabei Rechner, Prozessor-Kerne, Mitarbeiter usw. sein. In Szenarien mit beschränkter Breite wächst die Anzahl der Schnitte des Szenarios nicht exponentiell und zum Teil schwächer als die Anzahl der transitiven Kanten. Um so mehr Abhängigkeiten ein Szenario besitzt, um so kleiner ist die Anzahl der Schnitte. Im Gegensatz dazu wird der Markenfluss Algorithmus mit der Anzahl der vorhandenen Abhängigkeiten komplexer. Ein weiterer Grund dafür, dass der aktivierte Schnitte Algorithmus in vielen Fällen eine gute Laufzeit aufweist, ist, dass er von einer konkreten Verteilung der Marken abstrahiert. Produziert jedes Präfix genügend Marken

für den nachfolgenden Schnitt, spielt es keine Rolle, welches Ereignis welche dieser Marken produziert.
Zusammengefasst wollen wir eine Charakterisierung der Szenario-Sprache entwickeln, die

- auf den Kanten des Szenarios definiert ist (die Anzahl der Kanten wächst nicht exponentiell in der Größe der Eingabe),
- von Identitäten und der konkreten Verteilung einzelner Marken oder Bedingungen abstrahiert,
- nicht nur durch Aufzählen aller gültigen Szenarien gefunden, sondern direkt konstruiert werden kann.

Zu diesem Zweck definieren wir in dieser Arbeit den Begriff eines kompakten Markenflusses. Ein kompakter Markenfluss ist eine Abbildung aus der Menge des Skelettes aller Abhängigkeiten eines Szenarios und dessen Ereignissen in die nicht negativen ganzen Zahlen. Ein kompakter Markenfluss ist damit eine Abbildung auf der Menge der kleinsten Repräsentation aller Abhängigkeiten.
Analog zu den Begriffen eines Szenarios und einer Szenario-Spezifikation definieren wir zunächst den Begriff eines kompakten Szenarios.

Definition 5.1.1 (Kompaktes Szenario)

DEFINITION 5.1.1 (KOMPAKTES SZENARIO)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario und* $\lhd$ *das Skelett der Ordnung* $<$*. Die Szenario-Spezifikation* $(E, \lhd, l)$ *heißt kompaktes Szenario von* bpo.
Jedes kompakte Szenario beschreibt durch $(E, \lhd^*, l)$ *eindeutig das zu dem kompakten Szenario gehörende Szenario.*

Für ein Szenario definieren wir einen kompakten Markenfluss auf dem Skelett der Abhängigkeiten und damit auf dem kompakten Szenario.

Definition 5.1.2 (Kompakter Markenfluss)

DEFINITION 5.1.2 (KOMPAKTER MARKENFLUSS)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario und* $\lhd$ *das Skelett der Ordnung* $<$*. Eine Funktion* $x : (\lhd \cup E) \to \mathbb{N}_0$ *ist ein kompakter Markenfluss. Für ein Ereignis* $e \in E$ *und einen kompakten Markenfluss ist*

$$zu^{\lhd}(e) = x(e) + \sum_{e' \lhd e} x(e', e) \text{ der Markenzufluss von } e \text{ und}$$

$$ab^{\lhd}(e) = \sum_{e \lhd e'} x(e, e') \text{ der Markenabfluss von } e.$$

Für eine Menge von Ereignissen $V \subseteq E$ *sei*

$$V^{\ll} = \lhd \cap ((E \setminus V) \times V) \text{ die Menge der Skelettkanten, die zu } V \text{ führen,}$$

$$V^{\gg} = \lhd \cap (V \times (E \setminus V)) \text{ die Menge der Skelettkanten, die aus } V \text{ führen.}$$

Damit ist

$$ZU(V) = \sum_{(e,e') \in V^{\ll}} x(e, e') + \sum_{e \in V} x(e) \quad \text{der Markenzufluss zu } V,$$

$$AB(V) = \sum_{(e,e') \in V^{\gg}} x(e, e') + \sum_{e \in E \setminus V} x(e) \quad \text{der Markenabfluss aus } V.$$

Abbildung 35 zeigt den Unterschied zwischen der Definitionsmenge eines Markenflusses und der eines kompakten Markenflusses. Im linken Teil ist eine Szenario-Spezifikation abgebildet. Die „später als“-Relation beschreibt dabei alle Abhängigkeiten, die in dem Szenario zwischen den Ereignissen bekannt sind, oder die Teilmenge der Abhängigkeiten, die explizit modelliert und beschrieben werden soll. Diese Form ist eine sehr intuitive Art der Beschreibung halbgeordneten Verhaltens. Im mittleren Teil der Abbildung ist das zugehörige Szenario abgebildet. Dieses erweitert eine Szenario-Spezifikation um die Menge der transitiven Abhängigkeiten. Selbst in diesem kleinen Beispiel verdreifacht sich die Anzahl der Abhängigkeiten. Auf dieser Menge der Abhängigkeiten eines Szenarios sind Markenflüsse definiert. Im rechten Teil der Abbildung 35 ist das zugehörige kompakte Szenario dargestellt. Es enthält das Skelett der Spezifikation. Ein kompaktes Szenario beschreibt die Abhängigkeiten, auf denen ein kompakter Markenfluss definiert ist. Diese bilden die kleinste Repräsentation aller Abhängigkeiten.

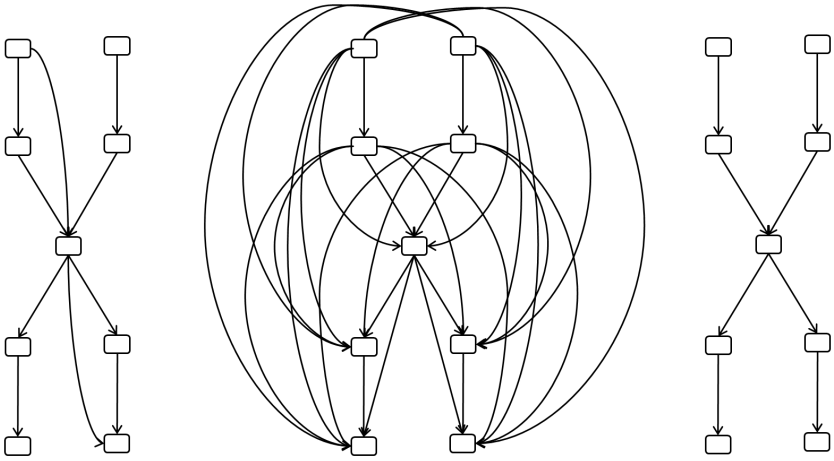

Abbildung 35: Links: Eine Szenario-Spezifikation. Mitte: Ein Szenario. Rechts: Ein kompaktes Szenario.

Ein kompakter Markenfluss beschreibt, genau wie ein Markenfluss oder ein Prozessnetz, wie Marken in einer Stelle eines S/T-Netzes während der Ausführung eines Szenarios durch Ereignisse produziert und konsumiert werden. Bei einem kompakten Markenfluss wird jedoch von der konkreten Verteilung der Marken abstrahiert und die Verteilung wird auf einer Teilmenge aller Abhängigkeiten betrachtet. Die Teilmenge ist das Skelett der Ordnung, was bedeutet, dass sich jede Abhängigkeit zwischen Ereignissen im Szenario durch einen Weg aus Abhängigkeiten des kompakten Szenarios darstellen lässt. Als Konsequenz müssen wir dafür sorgen, dass bei einem kompakten Markenfluss Marken nicht nur direkt verteilt werden, sondern dass Ereignisse Marken erhalten, die sie an spätere Ereignisse weiterleiten können, anstatt diese selber zu konsumieren.
Abbildung 36 zeigt den Unterschied zwischen einem kompakten Szenario und einem Prozess. Der linke Teil der Abbildung zeigt ein markiertes S/T-Netz. Es besteht aus drei Transitionen t_1, t_2 und t_3 und einer Stelle. In der Mitte der Abbildung ist ein kompaktes Szenario abgebildet. Der kompakte Markenfluss beschreibt die Ausführung des Szenarios bezüglich der Stelle in der folgenden Weise: Das mit t_1 beschriftete

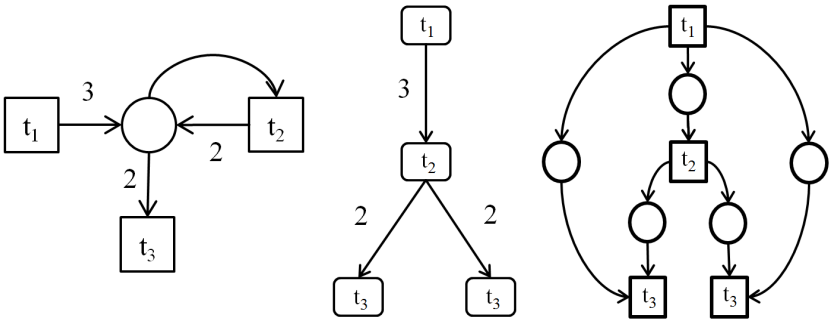

Abbildung 36: Links: Ein markiertes S/T-Netz. Mitte: Ein kompaktes Szenario. Rechts: Ein Prozess.

Ereignis produziert drei Marken, die an das mit t_2 beschriftete Ereignis weitergegeben werden. Damit es eintreten kann, benötigt das mit t_2 beschriftete Ereignis eine Marke und produziert durch sein Eintreten zwei. Da es drei Marken übergeben bekommt, kann es eine davon konsumieren und anschließend die zwei übrigen und die zwei neu produzierten weiterleiten. Zwei dieser vier Marken werden an das dritte Ereignis und die restlichen zwei an das vierte Ereignis weitergeben. Mit diesen Marken können nun die beiden mit t_3 beschrifteten Ereignisse eintreten. Für diese Ereignisse ist es egal, welche Marken von der Transition t_1 und welche Marken von der Transition t_2 produziert wurden. In Analogie zu Definition 4.3.1 beschreibt der kompakte Markenfluss die Anzahl der durch ein Präfix produzierten Marken. Der Unterschied zu Prozessen wird deutlich, wenn wir den rechten Teil der Abbildung betrachten. In dem Prozess ist die vollständige Verteilung der Marken zu erkennen.
Analog zu den in Abschnitt 4.5 beschriebenen Eigenschaften (I), (II) und (III) definieren wir drei neue Eigenschaften (i), (ii) und (iii) für einen kompakten Markenfluss. Erfüllt ein kompakter Markenfluss diese Eigenschaften, so beschreibt er eine gültige Ausführung eines Szenarios in einem markierten S/T-Netz. Eigenschaft (i) stellt sicher, dass jedes Ereignis genügend Marken erhält. Eigenschaft (ii) stellt sicher, dass jedes Ereignis maximal die Anzahl an Marken weiter gibt, die es nach seinem Eintreten übrig hat, plus die Anzahl an Marken, die es in der Stelle produzieren kann. Eigenschaft (iii) ist identisch zu Eigenschaft (III), die Summe der Marken, die aus der Anfangsmarkierung konsumiert werden, darf diese nicht überschreiten.

Definition 5.1.3 (Gültiger kompakter Markenfluss)

Definition 5.1.3 (Gültiger kompakter Markenfluss)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $\mathrm{bpo} = (E, <, l)$ *eine Szenario mit* $l(E) \subseteq T$, *und sei* $\lhd$ *das Skelett der Ordnung* $<$.
Für eine Stelle $p \in P$ *ist ein kompakter Markenfluss* x *gültig für* (N, m_0), *wenn er die folgenden Bedingungen erfüllt:*

(i) *Für alle* $e \in E$ *gilt* $zu^{\lhd}(e) \geqslant W(p, l(e))$,

(ii) *für alle* $e \in E$ *gilt* $ab^{\lhd}(e) \leqslant zu^{\lhd}(e) - W(p, l(e)) + W(l(e), p)$,

(iii) *es gilt* $\sum_{e \in E} x(e) \leqslant m_0(p)$.

Wir beweisen, dass sich aus dieser Definition eines gültigen kompakten Markenflusses eine Charakterisierung der Szenario-Sprache eines S/T-Netzes ergibt. Diesen Beweis können wir allerdings nicht analog zu dem entsprechenden Beweis für gültige (einfache) Markenflüsse führen, der auf dem starken Zusammenhang zwischen Markenflüssen und Prozessen beruht. Durch die komprimierte Darstellung der Verteilung der Marken bei einem kompakten Markenfluss geht dieser Zusammenhang verloren. Die Äquivalenz der neuen Charakterisierung zu den bereits existierenden zeigen wir in zwei Schritten. Zunächst geben wir eine Konstruktion an, mit der man aus einem gültigen einfachen Markenfluss einen gültigen kompakten Markenfluss konstruieren kann. Danach zeigen wir, dass die Existenz eines gültigen kompakten Markenflusses für jede Stelle eines S/T-Netzes gleichbedeutend mit der Aktivierheit aller Schnitte des Szenarios ist.
Ausgehend von einem gültigen einfachen Markenfluss konstruieren wir für ein gegebenes markiertes S/T-Netz einen kompakten Markenfluss. Die grundlegende Idee dabei ist, den gesamten Markenfluss, welcher auf allen Kanten des Szenarios verteilt ist, auf das Skelett der Ordnung des Szenarios zu verschieben. Anstatt direkt über eine transitive Kante zu einem Ereignis zu fließen, wird der Markenfluss von Ereignis zu Ereignis weitergegeben, bis er auf diese Weise sein Ziel erreicht. Für unsere Konstruktion ordnen wir also jeder Kante (e, e') des Szenarios einen Weg aus Skelettkanten von Ereignis e nach Ereignis e' zu. Dies können wir tun, da nach der Definition eines Skelettes mindestens ein solcher Weg für jede Kante existiert. Wir addieren den Wert des Markenflusses auf jeder Kante (e, e') auf den Wert jeder Skelettkante des zugeordneten Weges, während wir die Werte des Markenflusses auf Ereignissen beibehalten.

Definition 5.1.4 (Passender kompakter Markenfluss)

Definition 5.1.4 (Passender kompakter Markenfluss)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario,* $\lhd$ *das Skelett der Ordnung* $<$ *und* x *ein Markenfluss in* bpo. *Zu jeder Kante* (v, v') *existiert ein Weg aus Skelettkanten von* v *nach* v'. *Wir konstruieren eine Abbildung* weg *aus der Menge der Kanten in die Menge der Wege aus Skelettkanten* $\mathrm{weg} : <\to 2^{\lhd}$, *indem wir jeder Kante* $(v, v') \in <$ *einen Weg* ω *von* v *nach* v' *zuordnen.*
Für jede Skelettkante $(e, e') \in \lhd$ *definieren wir die Menge*

$$(e, e')^* = \{(v, v') \in < \mid (e, e') \in \mathrm{weg}(v, v')\}.$$

Wir definieren einen zu dem Markenfluss x *passenden kompakten Markenfluss* $x^{\lhd} : (\lhd \cup E) \to \mathbb{N}_0$ *durch:*

$$x^{\lhd}(a) = \begin{cases} \displaystyle\sum_{(v,v') \in (e,e')^*} x(v, v'), & \text{falls } a = (e, e') \in \lhd, \\ x(e), & \text{falls } a = e \in E. \end{cases}$$

Abbildung 37 zeigt im linken Teil ein Szenario mit einem Markenfluss. Der mittlere Teil der Abbildung deutet eine mögliche Abbildung *weg* aus Definition 5.1.4 an. Jede Kante wird auf einen Weg aus Skelettkanten abgebildet. In der Abbildung schmiegt sich die transitive Kante dafür an den entsprechenden Weg an, so wird zum Beispiel die Kante (e_1, e_5) dem Weg $(e_1, e_2)(e_2, e_4)(e_4, e_5)$ zugeordnet. Der Wert jeder Skelettkante des kompakten Markenflusses ergibt sich als die Summe der Werte aller Kanten, die unter der Abbildung *weg* diese

Skelettkante enthalten. Der sich ergebende kompakte Markenfluss ist im rechten Teil der Abbildung 37 dargestellt. Zum Beispiel ergibt sich der Wert des kompakten Markenflusses auf der Kante (e_2, e_4) aus der Summe der Werte des einfachen Markenflusses auf den Kanten $(e_1, e_4), (e_2, e_4), (e_2, e_5)$ und (e_1, e_5).

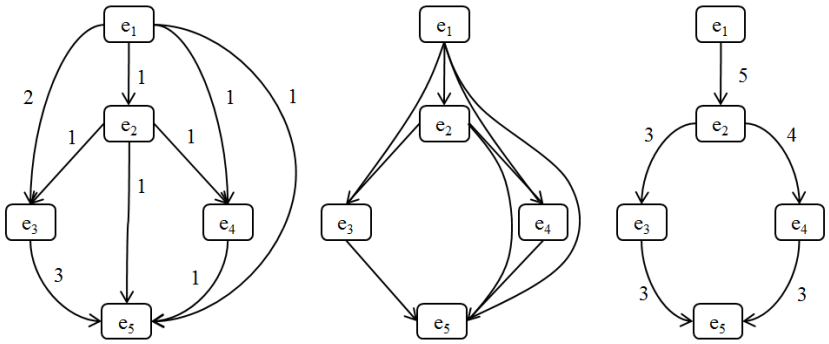

Abbildung 37: Links: Ein Szenario mit Markenfluss. Mitte: Die Abbildung *weg*. Rechts: Ein passender kompakter Markenfluss.

Abbildung 37 stellt die Werte der beiden Markenflüsse auf den Ereignissen nicht dar. Diese Werte im kompakten Markenfluss sind zu den entsprechenden Werten des einfachen Markenflusses identisch.
In Definition 5.1.4 ist der konstruierte kompakte Markenfluss nicht eindeutig. Jede Wahl der Abbildung *weg* kann zu einem anderen kompakten Markenfluss führen. Trotzdem können wir zeigen, dass jeder zu einem gültigen Markenfluss passende kompakte Markenfluss gültig ist.

SATZ 5.1.1 (ÜBER EINEN PASSENDEN KOMPAKTEN MARKENFLUSS)
Sei x ein gültiger Markenfluss. Jeder zu x passende kompakte Markenfluss ist gültig.

Satz 5.1.1 (über einen passenden kompakten Markenfluss)

BEWEIS: Sei (N, m_0) mit $N = (P, T, W)$ ein markiertes S/T-Netz, $bpo = (E, <, l)$ ein Szenario mit $l(E) \subseteq T$ und $\lhd$ das Skelett der Ordnung $<$. Sei x ein zu einer Stelle $p \in P$ und zu (N, m_0) gültiger Markenfluss. Sei $x^\lhd$ ein zu x passender kompakter Markenfluss.
Wir betrachten ein Ereignis e und das Bild der Abbildung *weg* bezüglich e. Es existieren vier Arten von Wegen in $weg(<)$: Wege, die bei e enden, Wege, die bei e beginnen, Wege, die über e führen, und Wege, in denen e weder als Start- noch als Endknoten einer Kante enthalten ist.
Der Zufluss $zu^\lhd(e)$ eines Ereignisses e im kompakten Markenfluss $x^\lhd$ ist die Summe von drei Werten: Dem Wert von x auf dem Ereignis e, der Summe der Werte der bei e endenden Wege und der Summe der über e führenden Wege.
Die Wege, die bei e enden, entstehen nach Konstruktion aus allen Kanten $\{(e', e) | e' \in E, e' < e\}$. Genau diese Menge wird auf Wege aus Skelettkanten abgebildet, deren letzte Kante zu e führt. Die Summe der Werte dieser Kanten $\sum_{e'<e} x(e', e)$ ergibt zusammen mit dem Wert $x(e)$ den Wert $zu(e)$, den Markenzufluss von e im einfachen Markenfluss x. Sei $\delta(e)$ die Summe der Werte auf Kanten, die auf Wege abgebildet werden, die über e führen. Der Markenzufluss von e in $x^\lhd$ ist demnach

$$zu^\lhd(e) = zu(e) + \delta(e).$$

Der Abfluss $ab^{\triangleleft}(e)$ eines Ereignisses e unter dem kompakten Markenfluss $x^{\triangleleft}$ ist die Summe von zwei Werten: Der Summe der Werte der bei e beginnenden Wege und der Summe der über e führenden Wege.
Die Wege, die bei e beginnen, entstehen nach Konstruktion aus allen Kanten $\{(e, e') | e' \in E, e < e'\}$. Genau diese Menge wird auf Wege aus Skelettkanten abgebildet, deren erste Kante bei e beginnt. Die Summe der Werte dieser Kanten $\sum_{e<e'} x(e, e')$ ist gleich dem Wert $ab(e)$, dem Markenabfluss von e unter dem einfachen Markenfluss x. Zusammen mit dem Wert $\delta(e)$, der Summe der Werte auf Kanten, die auf Wege abgebildet werden, die über e führen, ergibt sich der Markenabfluss von e in $x^{\triangleleft}$ als

$$ab^{\triangleleft}(e) = ab(e) + \delta(e).$$

Damit ergeben sich die Bedingungen (i), (ii) und (iii) wie folgt:

$$\begin{aligned} zu^{\triangleleft}(e) &= zu(e) + \delta(e) \\ &\geqslant zu(e) \\ &\overset{(I)}{=} W(p, l(e)), \end{aligned}$$

$$\begin{aligned} ab^{\triangleleft}(e) &= ab(e) + \delta(e) \\ &\overset{(I)}{=} ab(e) + \delta(e) + zu(e) - W(p, l(e)) \\ &= ab(e) + zu^{\triangleleft}(e) - W(p, l(e)) \\ &\overset{(II)}{\leqslant} W(l(e), p) + zu^{\triangleleft}(e) - W(p, l(e)), \end{aligned}$$

$$\begin{aligned} \sum_{e \in E} x^{\triangleleft}(e) &= \sum_{e \in E} x(e) \\ &\overset{(III)}{\leqslant} m_0(p). \end{aligned}$$

□

Die Gegenrichtung der Aussage von Satz 5.1.1 lässt sich nicht konstruktiv oder analog zum Beweis von Satz 4.5.1 führen. Dazu müsste ein gegebener gültiger kompakter Markenfluss durch „Auseinanderfalten“ zu einem einfachen gültigen Markenfluss gemacht werden. Solch eine Konstruktion ist allerdings schwer anzugeben, da es nicht klar ist, wie der Wert der Skelettkanten auf die transitiven Kanten verteilt werden muss. Um zu zeigen, dass eine Charakterisierung der Szenario-Sprache über gültige kompakte Markenflüsse möglich ist, beweisen wir deshalb den Zusammenhang zwischen einem gültigen kompakten Markenfluss und der Definition eines aktivierten Szenarios (Definition 4.1.4). Damit dies gelingt, beweisen wir zunächst zwei Lemmata. Das erste Lemma zeigt, auf welche Weise die Summe der Werte eines für eine Stelle

p gültigen kompakten Markenflusses, der ein Präfix an Ereignissen verlässt, mit der Anzahl an Marken zusammenhängt, die das gleiche Präfix in der Stelle p produziert.

LEMMA 5.1.1 (ÜBER DEN MARKENABFLUSS AUS EINEM PRÄFIX EINES GÜLTIGEN KOMPAKTEN MARKENFLUSSES)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $bpo = (E, <, l)$ *ein Szenario mit* $l(E) \subseteq T$, $\lhd$ *das Skelett der Ordnung* $<$ *und* $p \in P$ *eine Stelle. Sei* $x^{\lhd}$ *ein zu* (N, m_0) *und* p *gültiger kompakter Markenfluss in* bpo *und sei* $V \subseteq E$ *ein Präfix des Szenarios. Die Anzahl der in* p *durch das Präfix* V *produzierten Marken ist*

Lemma 5.1.1 (über den Markenabfluss aus einem Präfix eines gültigen kompakten Markenflusses)

$$prod(V) = m_0(p) + \sum_{e \in V} (W(l(e), p) - W(p, l(e))).$$

Für den kompakten Markenfluss $x^{\lhd}(e)$ *gilt:*

$$AB(V) \leqslant prod(V).$$

BEWEIS: Wir führen diesen Beweis durch vollständige Induktion. Für das minimale und leere Präfix $V = \emptyset$ gilt:

$$\begin{aligned} AB(\emptyset) &= \sum_{e \in E} x(e) \\ &\overset{(iii)}{\leqslant} m_0(p) \\ &= prod(\emptyset) \end{aligned}$$

und damit der Induktionsanfang.
Sein V' ein Präfix von bpo, für das $AB(V') \leqslant prod(V')$ gilt, und sei $e \in E \backslash V'$ ein in $E \backslash V'$ minimales Ereignis. Für das Präfix $V = V' \cup e$ gilt:

$$\begin{aligned} AB(V) &= AB(V') - zu^{\lhd}(e) + ab^{\lhd}(e) \\ &\overset{(ii)}{\leqslant} AB(V') - zu^{\lhd}(e) + zu^{\lhd}(e) - W(p, l(e)) + W(l(e), p) \\ &\leqslant prod(V') - W(p, l(e)) + W(l(e), p) \\ &= prod(V) \end{aligned}$$

und damit ist der Induktionsschritt gezeigt.
Durch das Hinzufügen von Ereignissen zum leeren Präfix lässt sich jedes Präfix des Szenarios erzeugen. Insgesamt folgt für jedes Präfix V damit $AB(V) \leqslant prod(V)$. □

Das zweite Lemma zeigt den Zusammenhang zwischen dem Markenzufluss zu einem Schnitt des Szenarios und dem Markenabfluss aus dem vor dem Schnitt liegenden Präfix.

Lemma 5.1.2 (über den Markenfluss zu Ereignissen eines Schnittes)

LEMMA 5.1.2 (ÜBER DEN MARKENFLUSS ZU EREIGNISSEN EINES SCHNITTES)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario,* $\triangleleft$ *das Skelett der Ordnung* $<$ *und* $x^{\triangleleft}$ *ein kompakter Markenfluss. Sei* $C \subseteq E$ *ein Schnitt aus Ereignissen im Szenario und sei* $^{\bullet}C = \{e \in E | e < C\}$. *Es gilt:*

$$ZU(C) \leqslant AB(^{\bullet}C).$$

BEWEIS: Jede Kante im Skelett des Szenarios, die an einem Ereignis im Schnitt C endet, geht von einem Knoten in $^{\bullet}C$ aus. Da C ein Schnitt ist, führt jede Kante, die ein Ereignis in $^{\bullet}C$ verlässt, nach C. Demnach sind die Werte $ZU(C)$ und $AB(^{\bullet}C)$ gleich groß, die Ungleichung ergibt sich aus den Werten des kompakten Markenflusses auf den Knoten, die hinter C geordnet sind.

$$\begin{aligned} ZU(C) &= \sum_{(e,e') \in C^{\ll}} x^{\triangleleft}(e,e') + \sum_{e \in C} x^{\triangleleft}(e) \\ &= \sum_{(e,e') \in {}^{\bullet}C^{\gg}} x^{\triangleleft}(e,e') + \sum_{e \in C} x^{\triangleleft}(e) \\ &\leqslant \sum_{(e,e') \in {}^{\bullet}C^{\gg}} x^{\triangleleft}(e,e') + \sum_{e \in \{E \setminus {}^{\bullet}C\}} x^{\triangleleft}(e) \\ &= AB(^{\bullet}C). \end{aligned}$$

□

Mit Hilfe von Lemma 5.1.1 und Lemma 5.1.2 können wir die Umkehrung von Satz 5.1.1 beweisen.

Satz 5.1.2 (über die Aktiviertheit gültiger kompakter Markenflüsse)

SATZ 5.1.2 (ÜBER DIE AKTIVIERTHEIT GÜLTIGER KOMPAKTER MARKENFLÜSSE)
Ein Szenario, für das für jede Stelle eines S/T-Netzes ein gültiger kompakter Markenfluss existiert, ist aktiviert.

BEWEIS: Sei (N, m_0) mit $N = (P, T, W)$ ein markiertes S/T-Netz und $\mathrm{bpo} = (E, <, l)$ ein Szenario mit $l(E) \subseteq T$. Für jeden Schnitt C von bpo und für jede Stelle $p \in P$ gilt:

$$\begin{aligned} \sum_{e \in C} W(p, l(e)) &\overset{(i)}{\leqslant} \sum_{e \in C} zu^{\triangleleft}(e) \\ &\overset{\text{(Lemma 5.1.2)}}{\leqslant} AB(^{\bullet}C) \\ &\overset{\text{(Lemma 5.1.1)}}{\leqslant} \mathrm{prod}(^{\bullet}C) \\ &= m_0(p) + \sum_{e \in {}^{\bullet}C} (W(l(e), p) - W(p, l(e))) \end{aligned}$$

und damit folgt nach Definition 4.3.1, dass bpo aktiviert ist. □

Wir fassen also zusammen, dass sich durch Definition 5.1.3 eine neue Charakterisierung der Szenario-Sprache ergibt. Diese Charakterisierung abstrahiert von den Identitäten der einzelnen Bedingungen oder Marken und zusätzlich von deren konkreter Verteilung. Mit Hilfe dieser neuen Charakterisierung ist eine effizientere Lösung des Szenario-Verifikation-Problems möglich. Doch bevor wir diese betrachten, halten wir den entscheidenden Satz dieses Abschnittes fest, der sich direkt aus Satz 5.1.1 und Satz 5.1.2 ergibt.

Satz 5.1.3 (über gültige kompakte Markenflüsse)

SATZ 5.1.3 (ÜBER GÜLTIGE KOMPAKTE MARKENFLÜSSE)
Die Menge aller Szenarien, für die zu einem markierten S/T-Netz (N, m_0) *zu jeder Stelle von* N *ein gültiger kompakter Markenfluss existiert, ist die Szenario-Sprache* $L(N, m_0)$ *von* (N, m_0).

Im nächsten Abschnitt beschreiben wir einen Algorithmus, der das Szenario-Verifikations-Problem über die Konstruktion gültiger kompakter Markenflüsse entscheidet.

5.2 DER KOMPAKTE MARKENFLUSS ALGORITHMUS

In diesem Abschnitt entwickeln wir einen Algorithmus, der das Szenario-Verifikations-Problem über die Definition der kompakten Markenflüsse entscheidet. Der Algorithmus arbeitet ähnlich wie der in Abschnitt 4.5 beschriebene direkte Test, der die einfache Version der Markenflüsse verwendet. Wir werden zu einem Szenario und für jede Stelle des S/T-Netzes ein Flussnetzwerk konstruieren, so dass sich ein maximaler Fluss direkt in einen kompakten Markenfluss übersetzen lässt. Erreicht der maximale Fluss einen bestimmten Wert, erfüllt der kompakte Markenfluss die Eigenschaften (i), (ii) und (iii). Für diesen Abschnitt sei $bpo = (E, <, l)$ ein Szenario, (N, m_0) mit $N = (P, T, W)$ ein markiertes S/T-Netz und $p \in P$ eine Stelle.
Um ein Flussnetzwerk zu einem kompakten Szenario und für eine Stelle p zu konstruieren, erzeugen wir für jedes Ereignis des Szenarios zwei Knoten und verwenden dabei die gleiche Notation wie in Abschnitt 4.5. Der erste Knoten wird dem Ereignis durch eine Abbildung $oben$, der zweite Knoten wird dem Ereignis durch eine Abbildung $unten$ zugeordnet. Für ein Ereignis e beschreibt der Fluss durch den Knoten $oben(e)$ die Anzahl an Marken, die das Ereignis erhält. Der Fluss durch den Knoten $unten(e)$ beschreibt die Anzahl an Marken, die das Ereignis an spätere Ereignisse weitergibt. Weiter konstruieren wir einen Knoten k_0, so dass der Fluss durch diesen Knoten den Markenfluss aus der Anfangsmarkierung beschreibt, eine Quelle q und eine Senke s.
Sind zwei Ereignisse e und e' im Szenario durch eine Skelettkante geordnet, so konstruieren wir im Flussnetzwerk eine Kante $(unten(e), oben(e'))$. Der Wert des Flusses auf dieser Kante beschreibt die Anzahl der Marken, die von Ereignis e an Ereignis e' weitergegeben werden. Bei einem kompakten Markenfluss kann ein Ereignis Marken erhalten und an spätere Ereignisse weiterleiten. Das Flussnetzwerk enthält dazu für jeden Knoten e eine Kante $(oben(e), unten(e))$. Um die Marken aus der Anfangsmarkierung von p zu verteilen, existiert für jeden Knoten e eine Kante $(k_0, oben(e))$. Wir nennen diese drei Arten von Kanten die inneren Kanten. Sie beschreiben die Verteilung der Marken durch das Szenario. Die Kapazitäten dieser inneren Kanten dürfen den Fluss nicht beschränken, so dass jedes Ereignis alle Marken weitergeben kann, die es zur Verfügung hat.

Wir begrenzen die Verteilung von Marken so, dass sie die Eigenschaften (i), (ii) und (iii) erfüllt. Kein Ereignis e darf zu viele Marken produzieren. Zwar kann ein Ereignis Marken an seinem Knoten $\mathtt{oben}(e)$ erhalten und diese zu seinem Knoten $\mathtt{unten}(e)$ weiterleiten, aber wir beschränken die Menge an Fluss, die zusätzlich an dem Knoten $\mathtt{unten}(e)$ ins Flussnetzwerk gelangt, indem wir für jedes Ereignis e eine Kante $(q, \mathtt{unten}(e))$ mit der Kapazität $W(l(e), p)$ konstruieren. Damit jedes Ereignis e genügend Marken erhält, konstruieren wir für jedes Ereignis e eine Kante $(\mathtt{oben}(e), s)$ mit der Kapazität $W(p, l(e))$ und überprüfen später, ob ein Fluss die Kapazität jeder dieser Kanten ausschöpft. Eigenschaft (iii) stellen wir sicher, indem wir den Durchfluss des Knotens k_0 begrenzen. Dies geschieht durch eine Kante (q, k_0) mit der Kapazität $m_0(p)$. Es ergibt sich das assoziierte kompakte Flussnetzwerk. Wenn wir einen Fluss konstruieren können, der alle der Senke eingehenden Kanten füllt, und wir die Verteilung des Flusses in einen kompakten Markenfluss im Szenario übersetzen, ist ein solcher kompakter Markenfluss gültig für das Szenario.

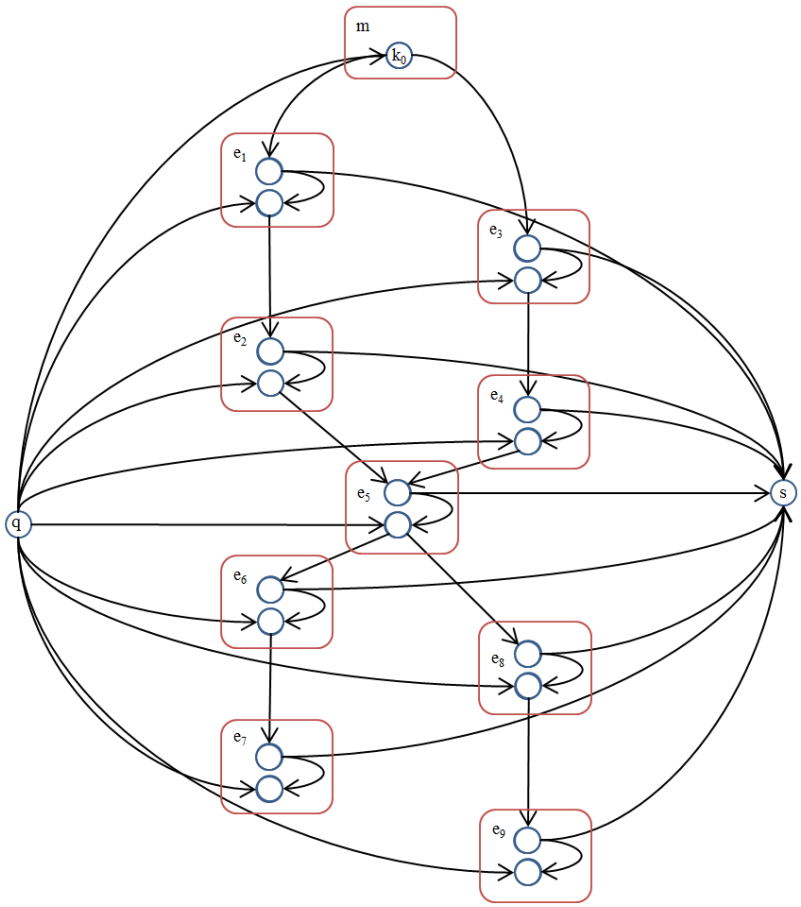

Abbildung 38: Das assoziierte kompakte Flussnetzwerk zu dem Szenario aus Abbildung 35.

Abbildung 38 zeigt das assoziierte kompakte Flussnetzwerk zum Szenario aus Abbildung 35 und einem beliebigen markierten S/T-Netz. Da wir das S/T-Netz nicht angeben, sind die Kanten ohne ihre Kapazitäten dargestellt. In dieser Abbildung sind die Knoten in Konturen gezeichnet, die mit den zugehörigen Ereignissen beschriftet sind. Der

jeweils obere Knoten in der Kontur ist der oben-Knoten, der untere ist der unten-Knoten. Die Kapazitäten der Kanten ergeben sich für ein konkretes markiertes S/T-Netz auf folgende Weise:

- Kanten, die die Quelle verlassen, haben die Kapazität der Anzahl an Marken, die die zugehörige Transition in der Stelle p produziert bzw. den Wert der Anfangsmarkierung $m_0(p)$.
- Innere Kanten haben die Kapazität $M_p(bpo, N)$ aus Definition 5.2.1 und sie beschränken den Fluss nicht.
- Kanten, die zur Senke führen, haben die Kapazität der Anzahl an Marken, die die zugehörige Transition aus der Stelle p konsumiert.

In Abbildung 38 ist der Knoten k_0 nur mit den Knoten verbunden, die zu Ereignissen gehören, die in dem Szenario minimal sind. Bevor wir das assoziierte kompakte Flussnetzwerk formal definieren, müssen wir beweisen, dass diese Vereinfachung zulässig ist. Der folgende Satz besagt, dass für jeden gültigen kompakten Markenfluss ein weiterer gültiger kompakter Markenfluss existiert, bei dem der Wert des Markenflusses auf allen Ereignissen, die in der Ordnung des Szenarios nicht minimal sind, gleich Null ist.

Satz 5.2.1 (über nicht minimale Ereignisse)

SATZ 5.2.1 (ÜBER NICHT MINIMALE EREIGNISSE)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $bpo = (E, <, l)$ *ein Szenario und* $\lhd$ *das Skelett der Ordnung* $<$*. Wir beschreiben die Menge der in der Ordnung von* bpo *minimalen Ereignisse mit* $min(E)$*.*
Existiert für eine Stelle $p \in P$ *ein gültiger kompakter Markenfluss* x*, so existiert für* p *ein gültiger kompakter Markenfluss* x_{min} *mit*

$$\sum_{e \in \{E \setminus min(E)\}} x_{min}(e) = 0.$$

BEWEIS: Sei $e' \in E \setminus min(E)$ ein Ereignis für das $x(e') > 0$ gilt. Es existiert ein Ereignis $e \in min(E)$, für das ein Weg aus Skelettkanten $\omega = (e_1, e_2)(e_2, e_3) \dots (e_{n-1}, e_n)$ mit $e_1 = e$ und $e_n = e'$ existiert.
Wir definieren einen kompakten Markenfluss $x' : (E \cup \lhd) \to \mathbb{N}_0$ durch

$$x'(a) = \begin{cases} 0, & \text{falls } a = e', \\ x(e) + x(e'), & \text{falls } a = e, \\ x(e_i, e_{i+1}) + x(e'), & \text{falls } a = (e_i, e_{i+1}) \in \omega, \\ x(a), & \text{sonst.} \end{cases}$$

Für das Ereignis e_n gilt:

$$zu_{x'}(e_n) = zu_x(e_n) - x(e_n) + x(e_n) = zu_x(e_n) \text{ und}$$

$$ab_{x'}(e_n) = ab_x(e_n).$$

Für alle $i \in \{1, \dots n-1\}$ gilt:

$$zu_{x'}(e_i) = zu_x(e_i) + x(e_n) \text{ und}$$

$$ab_{x'}(e_i) = ab_x(e_i) + x(e_n).$$

Für den kompakten Markenfluss x' gelten demnach weiterhin die Eigenschaften (i) und (ii). Weiter gilt:

$$\begin{aligned}\sum_{a\in E} x'(a) &= x'(e) + x'(e') + \sum_{a\in E\setminus\{e,e'\}} x'(a)\\ &= x(e) + x(e') + 0 + \sum_{a\in E\setminus\{e,e'\}} x(a)\\ &= \sum_{a\in E} x(a),\end{aligned}$$

und damit ist x' gültig für p.
Wiederholen wir diese Konstruktion für alle nicht minimalen Ereignisse, so erhalten wir den gesuchten gültigen kompakten Markenfluss. □

Satz 5.2.1 zeigt, dass es genügt, nach gültigen kompakten Markenflüssen zu suchen, die positive Werte auf Kanten und minimalen Ereignissen besitzen. Insgesamt ergibt sich das zu einem Szenario assoziierte kompakte Flussnetzwerk.

Definition 5.2.1 (Assoziiertes kompaktes Flussnetzwerk)

Definition 5.2.1 (Assoziiertes kompaktes Flussnetzwerk)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $bpo = (E, <, l)$ *ein Szenario mit* $E = \{e_1, \ldots, e_{|E|}\}$, $\lhd$ *das Skelett der Ordnung* $<$ *und* $p \in P$ *eine Stelle. Sei* $M_p(bpo, N) = \sum_{e\in E} W(p, l(e))$ *die Summe der durch die Ereignisse aus* p *konsumierten Marken. Wir definieren das assoziierte kompakte Flussnetzwerk* $G = (K, F, c, q, s)$*:*

$K = \{k_1, \ldots, k_{2|E|}\} \cup \{k_0, q, s\}$ *und* $oben : E \to K$, $unten : E \to K$ *ist definiert durch* $oben(e_i) = k_i$ *und* $unten(e_i) = k_{|E|+i}$.

$$\begin{aligned}F &= F_m \cup F_e \cup F_\lhd \cup F_q \cup F_0 \cup F_s \text{ mit}\\ F_m &= \{(k_0, oben(e)) \mid e \in min(E)\},\\ F_e &= \{(oben(e), unten(e)) \mid e \in E\},\\ F_\lhd &= \{(unten(e), oben(e')) \mid e \lhd e'\},\\ F_q &= \{(q, unten(e)) \mid e \in E\},\\ F_0 &= \{(q, k_0)\},\\ F_s &= \{(oben(e), s) \mid e \in E\}.\end{aligned}$$

$$c(k, k') = \begin{cases} W(l(unten^{-1}(k')), p), & \text{falls } k = q, k' \neq k_0,\\ m_0(p), & \text{falls } (k, k') = (q, k_0),\\ W(p, l(oben^{-1}(k))), & \text{falls } k' = s,\\ M_p(bpo, N), & \text{sonst.}\end{cases}$$

Der Vorteil des assoziierten kompakten Flussnetzwerks gegenüber dem einfachen assoziierten Flussnetzwerk ist die geringere Anzahl an Kanten. Die Größe des assoziierten kompakten Flussnetzwerks wächst nicht in der Größe der transitiven Ordnung des Szenarios,

sondern in der Größe des Skelettes dieser Ordnung. Nur für pathologische Beispiele stimmt das Skelett mit der transitiven Ordnung überein. Abbildung 35 zeigt ein Beispiel, bei dem der Unterschied an einem kleinen Szenario bereits deutlich zu erkennen ist. Die Menge der Kanten des assoziierten Flussnetzwerks beeinflusst die Laufzeit der Fluss-Maximierungs-Algorithmen, und diese haben großen Anteil an der Laufzeit der Szenario-Verifikation.
Ein Vergleich der Struktur der beiden assoziierten Flussnetzwerke zeigt, dass im einfachen assoziierten Flussnetzwerk alle Wege von der Quelle zur Senke die Länge drei besitzen. Es existieren keine längeren Wege, aber viele Sackgassen und Knoten, die nicht von der Quelle aus erreichbar sind. Bei dem assoziierten kompakten Flussnetzwerk ergibt sich die Länge der Wege aus der Länge der Wege im kompakten Szenario, dafür verzweigen die Wege an Knoten nicht so stark. In Kapitel 6 werden wir die Folgen dieser Unterschiede mit Hilfe von Laufzeit-Experimenten untersuchen.
Wir übersetzen einen Fluss im assoziierten kompakten Flussnetzwerk in einen kompakten Markenfluss im Szenario und erhalten den assoziierten kompakten Markenfluss.

Definition 5.2.2 (Assoziierter kompakter Markenfluss)

DEFINITION 5.2.2 (ASSOZIIERTER KOMPAKTER MARKENFLUSS)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $bpo = (E, <, l)$ *ein Szenario,* $\lhd$ *das Skelett der Ordnung* $<$, $p \in P$ *eine Stelle und* g *ein Fluss im assoziierten kompakten Flussnetzwerk* $G = (K, F, c, q, s)$.
Wir definieren den assoziierten kompakten Markenfluss $x : (E \cup \lhd) \to \mathbb{N}_0$ *als*

$$x(a) = \begin{cases} g(unten(e), oben(e')), & \text{falls } a = (e, e') \in \lhd, \\ g(k_0, oben(e)), & \text{falls } a = e \in min(E), \\ 0, & sonst. \end{cases}$$

Wir beweisen, dass für ein Szenario und für ein markiertes S/T-Netz zu einer Stelle $p \in P$ genau dann ein gültiger kompakter Markenfluss x existiert, wenn ein maximaler Fluss im assoziierten kompakten Flussnetzwerk den Wert $M_p(bpo, N)$ besitzt. Dazu zeigen wir den engen Zusammenhang zwischen Flüssen im einfachen assoziierten Flussnetzwerk und Flüssen im kompakten assoziierten Flussnetzwerk.

Satz 5.2.2 (über Flüsse in assoziierten Flussnetzwerken)

SATZ 5.2.2 (ÜBER FLÜSSE IN ASSOZIIERTEN FLUSSNETZWERKEN)
Sei (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz,* $bpo = (E, <, l)$ *ein Szenario,* $\lhd$ *das Skelett der Ordnung* $<$, $p \in P$ *eine Stelle,* $G = (K, F, c, q, s)$ *das assoziierte Flussnetzwerk und* $G^\lhd = (K, F^\lhd, c^\lhd, q, s)$ *das kompakte assoziierte Flussnetzwerk. Für jeden Fluss* g *in* G *existiert ein Fluss* $g^\lhd$ *in* $G^\lhd$ *und für jeden Fluss* $g^\lhd$ *in* $G^\lhd$ *existiert ein Fluss* g *in* G, *so dass* $w(g) = w(g^\lhd)$ *gilt.*

BEWEIS: Mit den Notationen aus Definition 4.5.7 und Definition 5.2.1 gilt: $F_q = F_q^\triangle$, $F_0 = F_0^\triangle$ und $F_s = F_s^\triangle$. Die Kapazitäten der Kanten c und $c^\lhd$ stimmen, wenn man sie auf die Mengen F_q, F_0 und F_s einschränkt, überein. Jede weitere Kante in G, sowie jede weitere Kante in $G^\lhd$, besitzt das Kantengewicht $M_p(bpo, N)$ und keine dieser Kanten beschränkt den Fluss, da die Kapazität des Schnittes $(K \backslash \{s\}, \{s\})$ in beiden Flussnetzwerken $M_p(bpo, N)$ beträgt.
Jeder Weg von der Quelle zur Senke in G besteht aus drei Kanten. Die erste und die letzte Kante begrenzen den Fluss und existieren identisch

in $G^{\triangleleft}$. Jede mittlere Kante dieser Wege entsteht aus der transitiven Ordnung des Szenarios und beschränkt den Fluss nicht. Zu dieser Kante existiert ein äquivalenter Weg aus inneren Kanten in $G^{\triangleleft}$, die den Fluss ebenfalls nicht beschränken, da sich die Kanten in $G^{\triangleleft}$ aus dem Skelett der Ordnung des Szenarios ergeben.
Jeder Weg von der Quelle zur Senke in $G^{\triangleleft}$ besteht aus einer Sequenz von Kanten. Die erste und die letzte Kante begrenzen den Fluss und existieren identisch in G. Die anderen Kanten des Weges entstehen aus dem Skelett der Ordnung des Szenarios und bilden einen Weg, der den Fluss nicht beschränkt. Zu diesem Weg existiert eine äquivalente direkte Kante in G, die den Fluss nicht beschränkt, da sich die Kanten in G direkt aus der (transitiven) Ordnung des Szenarios ergeben.
Die inneren Kanten beider Flussnetzwerke erlauben die gleichen Verteilungen von Fluss und die äußeren Kanten sind identisch. Zu jedem Fluss in G existiert ein Fluss mit gleichem Wert in $G^{\triangleleft}$ und andersherum. □

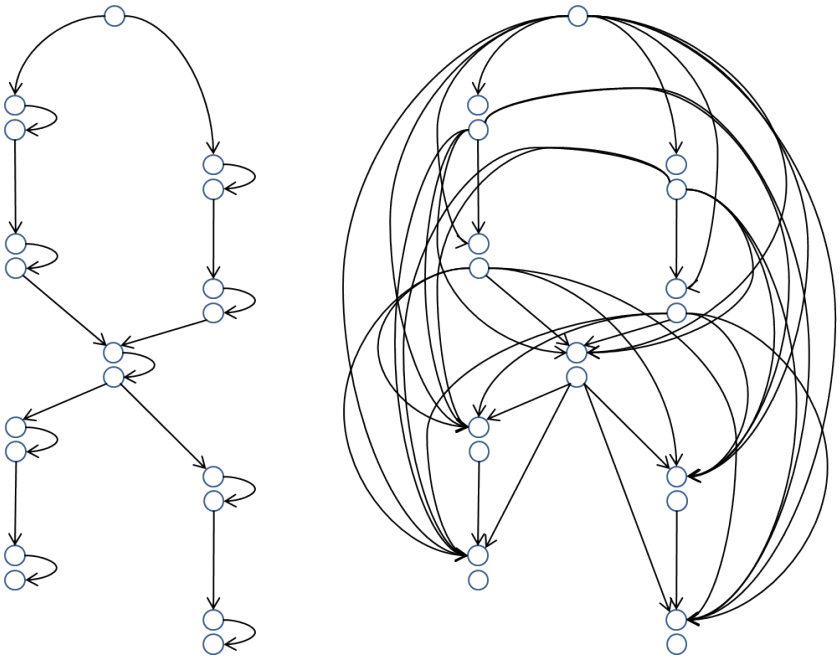

Abbildung 39: Links: Innere Kanten eines assoziierten kompakten Flussnetzwerks. Rechts: Inneren Kanten eines assoziierten einfachen Flussnetzwerks.

Abbildung 39 illustriert die Idee des Beweises von Satz 5.2.2. Sie zeigt die inneren Kanten der beiden Varianten von assoziierten Flussnetzwerken. Die linke Seite zeigt einen Ausschnitt aus Abbildung 38. Abgebildet sind die Menge aller Kanten, deren Kapazität $M_p(\mathrm{bpo}, N)$ beträgt, und alle Knoten bis auf Quelle und Senke. Die rechte Seite zeigt den gleichen Ausschnitt eines zugehörigen einfachen assoziierten Flussnetzwerks. Es wird deutlich, wie sich direkte Kanten im assoziierten Flussnetzwerk in Wege von Kanten im assoziierten kompakten Flussnetzwerk übersetzen. Aus Satz 5.2.2 ergibt sich das folgende Korollar.

KOROLLAR 5.2.1 (ÜBER DAS ASSOZIIERTE KOMPAKTE FLUSSNETZWERK)
Sei $\mathrm{bpo} = (E, <, l)$ *ein Szenario und* (N, m_0) *mit* $N = (P, T, W)$ *ein markiertes S/T-Netz. Das Szenario* bpo *ist genau dann in der Szenario-Sprache* $L(N, m_0)$ *enthalten, wenn für jede Stelle* $p \in P$ *ein maximaler Fluss im assoziierten kompakten Flussnetzwerk* G_p *den Wert* $M_p(\mathrm{bpo}, N)$ *besitzt.*

Korollar 5.2.1 (über das assoziierte kompakte Flussnetzwerk)

Korollar 5.2.1 beschreibt einen Algorithmus, der das Szenario-Verifikations-Problem über assoziierte kompakte Flussnetzwerke entscheidet.

ALGORITHMUS 5.2.1 (SZENARIO-VERIFIKATION ÜBER KOMPAKTE MARKENFLÜSSE)

Eingabe: *Szenario oder Szenario-Spezifikation* $\mathrm{bpo} = (E, <, l)$ *und ein markiertes Petrinetz* $N = (P, T, W, m_0)$
Ausgabe: *Entscheidet ob* $\mathrm{bpo} \in L(N, m_0)$ *gilt*

```
1   FOREACH p ∈ P
2   {
3       G ← Assoziiertes kompaktes Flussnetzwerk(p)
4       w ← Wert maximaler Fluss(G)
5       IF (w < M_p(bpo, N)) RETURN false
6   }
7   RETURN true
```

Algorithmus 5.2.1 (Szenario-Verifikation über kompakte Markenflüsse)

Algorithmus 5.2.1 entscheidet das Szenario-Verifikations-Problem mit Hilfe der Definition der gültigen kompakten Markenflüsse. Eine gegebene Szenario-Spezifikation kann für jede Stelle direkt in ein kompaktes Flussnetzwerk übersetzt werden, ohne dass zuvor die transitive Hülle der Ordnung berechnet werden muss (Zeile 3). Bei der Übersetzung kann man entweder nur das Skelett der Ordnung oder eine Szenario-Spezifikation betrachten. Zusätzliche innere transitive Kanten ändern an der Aussage von Korollar 5.2.1 nichts. Für jedes kompakte Flussnetzwerk wird ein maximaler Fluss berechnet (Zeile 4). In Zeile 5 wird dieser Wert mit dem Wert $M_p(\mathrm{bpo}, N)$ verglichen. Ist der Wert des Flusses zu klein, terminiert der Algorithmus mit negativer Antwort in Zeile 5. Erreicht der Algorithmus Zeile 7, ist das Szenario-Verifikations-Problem mit positiver Antwort entschieden.
Die Laufzeit von Algorithmus 5.2.1 setzt sich aus zwei Zeiten zusammen: Erstens der Zeit für die Konstruktionen der Flussnetzwerke, zweitens der Zeit zur Lösung der Fluss-Maximierungs-Probleme. Die Laufzeit der Konstruktion eines assoziierten Flussnetzwerks kann man gut durch die Anzahl der Elemente des Flussnetzwerks beschreiben. Es besteht aus $2|E| + 3$ Knoten und aus ungefähr $|\lhd| + 2 \mid E \mid$ Kanten. Die Laufzeiten der Fluss-Maximierungs-Algorithmen haben wir ausführlich in Kapitel 3 betrachtet. Dabei hängt die Laufzeit stark von der Anzahl der Kanten des assoziierten Flussnetzwerks ab. Insgesamt erhalten wir eine Schlechtester-Fall-Komplexität in $O(|P| \cdot |E|^3)$, wenn wir den Preflow-Push Algorithmus verwenden. Die Schlechtester-Fall-Komplexität entspricht der Komplexität des Algorithmus, der die einfachen kompakten Markenflüsse verwendet. Einen Unterschied erkennen wir, wenn wir die durchschnittlichen Laufzeiten betrachten. Diese wachsen wie die Funktion $|P| \cdot 2|\lhd|$. Der Faktor $|P|$ ergibt sich, da der Algorithmus für jede Stelle einen maximalen Fluss im assoziierten kompakten Flussnetzwerk berechnet. Der Faktor $|\lhd|$ ergibt sich zweimal, einmal aus

der Konstruktion des assoziierten Flussnetzwerks und einmal aus der Berechnung des maximalen Flusses bei einem günstigen Durchlauf mit dem Preflow-Push Algorithmus. Auf das gleiche Szenario angewandt sollte Algorithmus 5.2.1 um den Faktor $|<| \,/\, |\triangleleft|$ schneller sein als der Algorithmus, der das einfache assoziierte Flussnetzwerk verwendet. Die Implementierung dieses neuen Algorithmus als Plugin für das VipTool ist Teil dieser Arbeit. In Kapitel 6 werden wir seine Laufzeiten diskutieren und mit den anderen Szenario-Verifikations-Algorithmen vergleichen.

6

VERGLEICH DER SZENARIO-VERIFIKATIONS-ALGORITHMEN

Dieses Kapitel vergleicht die Laufzeiten der in den Kapitelen 4 und 5 vorgestellten Algorithmen zur Lösung des Szenario-Verifikations-Problems. Zu diesem Zweck wurden die entsprechenden Algorithmen als Plugins für das VipTool implementiert. Der nächste Abschnitt beschreibt vier Laufzeitexperimente, anhand derer die Komplexität der Algorithmen ausführlich untersucht und diskutiert wird. Der übernächste Abschnitt gibt einen kurzen Einblick in die Integration der Plugins in das VipTool.

6.1 LAUFZEITEXPERIMENTE

Kapitel 4 beschreibt zu Beginn zwei Charakterisierungen der Szenario-Sprache eines markierten S/T-Netzes, die Charakterisierung über Prozesse und die Charakterisierung über die Aktiviertheit von Schnitten des Szenarios. Beide Charakterisierungen führen zu Algorithmen, die das Szenario-Verifikations-Problem in Exponentialzeit entscheiden. Am Ende von Kapitel 4 ist eine Charakterisierung über einfache Markenflüsse dargestellt. Aus dieser Charakterisierung lassen sich zwei Verfahren ableiten, die das Szenario-Verifikations-Problem in Polynomialzeit entscheiden. Das erste Verfahren ist der iterative Test, das zweite ist der direkte Test. Der iterative Test wurde in [11] bereits für erste Laufzeitexperimente implementiert. Das direkte Verfahren wurde in [66] vorgeschlagen, allerdings bisher noch nicht implementiert.
Ein großer Kritikpunkt an einer Charakterisierung der Szenario-Sprache über einfache Markenflüsse ist, dass diese auf der vollständigen Ordnung des Szenarios definiert sind. Für lichte Szenarien führt die Verwendung der einfachen Markenflüsse zu erstaunlich schnellen Algorithmen. Sobald ein Szenario allerdings viele Abhängigkeiten zwischen den Ereignissen beschreibt, werden die gleichen Algorithmen mit der steigenden Anzahl an Kanten erheblich langsamer. Für ein dichtes Szenario ist oftmals der aktivierte Schnitte Algorithmus schneller als der in [11] implementierte iterative Algorithmus, der einfache Markenflüsse verwendet. Diesen Umstand haben wir in dieser Arbeit in Kapitel 5 mit der Entwicklung einer vierten Charakterisierung, den kompakten Markenflüssen, behoben. Diese erlaubt es, die Gültigkeit eines Szenarios mit Hilfe des Skelettes seiner Ordnung zu entscheiden. Das Ziel der entsprechenden Definition ist eine in jedem Fall effiziente Entscheidung des Szenario-Verifikations-Problems.
In diesem Abschnitt werden wir die durchschnittlichen Laufzeiten aller entwickelten Verfahren vergleichen und bewerten. Besonders gründlich werden wir dabei die neuen, direkten Verfahren untersuchen. Diese werden wir jeweils in zwei Versionen testen, die sich durch den ver-

wendeten Fluss-Maximierungs-Algorithmus unterscheiden. Abschnitt 3.6 beschreibt, warum die Fluss-Maximierungs-Algorithmen von Dinic (Algorithmus iii) und der Preflow-Push Algorithmus (Algorithmus vii) die für uns am besten geeigneten Algorithmen sind. Wir wollen beide einsetzen, um den optimalen Algorithmus für die speziellen Probleminstanzen der assoziierten Flussnetzwerke zu bestimmen.
Wir fassen die zu untersuchenden Algorithmen an dieser Stelle noch einmal zusammen:

Alg I	Szenario-Verifikation über Schnitte eines Szenarios (Algorithmus 4.3.1)
Alg II	Szenario-Verifikation über Prozessnetze eines S/T-Netzes (Algorithmus 4.4.1)
Alg III	Szenario-Verifikation über Markenflüsse, iterativer Test (Algorithmus 4.5.1)
Alg IV	Szenario-Verifikation über Markenflüsse, direkter Test unter Verwendung des Algorithmus von Dinic (Algorithmus 4.5.2)
Alg V	Szenario-Verifikation über Markenflüsse, direkter Test unter Verwendung des Preflow-Push Algorithmus (Algorithmus 4.5.2)
Alg VI	Szenario-Verifikation über kompakte Markenflüsse, direkter Test unter Verwendung des Algorithmus von Dinic (Algorithmus 5.2.1)
Alg VII	Szenario-Verifikation über kompakte Markenflüsse, direkter Test unter Verwendung des Preflow-Push Algorithmus (Algorithmus 5.2.1)

Um die folgenden Laufzeitexperimente durchzuführen, wurden diese sieben Algorithmen implementiert. Bevor wir jedoch die Laufzeitexperimente betrachten, überlegen wir, an welchen Beispielen wir diese durchführen. Bis zu diesem Zeitpunkt existieren keine Benchmarks, die S/T-Netze und deren Szenario-Sprache enthalten. In der Praxis existieren zwar sogenannte Log-Dateien großer Workflow-Systeme, die Szenarien mit bis zu 6000 Ereignissen enthalten, allerdings sind diese Szenarien nur Sequenzen von Ereignissen und beschreiben keine Nebenläufigkeit. Die einfachste Möglichkeit geeignete Test-Szenarien zu erhalten ist es, ein Szenario aus einer Szenario-Sprache eines markierten S/T-Netz direkt zu konstruieren. Wir betrachten ein Szenario, das wir sequentiell oder parallel wiederholen, um größere Szenarien zu konstruieren. Der Umstand, dass sich ein Szenario aus gleichen Teilen zusammensetzt, hat auf die Laufzeit der Algorithmen keinen Einfluss. Eine andere Möglichkeit ist die teilweise Berechnung der Szenario-Sprache eines S/T-Netzes, aus der man dann ein zufälliges Szenario auswählen kann. Im letzten Experiment dieses Kapitels bedienen wir uns dieser Methode.
Es bleibt zu bemerken, dass man durch beide Methoden nur Szenarien erhält, die in der Sprache des jeweiligen S/T-Netzes enthalten sind. In dieser Arbeit verzichten wir bewusst darauf, Laufzeitexperimente mit nicht ausführbaren Szenarien durchzuführen. Es ist leicht einzusehen, dass sich die Laufzeit aller Algorithmen verkürzt, wenn das

eingegebene Szenario nicht in der Sprache des S/T-Netzes enthalten ist. In diesem Fall können die Algorithmen früher abbrechen und eine negative Antwort liefern. Algorithmus I kann abbrechen, sobald ein nicht aktivierter Schnitt gefunden ist. Um so früher ein solcher Schnitt bearbeitet wird, um so schneller ist der Algorithmus, und Laufzeitexperimente liefern verzerrte Ergebnisse je nach Lage des Schnittes im Szenario. Algorithmus II kann abbrechen, sobald das in der topologischen Ordnung nächste Ereignis nicht an die Menge der bis zu diesem Zeitpunkt konstruierten Prozesse angehängt werden kann. Können die Ereignisse in ihrer topologischen Ordnung zu einem Prozess zusammengesetzt werden, kommt der Algorithmus erst nach dem Vergleich der konstruierten Sprache mit der Eingabe zu einem Ergebnis. Die Algorithmen III bis VII können abbrechen, sobald eine Stelle gefunden ist, für die es nicht möglich ist, einen gültigen Markenfluss zu konstruieren. Wieder ist die Reihenfolge, in der die Plätze bearbeitet werden, zufällig. Alle Algorithmen können um so früher abbrechen, je stärker ein Szenario von der eigentlichen Sprache des S/T-Netzes abweicht. Wann genau abgebrochen werden kann, ist dabei immer von der zufälligen Bearbeitungsreihenfolge abhängig, so dass Laufzeitexperimente nicht aussagekräftig sind.
Wir werden in den folgenden Abschnitten vier Experimente und deren Ergebnisse beschreiben. Der erste Abschnitt ist insbesondere dem Prozessnetz Algorithmus gewidmet, dessen Laufzeit für die in den darauf folgenden Abschnitten beschriebenen Experimente zu schlecht ist. Dann betrachten wir ein typisches Geschäftsprozessmodell und dichte Szenarien, bevor wir das gleiche Modell mit lichten Szenarien untersuchen. Der letzte Abschnitt beschreibt ein S/T-Netz, dessen Verhalten stark von der Verteilung von Ressourcen abhängt.

6.1.1 *Test des Prozessnetz Algorithmus*

Wir betrachten das Beispiel eines kollaborativen Geschäftsprozessesmodells. Abbildung 40 zeigt dazu ein markiertes S/T-Netz. Das S/T-Netz und der Geschäftsprozess sind ausführlich in [12] beschrieben. Wie für Geschäftsprozesse üblich, besitzt dieses S/T-Netz ganz links eine Stelle, die den Beginn, und ganz rechts eine Stelle, die das Ende des Geschäftsprozesses beschreibt. Für unsere Experimente fügen wir die zusätzliche Transition t ein, die eine Marke aus der Endstelle wieder in die Anfangsstelle zurücklegt. Auf diese Weise wird die Szenario-Sprache des S/T-Netzes um die Szenarien, die aus Wiederholungen der ursprünglichen Szenarien entstehen, ergänzt. Der abgebildete Geschäftsprozess besitzt eine übliche Struktur und enthält ein übliches Maß an Abhängigkeiten.
Abbildung 41 zeigt eine Szenario-Spezifikation aus der Szenario-Sprache des markierten S/T-Netzes aus Abbildung 40. Um ein erstes Experiment durchzuführen und dabei die Größe des zu prüfenden Szenarios variieren zu können, wiederholen wir das Szenario aus Abbildung 41 sequenziell. Die Anzahl der Wiederholungen determiniert die Länge der entstehende Szenarien. Auf diese Weise erhalten wir gut skalierbare Testfälle verschiedener Länge. Im ersten Experiment, vergleichen wir ausschließlich die beiden Algorithmen II und IV.

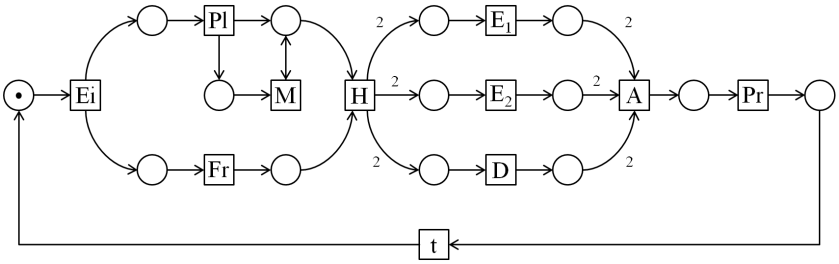

Abbildung 40: Das S/T-Netz eines Geschäftsprozesses.

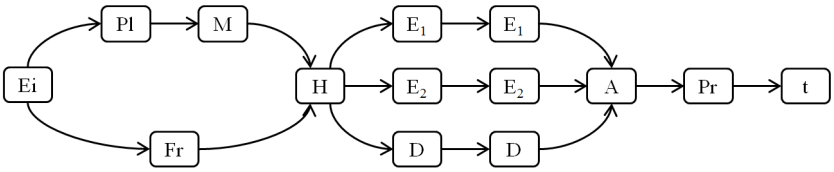

Abbildung 41: Eine Spezifikation des Geschäftsprozesses.

Experiment 6.1.1

EXPERIMENT 6.1.1

Wir betrachten sechs Szenarien $\mathrm{bpo}_1,\ldots,\mathrm{bpo}_6$ *aus der Szenario-Sprache des markierten S/T-Netzes aus Abbildung 40. Jedes Szenario besteht aus Iterationen des Szenarios aus Abbildung 41. Die sechs Szenarien entstehen, indem wir das Szenario 2, 5, 10, 20, 40 und 80 mal sequentiell wiederholen. In diesem Setting entscheiden wir das Szenario-Verifikations-Problem mit Hilfe der Algorithmen II und IV und vergleichen ihre Laufzeiten.*

Darstellung der Laufzeiten aus Experiment 6.1.1

	bpo_1	bpo_2	bpo_3	bpo_4	bpo_5	bpo_6
Wiederholungen	2	5	10	20	40	80
Ereignisse	28	70	140	280	560	1020
	Laufzeit in ms					
Prozessnetz Algorithmus (Alg II)	7	10	19	62	490	5860
Markenfluss Algorithmus, direkt, Dinic (Alg IV)	2	2	5	20	81	699

Experiment 6.1.1 zeigt, dass Algorithmus IV für jedes der sechs Szenarien $\mathrm{bpo}_1,\ldots,\mathrm{bpo}_6$ deutlich schneller läuft als Algorithmus II. Dabei sind die Testfälle hier so konstruiert, dass Algorithmus II nur einen einzigen Prozess und keine Teilmenge der Szenario-Sprache konstruieren muss. Das Experiment ist ein für Algorithmus II optimales Beispiel, trotzdem bleiben die Laufzeiten weit hinter den Laufzeiten des direkten Tests zurück.
Im nächsten Experiment wird das Szenario aus Abbildung 41 nicht nur sequentiell, sondern auch parallel wiederholt. Für ein Szenario,

das aus der nebenläufigen Ausführung zweier Instanzen des einfachen Szenarios entsteht, würde Algorithmus II ganze 2^{14} verschiedene Prozesse konstruieren. Jedes Ereignis kann an einen von zwei isomorphen Strängen innerhalb des Prozesses angehängt werden. Alle so entstehenden Prozesse sind isomorph. Um dies zu verhindern, kann man die Menge aller Prozesse nach jedem Schritt auf Isomorphie testen oder die Menge aller Prozesse in einer einzigen Struktur, dem sogenannten Branching-Prozess, konstruieren. Die zweite Möglichkeit würde allerdings wiederum einen Isomorphie-Test zwischen dem Branching-Prozess und dem eingegebenen Szenario erfordern. In keinem Fall würde sich die Laufzeiten aus Experiment 6.1.1, in dem Algorithmus II nur einen Prozess konstruiert muss, verbessern. Aus diesem Grund werden wir Algorithmus II in den folgenden Experimenten nicht mehr betrachten.

6.1.2 *Verifikation von dichten Szenarien*

In Experiment 6.1.1 haben wir die Anzahl der Ereignisse der eingegebenen Szenarien sukzessive erhöht. Dadurch, dass jede Wiederholung des Szenarios aus Abbildung 41 an das Ende des gesamten Szenarios geordnet wurde, entstehen Szenarien mit vielen Abhängigkeiten. Im nächsten Experiment vergleichen wir alle verblieben Algorithmen, indem wir drei nebenläufige Ausführungen des Szenarios aus Abbildung 41 iterieren und damit Szenarien größerer Breite betrachten. Die entstehenden Szenario-Spezifikationen sind in Abbildung 42 skizziert.

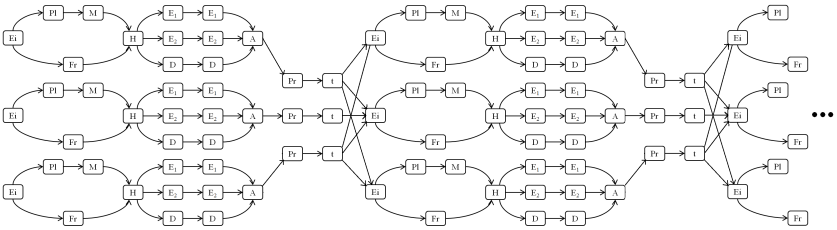

Abbildung 42: Eine dichte iterierte Spezifikation.

Experiment 6.1.2 *Experiment 6.1.2*

Wir betrachten fünf Szenarien $\mathrm{bpo}_1, \ldots, \mathrm{bpo}_5$ *aus der Szenario-Sprache des S/T-Netzes aus Abbildung 40. Jedes Szenario entsteht, indem wir drei parallel angeordnete Kopien des Szenarios aus Abbildung 41 sequentiell iterieren. Die Anzahl der Iterationen variiert über die Werte 2, 5, 10, 20 und 40. Die Anfangsmarkierung des S/T-Netzes wird zu diesem Zweck verdreifacht. Wir entscheiden das Szenario-Verifikations-Problem mit Hilfe des Algorithmus I und den Algorithmen III bis VII.*

Darstellung der Laufzeiten aus Experiment 6.1.2					
	bpo_1	bpo_2	bpo_3	bpo_4	bpo_5
Wiederholungen	2	5	10	20	40
Ereignisse	84	210	420	840	1680
Laufzeit in ms					
Schnitte Algorithmus (Alg I)	126	558	2424	12175	84605
Markenfluss Algorithmus, iterativ, Dinic (Alg III)	194	3602	32806	334647	-
Markenfluss Algorithmus, direkt, Dinic (Alg IV)	3	16	70	334	1955
Markenfluss Algorithmus, direkt, Preflow (Alg V)	3	14	65	342	1936
Kompakter Markenfluss, direkt, Dinic (Alg VI)	4	11	42	170	881
Kompakter Markenfluss, direkt, Preflow (Alg VII)	4	9	35	146	648
Quotient aufeinander folgender Laufzeiten					
Schnitte Algorithmus (Alg I)		4,4	4,3	5,0	6,9
Markenfluss Algorithmus, iterativ, Dinic (Alg III)		18,6	9,1	10,2	-
Markenfluss Algorithmus, direkt, Dinic (Alg IV)		5,3	4,4	4,8	5,9
Markenfluss Algorithmus, direkt, Preflow (Alg V)		4,7	4,6	5,3	5,7
Kompakter Markenfluss, direkt, Dinic (Alg VI)		2,8	3,8	4,0	5,2
Kompakter Markenfluss, direkt, Preflow (Alg VII)		2,3	3,9	4,2	4,4

In Experiment 6.1.2 besitzt der aktivierte Schnitte Algorithmus I die zweitlängsten Laufzeiten. Das Szenario in Abbildung 41 besitzt 36 Schnitte. Iterieren wir die dreimalig nebenläufige Ausführung dieses Szenarios, erhalten wir 36^3 Schnitte für jede Iteration. Betrachten wir die Anzahl der Schnitte in Abhängigkeit der Anzahl der Ereignisse der Eingabe, so führt eine Verdopplung der Anzahl der Ereignisse nur zu einer Verdopplung der Anzahl der Schnitte. Die Szenarien des Experiments 6.1.2 sind dadurch für Algorithmus I günstig. Dass sich

die Laufzeiten von Test zu Test mehr als verdoppeln, hängt damit zusammen, dass die Zeit für die Berechnung aller Schnitte stark wächst und die zu den Schnitten gehörenden Präfixe immer größer werden.
Experiment 6.1.2 beinhaltet Algorithmus II nicht. Wie bereits beschrieben, ist die Anzahl der zu konstruierenden Prozesse bereits für das erste Szenario zu groß. Der Algorithmus wurde nach einer Laufzeit von über 10 Minuten abgebrochen.
Algorithmus III besitzt die längsten Laufzeiten. Jede Laufzeit entspricht in etwa dem Produkt der Anzahl der Ereignisse des Szenarios und der entsprechenden Laufzeit des Algorithmus IV. Der Grund dafür ist, dass Algorithmus III ein Flussnetzwerk für jedes Ereignis des Szenarios und jede Stelle des S/T-Netzes konstruieren und lösen muss. Zwar sind die einzelnen Flussnetzwerke jeder Iteration des Algorithmus III schneller zu lösen als das Flussnetzwerk, welches in Algorithmus IV konstruiert wird, dieser Vorteil wird aber aufgehoben, da man in Algorithmus III den aktuellen Fluss nach jeder Iteration modifizieren muss.
Die Algorithmen IV bis VII laufen deutlich schneller als die übrigen Algorithmen. Die Methode, das Szenario für jede Stelle in nur ein einziges Flussnetzwerk zu übersetzen und in diesem einen maximalen Fluss zu finden, der einen gültigen Markenfluss beschreibt, falls er einen definierten Wert erreicht, ist deutlich effizienter als alle bisher implementierten Verfahren. Die Laufzeiten dieser vier Algorithmen setzen sich dabei aus zwei Teilen zusammen, der Laufzeit für die Konstruktion der Flussnetzwerke und der Laufzeit der eingesetzten Fluss-Maximierungs-Algorithmen. Im folgenden Experiment stellen wir die Ergebnisse der Algorithmen IV bis VII aus Experiment 6.1.2 noch einmal dar und unterscheiden dabei diese beiden Komponenten.

EXPERIMENT 6.1.3 *Experiment 6.1.3*

Wir geben die Laufzeit der Algorithmen IV, V, VI und VII aus Experiment 6.1.2 als Summe der Laufzeiten der Konstruktionen der assoziierten Flussnetzwerke und der Laufzeiten der Fluss-Maximierungs-Algorithmen an.

Darstellung der Laufzeiten aus Experiment 6.1.3

	bpo_1	bpo_2	bpo_3	bpo_4	bpo_5
Wiederholungen	2	5	10	20	40
Ereignisse	84	210	420	840	1680
	Laufzeit in ms				
Markenfluss Algorithmus, direkt, Dinic (Alg IV)	1+2	5+11	22+48	128+206	824+1131
Markenfluss Algorithmus, direkt, Preflow (Alg V)	1+2	5+9	22+43	128+214	824+1112
Kompakter Markenfluss, direkt, Dinic (Alg VI)	1+3	4+7	7+35	44+126	176+705
Kompakter Markenfluss, direkt, Preflow (Alg VII)	1+3	4+5	7+28	44+102	176+472

Die eingegebenen Szenarien in Experiment 6.1.3 sind Szenario-Spezifikationen. Die Algorithmen IV und V müssen aus der Szenario-Spezifikation zunächst das Szenario berechnen. Die Algorithmen VI und VII können direkt die Spezifikation verwenden, da diese das Skelett der Ordnung enthält, und sparen so Laufzeit.
Die Algorithmen IV und V brauchen für die Konstruktion des Flussnetzwerks weit länger, als die Algorithmen VI und VII. Die Szenarien aus Experiment 6.1.2 sind dicht und damit ist die Konstruktion der einfachen assoziierten Flussnetzwerke aufwändiger, als die Konstruktion der kompakten assoziierten Flussnetzwerke der Algorithmen VI und VII.
Auch die Fluss-Maximierungs-Algorithmen laufen in den Algorithmen VI und VII deutlich schneller als in den Algorithmen IV und V. Dies ist leicht zu erklären, da sich die Laufzeit der Fluss-Maximierungs-Algorithmen hauptsächlich aus der Größe der assoziierten Flussnetzwerke ergibt. Während in diesem Experiment die Anzahl der inneren Kanten des assoziierten einfachen Flussnetzwerks ungefähr $|E|^2/2$ beträgt, ist die Anzahl der inneren Kanten im kompakten assoziierten Flussnetzwerk nur ungefähr $2|E|$. Dabei beschreibt $|E|$ die Anzahl der Ereignisse des eingegebenen Szenarios. Die Laufzeit der Fluss-Maximierungs-Algorithmen wird zudem von der Struktur der assoziierten Flussnetzwerke beeinflusst. Eine Beobachtung ist, dass die Algorithmen IV und V sich kaum in der Verwendung des eingesetzten Fluss-Maximierungs-Algorithmus unterscheiden. Das einfache assoziierte Flussnetzwerk besitzt trotz der vielen Kanten nur Wege der Länge drei zwischen Quelle und Senke. Darüber hinaus ist in Experiment 6.1.2, wie für Geschäftsprozessmodelle üblich, jede Stelle nur mit wenigen Transitionen verbunden. Dadurch entstehen viele Sackgassen und Kanten, die nicht mit der Quelle oder der Senke verbunden sind. In diesem Fall ist es bezüglich der Laufzeit egal, ob man einen maximalen Fluss mit dem Algorithmus von Dinic oder dem Preflow-Push Algorithmus konstruiert. Vereinfacht betrachtet durchsucht der Algorithmus von Dinic nämlich in kurzen und breiten Flussnetzwerken die gleichen Kanten, die der Preflow-Push Algorithmus zum Schieben des Flusses verwendet.
Die Algorithmen VI und VII unterscheiden sich nach dem eingesetzten Fluss-Maximierungs-Algorithmus. Insgesamt besitzt das kompakte assoziierte Flussnetzwerk wenige, dafür aber lange Wege. Wieder sorgt die Struktur des eingegebenen markierten S/T-Netzes dafür, dass der zu konstruierende kompakte Markenfluss nur wenige Kanten des kompakten Szenarios wirklich benutzt, da die Stellen nicht stark verzweigen und im Szenario benachbarte Ereignisse meist auch im S/T-Netz benachbart sind. In vielen Fällen erhält ein Ereignis den nötigen Markenfluss direkt von seinem Vorgänger. Somit ist die Verwendung des Preflow-Push Algorithmus vorteilhaft, da das konstruierte Flussnetzwerk licht ist und der Algorithmus stoppen kann, sobald der produzierte initiale Markenfluss die Quelle erreicht.
In Experiment 6.1.2 sind im unteren Teil die Wachstumsfaktoren der Laufzeiten aufgelistet, die wir direkt mit den entsprechenden Quotienten aus Experiment 3.6.1 der Fluss-Maximierungs-Algorithmen vergleichen können. Die Laufzeiten der Algorithmen VI und VII wachsen wie die Laufzeiten der verwendeten Fluss-Maximierungs-Algorithmen quadratisch in der Anzahl der eingegebenen Ereignisse. Die Laufzeiten der Algorithmen IV und V wachsen etwas stärker als die Laufzeiten der

verwendeten Fluss-Maximierungs-Algorithmen, da die Anzahl der Kanten im Flussnetzwerk quadratisch von der Anzahl der eingegebenen Ereignisse abhängt.
Wir fassen die Ergebnisse des Experimentes 6.1.3 noch einmal zusammen. Sowohl bei der Konstruktion des assoziierten Flussnetzwerks als auch bei der Berechnung eines maximalen Flusses sind die Algorithmen VI und VII laufzeiteffizienter als die Algorithmen IV und V. Algorithmus VII, der auf der neuen Charakterisierung über kompakte Markenflüsse beruht, benötigt bereits bei 1680 Ereignissen nur gut ein Drittel der Laufzeit eines Algorithmus, der die einfache Version der Markenflüsse verwendet.

6.1.3 *Verifikation von lichten Szenarien*

Im vorangegangenen Abschnitt wurden die durchschnittlichen Laufzeiten der Verifikation dichter Szenarien untersucht. Wie wir bereits gesehen haben, beeinflusst die Struktur der eingegebenen Szenarien die Laufzeit der Verifikations-Algorithmen stark. Um ein detailliertes Bild der Laufzeiten auch bei lichten Szenarien zu erhalten, beschreiben wir in diesem Abschnitt ein weiteres Experiment. Dazu werden lichte Szenarien des in Abbildung 40 gezeigten Geschäftsprozessmodells konstruiert und verifiziert. Durch dieses Experiment wird Algorithmus I, der die Menge aller Schnitte des Szenarios untersucht, an einem für ihn komplexeren Beispiel getestet. Außerdem sollte der Unterschied zwischen den direkten Markenfluss Algorithmen kleiner ausfallen, da sich das Skelett weniger stark von der Ordnung des Szenarios unterscheidet. Wir erstellen für das nächste Experiment drei zueinander unabhängige Stränge von Wiederholungen des Szenarios aus Abbildung 41. Die entstehenden Szenario-Spezifikationen sind in Abbildung 43 skizziert.

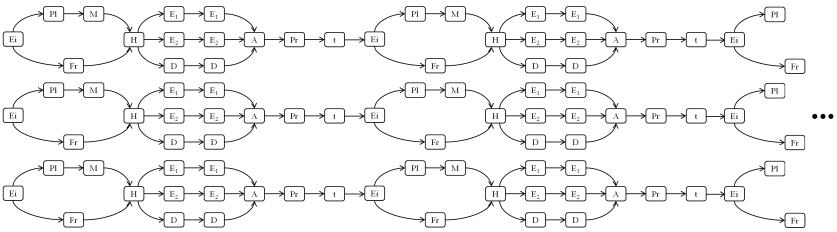

Abbildung 43: Eine lichte iterierte Spezifikation.

EXPERIMENT 6.1.4 *Experiment 6.1.4*

Wir betrachten fünf Szenarien $\mathrm{bpo}_1, \ldots, \mathrm{bpo}_5$ *aus der Szenario-Sprache des S/T-Netzes aus Abbildung 40. Jedes Szenario entsteht, indem wir das Szenarios aus Abbildung 41 sequentiell iterieren und drei Kopien des entstandenen Szenarios nebenläufig anordnen. Die Anzahl der Iterationen variiert über die Werte 2, 5, 10, 20 und 40. Die Anfangsmarkierung des S/T-Netzes wird zu diesem Zweck verdreifacht. Wir entscheiden das Szenario-Verifikations-Problem mit Hilfe des Algorithmus I und den Algorithmen III bis VII.*

Darstellung der Laufzeiten aus Experiment 6.1.4

	bpo_1	bpo_2	bpo_3	bpo_4	bpo_5
Wiederholungen	2	5	10	20	40
Ereignisse	84	210	420	840	1680
	Laufzeit in ms				
Schnitte Algorithmus (Alg I)	480	15091	-	-	-
Markenfluss Algorithmus, iterativ, Dinic (Alg III)	170	3002	25364	269862	-
Markenfluss Algorithmus, direkt, Dinic (Alg IV)	3	15	55	250	1336
Markenfluss Algorithmus, direkt, Preflow (Alg V)	3	11	47	216	1130
Kompakter Markenfluss, direkt, Dinic (Alg VI)	4	11	43	173	871
Kompakter Markenfluss, direkt, Preflow (Alg VII)	3	12	36	148	699
	Quotient aufeinander folgender Laufzeiten				
Schnitte Algorithmus (Alg I)		31,4	-	-	-
Markenfluss Algorithmus, iterativ, Dinic (Alg III)		17,7	8,4	10,6	-
Markenfluss Algorithmus, direkt, Dinic (Alg IV)		5,0	3,7	4,5	5,3
Markenfluss Algorithmus, direkt, Preflow (Alg V)		3,7	4,3	4,6	5,2
Kompakter Markenfluss, direkt, Dinic (Alg VI)		2,8	3,9	4,0	5,0
Kompakter Markenfluss, direkt, Preflow (Alg VII)		4,0	3,0	4,1	4,7

In Experiment 6.1.4 besitzt der aktivierte Schnitte Algorithmus I die längsten Laufzeiten. Innerhalb von 10 Minuten entscheidet er das Szenario-Verifikations-Problem nur für die beiden kleinsten Szenarien. Das Szenario aus Abbildung 41 besitzt 36 Schnitte. Damit besitzt jedes Szenario in Experiment 6.1.4 $(i \cdot 36)^3$ Schnitte, wenn i die Anzahl der Iterationen beschreibt. Die Anzahl der Schnitte ist somit viel höher als

die Anzahl der Schnitte in Experiment 6.1.2. In letzterem beträgt die Anzahl der Schnitte $i \cdot (36^3)$ und es ist klar, dass Algorithmus I nun eine wesentlich längere Laufzeit hat. Es hilft dabei kaum, dass die Breite aller in Experiment 6.1.4 betrachteten Szenarien durch neun begrenzt ist, und die Anzahl der Schnitte nicht exponentiell in der Anzahl der eingegebenen Ereignisse wächst.
Algorithmus III ist in Experiment 6.1.4 deutlich schneller als Algorithmus I und, nicht zu vergessen, auch deutlich schneller als Algorithmus II. Er liefert für die ersten vier Szenarien ein Ergebnis innerhalb von 10 Minuten. Da die Szenarien in diesem Experiment weniger Ordnung enthalten als die Szenarien in Experiment 6.1.2, sind die Laufzeiten des Algorithmus III nun deutlich schneller. Diese Robustheit gegenüber Nebenläufigkeit sorgt dafür, dass die Definition der Markenflüsse gegenüber den beiden anderen und älteren Definitionen über Prozessnetze und über die Aktiviertheit aller Schnitte vorteilhaft ist, sobald das betrachtete Szenario ein gewisses Maß an Nebenläufigkeit besitzt. Wieder ergibt sich die Laufzeit von Algorithmus III ungefähr als das Produkt der Anzahl der Ereignisse und der entsprechenden Laufzeit von Algorithmus IV.
Deutlich effizienter als die Algorithmen I bis III sind wieder die direkten Markenfluss Algorithmen IV bis VII. Experiment 6.1.5 gibt die Laufzeit dieser vier Algorithmen als Summe der Laufzeiten der Konstruktionen der assoziierten Flussnetzwerke und der Laufzeiten der Fluss-Maximierungs-Algorithmen an.

EXPERIMENT 6.1.5 *Experiment 6.1.5*

Wir geben die Laufzeit der Algorithmen IV, V, VI und VII aus Experiment 6.1.4 als Summe der Laufzeiten der Konstruktionen der assoziierten Flussnetzwerke und der Laufzeiten der Fluss-Maximierungs-Algorithmen an.

Darstellung der Laufzeiten aus Experiment 6.1.5

	bpo_1	bpo_2	bpo_3	bpo_4	bpo_5
Wiederholungen	2	5	10	20	40
Ereignisse	84	210	420	840	1680
	Laufzeit in ms				
Markenfluss Algorithmus, direkt, Dinic (Alg IV)	1+2	4+11	17+38	77+173	454+882
Markenfluss Algorithmus, direkt, Preflow (Alg V)	1+2	4+7	17+30	77+139	454+676
Kompakter Markenfluss, direkt, Dinic (Alg VI)	1+3	4+7	7+36	42+131	175+696
Kompakter Markenfluss, direkt, Preflow (Alg VII)	1+2	4+8	7+29	42+106	175+524

Die Laufzeiten der einfachen Markenfluss Algorithmen IV und V bei lichten Szenarien sind im Vergleich zu den Laufzeiten aus Experiment 6.1.3 deutlich schneller geworden. Beide Algorithmen sparen Zeit bei der Konstruktion der assoziierten Flussnetzwerke und können in diesen einen maximalen Fluss schneller berechnen. Weiter fällt auf, dass sich nun die Verwendung des Preflow-Push Algorithmus lohnt. Wie in Abbildung 18 aus Kapitel 3 beschrieben, ist der Preflow-Push Algorithmus vorteilhaft, wenn das betrachtete Flussnetzwerk licht ist.
Die Laufzeiten der direkten kompakten Markenfluss Algorithmen VI und VII sind nicht nur die schnellsten des Experimentes 6.1.5, sie entsprechen außerdem den Laufzeiten des Experiments 6.1.3. Für jede bis zu diesem Zeitpunkt bekannte Methode, das Szenario-Verifikations-Problem zu entscheiden, lassen sich Szenarien konstruieren, für die die Laufzeit ineffizient ist. Ist in einem zu prüfenden Szenario viel Nebenläufigkeit enthalten, so ist die Methode, die Schnitte des Szenarios zu prüfen, extrem laufzeitintensiv. Besitzt ein Szenario viele Prozesse, ist der Prozessnetz Algorithmus unbrauchbar. Ist das Szenario dicht und die Menge der Ereignisse stark geordnet, so ist die Methode der Markenflüsse ineffizient. Die neue Definition der kompakten Markenflüsse ist robust gegen jeden dieser Faktoren und somit in jedem Fall effizient. Obwohl die Szenarien aus Experiment 6.1.5 viele Abhängigkeiten enthalten, ist das Skelett der verwendeten Szenarien dem Skelett der Szenarien aus Experiment 6.1.3 sehr ähnlich. Das Ausmaß der beschriebenen Nebenläufigkeit spielt kaum eine Rolle. Die Laufzeiten der Algorithmen VI und VII sind auch hier fast ausschließlich von der Laufzeit des Fluss-Maximierungs-Algorithmus abhängig, sie wachsen quadratisch in der Größe der Eingabe.
Vergleichen wir die kompakten Markenfluss Algorithmus VI und VII in Bezug auf die verwendeten Fluss-Maximierungs-Algorithmen, so entsprechen die Ergebnisse den Erkenntnissen aus Experiment 6.1.3. Durch die Struktur des verwendeten S/T-Netzes erzielt auch in Experiment 6.1.5 der Preflow-Push Algorithmus etwas bessere Laufzeiten als der Algorithmus von Dinic.

6.1.4 *Verifikation ressourcenabhängiger Modelle*

An dieser Stelle führen wir ein letztes Experiment durch und betrachten ein S/T-Netz, dessen Verhalten stark von Ressourcen eines Systems abhängt. Das wohl bekannteste solche S/T-Netz ist das der essenden Philosophen. Eine bestimmte Anzahl an Philosophen sitzt um einen runden Tisch auf dem Stäbchen liegen. Jeder Philosoph hat links und rechts von sich ein Stäbchen, welches er sich mit dem jeweiligen Tischnachbarn teil. Jeder Philosoph denkt oder isst. Zum Essen benötigt er die beiden Stäbchen links und rechts von sich. Ein Philosoph kann nur mit dem Essen beginnen, wenn seine beiden Tischnachbarn denken und damit ihre Stäbchen, die Ressourcen dieses Prozesses, nicht benötigen. Beendet ein Philosoph das Essen und beginnt wieder zu denken, legt er beide Stäbchen zurück auf den Tisch.
Abbildung 44 zeigt das markierte S/T-Netz sechs essender Philosophen. Für das folgende Experiment variieren wir die Anzahl der Philosophen von 10 bis 80. Da immer die Hälfte aller Philosophen gleichzeitig essen kann, erhöhen wir mit der Anzahl der Philosophen neben den zu betrachtenden Stellen auch die mögliche Breite gültiger Szenarien. Neben der Anzahl der Philosophen variieren wir die Anzahl der Ereignisse

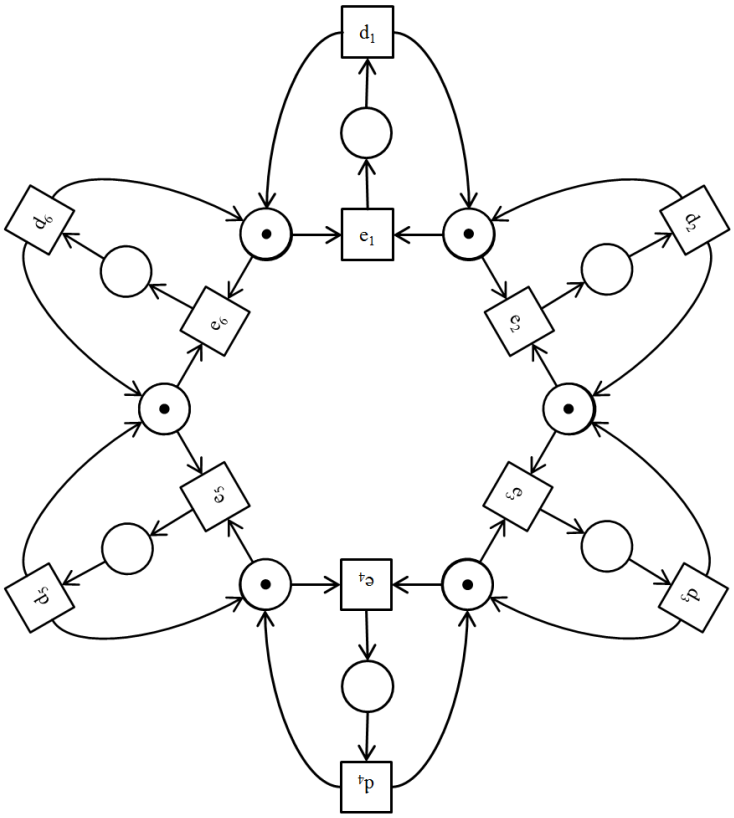

Abbildung 44: Sechs essende Philosophen.

der zu testenden Szenarien und erstellen diese zufällig aus der Prozess-Sprache des S/T-Netzes. Um diese Szenarien zu erstellen, entfalten wir Teile der Szenario-Sprache und wählen aus dieser 20 zufällige Szenarien aus. Im nächsten Experiment wiederholen wir dieses Vorgehen für jede Anzahl an Philosophen und jede Anzahl an Ereignissen und betrachten die durchschnittlichen Laufzeiten der Algorithmen IV bis VII. Um diese großen Probleminstanzen betrachten zu können, verzichten wir an dieser Stelle darauf, die Algorithmen I bis III zu testen. Deren Laufzeiten sind für diese Testfälle nicht akzeptabel.

EXPERIMENT 6.1.6 *Experiment 6.1.6*

Wir betrachten das Beispiel der essenden Philosophen und Szenarien verschiedener Größe. Wir betrachten 10, 20, 40 *oder* 80 *Philosophen und Szenarien mit* 200, 400, 800, 1600 *oder* 3200 *Ereignissen. Für jede sich ergebende Kombination erstellen wir zufällig* 20 *Szenarien der Szenario-Sprache und entscheiden das Szenario-Verifikations-Problem. Wir betrachten die durchschnittlichen Laufzeiten der Algorithmen IV bis VII für jede Kombination.*

Darstellung der Laufzeiten aus Experiment 6.1.6

Laufzeit in ms der Markenfluss Algorithmen

	mit Dinic (Alg IV)					*mit Preflow (Alg V)*				
	200	*400*	*800*	*1600*	*3200*	*200*	*400*	*800*	*1600*	*3200*
10	*29*	*121*	*539*	*2872*	*20414*	*25*	*110*	*547*	*2924*	*19644*
20	*48*	*216*	*942*	*4988*	*37573*	*40*	*189*	*865*	*4987*	*33730*
40	*87*	*357*	*1567*	*7813*	*58944*	*71*	*293*	*1299*	*6836*	*51137*
80	*127*	*577*	*2545*	*13503*	*125052*	*104*	*467*	*2052*	*9518*	*69326*

Laufzeit in ms der kompakten Markenfluss Algorithmen

	mit Dinic (Alg VI)					*mit Preflow (Alg VII)*				
	200	*400*	*800*	*1600*	*3200*	*200*	*400*	*800*	*1600*	*3200*
10	*25*	*80*	*317*	*1565*	*12602*	*19*	*60*	*242*	*1030*	*7188*
20	*43*	*162*	*644*	*3087*	*25949*	*32*	*123*	*490*	*2060*	*14276*
40	*81*	*315*	*1263*	*6112*	*49570*	*63*	*244*	*974*	*4133*	*28549*
80	*127*	*564*	*2342*	*11327*	*93006*	*100*	*438*	*1824*	*8103*	*55813*

Betrachten wir die beiden Algorithmen IV und V in Experiment 6.1.6. Die entsprechenden Laufzeiten sind in der oberen Hälfte der Tabelle dargestellt und variieren zwischen 25 Millisekunden und zwei Minuten. In einer Zeile der Tabelle wachsen die Laufzeiten mit der Anzahl der Ereignisse etwas stärker als die Laufzeiten der zugehörigen Fluss-Maximierungs-Algorithmen. Betrachtet man eine Spalte, so liegt der Faktor, um den sich die Laufzeiten erhöhen, wenn wir die Anzahl der Philosophen verdoppeln, meist deutlich unter dem Faktor Zwei. Verdoppelt man die Anzahl der Philosophen, so verdoppelt sich mit der Anzahl der Plätze auch die Anzahl der zu konstruierenden Markenflüsse. Der Faktor, um den sich die Laufzeiten in diesem Fall erhöhen, bleibt dennoch unter dem Wert Zwei, da mit einer Verdopplung der Philosophen auch der Grad der Nebenläufigkeit in den zufällig konstruierten Szenarien steigt. Die Algorithmen IV und V hängen stark von diesem Grad ab.
Betrachten wir nun die beiden Algorithmen VI und VII in Experiment 6.1.6. Die entsprechenden Laufzeiten sind in der unteren Hälfte der Tabelle dargestellt und variieren zwischen 19 Millisekunden und 1,5 Minuten. In einer Zeile wachsen die Laufzeiten mit der Anzahl der Ereignisse wie die Laufzeiten der zugehörigen Fluss-Maximierungs-Algorithmen. Betrachtet man eine Spalte, so liegt der Faktor, um den sich die Laufzeiten erhöhen, wenn wir die Anzahl der Philosophen verdoppeln, bei zwei. Verdoppelt man die Anzahl der Philosophen,

verdoppelt sich mit der Anzahl der Plätze auch die Anzahl der zu konstruierenden Markenflüsse. Die damit einhergehende Erhöhung des Grades der Nebenläufigkeit hat keinen großen Einfluss auf die kompakte Definition der Markenflüsse.
Vergleichen wir die Algorithmen nach den eingesetzten Fluss-Maximierungs-Algorithmen, so führt der Preflow-Push Algorithmus in diesem Experiment zu schnelleren Laufzeiten. Bei den Algorithmen, die die einfachen Markenflüsse verwenden, ist der Vorteil um so größer, je mehr Nebenläufigkeit die spezifizierten Szenarien besitzen. Bei den Algorithmen, die die kompakten Markenflüsse verwenden, ist der Preflow-Push Algorithmus in jedem Test deutlich schneller als der Algorithmus von Dinic.
Vergleichen wir die Algorithmen nach der verwendeten Definition der Markenflüsse, so führt die Verwendung der kompakten Markenflüsse zu schnelleren Laufzeiten. Bei den Algorithmen, die den Preflow-Push Algorithmus verwenden, benötigt der Algorithmus VII in Experiment 6.1.6 bei 3200 Ereignissen und 10 Philosophen nur knapp 36 Prozent der Laufzeit von Algorithmus V. Bei 3200 Ereignissen und 80 Philosophen benötigt der Algorithmus VII nur 85 Prozent der Laufzeit von Algorithmus V. Im ersten Fall handelt es sich um lange Szenarien mit einer maximalen Breite von fünf. Im zweiten Fall handelt es sich um sehr kurze Szenarien mit einer Breite von 80.

6.1.5 *Bewertung der Verifikations-Algorithmen*

Wir tragen an dieser Stelle die wichtigsten Ergebnisse dieses Kapitels zusammen und bewerten die sieben untersuchten Algorithmen abschließend.

- Algorithmus I, der die Menge aller Schnitte und deren Präfixe in einem Szenarios betrachtet, ist eine einfache und leicht zu implementierende Möglichkeit, das Szenario-Verifikations-Problem zu entscheiden. Ist das Szenario eine Sequenz, eine Schrittsequenz oder besitzt es sehr viel Ordnung, ist dieser Algorithmus eine brauchbare Alternative. Beschreibt ein Szenario nebenläufiges Verhalten, so wächst die Anzahl der Schnitte mit der Anzahl der Ereignisse so stark, dass Algorithmus I nicht mehr anwendbar ist.

- Algorithmus II, der das Szenario-Verifikations-Problem durch Konstruktion einer Teilmenge der Prozess-Sprache entscheidet, ist die schlechteste Möglichkeit, das Szenario-Verifikations-Problem zu entscheiden. In der für diese Arbeit beschriebenen Variante versuchen wir, von den Identitäten der Bedingungen so weit wie möglich zu abstrahieren, wodurch der Algorithmus auf kleinen Beispielen, die keine selbstnebenläufigen Transitionen enthalten, funktioniert. In diesem Fall konstruiert der Algorithmus gleichzeitig den zu Grunde liegenden Prozess. Die Laufzeit ist allerdings bereits für Beispiele mittlerer Größe nicht mehr akzeptabel.

- Algorithmus III, der ein iteratives Verfahren verwendet, um in dem Szenario einen einfachen Markenfluss für jede Stelle des S/T-Netzes zu konstruieren, stellt eine vernünftige Möglichkeit dar, das Szenario-Verifikations-Problem zu entscheiden. Ist das Szenario eine Sequenz, eine Schrittsequenz oder besitzt es viel Ordnung, ist das Verfahren jedoch ungeeignet. Es ist robust gegen

Nebenläufigkeit, insbesondere auch gegen Selbstnebenläufigkeit von Transitionen, wird aber in allen Experimenten von den direkten Methoden übertroffen.

- Algorithmus IV und Algorithmus V, die ein direktes Verfahren verwenden, um in dem Szenario einen einfachen Markenfluss für jede Stelle des S/T-Netz zu konstruieren, sind gute Möglichkeiten, das Szenario-Verifikations-Problem zu entscheiden. Beide Algorithmen sind robust gegen Nebenläufigkeit, nur etwas anfällig bei dichten Szenarien. Die Eingabe ist in jedem Fall ein vollständiges Szenario mit allen transitiven Abhängigkeiten. Bei großen Beispielen kostet das Arbeiten auf dieser Hülle Speicher und Laufzeit. Die Wahl des eingesetzten Fluss-Maximierungs-Algorithmus macht kaum einen Unterschied. Das assoziierte Flussnetzwerk besitzt nur Wege der Länge drei und so genügen die einfachsten Fluss-Maximierungs-Algorithmen, um akzeptable Laufzeiten zu erzielen. Im Gegensatz zum Preflow-Push Algorithmus mit Gap-Heuristik konstruiert der Algorithmus von Dinic einen vollständigen Fluss im assoziierten Flussnetzwerk. Ist man an einem konkreten gültigen Markenfluss im Szenario interessiert, sollte man also den Algorithmus von Dinic verwenden.
- Algorithmus VI und Algorithmus VII, die ein direktes Verfahren verwenden, um in dem Szenario einen kompakten Markenfluss für jede Stelle des S/T-Netz zu konstruieren, sind die besten Alternativen, das Szenario-Verifikations-Problem zu entscheiden. Sie laufen direkt auf einem Szenario, auf einer Szenario-Spezifikation oder auf dem Skelett eines Szenarios. Werden die Algorithmen auf dem Skelett oder einer Spezifikation ausgeführt, sind sie zuverlässig bei dichten und lichten Szenarien und auch bei selbstnebenläufigen Transitionen im eingegebenen S/T-Netz. Sie sind somit in jedem Fall die effizientesten Verifikations-Algorithmen. Wird der Algorithmus von Dinic eingesetzt, wird zusätzlich ein vollständiger kompakter Markenfluss berechnet. Der Preflow-Push Algorithmus ist das schnellste Verfahren zur Entscheidung des Szenario-Verifikations-Problems.

6.2 INTEGRATION IN DAS VIPTOOL

In diesem Abschnitt beschreiben wir die Integration der entwickelten Algorithmen als Plugins im VipTool. Das VipTool ist auf Algorithmen spezialisiert, die mit der Szenario-Sprache von S/T-Netzen in Zusammenhang stehen. Bis zu diesem Zeitpunkt sind verschiedenste Synthese-, Entfaltungs- und Verifikations-Algorithmen implementiert [37, 19, 22, 14]. Die aktuelle Version des VipTool ist auf der VipTool-Homepage zu finden:
www.fernuni – hagen.de/sttp/forschung/vip _tool.shtml.
Abbildung 45 zeigt einen Screenshot des neuen Flussnetzwerk-Plugins. Auf der linken Seite ist das Dateisystem abgebildet. In dem Projektordner Mississippi liegt ein XML-Dokument, dass das Flussnetzwerk aus Kapitel 3 enthält. Das Flussnetzwerk ist im Editor auf der rechten Seite des Screenshots geöffnet und kann dort bearbeitet werden. Über das Menü und die Toolbar kann man einen der für diese Arbeit implementierten Fluss-Maximierungs-Algorithmen auswählen und starten. Ein maximaler Fluss wird auf den Kanten des Flussnetzwerks dargestellt.

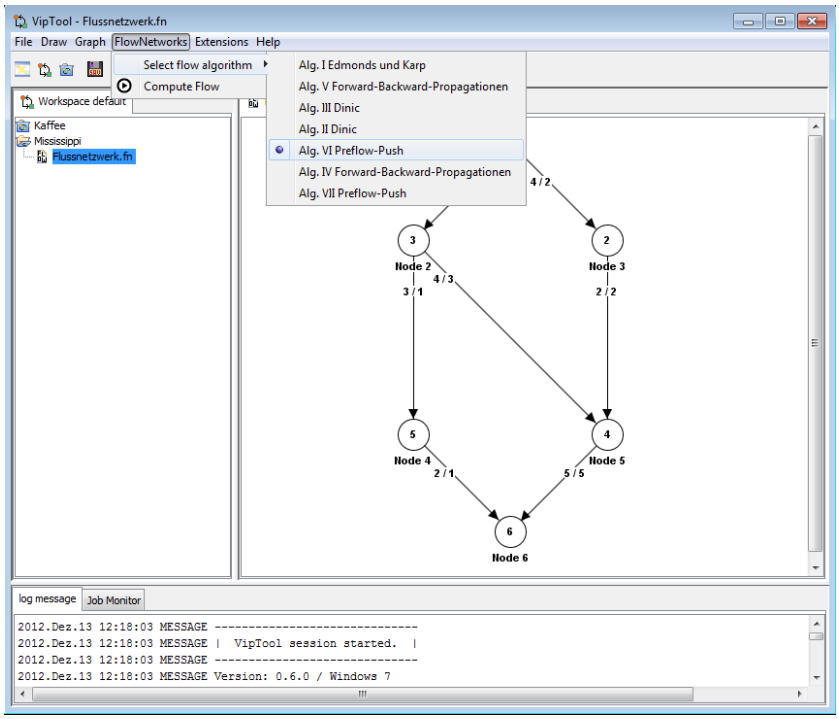

Abbildung 45: Screenshot des Flussnetzwerk-Plugins.

Abbildung 46 zeigt einen Screenshot des Szenario-Verifikations-Plugins. Auf der linken Seite ist wieder das Dateisystem abgebildet. In dem Projektordner Kaffee liegen die XML-Dokumente, die das S/T-Netz unserer Kaffee-Druckbrühmaschiene aus Kapitel 1, die zwei zugehörigen Szenarien und ein zusätzliches nicht gültiges Szenario. Das S/T-Netz und eines der Szenarien sind im Editor auf der rechten Seite des Screenshots geöffnet. Über den im VipTool vorhandenen S/T-Netz-Editor und den Szenario-Editor können diese bearbeitet werden. Über das Menü wird der entsprechende Szenario-Verifikations-Algorithmus ausgewählt. Der Algorithmus kann für eine Menge von Szenarien zu einem S/T-Netz ausgeführt werden. Dazu werden die entsprechenden Dateien im Projektbaum selektiert und der Verifikations-Algorithmus im Rechtsklick-Popup-Menü dieser Selektion gestartet. Daraufhin wird für jedes ausgewählte Szenario das Szenario-Verifikations-Problem bezüglich des ausgewählten S/T-Netzes entschieden. Ist das Szenario in der Sprache des S/T-Netzes enthalten, wird das entsprechende Icon des Szenarios im Projektbaum grün markiert, ist das Szenario nicht in der Sprache des S/T-Netzes enthalten, wird das entsprechende Icon des Szenarios im Projektbaum orange markiert. Damit ist man in der Lage, auch große Mengen von Szenarien schnell und unkompliziert in einem S/T-Netz zu testen und anschließend das S/T-Netz oder die Szenarien weiter zu bearbeiten.
Durch die Integration in das VipTool besteht eine gute Anbindung an weitere Plugins, und es ist möglich, ergänzend zu der Szenario-Verifikation andere Algorithmen auf der Menge der Szenarien durchzuführen. Sind zum Beispiel einige Szenarien nicht in der Sprache des S/T-Netzes enthalten, so kann man Synthese-Algorithmen verwenden, um ein alternatives S/T-Netz zu generieren. Entfaltungs-Algorithmen können Teile der Szenario-Sprache aus einem prototypischen S/T-Netz

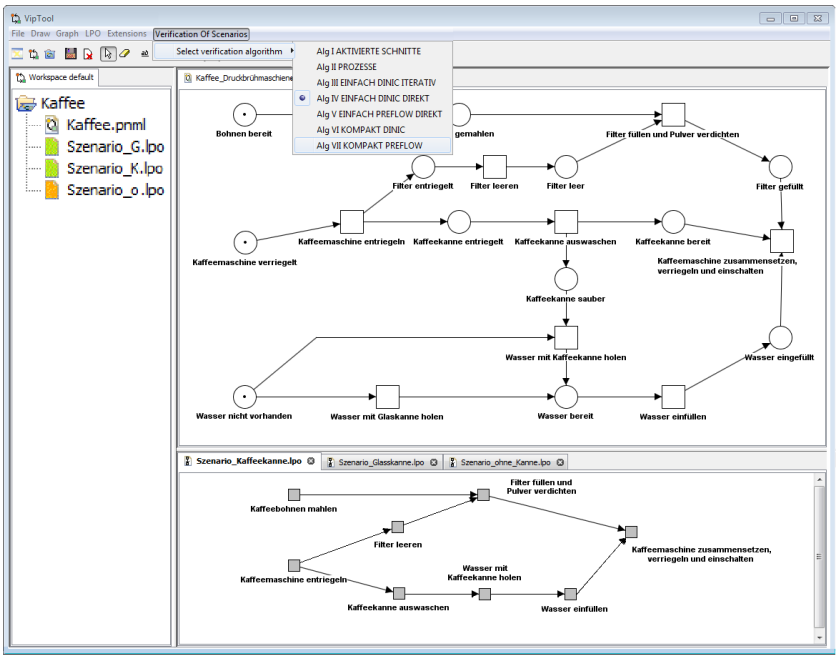

Abbildung 46: Screenshot des Verifikations-Plugins.

erstellen und vieles mehr. Insofern sind die neuen Plugins eine perfekte Ergänzung zu den bisherigen Funktionalitäten des VipTool und unterstützen alle in Kapitel 1 beschriebenen Anwendungen bestmöglich.

7

ABSCHLUSSBETRACHTUNGEN

In diesem Kapitel fassen wir die Inhalte der vorliegenden Arbeit noch einmal zusammen und geben einen Ausblick auf interessante weiterführende Forschungsthemen.

7.1 ZUSAMMENFASSUNG

Die vorliegende Arbeit behandelt ausführlich das Thema der Verifikation von Szenarien in S/T-Netzen. Zunächst wurde in Kapitel 1 die Bedeutung dieser Problemstellung im Rahmen der Modellierung von Systemen herausgestellt. Anschließend wurden bekannte Vorgehensmodelle zur Geschäftsprozessmodellierung beschrieben und ein neuer testgetriebener Ansatz skizziert. Der neue Ansatz zur Geschäftsprozessmodellierung eignet sich besonders, wenn Mitarbeiter eines Unternehmens stark in den Entwicklungsprozess eingebunden werden sollen. Die Möglichkeit, eine Szenario-Spezifikation effizient in einem Modell verifizieren zu können, ist essentiell für dieses Vorgehensmodell, aber auch für alle existierenden Alternativen. Die Möglichkeiten eines solchen testgetriebenen Ansatzes wurden an unserer Erfahrung mit der Audi AG und mit dem Therapaedicum Medifit dargestellt und diskutiert.
Neben diesen praktischen Anwendungen hat die in dieser Arbeit entwickelte kompakte Definition der Szenario-Sprache eines S/T-Netzes einen Wert an sich. Eine formale, effiziente und verständliche Definition der Halbordnungssemantik von S/T-Netzen ist Grundlage für alle weiteren Verfahren, die halbgeordnetes Verhalten von Petrinetzen betrachten. Die neben der Szenario-Verifikation bekanntesten Probleme, die direkt auf der Definition ausführbarer Szenarien beruhen, sind die Synthese aus Szenarien und das Berechnen der Szenario-Sprache eines S/T-Netzes.
Nach den einführenden Ausführungen und den formalen Grundlagen haben wir in Kapitel 3 Flussnetzwerke betrachtet. Der Fokus des Kapitels liegt dabei auf den verschiedenen Strategien der Fluss-Maximierungs-Algorithmen und deren durchschnittlichen Laufzeiten. Die in den Kapiteln 4 und 5 beschriebenen Verifikations-Algorithmen verwenden Fluss-Maximierungs-Algorithmen, und ist es notwendig, diese Verfahren gut abzuschätzen und bewerten zu können. Am Ende des Kapitel folgt ein ausführlicher Laufzeitvergleich der Fluss-Maximierungs-Algorithmen, um darauf aufbauend den optimalen Szenario-Verifikations-Algorithmus bestimmen zu können. Die Ergebnisse der Experimente zeigen, dass der Algorithmus von Dinic und der Preflow-Push Algorithmus mit Gap-Heuristik und maximaler initialer Höhenfunktion die beiden besten Verfahren sind.

In Kapitel 4 wurden S/T-Netze, die Szenario-Sprache von S/T-Netzen und das sich daraus ergebende Szenario-Verifikations-Problem definiert. Alle bis zu diesem Zeitpunkt existierenden Charakterisierungen der Szenario-Sprache wurden dargestellt und es wurde jeweils ein Verfahren abgeleitet, das Szenario-Verifikations-Problem zu entscheiden. Die Möglichkeiten und Beschränkungen der einzelnen Verfahren wurden ausführlich diskutiert, um die verschiedenen Ansätze bewerten zu können und diese anschließend als Basis für eine in jedem Fall effiziente Charakterisierung zu verwenden. Die Charakterisierung über Schnitte des eingegebenen S/T-Netzes abstrahiert so weit wie möglich von den Marken eines S/T-Netzes, während die Charakterisierung über Markenflüsse robuster gegen die Struktur des eingegebenen Szenarios ist, dabei aber die vollständige Verteilung aller Marken konstruiert. Die Charakterisierung über die Prozesse eines S/T-Netzes eignet sich nicht für die Szenario-Verifikation.
In Kapitel 5 wurde eine für das Problem der Szenario-Verifikation effiziente Charakterisierung der Szenario-Sprache entwickelt. Ausgehend von den einfachen Markenflüssen haben wir von der konkreten Verteilung der Marken abstrahiert und den Begriff der kompakten Markenflüsse eingeführt. Danach wurde die Äquivalenz der neuen Definition zu den bis zu diesem Zeitpunkt existierenden Charakterisierungen bewiesen. Im zweiten Teil des Kapitels haben wir einen entsprechenden Algorithmus entwickelt, der das Szenario-Verifikations-Problem mit Hilfe der neuen Definition entscheidet.
Für Kapitel 6 wurden zunächst sowohl die Fluss-Maximierungs-Algorithmen als auch die verschiedenen in Kapitel 4 und 5 beschriebenen Szenario-Verifikations-Algorithmen implementiert. Bei der Implementierung wurde besonderer Wert auf eine gute Laufzeit der Verfahren gelegt. Insgesamt entstand eine Sammlung von Szenario-Verifikations-Algorithmen, die alle grundlegenden Möglichkeiten, das Szenario-Verifikations-Problem zu entscheiden, abdeckt. Diese Algorithmen wurden in Laufzeitexperimenten getestet, deren Ergebnisse in Kapitel 6 dargestellt wurden. Insgesamt konnte der effizienteste Verifikations-Algorithmus bestimmt werden. Gleichzeitig wurde ein Eindruck von den Faktoren vermittelt, die die Laufzeit beeinflussen.
Es zeigte sich, dass der schnellste Verifikations-Algorithmus derjenige ist, welcher das Szenario-Verifikations-Problem mit Hilfe der neuen Charakterisierung über kompakte Markenflüsse und über den Wert eines maximalen Flusses im kompakten assoziierten Flussnetzwerk mit der Methode der Preflow-Push Algorithmen unter Verwendung der Gap-Heuristik berechnet. Die schnelle Laufzeit dieses Algorithmus wird kaum von der Struktur des Szenarios beeinflusst.

7.2 AUSBLICK

Diese Arbeit entwickelt eine neue Definition der Szenario-Sprache eines markierten S/T-Netzes. Für weiterführende Arbeiten bietet es sich an, diese Definition auf ihre Anwendbarkeit für andere, erweiterte Petrinetzklassen, zu testen. Ein mögliches Beispiel ist die Klasse der S/T-Netze mit Inhibitorkanten. Ihre Semantik kann mit einfachen Markenflüssen elegant beschrieben werden [68]. Es bleibt zu überprüfen, inwiefern sich die Ergebnisse dieser Arbeit auch auf diese Petrinetzklasse übertragen lassen.
Der Begriff der Ausführbarkeit eines Szenarios in einem S/T-Netz ist essentiell für verwandte Probleme. So ist beispielsweise die Synthese von Petrinetz-Modellen aus Verhaltensbeschreibungen ein interessantes Forschungsgebiet. Für die Synthese aus Szenarien lässt sich eine auf den einfachen Markenflüssen basierende Definition einer sogenannten Region auf einer Menge von Szenarien beschreiben. Die sich daraus ergebenden Synthese-Verfahren sind sehr genau, allerdings auch sehr rechenzeitintensiv. Einer der Hauptgründe für die schlechte Laufzeit dieser Algorithmen ist, dass sie in ihrem Verlauf ein Gleichungssystem lösen, bei dem der Wert eines Markenflusses auf jeder Kante jedes Szenarios einer Variable entspricht. Sollte es möglich sein, den Begriff einer Region kompakt auf dem Skelett eines Szenarios zu definieren, könnte damit die Größe der Gleichungssysteme erheblich reduziert werden.
Ein weiteres anerkanntes Forschungsgebiet ist das Modelchecking von Petrinetzen. Dabei werden Eigenschaften eines Modells als formaler Ausdruck spezifiziert und überprüft. Viele Methoden des Modelcheckings sind auf dem Erreichbarkeitsgraphen eines Petrinetzes durchführbar. Andere Methoden versuchen auch das halbgeordnete Verhalten von Petrinetzen zu erfassen. Diese Methoden sind auf eine effiziente Charakterisierung der Szenario-Sprache eines Petrinetzes angewiesen. Die Anwendbarkeit der neuen Definition der kompakten Markenflüsse im Bereich des Modelcheckings zu prüfen, ist eine weitere Herausforderung, die sich im Anschluss an die vorliegende Arbeit ergibt.
Eine andere Aufgabe ist es, die vollständige Szenario-Sprache eines S/T-Netzes mit Hilfe der kompakten Markenflüsse zu berechnen. Ein Ansatz, der die Szenario-Sprache mit Hilfe der einfachen Markenflüsse berechnet, wurde in [22] entwickelt. Wahrscheinlich ist die Laufzeitersparnis in diesem Bereich weniger deutlich als bei anderen Problemstellungen. Dennoch kann es sich lohnen zu untersuchen, inwiefern sich die kompakten Markenflüsse auch für die Berechnung der Szenario-Sprache eines S/T-Netzes eignen.
Zum Abschluss betrachten wir den vielleicht wichtigsten Bereich für weiterführende Forschungsarbeiten, nämlich die Anwendung der vorgestellten Verfahren zur Verifikation von Szenarien in der Praxis. Wir haben in Kapitel 1 bereits einige Vorschläge zur Anwendung dieser Verfahren im Rahmen von Projekten vorgestellt. Dabei kann der neue, effiziente Algorithmus die bestehenden Vorgehensmodelle ergänzen und zur Basis eines neuen, testgetriebenen Ansatzes werden. Es wird äußerst spannend, diese Methoden in weiteren Projekten zu untersuchen und zu evaluieren.

LITERATURVERZEICHNIS

[1] Aalst, Wil M. P. d. ; Desel, Jörg ; Kindler, Ekkart: *On the semantics of EPCs: A vicious circle.* In: Nüttgens, Markus (Hrsg.) ; Rump, Frank J. (Hrsg.): *Proc. of EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, GI-Arbeitskreistreffen, 2002, S. 71–79 (Zitiert auf Seite 12.)

[2] Aalst, Wil M. P. d. (Hrsg.) ; Desel, Jörg (Hrsg.) ; Oberweis, Andreas (Hrsg.): *Business Process Management, Models, Techniques, and Empirical Studies*. LNCS 1806, Springer, 2000 (Zitiert auf Seite 12.)

[3] Aalst, Wil M. P. d. ; Rubin, Vladimir ; Dongen, Boudewijn F. v. ; Kindler, Ekkart ; Guenther, C.W: *Process Mining: A Two-Step Approach using Transition Systems and Regions.* BPM Center Report: BPM-06-30, 2006 (Zitiert auf Seite 24.)

[4] Ahuja, Ravindra K. ; Magnanti, Thomas L. ; Orlin, James B.: *Network flows: Theory, algorithms, and applications*. Prentice Hall (Englewood Cliffs), 1993 (Zitiert auf Seiten 35 und 57.)

[5] Anderson, Ann ; Beattie, Ralph ; Beck, Kent: *Chrysler Goes to "Extremes"*. Chrysler Corporation – Distributed Computing, 1998 (Zitiert auf Seite 24.)

[6] Badouel, Éric ; Caillaud, Benoît ; Darondeau, Philippe: *Distributing Finite Automata Through Petri Net Synthesis.* In: *Formal Aspects of Computing Nr. 6*, 2002, S. 447–470 (Zitiert auf Seite 25.)

[7] Baumgarten, Bernd: *Petri-Netze: Grundlagen und Anwendungen*. Spektrum (Heidelberg), 1996 (Zitiert auf Seite 69.)

[8] Beck, Kent: *Extreme Programming: A Discipline of Software Development*. In: Nierstrasz, Oskar (Hrsg.) ; Lemoine, Michel (Hrsg.): *Software Engineering - ESEC/FSE*, Bd. 1687, Springer, 1999 (Zitiert auf Seiten 20 und 21.)

[9] Beck, Kent ; Andres, Cynthia: *Extreme programming explained: Embrace change*. Addison-Wesley (Boston), 2005 (Zitiert auf Seite 21.)

[10] Beck, Kent ; Beedle, Mike ; Bennekum, Arie van ; Cockburn, Alistair ; Cunningham, Ward ; Fowler, Martin ; Grenning, James ; Highsmith, Jim ; Hunt, Andrew ; Jeffries, Ron ; Kern, Jon ; Marick, Brian ; Martin, Robert C. ; Mellor, Steve ; Schwaber, Ken ; Sutherland, Jeff ; Thomas, Dave: *Manifesto for Agile Software Development*. 2001 (Zitiert auf Seite 21.)

[11] Bergenthum, Robin: *Algorithmen zur Verifikation von halbgeordneten Petrinetz-Abläufen: Implementierung und Anwendungen*. Diplomarbeit, Katholische Universität Eichstätt-Ingolstadt, 2006 (Zitiert auf Seiten 10, 25, 26, 87, 90, 92, 105, und 123.)

[12] Bergenthum, Robin ; Desel, Jörg ; Harrer, Andreas ; Mauser, Sebastian: *Learnflow Mining*. In: Seehusen, Silke (Hrsg.) ; Lucke,

Ulrike (Hrsg.) ; Fischer, Stefan (Hrsg.): *Proc. of DeLFI 2008*, LNI 132, GI, 2008 (Zitiert auf Seite 125.)

[13] Bergenthum, Robin ; Desel, Jörg ; Harrer, Andreas ; Mauser, Sebastian: *Modellierung und Mining kollaborativer Learnflows*. *Proc. of AWPN 2009*, 2009 (Zitiert auf Seite 14.)

[14] Bergenthum, Robin ; Desel, Jörg ; Juhás, Gabriel ; Lorenz, Robert: *Can I Execute My Scenario in Your Net? VipTool Tells You!* In: Donatelli, Susanna (Hrsg.) ; Thiagarajan, Pazhamaneri S. (Hrsg.): *Proc. of Petri Nets and Other Models of Concurrency 2006*, LNCS 4024, Springer, 2006, S. 381 – 390 (Zitiert auf Seiten 10, 11, 25, 87, 90, und 138.)

[15] Bergenthum, Robin ; Desel, Jörg ; Kölbl, Christian ; Mauser, Sebastian: *Experimental Results on Process Mining Based on Regions of Languages. Proc. of Satellite Workshop of ICATPN, CHINA 2008*, 2008 (Zitiert auf Seiten 24 und 106.)

[16] Bergenthum, Robin ; Desel, Jörg ; Lorenz, Robert ; Mauser, Sebastian: *Process Mining Based on Regions of Languages*. In: Alonso, Gustavo (Hrsg.) ; Dadam, Peter (Hrsg.) ; Rosemann, Michael (Hrsg.): *Proc. of Business Process Management, BPM 2007*, LNCS 4714, Springer, 2007, S. 375–383 (Zitiert auf Seiten 19 und 24.)

[17] Bergenthum, Robin ; Desel, Jörg ; Lorenz, Robert ; Mauser, Sebastian: *Synthesis of Petri Nets from Finite Partial Languages*. In: *Fundamenta Informaticae 88, Nr. 4*, 2008, S. 437–468 (Zitiert auf Seite 106.)

[18] Bergenthum, Robin ; Desel, Jörg ; Lorenz, Robert ; Mauser, Sebastian: *Synthesis of Petri Nets from Scenarios with VipTool*. In: Hee, Kees Max v. (Hrsg.) ; Valk, Rüdiger (Hrsg.): *Proc. of Petri nets*, LNCS 5062, Springer, 2008, S. 388–398 (Zitiert auf Seite 18.)

[19] Bergenthum, Robin ; Desel, Jörg ; Mauser, Sebastian: *Comparison of Different Algorithms to Synthesize a Petri Net from a Partial Language*. In: Jensen, Kurt (Hrsg.) ; Billington, Jonathan (Hrsg.) ; Koutny, Maciej (Hrsg.): *Transactions on petri nets and other models of concurrency 3*, LNCS 5800, Springer, 2009 , S. 216–243 (Zitiert auf Seiten 26, 106, und 138.)

[20] Bergenthum, Robin ; Desel, Jörg ; Mauser, Sebastian ; Lorenz, Robert: *Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages*. In: *Fundamenta Informaticae 95*, Nr. 1, 2009, S. 187–217 (Zitiert auf Seite 106.)

[21] Bergenthum, Robin ; Juhás, Gabriel ; Lorenz, Robert ; Mauser, Sebastian: *Unfolding Semantics of Petri Nets Based on Token Flows*. In: *Fundamenta Informaticae 94*, Nr. 3-4, 2009, S. 331–360 (Zitiert auf Seiten 82 und 106.)

[22] Bergenthum, Robin ; Lorenz, Robert ; Mauser, Sebastian: *Faster Unfolding of General Petri Nets*. In: Philippi, Stephan (Hrsg.) ; Pinl, Alexander (Hrsg.): *Proc of Workshop Algorithmen und Werkzeuge für Petrinetze 2007*, 2007, S. 63–68 (Zitiert auf Seiten 138 und 143.)

[23] Bergenthum, Robin ; Lorenz, Robert ; Mauser, Sebastian: *Faster Unfolding of General Petri Nets Based on Token Flows*. In: Hee, Kees Max v. (Hrsg.) ; Valk, Rüdiger (Hrsg.): *Proc. of Applications and theory of Petri nets*, LNCS 5062, Springer, 2008, S. 13–32 (Zitiert auf Seiten 24, 87, und 106.)

[24] Bergenthum, Robin ; Mauser, Sebastian: *Synthesis of Petri Nets from Infinite Partial Languages with VipTool*. In: Lohmann, Niels (Hrsg.) ; Wolf, Karsten (Hrsg.): *Proc. of Workshop Algorithmen und Werkzeuge für Petrinetze 2008*, CEUR-WS.org 380, 2008, S. 81–86 (Zitiert auf Seite 106.)

[25] Bergenthum, Robin ; Mauser, Sebastian: *Folding Partially Ordered Runs*. *Proc. of Workshop Applications of Region Theory 2011*, 2011 (Zitiert auf Seiten 12, 14, 17, 18, und 19.)

[26] Bergenthum, Robin ; Mauser, Sebastian: *Mining with User interaction*. *Proc. of Workshop Applications of Region Theory 2011*, 2011 (Zitiert auf Seite 24.)

[27] Best, Eike ; Devillers, Raymond: *Sequential and concurrent behaviour in Petri net theory*. In: *Theoretical Computer Science 55*, Nr. 1, S. 87 – 136 (Zitiert auf Seite 73.)

[28] Billington, Jonathan (Hrsg.) ; Diaz, Michel (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Application of Petri nets to communication networks: Advances in Petrinets*, Springer (Berlin), 1999 (Zitiert auf Seite 25.)

[29] Bringhurst, Robert: *The elements of typographic style: Version 3.0*. Point Roberts Wash.: Hartley & Marks (Publishers), 2004 (Zitiert auf Seite 153.)

[30] Bron, Coenraad ; Kerbosch, Joep: *Finding All Cliques of an Undirected Graph (Algorithm 457)*. In: *Communications of The ACM 16*, Nr. 9, 1973, S. 575 – 576 (Zitiert auf Seite 78.)

[31] Cassandras, Christos G. ; Lafortune, Stéphane: *Introduction to discrete event systems*, Kluwer Academic (Boston), 1999 (Zitiert auf Seite 25.)

[32] Cazals, Frédéric ; Karande, Chinmay D.: *A note on the problem of reporting maximal cliques*. In: *Theoretical Computer Science 407*, Nr. 1-3, 2008, S. 564–568 (Zitiert auf Seite 78.)

[33] Cortadella, Jordi ; Kishinevsky, Michael ; Kondratyev, Alex ; Lavagno, Luciano ; Yakovlev, Alex: *Hardware and Petri Nets Application to Asynchronous Circuit Design*. In: Nielsen, Mogens (Hrsg.) ; Simpson, Dan (Hrsg.): *Proc. of Application and Theory of Petri Nets*, LNCS 1825, Springer (Berlin), 2000, S. 1–15 (Zitiert auf Seite 25.)

[34] Darondeau, Philippe: *Region Based Synthesis of P/T-Nets and Its Potential Applications*. In: Nielsen, Mogens (Hrsg.) ; Simpson, Dan (Hrsg.): *Proc. of Application and Theory of Petri Nets*, LNCS 1825, Springer (Berlin), 2000, S. 16–23 (Zitiert auf Seiten 24 und 25.)

[35] Desel, Jörg: *From Human Knowledge to Process Models*. In: Kaschek, Roland (Hrsg.) ; Kop, Christian (Hrsg.) ; Steinberger, Claudia (Hrsg.) ; Fliedl, Günther (Hrsg.): *Proc. of Information Systems and e-Business Technologies*, Lecture Notes in Business Information 5, Springer, 2008, S. 84–95 (Zitiert auf Seite 12.)

[36] Desel, Jörg ; Juhás, Gabriel: *"What Is a Petri Net?"*. In: Ehrig, Hartmut (Hrsg.) ; Juhás, Gabriel (Hrsg.) ; Padberg, Julia (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Unifying Petri Nets, Advances in Petri Nets*, LNCS 2128, Springer, 2001, S. 1–25 (Zitiert auf Seite 69.)

[37] Desel, Jörg ; Juhás, Gabriel ; Lorenz, Robert ; Neumair, Christian: *Modelling and Validation with VipTool*. In: Aalst, Wil M. P. d. (Hrsg.) ; Hofstede, Arthur H. M. (Hrsg.) ; Weske, Mathias (Hrsg.): *Proc. of Business Process Management 2003*, LNCS 2678, Springer, 2003, S. 380–389 (Zitiert auf Seiten 12, 27, 32, und 138.)

[38] Desel, Jörg ; Milijic, Vesna ; Neumair, Christian: *Model Validation in Controller Design*. In: Desel, Jörg (Hrsg.) ; Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Lectures on Concurrency and Petri Nets*, LNCS 3098, Springer, 2004, S. 467 – 495 (Zitiert auf Seite 25.)

[39] Desel, Jörg ; Petrucci, Laure: *Aggregating views for Petri net model construction*. In: *Proc. of Workshop on Petri Nets and Distributed Systems*, 2008, S. 17–31 (Zitiert auf Seiten 12 und 17.)

[40] Desel, Jörg ; Reisig, Wolfgang: *Place/Transition Petri Nets*. In: Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, LNCS 1491, Springer, 1998, S. 122 – 173 (Zitiert auf Seite 69.)

[41] Dinic, E. A.: *Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation*. In: *Soviet Math Doklady 11*, 1970, S. 1277 – 1280 (Zitiert auf Seiten 45 und 47.)

[42] Dongen, Boudewijn F. v. ; Aalst, Wil M. P. d.: *Multi-Phase Process Mining: Building Instance Graphs*. In: Atzeni, P. (Hrsg.) ; Chu, W.W (Hrsg.) ; Lu, H. (Hrsg.) ; Zhou, S. (Hrsg.) ; Ling, T.W (Hrsg.): *Proc. of Conceptual Modeling 2004*, LNCS 3288, Springer, 2004, S. 362–376 (Zitiert auf Seiten 12 und 17.)

[43] Dongen, Boudewijn F. v. ; Aalst, Wil M. P. d.: *Multi-phase Process mining: Aggregating Instance Graphs into EPCs and Petri Nets*. In: *Proc. of Workshop PNCWB 2005*, 2005, S. 35–58 (Zitiert auf Seite 18.)

[44] Dongen, Boudewijn F. v. ; Desel, Jörg ; Aalst, Wil M. P. d.: *Aggregating Causal Runs into Workflow Nets*. In: Jensen, Kurt (Hrsg.) ; Aalst, Wil M. P. d. (Hrsg.) ; Ajmone Marsan, Marco (Hrsg.) ; Franceschinis, Giuliana (Hrsg.) ; Kleijn, Jetty (Hrsg.) ; Kristensen, LarsMichael (Hrsg.): *Transactions on Petri Nets and Other Models of Concurrency 6*, LNCS 7400, Springer (Berlin), 2012, S. 334–363 (Zitiert auf Seite 19.)

[45] Dongen, Boudewijn F. v. ; Pinna, G. M. ; Aalst, Wil M. P. d.: *An Iterative Algorithm for Applying the Theory of Regions in Process Mining*. In: Reisig, Wolfgang (Hrsg.) ; Hee, Kees Max v. (Hrsg.)

; WOLF, Karsten (Hrsg.): *Proceedings of Workshop on Formal Approaches to Business Processes and Web Services 2007*, University of Podlasie, 2007, S. 36–55 (Zitiert auf Seite 24.)

[46] DUMAS, Marlon (Hrsg.) ; AALST, Wil M. P. d. (Hrsg.) ; HOFSTEDE, Arthur H. M. (Hrsg.): *Process-Aware Information Systems: Bridging People and Software Through Process Technology*, Wiley, 2005 (Zitiert auf Seite 12.)

[47] EDMONDS, Jack ; KARP, Richard M.: *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*. In: *J. ACM 19*, Nr. 2, 1972, S. 248–264 (Zitiert auf Seiten 38, 42, und 43.)

[48] ENGELFRIET, Joost: *Branching processes of Petri nets*. In: *Acta Informatica 28*, Nr. 6, 1991, S. 575–591 (Zitiert auf Seite 82.)

[49] ESPARZA, Javier ; RÖMER, Stefan ; VOGLER, Walter: *An Improvement of McMillan's Unfolding Algorithm*. In: *Formal Methods in System Design 20*, Nr. 3, 2002, S. 285–310 (Zitiert auf Seite 24.)

[50] FANCHON, Jean ; MORIN, Rémi: *Pomset Languages of FiniteStep Transition Systems*. In: FRANCESCHINIS, Giuliana (Hrsg.) ; WOLF, Karsten (Hrsg.): *Proc. of Applications and Theory of Petri Nets 2009*, LNCS 5606, Springer, 2009, S. 83–102 (Zitiert auf Seite 73.)

[51] FLOYD, Robert W.: *Algorithm 97: Shortest path*. In: *Communications of The ACM 5*, Nr. 6, 1962, S. 345 (Zitiert auf Seite 77.)

[52] FORD, L.R ; FULKERSON, D.R: *Maximal Flow Through A Network*. In: *Canadian Journal of Mathematics 8*, 1956, S. 399–404 (Zitiert auf Seiten 26, 38, 39, und 41.)

[53] GLABBEEK, R.J. v.: *The Individual and Collective Token Interpretations of Petri Nets*. In: *Porc. of CONCUR 2005*, Springer (London), 2005, S. 323–337 (Zitiert auf Seite 81.)

[54] GOLTZ, Ursula ; REISIG, Wolfgang: *Processes of Place/Transition-Nets*. In: DIAZ, J. (Hrsg.): *Automata Languages and Programming*, Bd. 154, Springer (London), 1983, S. 264–277 (Zitiert auf Seiten 10, 25, 26, und 79.)

[55] GOLTZ, Ursula ; REISIG, Wolfgang: *The Non-sequential Behavior of Petri Nets*. In: *Information and Control* 57, Nr. 2/3, 1983, S. 125–147 (Zitiert auf Seiten 10, 25, 26, und 79.)

[56] GRABOWSKI, J.: *On partial languages*. In: *Fundamenta Informaticae* 4, Nr. 2, 1981, S. 427–498 (Zitiert auf Seiten 10, 25, 26, und 76.)

[57] JANICKI, Ryszard ; KOUTNY, Maciej: *Structure of concurrency*. In: *Theoretical Computer Science* 112 , Nr. 1, 1993, S. 5–52 (Zitiert auf Seite 73.)

[58] JOSEPHS, Mark B. ; FUREY, Dennis P.: *A Programming Approach to the Design of Asynchronous Logic Blocks*. In: CORTADELLA, J. (Hrsg.) ; YAKOVLEV, Alex (Hrsg.) ; ROZENBERG, Grzegorz (Hrsg.): *Concurrency and hardware design*, LNCS 2549, Springer, 2002, S. 34–60 (Zitiert auf Seite 25.)

[59] JUHÁS, Gabriel: *Are these events independent? It depends!* Habilitation, Katholische Universität Eichstätt-Ingolstadt, 2005 (Zitiert auf Seite 73.)

[60] JUHÁS, Gabriel ; LORENZ, Robert ; DESEL, Jörg: *Can I Execute My Scenario in Your Net?* In: CIARDO, Gianfranco (Hrsg.) ; DARONDEAU, Philippe (Hrsg.): *Proc. of Applications and Theory of Petri Nets 2005*, LNCS 3536, Springer, 2005, S. 289–308 (Zitiert auf Seiten 10, 25, 26, 73, 75, 87, 88, 89, 90, und 105.)

[61] JUNGNICKEL, D.: *Graphs, networks, and algorithms*. Springer (Berlin), 1999 (Zitiert auf Seite 35.)

[62] JUNGNICKEL, D.: *Algorithms and Computation in Mathematics 5, Graphs, networks, and algorithms 2*, Springer (Berlin), 2005 (Zitiert auf Seite 32.)

[63] KARZANOV, A. V.: *Determining the maximal flow in a network by the method of preflows*. In: *Soviet Mathematics Doklady* 15, 1974, S. 434–437 (Zitiert auf Seite 54.)

[64] KIEHN, Astrid: *On the Interrelation Between Synchronized and Non-Synchronized Behaviour of Petri Nets*. In: *Elektronische Informationsverarbeitung und Kybernetik* 24, Nr. 1/2, 1988, S. 3–18 (Zitiert auf Seiten 25, 75, 81, und 105.)

[65] KING, V. ; RAO, S. ; TARJAN, S.: *A Faster Deterministic Maximum Flow Algorithm*. In: *Journal of Algorithms* 17, Nr. 3, 1994, S. 447–474 (Zitiert auf Seite 61.)

[66] LORENZ, Robert: *Szenario-basierte Verifikation und Synthese von Petrinetzen: Theorie und Anwendungen*. Habilitation, Katholische Universität Eichstätt-Ingolstadt, 2006 (Zitiert auf Seiten 11, 25, 90, 98, 102, 103, 105, und 123.)

[67] LORENZ, Robert ; JUHÁS, Gabriel ; MAUSER, Sebastian: *Partial Order Semantics of Types of Nets*. In: NIELSEN, Mogens (Hrsg.) ; KUČERA, Antonín (Hrsg.) ; MILTERSEN, Peter B. (Hrsg.) ; PALAMIDESSI, Catuscia (Hrsg.) ; TRUMA, Petr (Hrsg.) ; VALENCIA, Frank D. (Hrsg.): *Proc. of SOFSEM 2009* LNCS 5404, Springer, 2009, S. 388–400 (Zitiert auf Seite 73.)

[68] LORENZ, Robert ; MAUSER, Sebastian ; BERGENTHUM, Robin: *Testing the executability of scenarios in general inhibitor nets*. In: BASTEN, Twan (Hrsg.) ; JUHÁS, Gabriel (Hrsg.) ; SHUKLA, Sandeep K. (Hrsg.): *Proc. of Application of Concurrency to System Design 2007*, IEEE Computer Society, 2007, S. 167–176 (Zitiert auf Seiten 106 und 143.)

[69] LORENZ, Robert ; MAUSER, Sebastian ; BERGENTHUM, Robin: *Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios*. In: KLEIJN, Jetty (Hrsg.) ; YAKOVLEV, Alexandre (Hrsg.): *Proc. of Petri Nets and Other Models of Concurrency 2007*, LNCS 4546, Springer, 2007, S. 342–361 (Zitiert auf Seite 106.)

[70] MALHOTRA, Vishv M. ; KUMAR, M. P. ; MAHESHWARI, S. N.: *An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks*. In: *Information Processing Letters* 7, Nr. 6, 1994, S. 277–278 (Zitiert auf Seite 49.)

[71] MAUSER, Sebastian: *Synthese von Petrinetzen aus halbgeordneten Abläufen*, Dissertation, Fernuniversität in Hagen, 2011 (Zitiert auf Seiten 17 und 19.)

[72] Mayr, Heinrich C. ; Kop, Christian ; Esberger, Daniela: *Business Process Modeling and Requirements Modeling*. In: *International Conference on the Digital Society*, 2007, S. 8–14 (Zitiert auf Seiten 11 und 12.)

[73] McMillan, K.L: *Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits*. In: Bochmann, Gregor (Hrsg.) ; Probst, David K. (Hrsg.): *Computer Aided Verification*, Bd. 663, Springer (Berlin), 1993, S. 164–177 (Zitiert auf Seite 82.)

[74] McMillan, K.L ; Probst, David K.: *A Technique of State Space Search Based on Unfolding*. In: *Formal Methods in System Design*, 1992, S. 45–65 (Zitiert auf Seite 24.)

[75] Oestereich, Bernd: *Objektorientierte Geschäftsprozessmodellierung und modellgetriebene Softwareentwicklung*. In: *HMD - Praxis Wirtschaftsinform* 241, 2005 (Zitiert auf Seite 11.)

[76] Ottmann, Thomas ; Widmayer, Peter: *Algorithmen und Datenstrukturen 5*. Spektrum Akad. Verl. (Heidelberg), 2011 (Zitiert auf Seite 32.)

[77] Peterson, James L.: *Petri net theory and the modeling of systems*. Prentice-Hall (Englewood Cliffs), 1981 (Zitiert auf Seite 69.)

[78] Petri, Carl A.: *Kommunikation mit Automaten*, Dissertation, Technische Universität Darmstadt, 1962 (Zitiert auf Seiten 6 und 69.)

[79] Pratt, Vaughan: *Modelling Concurrency with Partial Orders*. In: *International Journal of Parallel Programming* 15, 1986 (Zitiert auf Seite 73.)

[80] Reisig, Wolfgang: *Petrinetze: Eine Einführung*. Springer (Berlin), 1986 (Zitiert auf Seite 69.)

[81] Rozenberg, Grzegorz (Hrsg.) ; Salomaa, Arto (Hrsg.): *Handbook of formal languages: Vol 3 Beyond Words*. Springer (Berlin), 1997 (Zitiert auf Seite 73.)

[82] Salbrechter, Alexander ; Mayr, Heinrich C. ; Kop, Christian: *Mapping pre-designed business process models to UML*. In: *Proc. of IASTED 2004*, 2004, S. 400–405 (Zitiert auf Seite 12.)

[83] Scheer, August-Wilhelm: *ARIS - vom Geschäftsprozess zum Anwendungssystem*. Springer (Berlin), 2002 (Zitiert auf Seiten 11, 12, 13, und 17.)

[84] Schlingloff, Holger ; Martens, Axel ; Schmidt, Karsten: *Modeling and Model Checking Web Services*. In: *Electron. Notes Theor. Comput. Sci.* 126, 2005, S. 3–26 (Zitiert auf Seite 25.)

[85] Software AG: *ARIS Process Performance Manager*, *www.softwareag.com/corporate/products/aris_platform/aris_controlling/aris_process_performance/overview/default.asp* (Zitiert auf Seiten 12, 13, 14, und 17.)

[86] Solé, Marc ; Carmona, Josep: *Process Mining from a Basis of State Regions*. In: Lilius, Johan (Hrsg.) ; Penczek, Wojciech (Hrsg.): *Proc. of Applications and Theory of Petri Nets 2010*, LNCS 6128, Springer (Berlin), 2010, S. 226–245 (Zitiert auf Seiten 24 und 106.)

[87] VOGLER, Walter: *Modular construction and partial order semantics of Petri nets*. LNCS 625, Springer (Berlin), 1992 (Zitiert auf Seiten 25, 75, 81, und 105.)

[88] VOGLER, Walter: *Partial words versus processes: A short comparison*. In: ROZENBERG, Grzegorz (Hrsg.): *Advances in Petri Nets 1992*, Bd. 609, Springer (Berlin), 1992, S. 292–303 (Zitiert auf Seite 76.)

[89] WARSHALL, Stephen: *A Theorem on Boolean Matrices*. In: *J. ACM* 9, Nr. 1, 1962, S. 11–12 (Zitiert auf Seite 77.)

[90] WESKE, Mathias: *Business process management: Concepts, languages, architectures*. Springer (Berlin), 2007 (Zitiert auf Seite 12.)

[91] WINSKEL, Glynn: *Event structures*. In: BRAUER, W. (Hrsg.) ; REISIG, W. (Hrsg.) ; ROZENBERG, G. (Hrsg.): *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Bd. 255, Springer (Berlin), 1987, S. 325–392 (Zitiert auf Seite 73.)

[92] ZHOU, MengChu ; DICESARE, Frank: *Petri net synthesis for discrete event control of manufacturing systems*. Kluwer Academic (Boston), 1993 (Zitiert auf Seite 25.)

KOLOPHON

Diese Dissertation wurde mittels LaTeX 2ε unter der Verwendung der Schriftbilder Palatino von Hermann Zapf und Euler gesetzt (als Type 1 PostScript Fonts wurden URW Palladio L und FPL verwendet). Aufzählungen wurden dagegen mit Bera Mono gesetzt. Bera Mono wurde ursprünglich von Bitstream Inc. als Bitstream Vera entwickelt (Type 1 PostScript Fonts wurden von Malte Rosenau und Ulrich Dirr zugänglich gemacht).
Der typografische Stil wurde von Bringhursts Arbeit *The Elements of Typographic Style* [29] inspiriert. Er ist für LaTeX über CTAN unter der Bezeichnung „`classicthesis`“ verfügbar.
Die spezifische Größe des Textblocks wurde nach Bringhursts Vorgaben berechnet. 10 pt Palatino benötigt 133.21 pt für die Zeichenfolge „abcdefghijklmnopqrstuvwxyz“. Dies ergibt eine gute Zeilenlänge zwischen 24–26 pc (288–312 pt). Die Wahl eines „double square textblock“ mit einem 1:2 Verhältnis resultiert in einem Textblock der Maße 312:624 pt (welcher die Kopfzeile in diesem Design einschließt).

CURRICULUM VITAE

PERSÖNLICHE ANGABEN

Familienname	Bergenthum
Vorname	Robin
Geburtsdatum	30. Oktober 1980
Geburtsort	Düsseldorf
Familienstand	ledig
Nationalität	deutsch
Konfession	evangelisch
Anschrift	Bürgerstr. 26, 58097 Hagen
E-Mail	robin.bergenthum@fernuni-hagen.de

AUSBILDUNGSGANG

08.1987 – 08.1991	Besuch der Grundschule Süd Oberursel
09.1991 – 12.1995	Besuch des Gymnasiums Oberursel
01.1996 – 06.2000	Besuch des Weidig-Gymnasiums in Butzbach Leistungskurse: Mathematik, Geschichte
17.06.2000	Abitur mit Note 1,7
09.2000 – 06.2001	Grundwehrdienst im 314. Fallschirmjägerbataillon in Oldenburg

10.2001 – 03.2006	Studium der Wirtschaftsmathematik an der Katholischen Universität Eichstätt-Ingolstadt Studienschwerpunkte: Wahrscheinlichkeitstheorie, Finanzmathematik, Numerische Mathematik, Finance & Banking, Kapitalmärkte, Wertpapiermanagement, Petrinetztheorie, Softwareentwicklung
30.09.2003	Vordiplom in Wirtschaftsmathematik mit Note 1,97
30.03.2006	Diplom in Wirtschaftsmathematik mit Note 1,73 Titel der Diplomarbeit: Algorithmen zur Verifikation von halbgeordneten Petrinetz-Abläufen: Implementierung und Anwendung

BERUFLICHE TÄTIGKEITEN

04.2002 – 01.2006	Wissenschaftliche Hilfskraft an dem Lehrstuhl für Angewandte Informatik der Katholischen Universität Eichstätt-Ingolstadt von Prof. Dr. Jörg Desel Aufgabenfelder: Korrekturarbeiten, Tutorium und Softwarentwicklung
10.2004 – 02.2005	Wissenschaftliche Hilfskraft an der Professur für Informatik der Katholischen Universität Eichstätt-Ingolstadt von Prof. Dr. Stephan Diehl Aufgabenfeld: Tutorium
08.2005 – 10.2005	Praktikum bei der Deka Investment GmbH in Frankfurt im Bereich Quantitative Methoden
04.2006 – 08.2010	Wissenschaftlicher Angestellter am Lehrstuhl für Angewandte Informatik der Katholischen Universität Eichstätt-Ingolstadt von Prof. Dr. Jörg Desel
09.2010 – heute	Wissenschaftlicher Angestellter am Lehrgebiet Softwaretechnik und Theorie der Programmierung der FernUniversität in Hagen von Prof. Dr. Jörg Desel

FORSCHUNGSGEBIETE

Verifikation, Synthese und Entfaltung von Petrinetzen, Petrinetzsemantiken, konzeptuelle Modellierung von Systemen und Szenarien, Geschäftsprozessmodellierung, Process Mining, Lernprozessmodellierung.