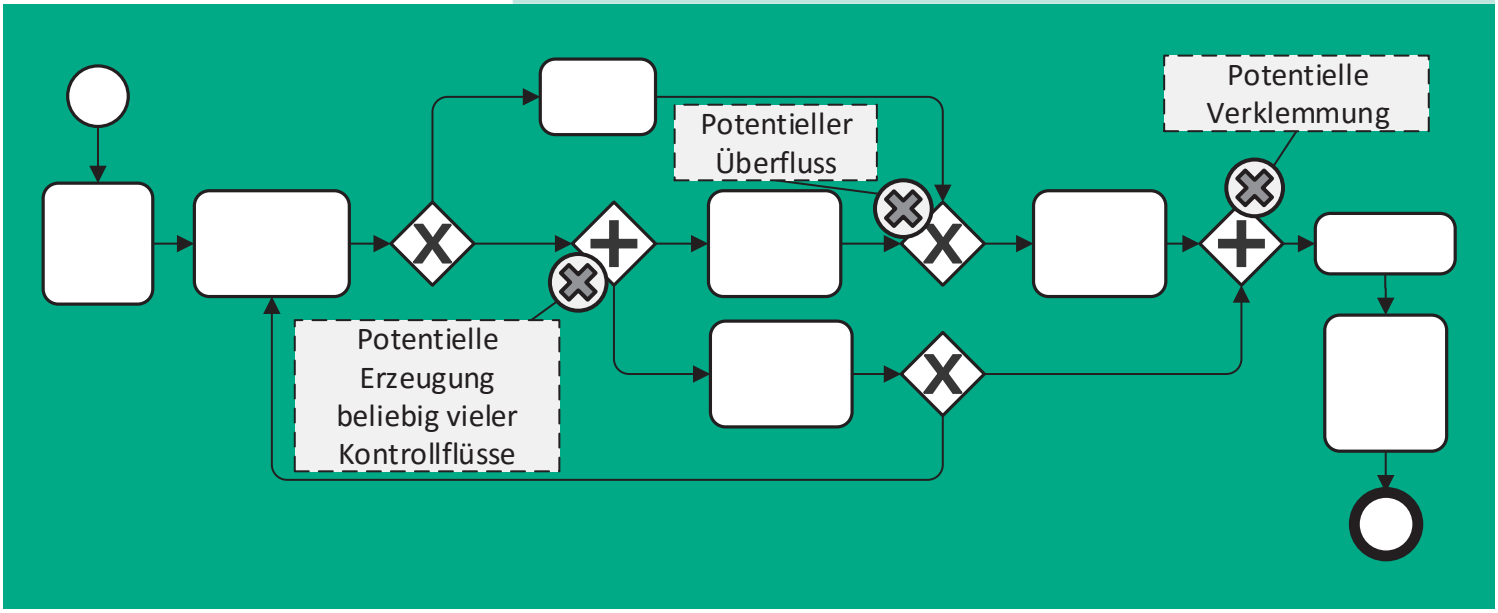
FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Thomas M. Prinz

Entwicklung von kontrollflussbasierten Methoden und Techniken für einen benutzerfreundlichen Entwurf von sicheren Geschäftsprozessen



DISSERTATIONSSCHRIFT

Entwicklung von kontrollflussbasierten Methoden und Techniken für einen benutzerfreundlichen Entwurf von sicheren Geschäftsprozessen

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena

von

Diplom-Informatiker Thomas Martin Prinz

geboren am 15. Oktober 1986 in Gera

Gutachter

1: Prof. Dr. habil. Wolfram Amme

2: Prof. Dr. habil. Stefanie Rinderle-Ma

Tag der öffentlichen Verteidigung:

27. Oktober 2017

Für Cassandra, Lorelai und Eleen.
Danke, dass es euch gibt.

Schaue ich bisher auf mein Leben, dann sind es nicht die großen Entscheidungen, die es prägen. Es sind die kleinen, zunächst unscheinbaren Ereignisse, die das tun: Wie ich meine liebe Frau Cassandra kennengelernt habe; wie ich in die Forschung über Wolfram geraten bin; wie ich wissenschaftlicher Mitarbeiter durch Wilhelm Rossak wurde; um nur einige solcher wichtigen Momente zu erwähnen. Es ist vergleichbar mit winzigen Ursachen, die zu großen Ereignissen in der Zukunft heranwachsen: Die Geburt meiner geliebten Kinder Lorelai und Eleen; und im vorliegenden Fall, meine fertige Dissertation – ein richtiger *Schmetterlingseffekt* (Edward N. Lorenz).

Natürlich wurde dieser Schmetterlingseffekt durch wichtige Menschen gelenkt. So sei an dieser Stelle meinen Eltern Andrea und Volker und meinen Geschwistern Janika und Jan gedankt. Auch sei Wolfram als mein Doktor*vater* an dieser Stelle herzlich gedankt für seinen Rat und seine Geduld in allen Phasen meiner Dissertation.

Natürlich möchte ich mich auch bei meiner Frau Cassandra bedanken. Sie ist mein roter Faden im Leben. Sie bringt mich zum Lachen und Weinen, Nachdenken und Aufregen. Ohne sie hätte ich nicht meine geliebten Kinder.

Meinen Kollegen und Freunden am Institut für Informatik, bei ULe und natürlich im Privaten sei auch herzlichst gedankt. So manches interessante Gespräch, so manche hitzige Diskussion sind notwendig, Wogen aus Wellen zu glätten, um den Blick auf die Einfachheit der Dinge zu bekommen. Der Horizont ist schwer zu erkennen, wenn das Meer aufgewühlt ist. An dieser Stelle möchte ich besonders Wilhelm, Arndt, Kai, Axel, Heiko, René, Steffen, Stefan, Johannes, Sebastian, Manuela, Martin, Sören, Christian, Volkmar, Conny, Thomas, Paul, Katharina, Marianne, Rolf Steyer, Anja, Linda, Anna, Jan, Christoph, Marcel, Raphael, Katrin, Hannes, Jasmin, Elisa, Torsten, Magdalena, Max, Mario, Sebastian, Hans, Bianca und Kezia danken. Wenn ich jemanden vergessen habe, so ist es nicht mit Absicht. Er/sie soll sich ebenfalls gedankt fühlen.

Zum Schluss möchte ich noch meinen Dank den Gutachtern, Stefanie Rinderle-Ma und Wolfram Amme, aussprechen, die bereitwillig zugestimmt haben, diese Dissertationsschrift zu bewerten.

Abstract

Processes are omnipresent: Whether in companies, hospitals, or in the daily life of each person, they determine our lives. In most cases, the consideration, detection, and record of such processes has a benefit, e.g. companies can reach their business goals and human lives can be saved. If they are formalized, they are called *workflows*.

Processes are formalized in to a workflow mostly in a visual manner as a graph consisting of nodes and edges. Different kinds of nodes determine different semantics, i.e. some nodes describe tasks and other nodes describe decisions and parallelisms.

The operation from the consideration of a process to its formalization in to a workflow is difficult. Therefore, it is done by trained experts. However, there are still faults in the workflows. Such faults are very difficult to identify in complex workflows and lead to undesired process behavior. An undesired behavior can result in the non-achievement of business goals and even the loss of human lives. Briefly: Wrong behaviors are expensive and dangerous; practical applications should avoid them. The present work considers such undesired behaviors and uncovers their *causes before* they lead to damages.

The notion of *soundness* forms the basis for the detection of the causes of wrong behaviors. Soundness considers the control flow only and excludes the faults *deadlocks* and *abundances* (lacks of synchronization). To reveal the *causes* for both faults, this work introduces partial analyses which allow for considerations from different points of a workflow. This makes it possible to detect *potential* errors behind others.

Based on partial analyses, this work presents two new techniques: The first technique detects the causes of all immediate occurring potential deadlocks; the second technique derives the causes of all potential abundances. Both techniques have in common that they have a cubic asymptotic runtime complexity for realistic processes depending on the number of edges in the workflow. Furthermore, they provide detailed diagnostic information, which can be used beneficially for the visualization and description of the detected errors. Both properties of these techniques certificate them for their application as supporting analyses, e.g. for their execution in each step of construction.

As proof of efficiency and quality, this work introduces the tool *Mojo* which implements the presented techniques. This tool is used afterwards to evaluate the algorithms with a benchmark of over 1,300 practical relevant workflows. Furthermore, those techniques will be compared with two approaches of the state-of-the-art.

Workflows, Deadlocks, Abundances, Verification, Compiler

Zusammenfassung

Oft kann die Untersuchung, Erkennung und Niederschrift von impliziten Abläufen von großem Nutzen sein. Dadurch können Unternehmen ihre Ziele erreichen und sogar Menschenleben gerettet werden. Eine formale Niederschrift eines solchen Ablaufs heißt *Arbeitsprozess* (Workflow).

Arbeitsprozesse werden sehr häufig in visuellen Notationen als Graphen aus Knoten und Kanten formalisiert. Dabei gibt es verschiedene Arten von Knoten mit unterschiedlichen Semantiken. Durch diese können einzelne Aufgaben beschrieben sowie Entscheidungen und Parallelitäten modelliert werden.

Der Vorgang von der Untersuchung eines Ablaufs bis zur Formalisierung zu einem Arbeitsprozess ist schwierig und wird deswegen von geschulten Experten übernommen. Leider treten dabei dennoch häufig Fehler auf. Diese Fehler sind gerade in umfangreichen Arbeitsprozessen sehr schwierig zu identifizieren und führen zu einem inkorrekten Prozessverhalten. Ein inkorrektes Verhalten kann zum Nichterreichen von Unternehmenszielen und sogar im schlimmsten Fall zum Verlust von Menschenleben führen. Kurzum: Fehlverhalten sind teuer und gefährlich! Die vorliegende Arbeit untersucht solches Fehlverhalten und deckt deren *Ursachen* auf, *bevor* sie Schaden verursachen können.

Als Grundlage zur Feststellung einer Ursache eines fehlerhaften Verhaltens dient der Korrektheitsbegriff (Soundness). Er schließt das Auftreten von *Verklemmungen* und *Überflüssen* (Lacks of Synchronization) aufgrund des Kontrollflusses aus. Um die *Ursachen* dieser Fehler*wirkungen* aufzudecken, wird eine partielle Analyse eingeführt. Diese ermöglicht die Untersuchung ausgehend von verschiedenen Punkten innerhalb eines Arbeitsprozesses. Dadurch können *potentielle* Fehlerursachen hinter anderen Fehlern entdeckt werden.

Darauf aufbauend präsentiert diese Arbeit zwei neue Techniken: Die erste Technik findet die Ursachen aller unmittelbar auftretenden, potentiellen Verklemmungen; die zweite ermittelt die Ursachen aller potentiellen Überflüsse. Beide Techniken besitzen eine *kubisch* asymptotische Laufzeitkomplexität für realitätsnahe Prozesse hinsichtlich der Anzahl der Kanten. Zudem liefern sie noch detaillierte diagnostische Informationen, die hervorragend zur Visualisierung und Erklärung der gefundenen Fehlerursachen dienen können. Beide Eigenschaften der Algorithmen zertifizieren sie für den Einsatz als unterstützendes Analyseverfahren, wie deren Anwendung in jedem Konstruktionsschritt.

Als Nachweis der Effizienz und Qualität dieser Techniken wird zudem ein Werkzeug, *Mojo*, eingeführt, welches diese implementiert. Dieses Werkzeug wird außerdem für eine Evaluation mit über 1.300 Arbeitsprozessen aus der Praxis genutzt und mit zwei anderen Techniken des aktuellen Stands der Forschung verglichen.

Arbeitsprozesse, Verklemmungen, Überflüsse, Verifikation, Übersetzer

Inhaltsverzeichnis

Kapitel 1

Einleitung und Motivation

Abläufe durchdringen Behörden, Unternehmen, Krankenhäuser und selbst den Alltag eines jeden Menschen. Kurzum: Abläufe sind überall. In vielen Fällen ist es sinnvoll solche Abläufe zu untersuchen, zu erkennen und aufzuschreiben. Dadurch können Behördenvorgänge beschleunigt, Unternehmensziele gesichert und sogar Patientenleben in Krankenhäusern gerettet werden. Werden Abläufe formal modelliert, so wird von *Arbeitsprozessen* (Workflows) gesprochen. In diesen Arbeitsprozessen werden Aufgaben in einer geordneten Reihenfolge für die Erreichung eines Ziels definiert und in Folge dessen der Ablauf formalisiert [15, S. 5][40, S. 87 f.].

In dieser Arbeit wird das folgende Beispiel eines (vereinfachten) Ablaufs bei der Behandlung eines Patienten in einem Krankenhaus verwendet: Ein Patient kommt mit gesundheitlichen Beschwerden in ein Krankenhaus. Als erstes untersucht ihn ein Arzt. Aufgrund dieser Untersuchung entscheidet sich der Arzt, ob es sich um eine einfache Behandlung handelt oder nicht. Ist sie einfach, so führt er sie sofort durch. Anderenfalls informiert er den Oberarzt und beginnt bereits gleichzeitig mit einer Behandlung, um die Beschwerden schnell zu lindern. Der Oberarzt begutachtet in der Zwischenzeit die Behandlungsmaßnahmen und das Beschwerdebild. Hält er andere Maßnahmen für notwendig, so gibt er diese Informationen an den behandelnden Arzt weiter, der daraufhin den Patienten erneut untersucht. Im einfachen Fall bestätigt der Oberarzt die Behandlung seines Kollegen. In einer abschließenden Behandlung wird der Patient nochmals untersucht und vom Arzt die Entlassungspapiere unterzeichnet. Mit diesen verlässt der Patient das Krankenhaus.

Dieser Ablauf könnte von einem Modellierer in einen Arbeitsprozess der Sprache Business Process Model and Notation (BPMN) 2.0 [52] überführt werden, welcher in Abbildung 1.1 zu sehen ist. Dabei ist der Prozess üblicherweise ein Graph mit Kanten und Knoten, wobei die Knoten des Graphen unterschiedliche Semantiken aufweisen. Es gibt Start- und Endereignisse (Kreise mit dünner und dicker Linie), Aufgaben (Rechtecke) sowie Entscheidungs- (Diamanten mit Kreuz) und Parallelitätsknoten (Diamanten mit Plus). Außerdem werden in der Abbildung sowohl Rollen

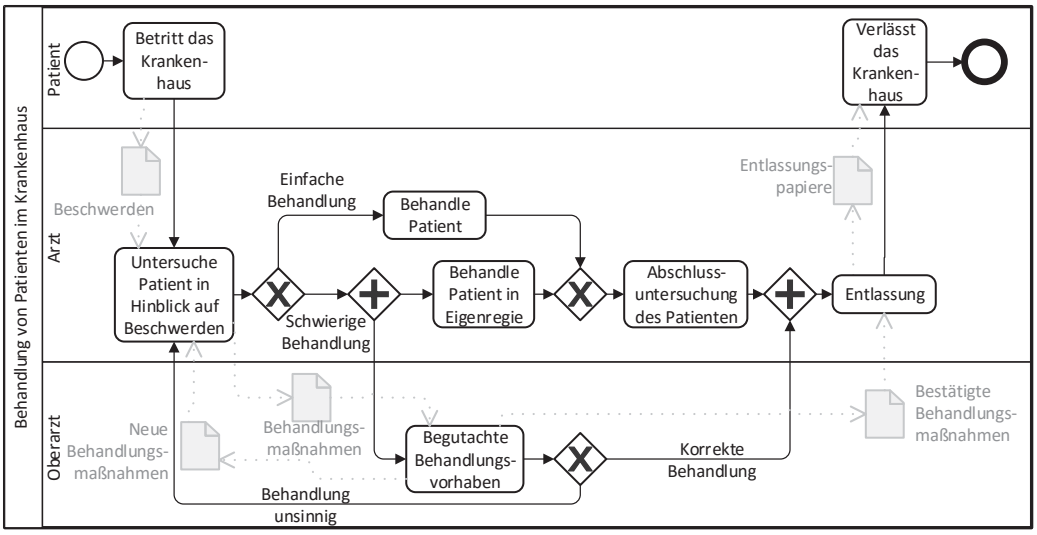


Abbildung 1.1: Ein Arbeitsprozess in BPMN

(Bahnen mit den Bezeichnungen Patient, Arzt und Oberarzt) als auch Informationen (Dokumente) dargestellt.

Der Vorgang von der Untersuchung eines Ablaufs bis zur Formalisierung zu einem Arbeitsprozess ist schwierig und wird deswegen von geschulten Experten übernommen. Dennoch schleichen sich dabei häufig Fehler in den Prozess ein, welche gerade in komplexeren Arbeitsprozessen sehr schwierig zu finden sind und bei Ausführung zu einem inkorrekten Prozessverhalten führen. Ein inkorrektes Verhalten eines Prozesses kann zu falsch ausgestellten behördlichen Urkunden, zum Nichterreichen eines Unternehmensziels, wenn nicht sogar zum Verlust eines Menschenlebens führen. Das heißt, Fehlverhalten sind teuer und gefährlich, so dass diese besser vermieden werden sollten [18]. Die vorliegende Arbeit untersucht solches Fehlverhalten inkorrekter Prozesse und deckt diese auf, *bevor* sie Schaden verursachen können.

In der Literatur werden unterschiedliche semantische Fehler von Prozessen untersucht und dementsprechend die Korrektheit des Prozessverhaltens auch verschieden definiert. Der in dieser Arbeit verwendete Begriff der Korrektheit wird klassisch über die Abwesenheit sogenannter *Verklemmungen* und *Überflüsse* festgelegt [68][78]. Bei einem Überfluss gelangt der Prozess in eine Situation, in dem ein und dieselbe Aufgabe unerwünscht mehrfach direkt hintereinander (überflüssigerweise) ausgeführt wird. Bei einer Verklemmung hingegen blockiert die Ausführung ab einem Punkt im Prozess und findet daher kein ordnungsgemäßes Ende.

Der Prozess aus Abbildung 1.1 beinhaltet Fehlverhalten in Form von Überflüssen sowie Verklemmungen und ist damit inkorrekt vom Prozessentwickler modelliert. Beispielsweise kann eine Verklemmung auftreten, sobald die Behandlung eines Patienten einfach ist. Wie Abbildung 1.2 an einer vereinfachten Version des Prozesses

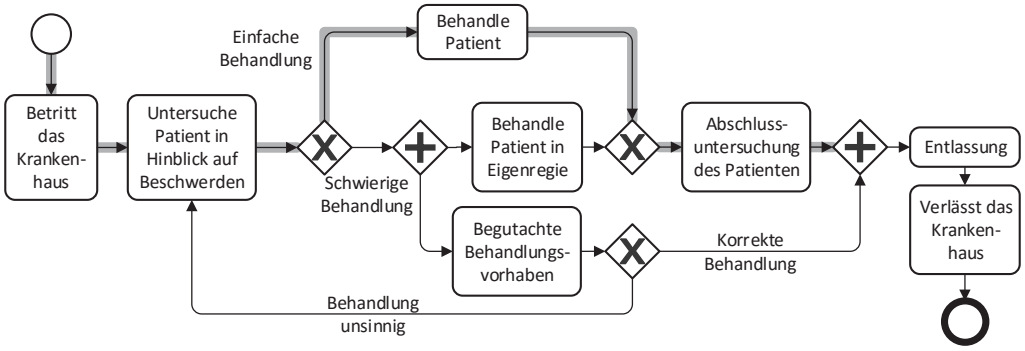


Abbildung 1.2: Eine erreichbare Verklemmung (grauer Pfad) im Beispielprozess

zeigt, kann eine Ausführung (durch dunkelgraue Kanten illustriert) vor dem rechten Diamant mit Pluszeichen zum Erliegen kommen. Dies geschieht, da dieser Knoten parallele Ausführungen synchronisieren möchte. Dies ist jedoch offenbar nicht gegeben, wodurch der Knoten unausgeführt bleibt und die Ausführung verklemmt.

Zur Vermeidung solcher Fehlverhalten in Arbeitsprozessen wurden bereits einige Techniken zur Bestimmung von Verklemmungen und Überflüssen entwickelt. Diese untersuchen in der Regel alle möglichen Situationen, in welche die Ausführung eines Prozesses möglicherweise gerät. Da es jedoch potentiell beliebig viele solcher Situationen geben kann, können viele dieser Techniken jedoch mitunter zu keinem Ergebnis kommen [18].

Viel schwerwiegender jedoch ist die ausschließliche Betrachtung der *Wirkung* [32, S. 30][50, S. 2] anstelle der *Ursache* [32, S. 30][50, S. 2] von Fehlverhalten in fast allen Techniken. Das heißt, es werden nur die Situationen gesucht, in welchen die Ausführung verklemmt oder einen Überfluss aufweist, anstelle die Ursache zu bestimmen, *warum* dies passiert. Es wird jedoch die Ursache benötigt, um den Modellierungsfehler zu lokalisieren und zu beseitigen. Solch eine Fehlerursache aus der Wirkung abzuleiten, erweist sich aber bereits in der Softwarequalitätssicherung als schwierig, denn der Fehler liegt bereits im Prozessmodell, äußert sich aber erst zur Ausführung.

Die ausschließliche Untersuchung der Wirkungen von Fehlern führt außerdem dazu, dass nicht jedes Fehlverhalten im Prozess untersucht werden kann; beispielsweise, weil ein Fehlverhalten – wie die Verklemmung – das Erreichen eines anderen Fehlverhaltens verhindert und somit *blockiert* [23]. Andererseits kann ein Fehlverhalten auch einen anderen Fehler aufheben (*maskieren* [32, S. 31]). Somit fällt mitunter die Wirkung eines Fehlers gar nicht auf. Manchmal fällt ein Fehler auch nicht auf, weil sich das Fehlverhalten anders äußert als erwartet. So kann aus einer Ursache eines Überflusses beispielsweise eine Verklemmung resultieren. Es wird sich also mitunter auch in der Ursache eines Fehlverhaltens *getäuscht* [57].

Wenn aber nun ein Werkzeug zur Überprüfung der Korrektheit eines Prozesses lediglich Fehler*wirkungen*, wie Verklemmungen und Überflüsse, anstelle deren Ursachen findet: *Woher soll ein (vielleicht unerfahrener) Prozessentwickler wissen,* ***was*** *die Ursache eines Fehlverhaltens ist,* ***wo*** *er sie finden kann und* ***ob*** *die gefundene Fehlerwirkung überhaupt eine adäquate Ursache besitzt? Im aktuellen Stand der Forschung konnte keine Technik gefunden werden, welche diese Fragen für beliebige Prozesse aus Sicht eines Anwenders vollständig und in annehmbarer Zeit beantwortet.*

Die vorliegende Arbeit beantwortet diese Fragen und untersucht dafür erstmals die *Ursachen* von Verklemmungen und Überflüssen in Prozessen. Weiterhin werden aus diesen Untersuchungen effiziente Algorithmen abgeleitet, welche die notwendigen Ursachen von Verklemmungen und Überflüssen mit detaillierten diagnostischen Informationen finden.

Aufbau dieser Arbeit. Diese Arbeit untergliedert sich in 10 Kapitel. Im Anschluss an diese Einleitung findet in Kapitel 2 die Einführung grundlegender Begriffe statt. Darauf aufbauend folgt eine ausführliche Betrachtung des Stands der Forschung (Kapitel 3). Diese Untersuchung mündet in Kapitel 4 in die Aufstellung des wissenschaftlichen Beitrags und der Thesen dieser Arbeit. Danach folgt die Einführung und Herleitung einer partiellen Analyse von Prozessen in Kapitel 5. Kapitel 6 untersucht darauf aufbauend die Ursachen von Verklemmungen. Im daran anschließenden Kapitel 7 werden die Ursachen von Überflüssen bestimmt. In Kapitel 8 folgt eine kurze Einführung in unser Analysewerkzeug *Mojo*. Dieses wird in der anschließenden Evaluation (Kapitel 9) genutzt. Im direkten Nachgang endet diese Arbeit in Kapitel 10 mit einer Diskussion der Ergebnisse, einer Zusammenfassung und dem Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlegende Begriffsbildung

In diesem Kapitel werden grundlegende Begriffe und Notationen definiert, die für das Verständnis der weiteren Arbeit von Belang sind.

2.1 Multimengen

Mengen beinhalten verschiedene Elemente, wie Zahlen, Kanten oder ähnliches. Dabei unterscheidet eine Menge nicht, ob Elemente mehrmals in ihr enthalten sind. Für die vorliegende Arbeit ist es jedoch an vielen Stellen notwendig zu wissen, *wie oft* ein Element in einer Menge ist. Dafür werden in der Literatur *Multimengen* verwendet. Die folgende Definition von Multimengen basiert auf Reisig [64].

Definition 2.1 (Multimenge)**.**

Eine *Multimenge* S über eine Menge M ist eine totale Funktion von M auf die natürlichen Zahlen $\mathbb{N}$ ($S{:}\, M \mapsto \mathbb{N}$). S weist jedem Element aus M eine Zahl zu, welche der Anzahl des Vorkommens des Elements in S entspricht. Wir schreiben $[\![m_0, \ldots, m_k]\!], k \geq 0$, $\{m_0, \ldots, m_k\} \subseteq M$.

Beispiel 2.2.

Für eine Menge $M = \{a, b, c, d\}$ ist $S = [\![a, a, b, b, b, d]\!]$ (oder auch $S = [\![a^2, b^3, d]\!]$) eine Multimenge. Die Funktion S gibt uns zudem für jedes Element x aus M die Häufigkeit seines Vorkommens zurück. Es gilt $S(a) = 2$, $S(b) = 3$, $S(c) = 0$ und $S(d) = 1$.

Um Multimengen gewohnt wie Mengen in dieser Arbeit nutzen zu können, führen wir weitere Notationen für und Operationen auf Multimengen ein. Dafür nehmen wir für den Rest dieses Abschnitts eine Multimenge S über eine Menge M an. Die folgenden Definitionen sind angelehnt an Rosen [67, S. 138] und Reisig [64], weichen jedoch beispielsweise bei der Vereinigung ab.

Die *Grundmenge* S^G beinhaltet alle Elemente der Menge M, die mindestens einmal in S enthalten sind:

$$S^G = \{m : m \in M \wedge S(m) \geq 1\} \tag{2.1}$$

Damit ist S^G die Korrespondenz von S zu einer gewöhnlichen Menge. Analog zu Mengen ist ein Element $m \in M$ ein *Element* von S (geschrieben $m \in S$), wenn m in S mindestens einmal enthalten ist bzw. $m \in S^G$ gilt:

$$m \in S \iff S(m) \geq 1 \iff m \in S^G \tag{2.2}$$

Eine Multimenge S ist *Teilmenge* einer Multimenge O, wenn jedes Element m der Grundmenge von S, $m \in S^G$, mindestens so häufig auch in O vorkommt:

$$S \subseteq O \iff \underset{m \in S^G}{\forall} S(m) \leq O(m) \tag{2.3}$$

In Analogie zu gewöhnlichen Mengen enthält die *Schnittmenge* C zweier Multimengen S und O (geschrieben $C = S \cap O$) die minimale Anzahl der Elemente, die in beiden Multimengen S und O vorhanden sind:

$$S \cap O = C \iff \underset{m \in S^G}{\forall} C(m) = \min(S(m), O(m)) \tag{2.4}$$

Die *Vereinigung* V zweier Multimengen S und O (geschrieben $V = S \cup O$) wiederum enthält die Summe aller Elemente beider Multimengen:

$$S \cup O = V \iff \underset{m \in (S^G \cup O^G)}{\forall} V(m) = S(m) + O(m) \tag{2.5}$$

Beim *Komplement* K zweier Multimengen S und O (geschrieben $K = S \setminus O$) wird die Anzahl derjenigen Elemente reduziert, die in O enthalten sind. Minimal sind natürlich 0 Elemente eines Elements enthalten:

$$S \setminus O = K \iff \underset{m \in S^G}{\forall} K(m) = \max\big(S(m) - O(m), 0\big) \tag{2.6}$$

Im weiteren Verlauf dieser Arbeit werden die gerade eingeführten Notationen auch in Kombination mit gewöhnlichen Mengen verwendet. In diesen Fällen gehen wir davon aus, dass die gewöhnliche Menge G der Multimenge S entspricht mit:

$$S^G = G \wedge \underset{m \in G}{\forall} S(m) = 1 \tag{2.7}$$

Damit haben wir alle notwendigen Notationen und Operationen bezüglich Multimengen eingeführt. Diese werden mit einem abschließenden Beispiel nochmals in ihrer Gesamtheit aufgegriffen.

Beispiel 2.3.

Zwei Multimengen $S_1 = [\![a, a, b, b, b, c]\!] = [\![a^2, b^3, c]\!]$ und $S_2 = [\![a, a, a, b, b, b, b, b, c, d, d]\!] = [\![a^3, b^5, c, d^2]\!]$ über die Menge $M = \{a, b, c, d, e\}$ seien gegeben. Die Grundmengen von S_1 und S_2 sind dadurch

$$S_1^G = \{a, b, c\} \quad \text{und} \quad S_2^G = \{a, b, c, d\}$$

und für das Element a aus M gilt: $a \in S_1$ und $a \in S_2$.

Die Multimenge S_1 ist eine Teilmenge von S_2 ($S_1 \subseteq S_2$), da sowohl a zweifach als auch b dreifach und c einfach in S_2 vorkommen. So entspricht auch die Schnittmenge von S_1 und S_2 genau der Multimenge S_1 ($S_1 = S_1 \cap S_2$). Die Vereinigung hingegen beinhaltet alle Elemente aus S_1 und S_2:

$$S_1 \cup S_2 = [\![a, a, a, a, a, b, b, b, b, b, b, b, b, c, c, d, d]\!] = [\![a^5, b^8, c^2, d^2]\!]$$

Die Multimenge S_2 ohne S_1 ($S_2 \setminus S_1$) entspricht der Multimenge

$$[\![a, b, b, d, d]\!] = [\![a, b^2, d^2]\!] = S_2 \setminus S_1$$

Wiederum entspricht die Multimenge S_1 ohne S_2 der leeren Menge.

2.2 Graphen und Wege

Ein wichtiges Konzept in der Informatik und im Speziellen im Übersetzerbau sind *Graphen*, insbesondere *gerichtete* Graphen. Für Graphen gibt es viele unterschiedliche Arten der Definition. Die folgende Definition greift die Idee von Chartrand und Zhang [9, S. 432 ff.] auf, die Bestandteile des Graphen als Funktionen zu definieren. Ansonsten folgt die Definition der klassischen Definition eines gerichteten Graphen, wie sie zum Beispiel auch von Rosen [67, S. 641] und Cormen et al. [13, S. 1179] aufgestellt wird:

Definition 2.4 (Gerichteter Graph)**.**

Ein *gerichteter Graph* oder *Digraph* G besteht aus einer Menge $N = N(G)$, genannt *Knoten* von G, und einer Menge $E = E(G)$ von geordneten Paaren, $E \subseteq N \times N$, genannt *Kanten* von G. Wir schreiben $G = (N, E)$.

Im Folgenden werden weitere Notationen für Digraphen eingeführt, die im Verlaufe der Arbeit verwendet werden. Dafür gehen wir von einem gerichteten Graphen $G = (N, E)$ aus.

Für eine Kante $e = (s, t) \in E$ bezeichnen wir den Knoten s als *Quelle* und wir schreiben $src(e)$ (vom Englischen *Source*). Zum Knoten t sagen wir *Ziel* und schreiben $tgt(e)$ (vom Englischen *Target*):

$$src((s,t)) = s \iff (s,t) \in E \quad \text{und} \quad tgt((s,t)) = t \iff (s,t) \in E \tag{2.8}$$

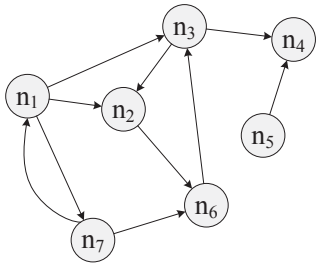


Abbildung 2.1: Ein Digraph

Häufig in der Theorie der Petrinetze [54] verwendete Notationen für Digraphen sind die für einen Knoten $n \in N$ *eingehenden* und *ausgehenden* Kanten. Die Menge der eingehenden Kanten beschreibt $\triangleright n$ und umfasst für n alle Kanten, die n als Ziel haben ($\{(p,n) = e \in E\}$). Analog wird die Menge der ausgehenden Kanten $n\triangleleft$ durch die Menge der Kanten beschrieben, die n als Quelle besitzen ($\{(n,s) = e \in E\}$):

$$\triangleright n = \{(p,n) = e \in E\} \quad \text{und} \quad n\triangleleft = \{(n,s) = e \in E\} \tag{2.9}$$

Beispiel 2.5.

Abbildung 2.1 zeigt einen gerichteten Graphen $G = (N,E)$ mit den Knoten

$$N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$

und den Kanten

$$E = \{(n_1,n_2),(n_1,n_3),(n_1,n_7),(n_2,n_6),(n_3,n_2),\\ (n_3,n_4),(n_5,n_4),(n_6,n_3),(n_7,n_1),(n_7,n_6)\}$$

Für diesen Graphen ist die Menge der eingehenden Kanten des Knotens n_3 gleich $\{(n_1,n_3),\ (n_6,n_3)\} = \triangleright n_3$. Die ausgehenden Kanten von n_3 bilden die Menge $n_3\triangleleft = \{(n_3,n_4),\ (n_3,n_2)\}$. Für die Kante (n_3,n_4) ist die Quelle $n_3 = src((n_3,n_4))$ und das Ziel $n_4 = tgt((n_3,n_4))$.

In der Visualisierung von Graphen lässt sich sehr gut erkennen, dass Knoten miteinander verbunden sind. Diese Verbundenheit beschreiben wir durch *Wege*.

Definition 2.6 (Weg).

In einem Digraphen $G = (N,E)$ nennen wir eine Sequenz $W = (e_0,\ldots,e_m)$, $m \geq 0$, von Kanten aus E, $\{e_0,\ldots,e_m\} \subseteq E$, einen *Weg* W, wenn jedes Ziel einer Kante der nächsten Kante Quelle entspricht:

$$\underset{0\leq i<m}{\forall} tgt(e_i) = src(e_{i+1}) \tag{2.10}$$

Im Gegensatz zur häufigsten Definition von Wegen über Knoten (zum Beispiel von Cormen et al. [13, S. 1180]) verwenden wir Wege über Kanten, wie auch Bossert

und Breitbach [8, S. 246]. Diese sind in Bezug auf den im Weg benutzten Kanten eindeutig und bedürfen keiner expliziten Nennung der *besuchten* Knoten.

Wir verwenden im Folgenden für eine in einem Weg $W = (e_0, \ldots, e_m)$, $m \geq 0$, enthaltenen Kante e die Elementnotation $e \in W$.

Ein wichtiger Begriff in Bezug auf Wege ist die *Schleife*, manchmal auch *Zyklus* genannt. Jeder Weg W, in dem eine Kante doppelt vorkommt, enthält eine Schleife. So sind nicht alle Kanten von W paarweise verschieden. Für einen Graphen gilt dann, dass er eine Schleife enthält, wenn es mindestens einen Weg gibt, der eine Schleife hat. Andernfalls wird ein solcher Graph *zyklenfrei* oder auch *azyklisch* genannt.

Im Rest der Arbeit benutzen wir die Notation $W_{a \to b}$ für einen Weg W von einer Kante $a \in E$ zu einer Kante $b \in E$. Analog schreiben wir $\mathcal{W}_{a \to b}$ für die Menge aller Wege von a nach b.

Abschließend ist die *Länge* eines Wegs $W = (e_0, \ldots, e_m)$, $m \geq 0$, durch die Anzahl m der in der Sequenz enthaltenen Kanten gegeben, geschrieben $|W| = m$.

Beispiel 2.7.

Für einen Weg

$$W_{(n_1,n_2)\to(n_3,n_4)} = \big((n_1,n_2),(n_2,n_6),(n_6,n_3),(n_3,n_4)\big)$$

aus Abbildung 2.1 ist die Kante (n_2, n_6) eine Wegkante und somit $(n_2, n_6) \in W$. Die Länge des Weges ist 4. Die Menge aller Wege $\mathcal{W}_{(n_1,n_2)\to(n_3,n_4)}$ von (n_1, n_2) nach (n_3, n_4) umfasst beliebig viele Wege aufgrund der Schleife $\big((n_3,n_2),(n_2,n_6),(n_6,n_3),(n_3,n_2)\big)$.

Wichtige Graphen sind die *Kontrollflussgraphen*. Sie sind gewöhnliche Digraphen mit nur genau einem Start- und Endknoten (bspw. [92, S.57]).

Definition 2.8 (Kontrollflussgraph)**.**

Ein Kontrollflussgraph $CFG = (N, E)$ ist ein Digraph (N, E) mit den folgenden Eigenschaften:

1. Es gibt genau einen *Startknoten*

$$|\{n \in N \colon \underset{e \in E}{\forall} n \neq tgt(e)\}| = |\{n_{Start}\}| = 1 \tag{2.11}$$

   mit genau einer ausgehenden Kante ($|n_{Start}\triangleleft| = |\{e_{Start}\}| = 1$).

2. Es gibt genau einen *Endknoten*

$$|\{n \in N \colon \underset{e \in E}{\forall} n \neq src(e)\}| = |\{n_{End}\}| = 1 \tag{2.12}$$

   mit genau einer eingehenden Kante ($|\triangleright n_{End}| = |\{e_{End}\}| = 1$).

3. Jede Kante $e \in E$ liegt auf einem Weg von e_{Start} zu e_{End}.

$$\underset{e \in E}{\forall} \quad \underset{W \in \mathcal{W}_{e_{Start} \to e_{End}}}{\exists} \quad e \in W \tag{2.13}$$

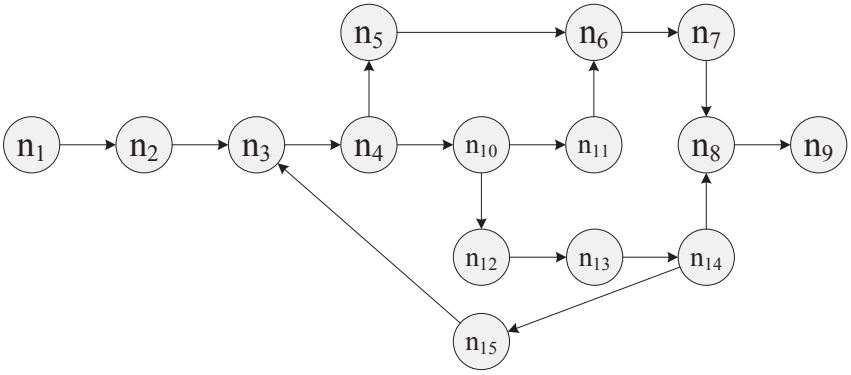


Abbildung 2.2: Ein Kontrollflussgraph

Wir bezeichnen die Kante e_{Start} im Folgenden als *Startkante* und die Kante e_{End} als *Endkante* von CFG und analog den Knoten n_{Start} als *Start-* und den Knoten n_{End} als *Endknoten*.

Beispiel 2.9.

Der in Abbildung 2.2 abgebildete Digraph ist ein Kontrollflussgraph. Er ähnelt von der Struktur unserem Beispielprozess (Abbildung 1.1, S. 2) aus Kapitel 1 und besitzt als Startknoten $n_{Start} = n_1$ mit der ausgehenden Startkante $e_{Start} = (n_1, n_2)$ und als Endknoten $n_{End} = n_9$ mit der eingehenden Endkante $e_{End} = (n_8, n_9)$. Offensichtlich liegt auch jede Kante auf einem Weg zwischen der Start- und Endkante.

Wichtige Relationen auf Kontrollflussgraphen sind die Dominator- und Postdominatorbeziehungen zwischen zwei Kanten [5, S. 20][49][62]:

Definition 2.10 (Postdominatorrelation)**.**

Eine Kante $a \in E$ *postdominiert* eine Kante $b \in E$ in einem Kontrollflussgraphen $CFG = (N, E)$, wenn auf allen Wegen von b zur Endkante e_{End} die Kante a liegt. Wir schreiben für die Postdominatorrelation *pdom*. Für den Fall, dass $(a, b) \in pdom$ gilt, notieren wir auch $a\ pdom\ b$.

$$a\ pdom\ b \iff \underset{W \in \mathcal{W}_{b \to e_{End}}}{\forall} a \in W \tag{2.14}$$

a postdominiert b *echt*, wenn $a\ pdom\ b$ und $a \neq b$ gilt. Außerdem postdominiert a die Kante b *unmittelbar*, wenn a die Kante b echt postdominiert und alle anderen echten Postdominatoren von b auch echte Postdominatoren von a sind.

$$a\ pdom\ b \text{ unmittelbar} \iff (a\ pdom\ b \text{ echt}) \wedge \left(\underset{c \in E}{\forall} c\ pdom\ b \text{ echt} \longrightarrow c\ pdom\ a \text{ echt}\right) \tag{2.15}$$

Die Dominatorrelation ist analog definiert, nur dass eine Kante $a \in E$ eine Kante $b \in E$ *dominiert*, wenn auf allen Wegen von der Startkante e_{Start} zur Kante b die Kante a liegt.

2.3 Workflowgraphen

Prozesse werden häufig in Form von skriptbasierten (wie der Business Process Execution Language (BPEL) [35]) oder graphischen Entwicklungssprachen (wie BPMN [52] oder der Yet Another Workflow Language (YAWL) [81]) angegeben. Diese Darstellungen von Prozessen enthalten zu viele Informationen, die für die Findung von Kontrollflussfehlern unrelevant sind. Aus diesem Grund wird bei Verfahren der Fehlersuche entweder auf Workflow*netzen* [77] oder auf *Workflowgraphen* [68] zurückgegriffen. Beide Konzepte sind gleichartig dafür geeignet, Kontrollflussanalysen durchzuführen, denn eine Überführung von Workflownetzen in Workflowgraphen und von Workflowgraphen zu Workflownetzen ist trivial [20].

In dieser Arbeit benutzen wir die von Sadiq und Orlowska [68] eingeführten Workflowgraphen, da sie spezielle Kontrollflussgraphen darstellen und somit zur Kontrollfluss-basierten Analyse von Prozessen passen. Da Workflowgraphen keinem wirklich festen Modell wie etwa die Workflownetze entsprechen, ist außerdem die Einführung von weiteren Arten von Kontrollflusselementen intuitiver [20].

Im Rahmen dieser Arbeit wird eine Definition von Workflowgraphen genutzt, die einen Workflowgraphen auf einen Kontrollflussgraphen zurückführt, für den für jeden seiner Knoten eine spezielle Beschriftung hinzugefügt wird.

Definition 2.11 (Workflowgraph)**.**

Ein *Workflowgraph* WFG umfasst einen Kontrollflussgraphen (N, E) und eine linkstotale Abbildung

$$l: N \mapsto \{Start, End, Task, Split, Merge, Fork, Join\} \tag{2.16}$$

die jedem Knoten eine *Beschriftung* hinzufügt und damit die Art und später auch die Semantik des Knotens definiert. Wir schreiben $WFG = (N, E, l)$.

Die Menge der Knoten N ist unterteilt in disjunkte Teilmengen:

$$N = \{n_{Start}, n_{End}\} \cup N_{Task} \cup N_{Split} \cup N_{Merge} \cup N_{Fork} \cup N_{Join} \tag{2.17}$$

Alle Knoten der gleichen Teilmenge besitzen die selbe Beschriftung:

$$l(n_{Start}) = Start, \text{ der } Startknoten \tag{2.18}$$

$$l(n_{End}) = End, \text{ der } Endknoten \tag{2.19}$$

$$\underset{n \in N_{Task}}{\forall} l(n) = Task, \text{ die } Taskknoten \tag{2.20}$$

$$\underset{n \in N_{Split}}{\forall} l(n) = Split, \text{ die } Splitknoten \tag{2.21}$$

$$\underset{n \in N_{Merge}}{\forall} l(n) = Merge, \text{ die } Mergeknoten \tag{2.22}$$

$$\underset{n \in N_{Fork}}{\forall} l(n) = Fork, \text{ die } Forkknoten, \text{ und} \tag{2.23}$$

$$\underset{n \in N_{Join}}{\forall} l(n) = Join, \text{ die } Joinknoten. \tag{2.24}$$

Weiterhin besitzt WFG die folgenden Eigenschaften:

1. Jeder Taskknoten $n_t \in N_{Task}$ hat genau eine eingehende und ausgehende Kante.

$$\underset{n_t \in N_{Task}}{\forall} |\triangleright n_t| = |n_t \triangleleft| = 1 \tag{2.25}$$

2. Jeder Split- und Forkknoten $n_{sf} \in (N_{Split} \cup N_{Fork})$ hat genau eine eingehende Kante und mindestens zwei ausgehende Kanten.

$$\underset{n_{sf} \in (N_{Split} \cup N_{Fork})}{\forall} |\triangleright n_{sf}| = 1 \wedge |n_{sf} \triangleleft| \geq 2 \tag{2.26}$$

3. Jeder Merge- und Joinknoten $n_{mj} \in (N_{Merge} \cup N_{Join})$ hat mindestens zwei eingehende Kanten und genau eine ausgehende Kante.

$$\underset{n_{mj} \in (N_{Merge} \cup N_{Join})}{\forall} |\triangleright n_{mj}| \geq 2 \wedge |n_{mj} \triangleleft| = 1 \tag{2.27}$$

Zur Darstellung von Workflowgraphen benutzen wir für die verschiedenen Arten von Knoten ein spezielles Symbol, wie Abbildung 2.3 zeigt. Startknoten werden durch einen einfachen Kreis illustriert. Einfache Taskknoten sind durch nicht gefüllte Rechtecke zu erkennen. Split- und Mergeknoten besitzen als Darstellung eine Raute, wobei die Raute der Mergeknoten dicker gezeichnet wird, als die der Splitknoten.

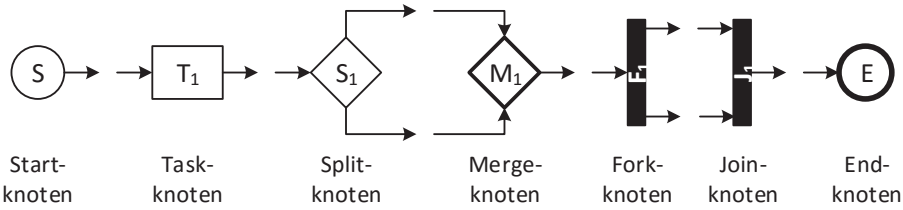


Abbildung 2.3: Darstellung der verschiedenen Knotenarten eines Workflowgraphen

Fork- und Joinknoten lassen sich durch schwarze Querbalken darstellen. Sie lassen sich anhand der unterschiedlichen Anzahl eingehender und ausgehender Kanten und wegen des Namens gut unterscheiden. Zu guter Letzt wird der Endknoten analog zum Startknoten durch einen Kreis illustriert, der aber im Gegensatz zum Startknoten etwas dicker umrandet ist.

Beispiel 2.12.

Abbildung 2.4 stellt einen Workflowgraphen $WFG = (N, E, l)$ dar, dessen Knoten durch die Menge

$$N = \{S, T_1, T_2, T_3, T_4, T_5, T_6, S_1, S_2, M_1, M_2, F_1, J_1, E\}$$

und Kanten durch die Menge

$$\begin{aligned} E = \big\{&(S, T_1), (T_1, M_1), (M_1, S_1), (S_1, T_2), (T_2, M_2), (M_2, T_3), \\ &(T_3, J_1), (S_1, F_1), (F_1, T_4), (T_4, M_2), (F_1, T_5), (T_5, S_2), \\ &(S_2, T_6), (T_6, M_1), (S_2, J_1), (J_1, E)\big\} \end{aligned}$$

repräsentiert werden. WFG entspricht dabei in etwa unserem Beispielprozess aus Kapitel 1. Die Abbildung der Knoten auf einen entsprechenden Typ über l ist definiert durch

$$\begin{aligned} l(T_1) = l(T_2) = l(T_3) = l(T_4) = l(T_5) = l(T_6) &= Task \\ l(S) = Start \quad l(E) &= End \\ l(S_1) = l(S_2) = Split \quad l(M_1) = l(M_2) &= Merge \\ l(F_1) = Fork \quad l(J_1) &= Join \end{aligned}$$

Aus diesem Grund ergeben sich die Teilmengen von N zu

$$\begin{aligned} N_{Task} = \{T_1, T_2, T_3, T_4, T_5, T_6\} \quad n_{Start} = S \quad n_{End} = E \\ N_{Split} = \{S_1, S_2\} \quad N_{Merge} = \{M_1, M_2\} \quad N_{Fork} = \{F_1\} \quad N_{Join} = \{J_1\} \end{aligned}$$

2.4 Semantik von Workflowgraphen

Die Ausführungssemantik eines Workflowgraphen wird hauptsächlich durch die Beschriftung der jeweiligen Knoten bestimmt. Um eine solche Semantik mathematisch zu definieren, nutzen wir, wie andere auch schon in diesem Forschungsgebiet (siehe etwa Vanhatalo et al. [85] und Völzer [88]), Definitionen der Petrinetze [54]. Im Wesentlichen basiert die Semantik von Petrinetzen auf *Zuständen*. Ein Zustand repräsentiert dabei eine aktuelle (oder erreichbare) Ausführungssituation.

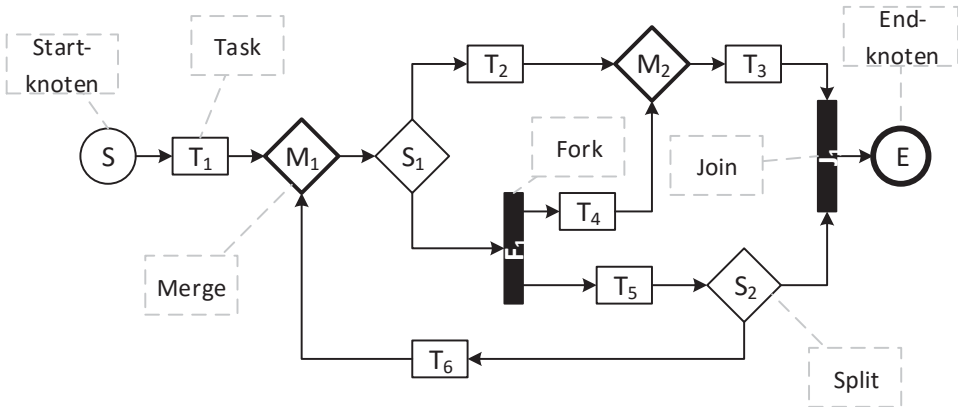


Abbildung 2.4: Ein Workflowgraph mit Verzweigungen, Parallelität und Schleife

Definition 2.13 (Zustand)**.**

Ein *Zustand* eines Workflowgraphen $WFG = (N, E, l)$ ist definiert als eine Multimenge Z über die Menge der Kanten E. Sie ordnet jeder Kante $e \in E$ eine natürliche Anzahl an *Marken* zu, $Z(e)$.

Die Menge aller überhaupt möglichen Zustände über eine Kantenmenge E bezeichnen wir mit $\mathcal{Z}(E)$.

Beispiel 2.14.

Sei der Workflowgraph $WFG = (N, E, l)$ aus Beispiel 2.12 gegeben, der nochmals in Abbildung 2.5 dargestellt ist. Die schwarzen Punkte auf den Kanten (T_4, M_2) und (S_2, T_6) entsprechen den Positionen der Marken im aktuellen Zustand der Ausführung des Workflowgraphen. Diese Darstellungsform eines Zustands werden wir auch im weiteren Verlauf dieser Arbeit verwenden.

Damit ist WFG im Beispiel in folgendem Zustand:

$$[\![(T_4, M_2), (S_2, T_6)]\!] = Z$$

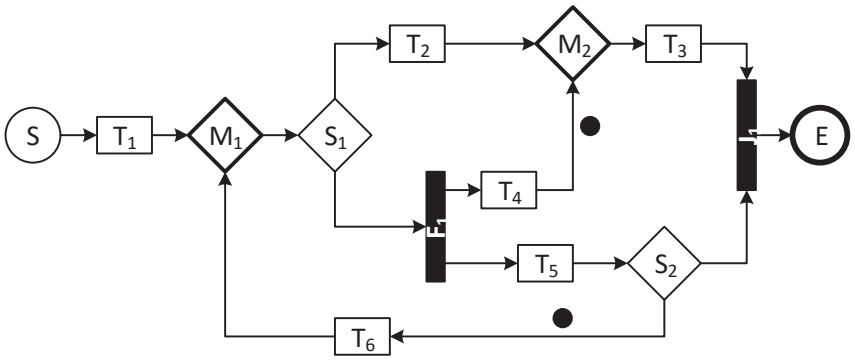


Abbildung 2.5: In einem Zustand stellen wir Marken durch schwarze Punkte dar

Eine Kante e eines Workflowgraphen $WFG = (N, E, l)$ *trägt*, *besitzt* oder einfach nur *hat* eine Marke in einem Zustand Z, wenn e ein Element der Multimenge Z ist bzw. $Z(e) \geq 1$ gilt. Im *Startzustand* trägt nur die Startkante e_{Start} und im *Endzustand* nur die Endkante e_{End} eine einzige Marke.

Ausgehend von einem Zustand kann ein Workflowgraph in einen anderen Zustand *wechseln*. Dazu muss jedoch ein Knoten des Workflowgraphen *ausführbar* sein.

Definition 2.15 (Ausführbarkeit)**.**

Sei $WFG = (N, E, l)$ ein Workflowgraph in einem Zustand Z.

Der Start- und Endknoten ist in keinem Zustand ausführbar. Alle anderen Knoten $n \in N \setminus \{n_{Start}, n_{End}\}$ sind dann *ausführbar* in Z, wenn entweder (1) n *kein* Joinknoten ist und mindestens eine seiner eingehenden Kanten eine Marke trägt, oder (2) jede seiner eingehenden Kanten eine Marke besitzt.

$$n \text{ ausführbar in } Z \iff (n \notin N_{Join} \wedge \triangleright n \cap Z \neq \varnothing) \vee (\triangleright n \subseteq Z) \tag{2.28}$$

$\mathcal{E}xec(Z)$ steht für die Menge aller ausführbaren Knoten im Zustand Z:

$$\mathcal{E}xec(Z) = \{n \in N : n \text{ ist ausführbar in } Z\} \tag{2.29}$$

Beispiel 2.16.

Im Workflowgraph $WFG = (N, E, l)$ im Zustand Z aus Abbildung 2.5 sind die Knoten M_2 und T_6 ausführbar. Sie sind keine Joinknoten und auf jeweils einer eingehenden Kante besitzen sie eine Marke. Die Menge aller ausführbaren Knoten ist dadurch gegeben als

$$\mathcal{E}xec(Z) = \{M_2, T_6\}$$

Der Joinknoten J_1 ist in jedem Zustand $Z' \supseteq \{(T_3, J_1), (S_2, J_1)\}$ ausführbar.

In der vorangegangenen Definition 2.15 wurde zum ersten Mal die Beschriftung eines Knotens im Zuge der Einführung einer Semantik von Workflowgraphen genutzt. Jede Art eines Knotens innerhalb eines Workflowgraphen besitzt eine spezielle Semantik. Gerade wurde die Ausführbarkeit als spezielle Eigenschaft der Joinknoten eingeführt, denn Joinknoten können erst dann in einem Zustand ausgeführt werden, wenn auf allen eingehenden Kanten eine Marke liegt.

Die speziellen Eigenschaften der anderen Knoten machen sich bei einem *Zustandsübergang* bemerkbar.

Definition 2.17 (Zustandsübergang)**.**

Nehmen wir einen Zustand Z eines Workflowgraphen $WFG = (N, E, l)$ an, in dem der Knoten $n \in N$ ausführbar ist.

Nach der Ausführung des Knotens n *wechselt* WFG in den Zustand Z'. Wir schreiben $Z \xrightarrow{n} Z'$. Der Zustand Z' ist, wie folgt, für den Knoten n definiert:

$n \in (N_{Task} \cup N_{Fork} \cup N_{Join})$**:** Von jeder eingehenden Kante e_{in} von n wird eine Marke entnommen ($Z' = Z \setminus \{e_{in}\}$) und auf jede ausgehende Kante e_{out} wird eine Marke gelegt ($Z' = Z \cup \{e_{out}\}$).

$$Z' = (Z \setminus \triangleright n) \cup n \triangleleft \tag{2.30}$$

$n \in (N_{Split} \cup N_{Merge})$**:** Von genau einer eingehenden Kante e_{in} von n, die eine Marke trägt, wird eine Marke entnommen ($Z' = Z \setminus \{e_{in}\}$) und auf genau eine ausgehende Kante e_{out} wird eine Marke gelegt ($Z' = Z \cup \{e_{out}\}$).

$$Z' = (Z \setminus \{e_{in}\}) \cup \{e_{out}\}, e_{in} \in (\triangleright n \cap Z) \tag{2.31}$$

Beispiel 2.18.

Nach dem letzten Beispiel 2.16 sind im Workflowgraphen $WFG = (N, E, l)$ aus Abbildung 2.5 die Knoten M_2 und T_6 im aktuellen Zustand Z ausführbar. Nichtdeterministisch können nach der vorangegangenen Definition 2.17 zwei mögliche Zustandswechsel folgen: Entweder schaltet der Taskknoten T_6 zuerst oder der Mergeknoten M_2.

Beispielsweise geht der aktuelle Zustand $Z = [\![(T_4, M_2), (S_2, T_6)]\!]$ nach Ausführung des Knotens T_6 in den Zustand Z' über, wobei der Kante (S_2, T_6) eine Marke entnommen und der Kante (T_6, M_1) eine hinzugefügt wird. Dies wird geschrieben als $Z \xrightarrow{T_6} Z'$ und der Zustand Z' ergibt sich zu

$$Z' = [\![(T_4, M_2), (T_6, M_1)]\!]$$

Definitionen 2.15 und 2.17 legen die Semantik der unterschiedlichen Arten von Knoten fest. Zusammengefasst besitzen der Start- und Endknoten keine Ausführungssemantik. Sie dienen damit lediglich der Markierung des Starts und Endes eines Workflowgraphen; von wo die Ausführung beginnt und wo sie endet. Alle Knoten mit Ausnahme der Joinknoten können ausgeführt werden, sobald mindestens eine Marke auf mindestens einer eingehenden Kante liegt. Ein Taskknoten nimmt dann eine Marke von seiner eingehenden Kante und positioniert sie auf seiner ausgehenden Kante. Split- und Mergeknoten treffen nicht-deterministische Entscheidungen. Ein Splitknoten entnimmt von seiner eingehenden Kante eine Marke und legt sie zufällig auf eine seiner ausgehenden Kanten. Ein Mergeknoten hingegen entnimmt zufällig von einer seiner eingehenden Kanten (natürlich mit Marke) eine Marke und positioniert sie auf seiner einzigen ausgehenden Kante. Forkknoten nehmen auch eine Marke von ihren eingehenden Kanten, legen aber auf jede ihrer ausgehenden Kante eine Marke ab. Damit erzeugen Forkknoten Parallelität. Diese Parallelität kann nur durch einen Joinknoten synchronisiert werden, denn dieser ist erst ausführbar, wenn

auf all seinen eingehenden Kanten mindestens eine Marke liegt. Bei Ausführung wird von jeder dieser Kanten eine Marke entnommen und nur eine auf seine ausgehende Kante gelegt. Die Semantik der unterschiedlichen Knoten entspricht der gängigen Semantik von Workflowgraphen in der Literatur (vgl. Sadiq und Orlowska [68]).

Durch die Semantik eines Knotens werden auch die möglichen Zustände definiert, die von dem aktuellen Zustand *erreicht* werden können.

Definition 2.19 (Erreichbarkeit)**.**

Ein Zustand Z_{to} ist von einem Zustand Z_{from} *direkt erreichbar* (geschrieben $Z_{from} \rightarrow Z_{to}$), wenn in Z_{from} ein ausführbarer Knoten n existiert, nach dessen Ausführung Z_{from} nach Z_{to} wechselt.

$$Z_{from} \rightarrow Z_{to} \iff \underset{n \in \mathcal{E}xec(Z_{from})}{\exists} Z_{from} \overset{n}{\rightarrow} Z_{to} \tag{2.32}$$

Z_{to} ist vom Zustand Z_{from} *erreichbar* (wir schreiben $Z_{from} \rightarrow^* Z_{to}$), wenn eine Sequenz von Zuständen $Z_0, \ldots, Z_m$, $m \geq 1$, existiert, so dass $Z_0 \rightarrow Z_1 \rightarrow \ldots \rightarrow Z_{m-1} \rightarrow Z_m$ und $Z_0 = Z_{from}$, $Z_m = Z_{to}$.

$$Z_{from} \rightarrow^* Z_{to} \iff \underset{\substack{Z_0,\ldots,Z_m, m \geq 1 \\ Z_0 = Z_{from}, Z_m = Z_{to}}}{\exists} \underset{i \in \{0,\ldots,m-1\}}{\forall} Z_i \rightarrow Z_{i+1} \tag{2.33}$$

Die Erreichbarkeit von Zuständen, insbesondere die direkte Erreichbarkeit, ist eine Relation auf der Menge der Zustände. Der durch die Relation aufgespannte Graph wird *Zustandsraum* genannt. In diesem stellt jeder Knoten einen Zustand dar. Eine Kante existiert zwischen jenen Zuständen, die direkt erreichbar sind.

2.5 Korrektheit

Der Begriff der *Korrektheit* (im Englischen *Soundness*) [76] tauchte zum ersten Mal in einer frühen Arbeit von van der Aalst im Zusammenhang mit *Free-Choice-Petrinetzen* [65, S. 104], speziellen Petrinetzen, in der Logistik auf. Diese Eigenschaft verspricht, dass jede Ausführung eines Free-Choice-Petrinetzes in einem Zustand terminiert, in dem nur der Platz genau eine Marke besitzt, der das Ende des Workflows repräsentiert [77]. Wissend um diesen Begriff, definierten Sadiq und Orlowska [68] dennoch einige Jahre später die Workflowgraphen und den Begriff der Korrektheit neu, da nach ihrer Auffassung kein Produkt zu diesem Zeitpunkt auf dem Markt war, dass tatsächlich Petrinetze für das Workflowmanagement benutzte. Aus diesem Grund besitzen Workflowgraphen meist die gängigen Arten von Knoten, die auch in der Arbeitsprozessmodellierung Anwendung finden. In diesem Kontext fanden sie speziellere Arten von Fehlern, als dies van der Aalst tat: die Verklemmung (im Englischen *Deadlock*) und die fehlende Synchronisierung (im Englischen *Lack of Synchronization*). Bei einer Verklemmung ist die Ausführung eines Workflowgraphen

in einem Zustand, der nicht dem Endzustand entspricht und dennoch kein weiterer Zustand erreichbar ist.

Definition 2.20 (Verklemmungszustand)**.**

Ein Zustand $Z_{dead} \neq Z_{End}, Z_{dead} \in \mathcal{Z}(E)$, eines Workflowgraphen $WFG = (N, E, l)$ heißt *Verklemmungszustand* oder einfach nur *Verklemmung*, wenn in Z_{dead} kein Knoten ausführbar ist. Das heißt, von Z_{dead} ist kein anderer Zustand direkt erreichbar.

$$Z_{dead} \text{ ist Verklemmung} \iff \mathcal{E}xec(Z_{dead}) = \varnothing \iff \underset{Z' \in \mathcal{Z}(E)}{\forall} Z_{dead} \not\rightarrow Z' \tag{2.34}$$

Beispiel 2.21.

Der Zustand Z_{dead} mit

$$Z_{dead} = [\![(T_3, J_1)]\!]$$

ist beispielsweise eine Verklemmung unseres Beispielworkflowgraphen aus Abbildung 2.6. In diesem Zustand besitzt zwar der Joinknoten J_1 auf einer eingehenden Kante eine Marke, kann nach Definition 2.15 jedoch nicht ausgeführt werden. Damit ist von Z_{dead} kein anderer Zustand erreichbar. Zudem ist die Kante (T_3, J_1) ungleich der Endkante und somit Z_{dead} nicht der Endzustand.

Der zweite von Sadiq und Orlowska eingeführte Fehlerzustand – die fehlende Synchronisierung – beschreibt Zustände, in denen mindestens eine Kante mindestens zwei Marken trägt. An dieser Stelle ist nach unserer Ansicht der Begriff unglücklich gewählt, denn „Fehlende Synchronisierung“ beschreibt die Ursache des Fehlers. Hingegen beschreibt der Begriff der „Verklemmung“ die Wirkung des Fehlers. Aus diesem Grund bevorzugen wir den Begriff des *Überflusszustands* anstelle der fehlenden Synchronisierung, da dieser dann einheitlich auch die Wirkung des Fehlers beschreibt.

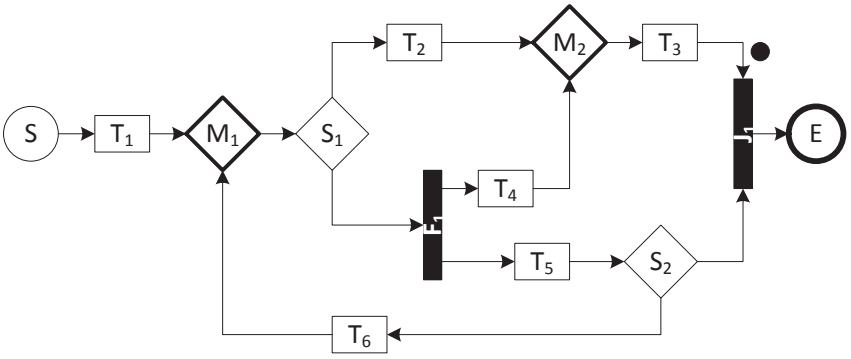


Abbildung 2.6: Ein Verklemmungszustand in unserem fortlaufenden Beispiel

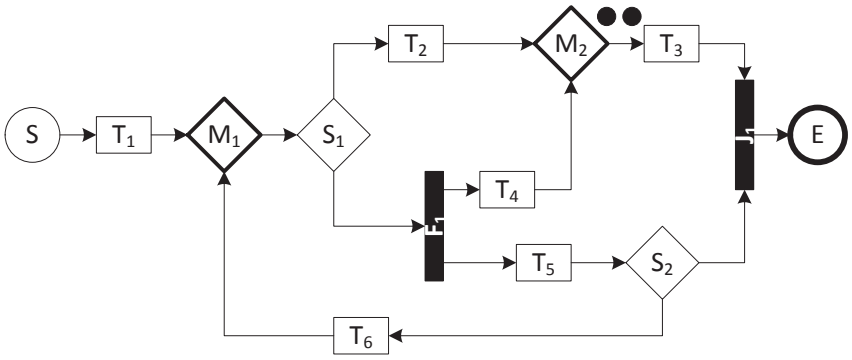


Abbildung 2.7: Ein Überflusszustand in unserem Workflowgraphen

Definition 2.22 (Überflusszustand)**.**

Ein Zustand $Z_{abundance} \in \mathcal{Z}(E)$ eines Workflowgraphen $WFG = (N, E, l)$ heißt *Überflusszustand*, wenn in $Z_{abundance}$ mindestens eine Kante mehr als eine Marke besitzt.

$$Z_{abundance} \text{ ist Überflusszustand} \iff \underset{e \in E}{\exists} Z_{abundance}(e) \geq 2 \tag{2.35}$$

Beispiel 2.23.

Ein Beispiel für einen Überflusszustand stellt der Zustand

$$Z_{abundance} = [\![(M_2, T_3), (M_2, T_3)]\!]$$

dar. In ihm besitzt die Kante (M_2, T_3) mehr als eine Marke. Abbildung 2.7 visualisiert diesen Überflusszustand.

Ein beliebiger Workflowgraph ist *korrekt*, wenn vom Startzustand weder Verklemmung noch Überflusszustand erreichbar ist [85].

Definition 2.24 (Korrektheit)**.**

Ein Workflowgraph $WFG = (N, E, l)$ wird *korrekt* genannt, wenn vom Startzustand weder ein Verklemmungs- noch ein Überflusszustand erreichbar ist.

$$WFG \text{ ist korrekt} \iff \underset{\substack{Z \in \mathcal{Z}(E) \\ Z_{Start} \to^* Z}}{\forall} \underbrace{(\mathcal{E}xec(Z) \neq \varnothing \vee Z = Z_{End})}_{Z \text{ ist keine Verklemmung}} \tag{2.36}$$

$$\wedge \underbrace{\underset{e \in E}{\forall} Z(e) \leq 1}_{Z \text{ ist kein Überflusszustand}} \tag{2.37}$$

Beispiel 2.25.

Unser Beispielworkflowgraph aus Abbildung 2.7 ist *nicht* korrekt, da die Verklemmung und der Überflusszustand aus den Beispielen 2.21 und 2.23 vom Startzustand erreichbar sind. Einen korrekten Workflowgraph, der zudem noch zu unserem Beispielgraph Ähnlichkeit besitzt, zeigt Abbildung 2.8.

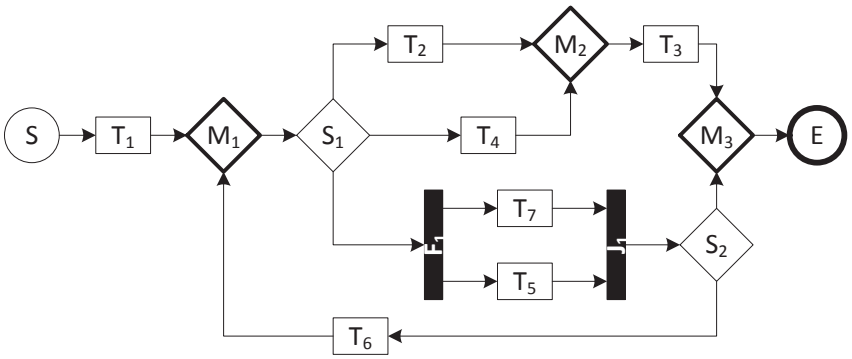


Abbildung 2.8: Ein zu Abbildung 2.4 ähnlicher Workflowgraph

Die vorangegangene Definition über die Korrektheit von Workflowgraphen hat nur unter der Annahme der sogenannten *Fairness* Bestand. Im Allgemeinen bewirkt die Fairnessannahme, dass nicht-deterministische Entscheidungen (von Split- und Mergeknoten) auch tatsächlich nicht-deterministisch getroffen werden und somit beispielsweise Schleifen nicht beliebig oft von einem „Kontrollfluss“ durchlaufen werden [7][39][51][79]. Für den Rest dieser Arbeit gehen wir von der Fairness der betrachteten Workflowgraphen aus.

Kapitel 3

Stand der Forschung

Die Kontrollflussanalyse von Arbeitsprozessen hat eine lange Tradition. Bereits 1995 fiel zum ersten Mal der Begriff der *Korrektheit* im Zusammenhang mit Geschäftsprozessen in einer Arbeit von van der Aalst [78]. Dieser Begriff hat sich dann mit der Zeit weiterentwickelt und bis heute entstanden verschiedene Varianten des Korrektheitsbegriffs, wie relaxte oder schwache Korrektheit (einen guten Überblick geben Puhlmann [63] und van der Aalst et al. [82]). Mittlerweile ist die Kontrollflussanalyse von Prozessen soweit fortgeschritten, dass sie auf Prozesse der realen Welt angewandt werden kann [91].

Für die vorliegende Arbeit wurde die bekannteste Definition von Korrektheit [78], auch *klassische* Korrektheit genannt, gewählt. Van der Aalst definiert die klassische Korrektheit auf speziellen Petrinetzen mit genau einem Start- und einem Endplatz, sogenannten *Geschäftsprozedurnetzen* (später besser bekannt als *Workflownetze*). Zudem sind diese Prozedurnetze zusammenhängend [53, S. 547] und gehören der Klasse der *Free-Choice*-Petrinetze[1] [16, S. 5 ff.] an. Ein solches Netz ist nach van der Aalsts Auffassung genau dann korrekt, wenn zum einen von jedem erreichbaren Zustand der Terminierungszustand erreicht werden kann, das heißt, keine Verklemmung vorliegt; und zum anderen ist der Terminierungszustand der einzige Zustand, in dem der Endplatz des Netzes eine Marke trägt – es gibt demnach keine Überflüsse.

Zusätzlich zu dieser Definition beschreibt van der Aalst zugleich, wie die Korrektheit eines solches Geschäftsprozedurnetzes nachgewiesen werden kann. Diese Lösung zum Nachweis galt lange Zeit und für viele Arbeiten noch bis heute als eine der effizientesten: Die Anwendung des Rangsatzes [16, S. 111 ff.].

Der Rangsatz kann auf Free-Choice-Petrinetzen angewandt werden, um die Eigenschaften der Lebendigkeit und Beschränktheit nachzuweisen. Damit van der Aalst

[1] Free-Choice-Petrinetze besitzen die Eigenschaft, dass für jeden Platz mit mindestens zwei ausgehenden Kanten gilt, dass die durch diese Kanten verbundenen Transitionen nur genau eine eingehende Kante besitzen. Dadurch wird jede Entscheidung, die durch den Platz repräsentiert wird, unabhängig in seiner Entscheidung.

den Rangsatz für den Nachweis der Korrektheit nutzen kann, führt er zunächst die Verklemmungen und Überflüsse auf die Begriffe der Lebendigkeit und Beschränktheit zurück. So kann er den Rangsatz einsetzen, um zum einen die Lebendigkeit als auch, zum anderen, die Beschränktheit nachzuweisen. Dabei benötigt der Rangsatz als Eingabe die Inzidenzmatrix des Netzes. Für diese Matrix wird anschließend der Rang berechnet. Entspricht der berechnete Rang der Anzahl an Clustern des Netzes (minus 1), so gelten die Eigenschaften als nachgewiesen; ansonsten nicht. Ein solches Cluster besteht dabei aus der minimalen Menge an Knoten, für die jeder Platz inklusive aller direkten Nachfolger und jede Transition inklusive aller direkten Vorgänger enthalten ist [16, S. 65].

Der Vorteil dieses Vorgehens ist, dass aus der Petrinetztheorie bekannte Algorithmen und Ansätze angewandt werden können und das Laufzeitverhalten bekannt ist. So benötigt die Anwendung des Rangsatzes eine polynomielle Laufzeit [37], genauer gesagt eine kubische Zeit $O(N^3)$ (N ist das Maximum aus der Anzahl der Plätze, Transitionen und Kanten) [36]. Der große Nachteil der Anwendung des Rangsatzes besteht jedoch im Fehlen diagnostischer Informationen zur Beschreibung von Fehlern [21]. Aus diesem Grund entstanden in den darauffolgenden Jahren neue Modelle und Ansätze zur Beschreibung und Überprüfung von Arbeitsprozessen, die wir im weiteren Verlauf dieses Kapitels kurz vorstellen.

Formale Modelle zur Beschreibung von Arbeitsprozessen. Das neben den Workflownetzen wohl bekannteste formale Modell von Prozessen ist der Workflowgraph von Sadiq und Orlowska [68]. Workflowgraphen ähneln sehr den aus dem Übersetzerbau bekannten Kontrollflussgraphen (bspw. [92, S.57]) und erweitern diese um explizite Parallelität. Sie beinhalten jedoch analog zu den Workflownetzen nur die Modellelemente, die zur Beschreibung des Kontrollflusses notwendig sind. Vielmehr noch waren Workflowgraphen zu Beginn ihrer Einführung azyklisch und wurden erst später durch das Konstrukt der Schleife ergänzt.

Van der Aalst [80] hat kurze Zeit nach der Einführung der Workflowgraphen bereits nachgewiesen, dass jeder (azyklische) Workflowgraph in ein semantisch äquivalentes Free-Choice-Workflownetz übertragen werden kann. Somit kann der Rangsatz wiederum auch auf Workflowgraphen angewandt werden. Erst im Jahr 2015 zeigten Favre et al. die Äquivalenz gewöhnlicher Workflowgraphen und Free-Choice-Workflownetze [20]. Im Gegensatz zu Workflownetzen besitzen Workflowgraphen aber den Vorteil, dass sie wesentlich einfacher um zusätzliche Kontrollflusselemente erweitert werden können [20]. Bekannte und häufig verwendete andere Kontrollflusselemente sind die sogenannten Or-Joinknoten und die Abbruchregionen. Favre et al. zeigten beispielsweise, dass die Semantik von Or-Joinknoten nicht vollständig auf Free-Choice-Workflownetze abgebildet werden kann.

Diesen Nachteil von Workflownetzen erkannten auch van der Aalst und ter Hofstede und formulierten die Prozessmodellierungssprache YAWL [81]. Ausgehend von einer Analyse verschiedener Modellierungssprachen erweiterten sie basierend auf ihren Free-Choice-Workflownetzen diese um spezielle Muster (Patterns), wie den bereits genannten Or-Joinknoten und Abbruchregionen.

Eine semantisch äquivalente Variante der Abbruchregionen wurde bereits etwas eher von Chrząstowski-Wachtel et al. [11] vorgeschlagen. Sie eröffneten mit ihrer Arbeit auch eine weitere Variante der Darstellung von Prozessen: Die Darstellung des Prozesses als Baum, in dem die Kindknoten eines jeden Knotens des Baums eine Verfeinerung des Prozesses, quasi einen Unterprozess, darstellen. Durch diese Verfeinerung entsteht eine hierarchische Anordnung der Elemente des Prozesses. Chrząstowski-Wachtels Modell des Prozesses fordert dabei dessen Konstruktion durch eine schrittweise Detaillierung. Werden in jedem Schritt der Verfeinerung nur entsprechend korrekte Strukturen verwendet, so ist der entstandene Prozess per Konstruktion korrekt. Die Rückrichtung, sprich das Herausfiltern einer solchen hierarchischen Struktur aus einem Prozess, bleiben Chrząstowski-Wachtel et al. in ihrer Arbeit schuldig.

Nachweis der Korrektheit mittels Dekomposition. Die Ableitung einer hierarchischen Struktur, des *Prozessstrukturbaums*, aus einem Workflowgraphen untersuchten Vanhatalo et al. [85] und gingen damit den anderen Weg als Chrząstowski-Wachtel et al. [11]. Der Vorteil dieser Variante liegt in der intuitiven Entwicklung eines Prozesses durch einen Modellierer mittels einer graphischen Modellierungssprache. Die aus dieser graphischen Ansicht durch Dekomposition gewonnene hierarchische und manchmal auch abstrakte Struktur bleibt dem Modellierer verborgen, kann aber von einem Analysewerkzeug schrittweise untersucht werden. Für diese Untersuchung bieten Vanhatalo et al. einfache Regeln und Heuristiken an – die jedoch nachweislich unvollständig sind [85]. Sie sind unvollständig, wenn es sich bei den untersuchten Teilstücken des Graphen, den sogenannten Single-Entry-Single-Exit (SESE)-Fragmenten, um unstrukturierte Teilgraphen handelt.

Diese SESE-Dekomposition hat jedoch auch wesentliche Vorteile: Zum einen liefert sie eine hohe Güte an diagnostischen Informationen. So kann pro Fragment genau ein Fehler gefunden werden, selbst wenn solch ein fehlerhaftes Fragment aufgrund einer früheren Verklemmung niemals bei der Ausführung erreicht wird. Zum anderen können auf den Fragmenten beliebige Analysen durchgeführt werden, denn bei den Fragmenten handelt es sich wiederum um Workflowgraphen, wenn sie durch einen expliziten Start- und Endknoten erweitert werden. So ist zum Beispiel der adaptive Einsatz einer Zustandsraumerkundung für unstrukturierte Fragmente möglich.

Ein weiterer wichtiger Vorteil ist auch die Laufzeit des Analyseverfahrens; sie wird durch die Dekomposition des Prozesses und die Konstruktion des Prozessstruktur-

baums bestimmt. Diese beiden Teilschritte (die Dekomposition und Konstruktion) und der Prozessstrukturbaum sind an die Übersetzertechniken zur Erzeugung des *Programm*strukturbaums von Johnson et al. [33][34] für gewöhnliche Computerprogramme angelehnt. Diese Techniken haben eine lineare Laufzeit und wurden von Ananian im Zuge seiner Masterarbeit zur *Single Static Information Form* [5] korrigiert. Damit stellt die Arbeit von Vanhatalo et al. [85] aus unserer Sicht den ersten Ansatz dar, Arbeitsprozesse mit den Techniken des Übersetzerbaus zu untersuchen.

Diese Techniken wurden von Vanhatalo et al. in einer späteren Arbeit nochmals verfeinert, in dem Fragmente gefunden werden, die 3-zusammenhängend sind [84]. Solche 3-zusammenhängende Teilgraphen entsprechen Graphen mit genau einer ein- und einer ausgehenden Kante. Solche können effizient in Linearzeit mit dem Algorithmus von Hopcroft und Tarjan [30] für einen beliebigen Kontrollflussgraphen bestimmt werden und sind wesentlich feingranularer als die SESE-Fragmente. Dadurch sind beispielsweise der Korrektheitsnachweis und die gewonnenen diagnostischen Informationen viel akkurater.

Diese akkuraten Fragmente und Informationen können auch zur Neugestaltung des Prozesses genutzt werden [86]. Beispielsweise ist es möglich, zwei Fragmente, deren Ein- und Ausgänge sich überschneiden, durch Duplizieren dieser Ein- und Ausgänge eindeutig zu gestalten. Dieser Ansatz der Neugestaltung kann auch bereits während der (graphischen) Konstruktion des Prozesses genutzt werden. Dabei werden Knoten entsprechend dupliziert oder, unter Beibehaltung der Semantik, Fragmente, die während der Konstruktion noch nicht geschlossen wurden, automatisch mit entsprechend passenden, konvergierenden Knoten vollendet. Damit schließt sich auch der Kreis zu den Arbeiten von Chrząstowski-Wachtel et al. [11], denn es entstehen immer Strukturen, die korrekt sind. Und sind die einzelnen Fragmente korrekt, so ist es auch der vollständige Workflowgraph. Der praktische Einsatz dieses Konstruktionsansatzes mit ergänzenden syntaktischen Regeln wurde von Kühne et al. [42] gezeigt. So war es ihnen möglich, bereits während der Konstruktion zeitnahe Diagnoseinformationen zur Korrektheit für den konstruierten Prozess zu erhalten.

Modellprüfung und Rangsatz. Ein großer Nachteil der Dekompositionsansätze ist derzeit aber noch dessen Unvollständigkeit für unstrukturierte Fragmente. Für diese können aber, wie bereits erwähnt, andere Techniken verwendet werden. Allen voran werden das Modellprüfen (Model Checking) und der bereits vorgestellte Ansatz über die Anwendung des Rangsatzes genutzt.

Der Rangsatz ist, wie bereits erwähnt, in vielen Arbeiten das Mittel zur Wahl, um Prozesse auf Korrektheit zu überprüfen. Der dazu populärste Alternativansatz wurde wieder von van der Aalst geliefert. In seiner Arbeit „Verification of Workflow Nets“ beschreibt er, dass auch die Analyse per Zustandsraumerkundung (einem weiteren Petrinetz-geprägten Analyseverfahren) möglich ist [77]. Dabei wird Zustand

für Zustand abgeleitet und durch einen Graphen repräsentiert. Das Grundproblem einer solchen Zustandsraumkonstruktion ist dessen Komplexität. Selbst für kleine Graphen kann der dazugehörige Zustandsraum bereits eine beliebige, unbeschränkte Größe besitzen. Dies wird auch Zustandsraumexplosion genannt [18][75].

Um zumindest die Unbeschränktheit eines solchen Zustandsraums auf eine beschränkte Größe zu reduzieren, werden beispielsweise beliebig steigende Anzahlen von Marken in einem Platz durch ein Einselement als obere Schranke abstrahiert. Der daraus resultierende Graph wird *Coverability Tree* [51] genannt. Aber auch dessen Erstellung ist EXPSPACE-hart [10]. Im Gegensatz zur einfachen Anwendung des Rangsatzes liefert dieser aber immerhin diagnostische Informationen, wie den Weg im Zustandsraumgraphen zum ersten fehlerhaften Zustand.

Um diesen Ansatz zu präzisieren, wählten Lohmann und Fahland [48] verschiedenste Techniken um den Zustandsraum zu reduzieren. Ziel war es herauszufinden, welche Entscheidungen von Splitknoten zu fehlerhaftem Verhalten führen. Diese Informationen können dann als diagnostische Informationen zurückgeliefert werden. Leider handelt es sich bei diesen Arbeiten jedoch nur um Konzepte, für die es noch keine Implementierung gibt.

Für die allgemeine Zustandsraumanalyse von Prozessen gibt es jedoch zwei bekannte Implementierungen: Woflan [87] und LoLA [69].

Woflan ist ein *Wo*rk*fl*ow-*An*alysewerkzeug von Verbeek et al. Es ist das wohl derzeit vollständigste Werkzeug zur Überprüfung von Arbeitsprozessen auf Workflownetzbasis. Neben der Korrektheit überprüft es zahlreiche weitere Qualitätsmerkmale. Wolfan versucht zunächst das Workflownetz auf ein kleineres Netz zu reduzieren. Ist das resultierende Netz trivial, dann kann es als korrekt eingestuft werden. Wenn nicht, versucht Woflan über die sogenannte *S-Coverability* zu entscheiden, ob es diagnostische Informationen liefern kann oder nicht. Die S-Coverability beschreibt dabei die Möglichkeit der Zerlegung des Workflownetzes in S-Komponenten. S-Komponenten sind minimale Mengen, in denen für jeden Platz alle direkten Vorgänger und Nachfolger enthalten sind [16, S. 89]. Ist die Zerlegung in solche S-Komponenten nicht möglich, so ist das Workflownetz inkorrekt; es ist aber unklar, ob ein Überfluss oder eine Verklemmung vorliegt. Ist eine Zerlegung hingegen möglich, so ist das Workflownetz zumindest überflussfrei. Mit Hilfe einer Zustandsraumerkundung werden dann Verklemmungen im Workflownetz abgeleitet.

Insgesamt hat Woflan damit einen Exponentiallaufzeitalgorithmus implementiert, der sich in anderen Arbeiten in Laufzeitvergleichen offenbart. Dies zeigt beispielsweise die Arbeit von Fahland et al. in der die drei bereits erwähnten Ansätze zum Korrektheitsnachweis von Prozessen untersucht wurden [18]: Die Zustandsraumerkundung mit dem Werkzeug LoLA, die SESE-Dekomposition mit dem *IBM WebSphere Business Modeler* und die S-Coverability und Zustandsraumerkundung mit

dem Werkzeug Woflan. Das Ergebnis dieses Vergleichs war die Erkenntnis, dass die Korrektheitsprüfung zum Großteil auf Anfrage in überschaubarer Zeit möglich ist. Die zustandsraumbasierten Ansätze haben aber auch gezeigt, dass manche der 1.386 untersuchten Prozesse nicht vollständig untersucht werden können, weil diese Untersuchungen zu lange dauern oder eben niemals enden würden.

Unter diese zustandsraumbasierten Ansätze fällt, wie bereits erwähnt, auch das Werkzeug LoLA [69]. LoLA ist ein *Low Level Petri Net Analyzer*, das heißt, ein Analysewerkzeug für einfache Petrinetze. Es dient der allgemeinen Modellprüfung und nicht nur dem Nachweis der Korrektheit. So werden in LoLA die nachzuweisenden Eigenschaften innerhalb des Zustandsraums durch formale Gleichungen angegeben. Beispielsweise enthält die Formel zum Nachweis eines Überflusses die Bedingung, dass in einem Zustand mindestens ein Platz des Netzes mehr als eine Marke trägt.

LoLA benutzt für seine Analyse Reduktionstechniken und anschließend eine Zustandsraumerkundung. Die Reduktionstechniken verkleinern unter Einbehaltung der Semantik das Petrinetz und somit auch den zum Petrinetz gehörenden Zustandsraum. Dadurch kann in den meisten Fällen dieser Zustandsraum derart minimiert werden, dass dessen Konstruktion keine exponentielle Laufzeit benötigt.

Andere Ansätze zum Nachweis der Korrektheit basieren ebenfalls auf einer Reduktion des Prozesses. Sadiq und Orlowska [68] haben beispielsweise bei der Einführung der Workflowgraphen Reduktionsregeln angegeben, um Verklemmungen und Überflüsse zu finden. Diese Reduktionsregeln eliminieren den Start- und Endknoten des Workflowgraphen, fassen Sequenzen zu einer einfachen Kante zusammen, vereinigen Split-, Fork-, Merge- und Joinknoten, usw. Besteht der durch mehrmalige Anwendung dieser Regeln resultierende Graph dann aus nur einem Knoten, so ist der Workflowgraph korrekt. Andernfalls ist er inkorrekt.

Auch wenn das Laufzeitverhalten mit $O(N^2)$ sehr gut ist (N ist die Summe der Anzahl an Knoten und Kanten), hat bereits van der Aalst kurz darauf gezeigt, dass die Reduktionsregeln unvollständig sind [80]. Jedoch können anhand der Reduktionsregeln die sogenannten *Instanzgraphen* gezählt werden. Instanzgraphen sind Teilgraphen eines Workflowgraphen, die eine mögliche Ausführung repräsentieren [68]. Eine Fehlerbeschreibung ist dadurch mit Hilfe eines Instanzgraphen als Diagnose sehr gut.

Dies erkannten auch Eshuis und Kumar [17], die ihrerseits die bisherigen Ansätze über Zustandsraumerkundung und Rangsatz dahingehend kritisieren, dass sie keine zuverlässige Fehlerlokalisierung bieten. Eshuis und Kumar selbst wählten einen Ansatz über Integer Programming, um die von Sadiq und Orlowska vorgeschlagenen, fehlerbehafteten Instanzgraphen zu finden. Ein Instanzgraph ist genau dann inkorrekt, wenn er für einen Joinknoten nicht all seine direkten Vorgänger (Verklemmung) oder mehr als einen direkten Vorgänger eines Mergeknotens (Überfluss) beinhaltet. Damit sind diese Instanzgraphen auf azyklische Workflowgraphen beschränkt.

Für jeden der Instanzgraphen können die beschriebenen Fehler gefunden und sehr gut lokalisiert werden. Jedoch ist es nicht möglich, Fehler hinter anderen Fehlern, wie Verklemmungen, zu identifizieren. Außerdem besitzt ein solches ganzzahliges Programm eine theoretisch expontielle Laufzeit in Abhängigkeit der Größe des Workflowgraphen. Auch wenn Eshuis und Kumar mit ihrem Werkzeug *DiagFlow*[2] feststellen konnten, dass dieses schneller ist als Woflan [87], sind ein Zeitaufwand von über 140 Sekunden für einen Graphen mit 218 Knoten für ein schnelle Fehlerrückmeldung, beispielsweise in einem Prozessmodellierer, nicht vertretbar.

Muster- und übersetzerbasierte Ansätze. Gute Alternativen zu den bisher vorgestellten Ansätzen mit einem exponentiellen Laufzeitverhalten mit diagnostischen Informationen und den Polynomialzeitansatz mit dem Rangsatz ohne diagnostische Informationen bieten die strukturellen Ansätze, die anhand von sogenannten Gegenmustern oder Theorien aus dem Übersetzerbau deren Korrektheit versuchen nachzuweisen. Den bekanntesten dieser Ansätze über die SESE-Dekomposition haben wir bereits ausführlich im Zusammenhang mit einer allgemeinen Dekomposition von Arbeitsprozessen beschrieben (vgl. S. 23). Abschließend zum Stand der Forschung betrachten wir kurz weitere solcher Ansätze.

Den Beginn unternahmen Dongen et al. mit dem Aufstellen zweier Relationen, die kausale Fußabdrücke genannt werden [83]: Eine der Relationen sagt für einen Knoten aus, dass bei Ausführung dieses Knotens mindestens einer der Knoten, mit dem dieser in Relation steht, ebenfalls ausgeführt wird. Die andere Relation wiederum sagt aus, dass beim Ausführen dieser Knoten mindestens einer der Knoten, mit denen er in Relation steht, vorher ausgeführt worden sein muss. Aufgrund dieser kausalen Fußabdrücke konnten drei fehlerhafte Muster für Verklemmungen, Überflüsse und sogenannte Fallstricke gefunden werden. Dabei stellen Fallstricke Endlosschleifen über Forkknoten dar. Leider kann auch nur im Falle der Fallstricke gezeigt werden, dass der Prozess inkorrekt ist. Für die anderen beiden Patterns kann nur ausgesagt werden, dass dort potentiell eine Verklemmung bzw. ein Überfluss vorliegt.

Ebenso über Patterns und Relationen hat Favre seinen Algorithmus zur Fehlerfindung in azyklischen Workflowgraphen entwickelt [21]. In seinem Ansatz definiert er eine Immer-Parallelitätsrelation zwischen Kanten, die er mit einer Art Datenflusssystem ermittelt, in dem er die Informationen der Relationen über den Workflowgraphen propagiert. Über diese Informationen kann er dann Aussagen treffen, ob zwei Kanten immer parallel eine Marke tragen oder nicht. Gilt in diesem Zusammenhang für die eingehenden Kanten eines Joinknotens, dass diese immer parallel sind, so kann der Joinknoten immer ausgeführt werden. Andernfalls kann eine Verklemmung im Joinknoten auftreten.

[2] `http://is.ieis.tue.nl/staff/heshuis/DiagFlow/` (Mai 2017)

Für die Identifikation von Überflüssen definiert er analog eine Manchmal-Parallelitätsrelation. Dabei dürfen zwei eingehende Kanten eines Mergeknotens niemals in Relation stehen, denn sonst gibt es die Möglichkeit für einen Überfluss.

Der Ansatz ist sehr gut, liefert ansprechende diagnostische Informationen und kann in polynomieller Laufzeit durchgeführt werden. Jedoch ist die Beschränkung auf azyklische Workflowgraphen eine zu starke Einschränkung und das Finden von Fehlern hinter bereits aufgetretenen Fehlern ist nur bedingt möglich.

Favre et al. schlugen noch einen weiteren Ansatz zum Korrektheitsnachweis auf Basis von Gegenmustern vor [22]. Diese Muster sind ähnlich zu den Mustern von Dongens et al. [83] und können auch auf Prozessen mit Schleifen angewandt werden. Dabei weisen Favre et al. nach, dass bei Vorhandensein eines dieser Patterns innerhalb eines Prozesses, dieser nicht korrekt sein kann. Wurde diese Inkorrektheit nachgewiesen, so wird eine Fehlerdiagnose gestartet. Diese Fehlerdiagnose findet den zum Fehler gehörenden, fehlerbehafteten Teilgraphen des Prozesses, der als diagnostische Information verwendet wird. Dieser Teilgraph wird unter anderem wieder mit Hilfe des Rangssatzes bestimmt.

Das Laufzeitverhalten des Ansatzes von Favre et al. ist quintisch [22], liefert dafür aber sehr gute diagnostische Information zu einem gefundenen Fehler. Dies ist aber auch von Nachteil, da für die Findung lediglich eines Fehlers pro Prozess die Laufzeit zu langsam ist. Mit einem Veröffentlichungsdatum von 2016 zeigt diese Arbeit jedoch, dass ein detaillierter Korrektheitsnachweis immer noch hohe Relevanz besitzt.

Kapitel 4

Wissenschaftlicher Beitrag und Thesen

Die einschlägigen und insbesondere die aktuellen Beiträge in der Literatur ergaben offene Probleme im Bereich der Korrektheitsanalyse von Arbeitsprozessen. In diesem Kapitel werden diese Probleme benannt und beschrieben. Darauf aufbauend positionieren wir anschließend unseren wissenschaftlichen Beitrag zur Lösung ausgewählter Probleme. Des Weiteren werfen wir Thesen auf, deren Gültigkeit mit dieser Arbeit bestätigt werden sollen.

4.1 Offene Probleme

Die meisten der im Jahr 2007 existierenden Kontrollflussanalysen zur Korrektheitsüberprüfung sind entweder effizient oder liefern diagnostische Informationen, aber eben nicht beides [85]. Dieses Problem griff Favre in seiner Dissertationsschrift 2014 auf und formulierte daraus ein konkretes offenes Problem: Zu diesem Zeitpunkt gab es keine Technik, die in Polynomialzeit eine vollständige Korrektheitsanalyse durchführt und gleichzeitig diagnostische Informationen liefert [19, S. 9].

Dieses Problem konnte Favre lösen. Das Result dieser Arbeit sind zwei neue Techniken: Die erste Technik basiert auf allgemeinen, die zweite auf azyklischen Arbeitsprozessen. Letztere akzeptiert zudem noch Or-Joinknoten [22].

Nach dem Stand der Forschung können wir die berechtigte Annahme treffen, dass Favres Techniken die wohl derzeit effizientesten darstellen, die gleichzeitig diagnostische Informationen liefern. Für diese identifizieren wir jedoch noch wesentliche Schwächen:

1. *Qualität und Quantität der Fehlerfindung.* Sie liefern Fehler*wirkungen* ausgehend vom Startzustand anstelle der Fehler*ursachen*. Für die Korrektur eines Prozesses ist es aber notwendig, dass der Modellierer einen vollständigen Über-

blick der im Prozess befindlichen Fehlerursachen erhält. Dies umfasst unter anderem:

   (a) Die Findung der *Ursachen* von Kontrollflussfehlern anstelle derer Fehler*wirkungen* (als Verklemmung oder Überfluss).

   (b) Die Findung der Ursachen von Kontrollflussfehlern trotz Fehlerblockierungen, -maskierungen und -täuschungen [57].

   (c) Die Bereitstellung diagnostischer Informationen zu den Fehlerursachen.

2. *Die Limitierung.* Für bestimmte Prozesse ergeben sich aus den Techniken gesonderte Anwendungsfälle. Der allgemeingültige Ansatz kommt zum Einsatz für zyklische Prozesse, die keine Or-Joinknoten beinhalten dürfen. Der spezielle Ansatz wird genutzt, sobald der Prozess Or-Joinknoten besitzt, jedoch azyklisch ist. Dadurch ist die Anwendbarkeit beider Techniken auf eine limitierte Menge von Prozessen beschränkt (jene, die sich in Free-Choice-Workflownetze übertragen lassen, und jene, die azyklisch sind [20]). Unserer Ansicht nach sollte eine Technik allgemeingültig sein oder das Potential zur Erweiterung auf andere Anwendungsfälle besitzen.

3. *Das Laufzeitverhalten.* Die Technik für allgemeine Prozesse hat ein quintisches Laufzeitverhalten. Inklusive der Qualität und Quantität der gewonnenen Fehlerinformationen ist diese Technik jedoch als unterstützendes Analyseverfahren aufgrund seiner Laufzeit direkt während der Konstruktion eines Prozesses unbrauchbar. Dies zeigen die Erfahrungen aus dem Übersetzerbau.

Als Hauptkritik an den Techniken erachten wir die Qualität und Quantität der Fehlerfindung. Wie wir bereits in der Einleitung dieser Arbeit argumentiert haben, untersuchen die gängigsten Techniken (wie die Anwendung des Rangsatzes, die Zustandsraumerkundung und Favres Techniken) die Fehler*wirkungen* innerhalb eines Prozesses. Wie aus dem Bereich des Qualitätsmanagements der Softwareentwicklung, genauer des Softwaretestens, bekannt ist, können die eigentlichen *Ursachen* aus den Fehlerwirkungen nur schwer, wenn sogar manchmal überhaupt nicht gefunden werden. Finden wir beispielsweise eine Verklemmung in einem Prozess, so muss diese Fehlerwirkung nicht zwangsläufig aus einem fehlenden Forkknoten resultieren. Sie kann auch aus der Ursache eines Überflusses entstehen. Solch eine Ursache auf Grundlage eines abgeleiteten Zustands zu erkennen, ist sehr schwer. Nahezu unmöglich wird die Ursachenableitung in großen Prozessen, wenn die in der Einleitung beschriebenen Effekte der Fehlerblockierung, -maskierung und -täuschung auftreten. Nähere Ausführungen zu diesem Thema können aus [57] entnommen werden.

Wir fassen diese Ausführungen zur Qualität und Quantität der Fehlerfindung in folgendem offenen Problem zusammen:

Offenes Problem 1 (Ursachen der Fehler)**.**

Es gibt derzeit keine Technik, welche hinsichtlich der Korrektheit vollständig die Ursachen von Kontrollflussfehlern, d. h. Verklemmungen und Überflüssen, in einem zyklischen Workflowgraphen findet und diese Ursachen detailliert als diagnostische Informationen zur Verfügung stellen kann.

Neben der Qualität und Quantität der Fehlerfindung ist auch das Laufzeitverhalten derzeitiger Analysen, welche gleichzeitig diagnostische Informationen liefern, nicht für den Einsatz als unterstützendes Analyseverfahren direkt während der Konstruktion eines Prozesses geeignet. Ein Support, wie in modernen integrierten Entwicklungsumgebungen für Programmiersprachen, ist so nicht möglich. Wir formulieren dies als ein Problem der Effizienz:

Offenes Problem 2 (Effizienz)**.**

Es gibt derzeit keine Technik, welche den Korrektheitsnachweis inklusive diagnostischer Informationen als unterstützendes Analyseverfahren (wie dessen Durchführung in jedem Konstruktionsschritt eines Prozesses) erlaubt. Für einen solchen Einsatz für praktisch relevante Prozesse sind Algorithmen mit einem kubischen Laufzeitverhalten hinsichtlich der Prozessgröße gerade noch akzeptabel.

Ein weiteres offenes Problem des Stands der Forschung ergibt sich aus der Einschränkung der untersuchbaren Prozesse: Einige Techniken beschränken sich auf azyklische Prozesse, andere wiederum auf wohlgeformte. Wieder andere Techniken besitzen als Limitierung Free-Choice-Workflownetze, so dass nicht jeder Workflowgraph mit Or-Joinknoten untersucht werden kann [20]. Diese Limitierung ist ein offenes Problem. Auch wenn wir dieses Problem in dieser Arbeit nicht aufgreifen, sind wir überzeugt, dass bspw. Workflowgraphen mit Or-Joinknoten auch mit unseren Techniken untersucht werden können. Dazu kann auch unsere Einführung einer ersten vollständigen Semantik für Or-Joinknoten helfen [58].

Wir formulieren das offene Probleme der Limitierung:

Offenes Problem 3 (Limitierung)**.**

Es gibt derzeit keine Technik, welche eine Kontrollflussanalyse in *zyklischen* Prozessen inklusive Or-Joinknoten oder anderer komplexer Kontrollflusselemente erlaubt.

Das letzte große offene Problem des Stands der Forschung ist das Auftreten sogenannter falsch-positiver [70] und falsch-negativer [3] Analyseergebnisse [19][26]. Diese treten aufgrund der Ignorierung datenbasierter Entscheidungen auf. Dies sehen wir als ein allgemeines Problem der statischen Analyse an. Es ist weiterhin ein offenes Problem und wird im Rahmen dieser Arbeit nicht betrachtet:

Offenes Problem 4 (Falsch-positive und falsch-negative Analyseergebnisse)**.** Falsch-positive und falsch-negative Analyseergebnisse entstehen durch die Abstraktion datenbasierter Entscheidungen.

4.2 Wissenschaftlicher Beitrag

Die vorliegende Arbeit behandelt zwei neue Techniken zur übersetzerbasierten Kontrollflussanalyse von Prozessen (im Sinne von Workflowgraphen). Diese beiden Techniken untersuchen zum *ersten* Mal die *Ursachen* von Verklemmungen und Überflüssen. Dadurch entstehen neue Erkenntnisse zu diesen Fehlerwirkungen, wodurch sie sich wesentlich besser korrigieren lassen. Zudem sind beide Techniken effizient und liefern im Normalfall in kubischer sowie im schlechtesten Fall in biquadratischer asymptotischer Laufzeit hinsichtlich der Anzahl der Kanten die notwendigen Ursachen *potentieller Fehler*.

Durch die erste vorgestellte Analysetechnik können die Ursachen von Verklemmungen erkannt werden. Dazu betrachtet die Analyse aufgrund der Fehlertäuschungen, -blockierungen und -maskierungen ausschließlich Verklemmungen, die nicht durch andere Fehlerwirkungen zustande kommen. Solche Verklemmungen heißen *unmittelbar*. Um unmittelbare Verklemmungen zu vermeiden, muss bei Erreichen einer Marke bei einem Joinknoten *garantiert* sein, dass dieser ausgeführt wird. Diese Garantie erhält die Analysetechnik durch die Verwendung einer *Aktivierungsrelation*.

Die zweite Technik bestimmt die Ursachen von Überflüssen. Überflüsse können genau dann auftreten, wenn zwei Kontrollflüsse an den Stellen, wo sie zum ersten Mal aufeinandertreffen können, nicht synchronisiert werden. Diese Stellen heißen *Treffpunkte*. Um die Ursachen von Überflüssen zu erkennen, weisen wir nach, wie zwei Kontrollflüsse *unabhängig* von einem Forkknoten einen nicht-synchronisierenden Treffpunkt erreichen können.

Auf Basis dieser beiden Techniken überprüfen wir, dass übersetzerbasierte Kontrollflussanalysen als unterstützende Analyseverfahren direkt während der Konstruktion geeignet sind. Dazu implementierten wir unsere Techniken in einem eigenen Werkzeug *Mojo*. Dieses integrierten wir anschließend in einem Prozessdesigner. Dieser Prozessdesigner führt nun nach jedem Konstruktionsschritt unsere Analysen durch. Gefundene Fehlerursachen werden ohne wahrnehmbare Latenz direkt in den Prozess projiziert. Ein weiterer Test mit einer großen Prozessbibliothek zeigt, dass Mojo zuverlässiger Kontrollflussfehler findet als das zustandsraumbasierte Analysewerkzeug LoLA. Zudem ist es für diese Testfälle effizienter. Die Tests zeigen, dass unsere Analysetechniken in der Praxis eine maximal quadratisch asymptotische Laufzeit besitzen.

4.3 Thesen

Zusammengefasst belegt die vorliegende Arbeit die Gültigkeit der folgenden Thesen:

1. Korrektheit kann über die Abwesenheit der *Ursachen* anstelle der Wirkungen von Verklemmungen und Überflüssen charakterisiert werden.
2. Typische Begriffe und Algorithmen aus dem Übersetzerbau eignen sich qualitativ und quantitativ mindestens genauso gut zur Kontrollflussanalyse von Workflowgraphen, wie typischerweise eingesetzte, petrinetzbasierte Ansätze.
3. Eine partielle Zustandsraumanalyse ist möglich.
4. Es existieren mindestens zwei Algorithmen als unterstützende Analyseverfahren, welche die Ursachen von Verklemmungen und Überflüssen in Arbeitsprozessen finden und die folgenden qualitativen Eigenschaften besitzen:
   (a) Sie besitzen eine im schlechtesten Fall biquadratische Laufzeit hinsichtlich der Anzahl der Kanten.
   (b) Sie liefern exakte diagnostische Informationen zur Fehlervisualisierung und -beschreibung.
   (c) Sie entdecken die Ursachen von Verklemmungen und Überflüssen ungeachtet dessen, ob bereits Ursachen von Verklemmungen und Überflüssen gefunden wurden.

Kapitel 5

Partielle Analyse von Workflowgraphen

In Kapitel 2 wurden die grundlegenden Begriffe für diese Arbeit eingeführt, darunter auch die Begriffe der Verklemmung und des Überflusses sowie daraus resultierend der Begriff der Korrektheit. Die Korrektheit fordert, dass jedweder vom Startzustand erreichbare Zustand weder Verklemmung ist, noch Überfluss hat. Aufgrund dieser Definition der Korrektheit lassen sich Verklemmungen und Überflusszustände einerseits zwar relativ einfach (wenn auch ineffizient) durch die Erkundung des Zustandsraums finden (wie beispielsweise in Fahland et al. [18] beschrieben) – andererseits sind sie eben aber auch nicht mehr als die bloßen *Auswirkungen* von Fehlern im Entwurf des Workflowgraphen. Im Vergleich zu herkömmlichen Programmen entspricht dies in etwa dem Auftreten eines Fehlers im Verlauf des Programms durch den Speicherzugriff auf einen Nullzeiger. Die eigentliche Ursache des Fehlers, beispielsweise das Zuweisen einer Nullreferenz oder das Fehlen einer Initialisierung des Zeigers, bleibt dadurch vorerst unentdeckt. Gerade aber diese Fehlerursachen im Entwurf sind entscheidend zur Korrektur von Programmen und dadurch eben auch von Workflowgraphen [57]. Sie werden jedoch durch die alleinige Betrachtung des Zustandsraums nicht gefunden. Nach den Thesen aus Kapitel 4 wollen wir aber genau diese Fehlentwürfe im Workflowgraphen entdecken, die zu Verklemmungen und Überflüssen führen.

Eine unserer grundlegenden Ideen für die Findung der Ursachen von Verklemmungen und Überflüssen ist die Untersuchung verschiedener *Eintrittspunkte* zur Analyse von Workflowgraphen, d.h., anstelle die Ausführung eines Workflowgraphen ausschließlich vom Startzustand zu betrachten, begutachten wir ausgehend von *beliebigen* Zuständen partielle Ausführungen des Workflowgraphen, die im Folgenden *Berechnungen* genannt werden. Spezielle Berechnungen – genau jene, die von einer einzigen Marke auf einer einzigen Kante im Workflowgraphen ausgehen – entsprechen zudem dem üblichen Verständnis eines *Kontrollflusses*.

Kontrollflüsse eignen sich hervorragend zur partiellen Analyse von Workflowgraphen aufgrund ihrer minimalen Anforderungen (genau eine Marke auf genau einer Kante). So lässt sich zeigen, dass jeder Kontrollfluss auch im Zustandsraum existiert, sofern die zum Kontrollfluss gehörende Kante auch eine Marke erhalten kann. In diesem Kapitel führen wir die eben erwähnten Kontrollflüsse und Berechnungen ein und zeigen deren Anwendung zur partiellen Analyse von Workflowgraphen.

5.1 Berechnungen

Kindler und van der Aalst [39] beschreiben Berechnungen als die endliche oder unendliche Sequenz direkt hintereinander erreichbarer Zustände beginnend beim Startzustand. Sie stellen im Wesentlichen Teilgraphen des Zustandsraums dar, die eine vollständige, mögliche Ausführung des Workflowgraphen repräsentieren. Wir verallgemeinern diese Definition, so dass Berechnungen von einem beliebigen Zustand starten können. Dadurch erlangen wir kleinere Ausschnitte des Zustandsraums als dies über die Definition von Kindler und van der Aalst möglich wäre.

Definition 5.1 (Berechnung)**.**

Wie nennen die (beliebig lange) Sequenz von direkt erreichbaren Zuständen

$$Z_0 \to Z_1 \to Z_2 \to \ldots \qquad Z_0, Z_1, Z_2, \ldots \in \mathcal{Z}(E) \tag{5.1}$$

eines Workflowgraphen $WFG = (N, E, l)$ eine *Berechnung*, c_{Z_0}, ausgehend vom Zustand Z_0, wenn auf jeden Zustand Z_i, $i \geq 0$, ein von ihm direkt erreichbarer Zustand Z_{i+1} folgt (solange existent). Formal gilt für $c_{Z_0} = (Z_0, Z_1, \ldots)$:

$$\underset{i\geq 0}{\forall}\Big((Z_i \to Z_{i+1}) \quad \vee \quad \big(|c_{Z_0}| = i \wedge \underset{Z\in\mathcal{Z}(E)}{\forall} Z_i \not\to Z\big)\Big) \tag{5.2}$$

Ist die Länge einer Berechnung durch eine Konstante beschränkt, so heißt die Berechnung *endlich*. Andernfalls heißt sie *unendlich*. Wir fassen alle möglichen Berechnungen $c^0_{Z_0}, \ldots, c^m_{Z_0}$, $m \geq 0$, ausgehend von Z_0 unter der Menge $\mathcal{C}_{Z_0} = \{c^0_{Z_0}, \ldots, c^m_{Z_0}\}$ zusammen.

Beispiel 5.2.

Wir betrachten unseren Beispielworkflowgraphen aus Abbildung 5.1, für den die Kanten zur einfacheren Identifikation nun mit Namen versehen sind. Die Ausführung des Workflowgraphen befindet sich gerade im Zustand $[\![k, o]\!]$. Ausgehend von diesem Zustand sind mehrere, sogar beliebig viele, Berechnungen möglich. Eine Berechnung wäre zum Beispiel

$$\begin{aligned}([\![k,o]\!], \quad [\![f,o]\!], \quad [\![g,o]\!], \quad [\![g,p]\!],\\ [\![g,c]\!], \quad [\![g,d]\!], \quad [\![g,e]\!], \quad [\![g,f]\!], \quad [\![g^2]\!]) \in \mathcal{C}_{[\![k,o]\!]}\end{aligned}$$

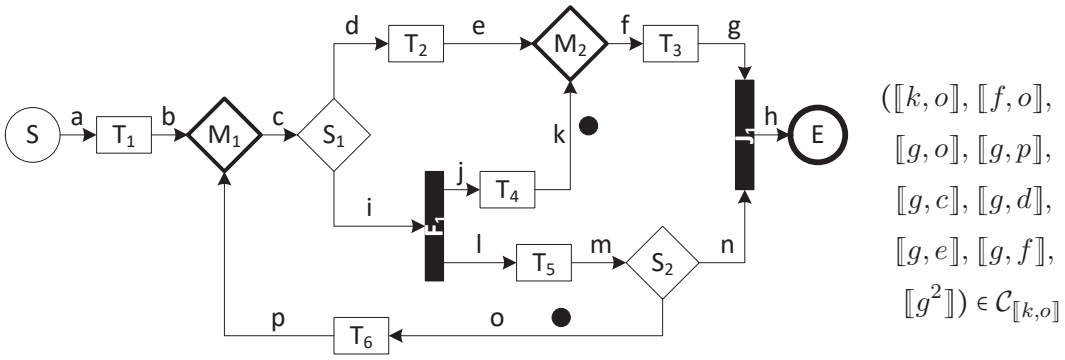


Abbildung 5.1: Ein Workflowgraph mit Berechnung ausgehend vom Zustand $[\![k, o]\!]$

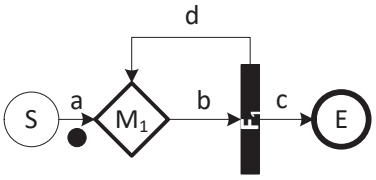


Abbildung 5.2: Ein Workflowgraph mit einer unendlichen Berechnung ausgehend vom Startzustand

Diese Berechnung endet in einer Verklemmung und gleichzeitig mit einem Überfluss auf der Kante g.

Eine andere Berechnung, deren Länge jedoch beliebig lang werden kann, führt die Splitknoten S_1 und S_2 so aus, dass diese stets eine Marke auf die Kante i bzw. o legen. Dadurch wird in jedem Durchlauf der Schleife

$$(i, l, m, o, p, c, i)$$

die Gesamtanzahl aller Marken im Workflowgraphen durch die mehrmalige Ausführung des Forkknotens F_1 um Eins erhöht. Auch wenn die dazugehörige Berechnung beliebig lang werden kann, bleibt sie jedoch stets endlich. Dies liegt daran, dass aufgrund der Fairness S_1 und S_2 irgendwann die Marken nicht mehr auf i und o legen. Dann terminiert die Ausführung oder gerät in einen Verklemmungs- bzw. Überflusszustand.

Abbildung 5.2 zeigt einen anderen Workflowgraphen, der eine unendliche Berechnung ausgehend vom Startzustand aufweist. Die Ausführung dieses Workflowgraphen terminiert nie, da der Forkknoten F_1 in jedem Durchlauf der Schleife (b, d, b) jeweils zwei neue Marken erzeugt, wobei die Marke auf d dann jedes Mal die Schleife erneut ausführt.

Wie im vorangegangenen Beispiel zu sehen war, gibt es Workflowgraphen, die bezüglich ihrer Zustände beliebig viele Berechnungen erzeugen können. Im Idealfall

(wie in jedem korrekten Workflowgraph) ist jedoch die Länge aller Berechnungen bezüglich eines Zustands durch eine Konstante beschränkt.

5.2 Kontrollflüsse

Aufbauend auf dem Begriff der Berechnungen definieren wir Kontrollflüsse als Berechnungen, die von genau einer Marke auf genau einer Kante ausgehen:

Definition 5.3 (Kontrollfluss)**.**

Ein *Kontrollfluss* f_e ab einer Kante e eines Workflowgraphen $WFG = (N, E, l)$ ist eine Berechnung $c_{[\![e]\!]}$ bezüglich des Zustands $[\![e]\!]$.

$$f_e = c_{[\![e]\!]} \tag{5.3}$$

Wir fassen alle möglichen Kontrollflüsse $f_e^0, \ldots, f_e^m$, $m \geq 0$, von e unter der Menge $\mathcal{F}_e = \{f_e^0, \ldots, f_e^m\} = \mathcal{C}_{[\![e]\!]}$ zusammen.

Beispiel 5.4.

Bezüglich des Workflowgraphen aus Abbildung 5.1 bildet die Sequenz

$$([\![k]\!]\ ,\ [\![f]\!]\ ,\ [\![g]\!]) = f_k$$

von Zuständen einen Kontrollfluss ab der Kante k. f_k endet im Zustand $[\![g]\!]$, da dieser ein Verklemmungszustand bzgl. der Kante k ist. Aufgrund des Determinismus dieser Zustandsübergänge ist f_k auch der einzige Kontrollfluss der Kante k.

$$\{f_k\} = \mathcal{F}_k$$

Ab der Kante o existieren hingegen beliebig viele Kontrollflüsse. Einer davon ist beispielsweise

$$([\![o]\!]\ ,\ [\![p]\!]\ ,\ [\![c]\!]\ ,\ [\![d]\!]\ ,\ [\![e]\!]\ ,\ [\![f]\!]\ ,\ [\![g]\!]) = f_o$$

Ein Kontrollfluss f_e ab einer Kante e eines Workflowgraphen beschreibt damit eine Auswahl möglicher Zustände, die aus einer einzigen Marke auf e resultieren können. Somit beschreibt der Kontrollfluss informell auch, auf welche anderen Kanten die Marke von e gelangen kann. Alle Kanten, auf welche die Marke von e im Kontrollfluss f_e gelangen kann, ist durch die Menge

$$\mathcal{E}dgesIn(f_e) = \bigcup_{Z \in f_e} Z \tag{5.4}$$

gegeben. Gelangt also eine Marke auf eine Kante a im Kontrollfluss f_e, so können wir dies auch mit $a \in \mathcal{E}dgesIn(f_e)$ ausdrücken. Gleichzeitig bedeutet dies aber auch, dass die Marke von e zu a wandern kann. Dann muss selbstverständlich auch ein Weg von e nach a existieren. Dies beschreibt die folgende Bemerkung.

Bemerkung 5.5 (Wegexistenz bei Kontrollfluss)**.**

Sei a eine in einem Kontrollfluss f_e ab einer Kante e enthaltene Kante innerhalb eines Workflowgraphen $WFG = (N, E, l)$. Dann folgt offenbar:

$$a \in \mathcal{E}dgesIn(f_e) \qquad \Longrightarrow \qquad \mathcal{W}_{e \to a} \neq \varnothing \tag{5.5}$$

Beispiel 5.6.

Die Marke auf der Kante k des Workflowgraphen aus Abbildung 5.1 muss zwangsläufig über einen Weg von der Startkante a bis zur Kante e gewandert sein (beispielsweise über (a, b, c, i, j, k)).

Tatsächlich können wir über solch einen Weg zwischen zwei Kanten erfreulicherweise auch immer abschätzen, ob ein Kontrollfluss existiert, in dem eine Marke über den betrachteten Weg von der einen Kante zur anderen Kante wandert (und möglicherweise danach noch weiter). Der folgende Satz weist nach, dass bei Existenz eines Weges zwischen zwei Kanten ebenso auch ein Kontrollfluss existiert, in dem zumindest eine Marke von der einen zur anderen Kante fließt, wenn nicht sogar weiter. Anderenfalls muss eine Verklemmung dies verhindern.

Satz 5.7 (Kontrollfluss bei Weg)**.**

Zwei Kanten $e, a \in E, e \neq a$, sind für einen Workflowgraphen (N, E, l) gegeben.

$$\mathcal{W}_{e \to a} \neq \varnothing \wedge W \in \mathcal{W}_{e \to a} \tag{5.6}$$

$$\Longrightarrow$$

$$\underset{f_e \in \mathcal{F}_e}{\exists} \Big(\underset{c \in W}{\forall} c \in \mathcal{E}dgesIn(f_e) \vee \tag{5.7}$$

$$\underset{Z \in f_e}{\exists} Z \text{ ist Verklemmungszustand} \Big) \tag{5.8}$$

Beweis (Satz 5.7). Wir führen den Beweis durch vollständige Induktion. Dabei gehen wir von der Existenz eines Weges $W_l = (e_0, \ldots, e_l), l \geq 1$, aus, über dessen Länge l wir induzieren möchten. Der Induktionsanfang, $l = 1$, ist trivial. Denn, entweder ist von $[\![e_0]\!]$ ein Zustand erreichbar, in dem e_1 eine Marke trägt und somit ein Kontrollfluss ab e_0 mit e_1 existiert. Oder aber es ist kein Zustand von $[\![e_0]\!]$ erreichbar, in dem e_1 eine Marke trägt. Dann muss jedoch der Kontrollfluss ab e_0 bereits in $[\![e_0]\!]$ verklemmen.

Dadurch kommen wir bereits zum Induktionsschritt, l; das heißt, wir gehen davon aus, dass der Satz für alle Wege der Länge l gilt. Damit muss er auch für die Länge $l + 1$ gelten. Es gibt nun genau zwei Fälle für den Weg W_l und einem Kontrollfluss f_{e_0} laut Satz:

1. Fall f_{e_0} beinhaltet einen Verklemmungszustand. Demzufolge endet dieser auch für $l + 1$ in einem Verklemmungszustand. ✓

2. Fall f_{e_0} beinhaltet alle Kanten des Weges W_l. Nehmen wir an $Z_l \in f_{e_0}$ sei der Zustand, der e_l beinhaltet. Für Z_l kann nun wiederum gelten:

1. Es gibt einen von Z_l erreichbaren Zustand Z_{l+1}, in dem e_{l+1} eine Marke trägt. Somit gibt es selbstverständlich einen Kontrollfluss ab e_0 der alle Kanten aus W_{l+1} beinhaltet. ✓
2. Es gibt keinen von Z_l erreichbaren Zustand, in dem e_{l+1} eine Marke trägt. Damit endet f_{e_0} in einer Verklemmung, da die Marke auf e_l verweilt. ✓

□

Selbstverständlich muss eine Marke nicht immer von einer Kante e zu einer Kante a wandern, auch wenn es einen Weg zwischen ihnen gibt. So können Entscheidungen von Splitknoten oder unterschiedliche Ausführungsreihenfolgen der Knoten dazu führen, dass a vom Kontrollfluss ab e *nicht* erreicht wird. Für jede dieser Ausführungsmöglichkeiten gibt es also einen eigenen Kontrollfluss ab e.

5.3 Partielle Zustandsraumerkundung

Kontrollflüsse eignen sich für eine *lokale* (partielle) Untersuchung des Verhaltens eines Workflowgraphen. Durch die explizite Annahme genau einer Marke auf genau einer Kante wird simuliert, welche (Teil-)Zustände erreicht werden können, sobald bei der Ausführung gerade diese Kante eine Marke erhält. Die Betrachtung einer einzelnen Kante und die daraus resultierenden Kontrollflüsse sind derart isoliert, dass sie sich hervorragend als Eintrittspunkte für weitergehende Analysen eignen. Unter *Eintrittspunkten* verstehen wir demnach eine beliebige Kante, auf der wir uns genau eine Marke vorstellen.

Die Betrachtung einer einzelnen Kante e als Eintrittspunkt hat den Vorteil, dass seine Kontrollflüsse tatsächlich auch Auszüge von Berechnungen beschreiben, sofern e in diesen eine Marke erhält. Dies beschreibt der folgende Satz:

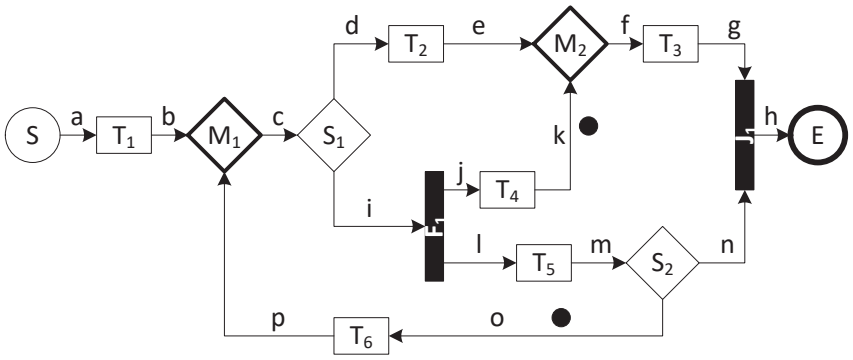
Satz 5.8.

Sei eine Kante $e \in E$ (der Eintrittspunkt) für einen Workflowgraphen $WFG = (N, E, l)$ in einem Zustand $Z, e \in Z$, gegeben.

$$\text{sei } f_e \in \mathcal{F}_e \tag{5.9}$$

$$\Longrightarrow$$

$$\exists_{c_Z \in \mathcal{C}_Z} \; \forall_{Z_e \in f_e} \; \exists_{Z_c \in c_Z} \; Z_e \subseteq Z_c \tag{5.10}$$



$$([\![k]\!]\,,\,[\![f]\!]\,,\,[\![g]\!]) = f_k$$

$$([\![\boldsymbol{k},o]\!],\quad [\![\boldsymbol{f},o]\!],\quad [\![\boldsymbol{g},o]\!],\quad [\![\boldsymbol{g},p]\!],\quad [\![\boldsymbol{g},c]\!],\quad [\![\boldsymbol{g},d]\!],\quad [\![\boldsymbol{g},e]\!],\quad [\![\boldsymbol{g},f]\!],\quad [\![\boldsymbol{g}^2]\!]) = c_Z$$

Abbildung 5.3: Unser Beispielworkflowgraph mit einem Kontrollfluss von k und einer Berechnung, die den Kontrollfluss beinhaltet.

Beweis (Satz 5.8). Sei ein Kontrollfluss $(Z_0, Z_1, Z_2, \ldots) = f_e \in \mathcal{F}_e$ ab der Kante e gegeben. Ist f_e unendlich, dann ergibt sich eine Berechnung, die den Kontrollfluss beinhaltet, einfach durch die Vereinigung jedes Zustands des Kontrollflusses mit dem Zustand Z:

$$(Z_0 \cup Z, Z_1 \cup Z, Z_2 \cup Z, \ldots) \in \mathcal{C}_Z$$

Ist f_e jedoch endlich und endet im Zustand Z_l, $l \in \mathbb{N}$, dann ergibt sich die Berechnung ähnlich, nur dass wir an ihr Ende noch eine Berechnung ab dem Zustand $Z_l \cup Z$ anhängen müssen, damit die Definition einer Berechnung erfüllt ist:

$$\big((Z_0 \cup Z, Z_1 \cup Z, Z_2 \cup Z, \ldots) + c_{Z_l \cup Z}\big) \in \mathcal{C}_Z, \quad c_{Z_l \cup Z} \in \mathcal{C}_{Z_l \cup Z}$$

□

Beispiel 5.9.

Abbildung 5.3 zeigt unseren Beispielworkflowgraphen im Zustand $[\![k, o]\!] = Z$. Für die Kante k mit einer Marke in Z existiert nach den vorangegangenen Beispielen der Kontrollfluss:

$$([\![k]\!]\,,\,[\![f]\!]\,,\,[\![g]\!]) = f_k$$

Aufgrund des letzten Satzes gibt es mindestens eine Berechnung $c_Z \in \mathcal{C}_Z$, die diesen Kontrollfluss beinhaltet. Ein Beispiel für c_Z ist:

$$([\![\boldsymbol{k},o]\!],\quad [\![\boldsymbol{f},o]\!],\quad [\![\boldsymbol{g},o]\!],\quad [\![\boldsymbol{g},p]\!],$$
$$[\![\boldsymbol{g},c]\!],\quad [\![\boldsymbol{g},d]\!],\quad [\![\boldsymbol{g},e]\!],\quad [\![\boldsymbol{g},f]\!],\quad [\![\boldsymbol{g}^2]\!]) = c_Z$$

Wenn jeder Zustand eines Kontrollflusses ein Teilzustand eines Zustands einer Berechnung ist, dann sagen wir vereinfacht, dass die Berechnung den Kontrollfluss

beinhaltet. Beispielsweise beinhaltet die Berechnung c_Z aus Abbildung 5.3 den Kontrollfluss f_k aus selbiger Abbildung. Formal beinhaltet eine Berechnung $c_Z \in \mathcal{C}_Z$ ausgehend vom Zustand Z eines Workflowgraphen $WFG = (N, E, l)$ einen Kontrollfluss $f_e \in \mathcal{F}_e, e \in E$, (geschrieben $f_e \; flowsIn \; c_Z$), wenn es für jeden Zustand $Z_{f_e} \in f_e$ des Kontrollflusses einen Zustand $Z_{c_Z} \in c_Z$ in der Berechnung gibt, der Z_{f_e} beinhaltet:

$$f_e \; flowsIn \; c_Z \iff \underset{Z_{f_e} \in f_e}{\forall} \; \underset{Z_{c_Z} \in c_Z}{\exists} \; Z_{f_e} \subseteq Z_{c_Z} \tag{5.11}$$

Für eine von einem Zustand Z ausgehende Berechnung c_Z existiert für jede Kante $e \in Z$ auch ein Kontrollfluss f_e, den die Berechnung beinhaltet, $f_e \; flowsIn \; c_Z$. Dies beschreibt der folgende Satz und schließt damit die Wechselbeziehungen zwischen Berechnungen und Kontrollflüssen ab:

Satz 5.10.

Für einen Workflowgraphen (N, E, l) sei ein Zustand $Z \in \mathcal{Z}(E)$ gegeben.

$$c_Z \in \mathcal{C}_Z \tag{5.12}$$

$$\Longrightarrow$$

$$\underset{e \in Z}{\forall} \; \underset{f_e \in \mathcal{F}_e}{\exists} \; f_e \; flowsIn \; c_Z \tag{5.13}$$

Beweis (Theorem 5.10). Wir nehmen für eine beliebige Kante $e \in Z$ eine größtmögliche (möglicherweise endlose) Sequenz von direkt erreichbaren Zuständen $(Z_0, \ldots, Z_l)$, $Z_0 = [\![e]\!]$, an, die in c_Z vertreten ist ($\underset{0 \leq i \leq l}{\forall} \; \underset{Z \in c_Z}{\exists} \; Z_i \subseteq Z$). Stellt diese Sequenz *keinen* Kontrollfluss ab e dar, dann muss es von Z_l einen direkt erreichbaren Zustand Z_{l+1} geben. Somit gibt es in Z_l einen ausführbaren Knoten n, nach dessen Ausführung in Z_{l+1} übergegangen wird. Dieser Knoten n muss aber natürlich auch in jedem Zustand $Z' \in c_Z$, $Z_l \subseteq Z'$, ausführbar sein. Dann aber haben wir offenbar keine größtmögliche Sequenz gefunden. Damit muss diese Sequenz einen Kontrollfluss darstellen. □

Beispiel 5.11.

Nehmen wir die Berechnung

$$\begin{aligned}(&[\![k,o]\!], \quad [\![f,o]\!], \quad [\![g,o]\!], \quad [\![g,p]\!],\\ &[\![g,c]\!], \quad [\![g,d]\!], \quad [\![g,e]\!], \quad [\![g,f]\!], \quad [\![g^2]\!]) = c_Z\end{aligned}$$

aus dem letzten Beispiel 5.9 für den Zustand $Z = [\![k,o]\!]$ des Workflowgraphen aus Abbildung 5.3 an. Ein Kontrollfluss $f_k \in \mathcal{F}_k$, welcher in c_Z enthalten ist, bildet:

$$f_k = ([\![k]\!] \, , \, [\![f]\!] \, , \, [\![g]\!])$$

Der Kontrollfluss, der von der Kante o in der Berechnung beinhaltet ist, ist:

$$([\![o]\!] \, , \, [\![p]\!] \, , \, [\![c]\!] \, , \, [\![d]\!] \, , \, [\![e]\!] \, , \, [\![f]\!] \, , \, [\![g]\!]) = f_o$$

Die letzten beiden Sätze machen es möglich, unsere Analysen an beliebigen Stellen des Workflowgraphen zu beginnen. Damit ist es auch möglich, Ursachen von Fehlern hinter anderen Fehlern zu finden.

Diese Errungenschaft verdanken wir der Betrachtung einer einzelnen Kante als Eintrittspunkt e, auf der wir uns eine Marke vorstellen. Die aus dieser Situation resultierenden Kontrollflüsse können nach dem Satz 5.8 Teil von jeder Berechnung sein, in welcher der gewählte Eintrittspunkt e eine Marke bekommt. Wenn also e im Zustand Z eine Marke bekommt, so liegt laut Satz 5.10 auch in jeder Berechnung ab Z mindestens ein Kontrollfluss ausgehend von e.

Finden wir also ab einem Eintrittspunkt e in einer Analyse einen erreichbaren Überflusszustand, so ist dieser Überflusszustand auch Teil mindestens einer Berechnung ausgehend von jedem Zustand, in dem e eine Marke trägt. Erhält e in keiner Ausführung des Workflowgraphen eine Marke, so muss es vorher bereits eine Verklemmung gegeben haben. Es wird demnach ein Überflusszustand nach einer Verklemmung gefunden.

Finden wir hingegen ab einem Eintrittspunkt e in einer Analyse eine Verklemmung, so muss sich diese leider in *keiner* Berechnung ab einem Zustand Z, $e \in Z$, manifestieren. Dies kann deswegen geschehen, weil andere Marken in Z die Verklemmung verhindern. An dieser Stelle hängt die Verklemmungsanalyse also von einer geeigneten Auswahl von Eintrittspunkten ab, wie wir später zeigen werden.

Zusammengefasst können Fehler hinter Fehlern gefunden werden, weil verschiedene Eintrittspunkte für Analysen gewählt werden können. Da die von diesen Eintrittspunkten gefundenen Fehlersituationen jedoch nicht zur Laufzeit auftreten müssen, da eine vorhergehende Verklemmung das Erreichen einer Marke auf den Eintrittspunkten verhindert, sprechen wir von *potentiellen* Fehlern. Da es jedoch immer eine erste erreichbare Verklemmung ausgehend vom Startzustand gibt, sind wir in jedem Fall vollständig hinsichtlich des Korrektheitsnachweises.

Definition 5.12 (Potentielle Fehlerzustände)**.**

Jeder Fehlerzustand Z eines Workflowgraphen $WFG = (N, E, l)$, der durch einen Kontrollfluss ab einem Eintrittspunkt $e \in E$ erreicht werden kann, nennen wir einen *potentiellen* Fehlerzustand von WFG bzgl. e.

Kapitel 6

Ursachen von Verklemmungen

Ein Workflowgraph ist inkorrekt, wenn vom Startzustand eine Verklemmung erreichbar ist. All diese Verklemmungen zu finden, ist durch eine Untersuchung des Zustandsraums und somit auch durch partielle Analysen möglich. Solche Zustandsraumerkundungen sind aber sehr zeitintensiv.

Unser Ziel ist es, zeitnah Ergebnisse zu liefern. Aus diesem Grund *verzichten* wir auf das Bestimmen *aller erreichbaren Verklemmungen* vom Startzustand. Stattdessen bestimmen wir die *Ursachen* der Verklemmungen, denn aus einer einzigen Ursache können eine Vielzahl von Verklemmungen abgeleitet werden. Außerdem lassen sich durch die Beseitigung der Ursachen auch die Verklemmungen verhindern.

Die Ursachen von Verklemmungen lassen sich im Grunde auf zwei Fälle reduzieren: (1) Einen Joinknoten erreichen zu wenig oder (2) zu viele Marken. Im ersten Fall kann es sein, dass (a) nicht genügend Marken während der Ausführung zur Verfügung stehen oder (b) andere Verklemmungen das Ankommen von Marken verhindern. Erreichen hingegen zu viele Marken einen Joinknoten (2), so kann dieser zwar ausgeführt werden, verklemmt danach aber, weil Marken auf seinen eingehenden Kanten übrig bleiben. Dieser Verklemmung kann also ein Überfluss vorangehen.

Es ist unter genügend großem Aufwand möglich, diese Ursachen von Verklemmungen in einem Workflowgraphen zu bestimmen. Bis auf Fall (1.a), dass nicht genügend Marken zur Verfügung stehen, hängen alle anderen Ursachen jedoch von anderen Fehlerursachen ab. Bestimmen wir also alle Verklemmungen (1.a) sowie alle Überflusssituationen, so erhalten wir implizit die anderen Ursachen von Verklemmungen. Es reicht also für eine schnelle Analyse aus, lediglich die Ursachen von Verklemmungen des Falles (1.a) zu untersuchen. Damit ist es natürlich *nicht* möglich, alle Verklemmungen präzise vorherzusagen.

Während der Betrachtung dieser Ursachen liegt der Fokus stets auf einem einzelnen Joinknoten j. Für diesen Joinknoten soll bestimmt werden, ob für ihn Ursachen für *potentielle* Verklemmungen vorliegen. Die Grundidee ist dabei sehr einfach: Sobald in einem Kontrollfluss eine Marke auf mindestens eine eingehende Kante von j

gelangt, muss *garantiert* sein, dass auf alle anderen eingehenden Kanten von j ebenfalls Marken gelangen. Das heißt, der Kontrollfluss muss einen Punkt im Workflowgraphen passieren, der auf allen eingehenden Kanten von j eine Marke gewährleistet. Diese Punkte nennen wir *Aktivierungskanten*. Ist dies jedoch nicht der Fall, dann ist auch nicht gewährleistet, dass der Joinknoten j in jedem Fall ausgeführt wird und somit kann er potentiell verklemmen.

Wichtig in diesem Zusammenhang ist die Auswahl geeigneter Eintrittspunkte, an denen solche Kontrollflüsse und somit die Ursachenanalysen starten sollten. Dabei hat sich herausgestellt, dass nur genau zwei Eintrittspunkte für jeden Joinknoten j untersucht werden müssen: Die *Startkante* des Workflowgraphen, damit die Zustände betrachtet werden können, *bevor* j ausgeführt wird; und die *ausgehende Kante out* von j, damit die Zustände betrachtet werden, *nachdem* j ausgeführt wurde.

Sind die Eintrittspunkte und die Aktivierungskanten bestimmt, so lässt sich leicht zeigen, dass auf jedem Weg von diesen Eintrittspunkten zu den betrachteten Joinknoten eine Aktivierungskante des Joinknotens liegen muss, um Verklemmungen zu vermeiden.

6.1 Eintrittspunkte und Eintrittsgraph

Der erste Schritt einer Ursachenbestimmung von Verklemmungen für einen einzelnen Joinknoten j ist die Auswahl geeigneter Eintrittspunkte für eine Analyse. Mit Hilfe einer partiellen Analyse ist es entweder nicht oder nur mit hohem Zeitaufwand möglich, jede Verklemmung in j zu bestimmen. Unser Ziel ist es jedoch zeitnahe Analyseergebnisse zu den *Ursachen* von Verklemmungen zu liefern, auch wenn diese nur *potentiell* oder mitunter auch gar nicht während der Ausführung auftreten können. Es hat sich herausgestellt, dass es ausreicht, dafür nur zwei Kanten als Eintrittspunkte zu betrachten: (1) Die Startkante und (2) die ausgehende Kante *out* von j. Ab der Startkante wird der Zustandsraum (partiell) untersucht, *bevor* der Joinknoten j zum ersten Mal ausgeführt wird. Ab der ausgehenden Kante *out* wird der Zustandsraum *nach* der Ausführung von j analysiert. Wie der folgende Satz zeigt, ist der Workflowgraph inkorrekt, wenn von mindestens einem der beiden Eintrittspunkte eine Verklemmung erreichbar ist:

Satz 6.1 (Zwei unabhängige Eintrittspunkte)**.**
Sei ein beliebiger Workflowgraph $WFG = (N, E, l)$ mit seiner Startkante *entry* und ein beliebiger Joinknoten j mit seiner ausgehenden Kante *out* gegeben. Ist in einem Kontrollfluss ausgehend von *entry* oder *out* eine Ver-

klemmung in j möglich, so kann der Workflowgraph nicht korrekt sein:

$$\exists_{f_{entry} \in \mathcal{F}_{entry}} \exists_{Z \in f_{entry}} j \text{ hat Verklemmung in } Z \tag{6.1}$$

$$\vee \exists_{f_{out} \in \mathcal{F}_{out}} \exists_{Z \in f_{out}} j \text{ hat Verklemmung in } Z \tag{6.2}$$

$$\Longrightarrow$$

$$WFG \text{ ist nicht korrekt} \tag{6.3}$$

Beweis (Satz 6.1). Aus dem Satz ergeben sich direkt zwei voneinander unabhängige Fälle: (1) Ab der Startkante *entry oder* (2) ab der ausgehenden Kante *out* vom Joinknoten j gibt es einen Kontrollfluss, der in j verklemmt. Es wird nun behauptet, dass der Workflowgraph WFG in beiden Fällen nicht korrekt sein kann.

Zu (1): Jeder Kontrollfluss ausgehend von *entry* entspricht einer möglichen Ausführung von WFG ab dem Startzustand. Hat solch eine Ausführung ab dem Startzustand eine Verklemmung, so ist WFG per Definition inkorrekt. ✓

Zu (2): Beweis per Kontradiktion: In einem Kontrollfluss f_{out} ausgehend von *out* wird eine Verklemmung Z_{dead} in j erreicht und *dennoch* ist WFG korrekt.

Da WFG korrekt ist, gibt es einen Kontrollfluss ab der Startkante, in dem auf die ausgehende Kante *out* von j eine Marke gelangt. Sei Z_{out} solch ein vom Startzustand erreichbarer Zustand, in dem *out* eine Marke trägt. Dieser Zustand Z_{out} setzt sich aus der Kante *out* und einer Menge *Add* von anderen Kanten zusammen, $Z_{out} = [\![out]\!] \cup Add$. Da nach Annahme

$$[\![entry]\!] \rightarrow^* \big([\![out]\!] \cup Add\big) \rightarrow^* \big(Z_{dead} \cup Add\big)$$

gilt, muss $Add \neq \varnothing$ gelten. Anderenfalls ist die Verklemmung Z_{dead} erreichbar. Sei dazu nun $\mathcal{A}dd$ die Menge aller Kantenmengen *Add*, die zusammen mit *out* vom Startzustand erreichbar sind: $\mathcal{A}dd = \{Add \subseteq E : [\![entry]\!] \rightarrow^* [\![out]\!] \cup Add\}$.

Ausgehend von einem $Add \in \mathcal{A}dd$ können unabhängig von der Kante *out* Berechnungen $\mathcal{C}_{Add}$ betrachtet werden. Da eine unendliche Berechnung nicht im Endzustand $[\![end]\!]$ endet und WFG korrekt ist, ist jede Berechnung $c_{Add} \in \mathcal{C}_{Add}$ endlich. Demzufolge endet jede Berechnung in einem Zustand *Finished*, von dem aus kein Zustand mehr erreichbar ist. Da nun auch

$$[\![entry]\!] \rightarrow^* \big([\![out]\!] \cup Add\big) \rightarrow^* \big([\![out]\!] \cup Finished\big)$$

gilt, ist $Finished \neq \varnothing$ und $Finished \in \mathcal{A}dd$. Die Menge $\mathcal{F}inished = \{Finished \in \mathcal{A}dd : \forall_{Z \in \mathcal{Z}(E)} Finished \not\rightarrow^* Z\}$ beschreibt die Menge aller $Finished \in \mathcal{A}dd$, von denen kein Zustand erreichbar ist.

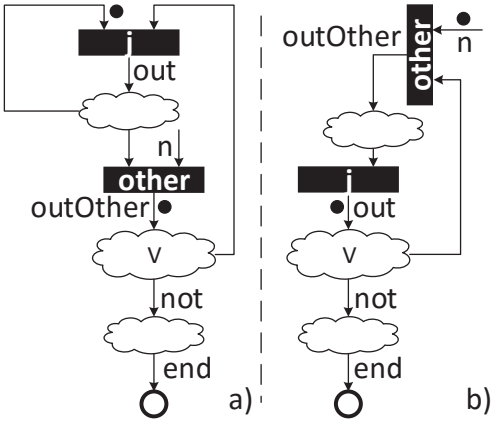


Abbildung 6.1: Verschiedene Fälle

Da von jedem $Finished \in \mathcal{F}inished$ kein Zustand erreichbar ist, ist jede Kante aus $Finished$ entweder eine eingehende Kante eines Joinknotens oder die Endkante end. Ist $end \in Finished$, dann kann von $[\![out]\!] \cup Finished$ der Zustand $[\![out, end]\!]$ erreicht und somit der Endzustand $[\![end]\!]$ nicht erreicht werden. WFG wäre inkorrekt. Demzufolge ist jede Kante aus $Finished \in \mathcal{F}inished$ eine eingehende Kante eines Joinknotens.

Weil ausgehend von jedem $Finished \in \mathcal{F}inished$ kein Zustand erreichbar ist, muss ausgehend von $[\![out]\!] \cup Finished$ die Marke auf out zur Ausführung aller Joinknoten führen, auf deren eingehenden Kanten in $Finished$ Marken liegen.

Sei ein $Finished \in \mathcal{F}inished$ und ein nach der Annahme von $[\![out]\!]$ erreichbarer Verklemmungszustand Z_{dead} in j gegeben. Weiterhin sei $Necessary \subseteq Finished$ eine notwendige nicht-leere Menge an Kanten, damit j ab $Z_{dead} \cup Necessary$ zur Ausführung kommt. Für jede Kante $n \in Necessary$ mit dem dazugehörigen Joinknoten $tgt(n) = other$ und dessen ausgehender Kante $outOther$ gelten abschließend zwei Fälle:

Fall 1: *outOther pdom out*: $outOther$ liegt auf allen Wegen von out zur Endkante end. Von $[\![out]\!] \cup Finished$ ist $Z_{dead} \cup Finished$ erreichbar. Da nun zur Ausführung von j der Joinknoten $other$ ausgeführt werden muss, gibt es einen von $Z_{dead} \cup Finished$ erreichbaren Zustand, in dem $other$ gerade ausgeführt wurde, j aber noch nicht. Dieser Zustand wird abstrakt in Abbildung 6.1 *a)* dargestellt. Es liegt also eine Marke auf der ausgehenden Kante $outOther$ von $other$. Da $outOther$ die Kante out echt postdominiert, gibt es einen Weg von $outOther$ zur Endkante ohne out. Auf diesem Weg gibt es außerdem eine erste Kante not, von der es keinen Weg mehr zu j gibt. Weil $outOther$ aber auch notwendig ist, um j auszuführen, muss es auch von $outOther$ zu j einen Weg geben. Abstrakt

wird diese Verzweigung in der Abbildung durch die Wolke V dargestellt. Diese Wolke kann nun zwei mögliche Semantiken besitzen:

Fall a: Wolke V hat die Semantik eines Splitknotens, das heißt, es gelangt o.B.d.A. entweder eine Marke nach j oder auf die Kante *not*. Gelangt die Kante nach *not*, kann daher die Marke nicht mehr zu j gelangen. Da die Marke der Kante n notwendig zur Ausführung von j ist, verklemmt j nun. $\lightning$

Fall b: V hat die Semantik eines Forkknotens, sprich, nach j und nach *not* gelangt o.B.d.A. jeweils eine Marke. Wird j ausgeführt, kann o.B.d.A. noch eine Marke auf *not* landen und somit einen Überfluss verursachen. $\lightning$

Fall 2: Es gibt einen Weg von *out* zur Endkante, auf dem nicht *outOther* liegt (*outOther* $\not\mathit{pdom}$ *out*). Dieser Fall ist im Zustand $[\![out]\!] \cup Finished$ abstrahiert in Abbildung 6.1 *b)* zu sehen. Es gibt nun einen Weg ohne *outOther* von *out* zur Endkante sowie einen Weg von *out* zu *outOther*. Demnach gibt es eine Verzweigung. Diese Verzweigung wird wieder abstrahiert durch die Wolke V in der Abbildung dargestellt. Wie gut anhand der Abbildungen *a)* und *b)* zu sehen ist, sind beide Situationen ähnlich und es gelten offenbar die selben Fälle *a* und *b*, wie für Fall 1. $\lightning$

Für alle Fälle kann die Ausführung in eine Verklemmung oder einen Überfluss geraten. Der Workflowgraph kann nicht korrekt sein. $\lightning$

□

Durch diesen Satz ist es möglich, von beiden Eintrittspunkten voneinander *unabhängige* partielle Analysen durchzuführen. Wird dabei eine Verklemmung gefunden, dann ist der Workflowgraph inkorrekt. Auch wenn die Ausführung dabei nicht zu hundertprozent in eine Verklemmung gerät, nehmen wir diese Ungenauigkeit für eine schnelle Analyse in Kauf. Unser Werkzeug gibt in solch einem Fall immer nur Warnungen aus, wohlwissend, dass der Workflowgraph trotzdem inkorrekt ist.

Unabhängige Analysen von beiden Eintrittspunkten können durch die einfache Trennung der eingehenden Kanten vom Joinknoten j von seiner ausgehenden Kante erreicht werden. Dazu wird j in einen zweiten Endknoten transformiert und ein neuer Startknoten wird die neue Quelle der ausgehenden Kante *out* (vgl. Abbildung 6.2). Beide Startknoten des Graphen sind demnach die Startknoten der Analysen bzw. deren ausgehende Kanten die Eintrittspunkte.

Wie bereits in der Einleitung zu diesem Kapitel diskutiert, gibt es ausgehend von diesen beiden Eintrittspunkten viele Ursachen, warum ein Joinknoten j potentiell verklemmen kann. So kann die Ursache einer Verklemmung in j eine Verklemmung eines anderen Joinknotens sein. Oder die Verklemmung in j resultiert aus einer Überanzahl an Marken, die auf seinen eingehenden Kanten ankommt. Dabei handelt

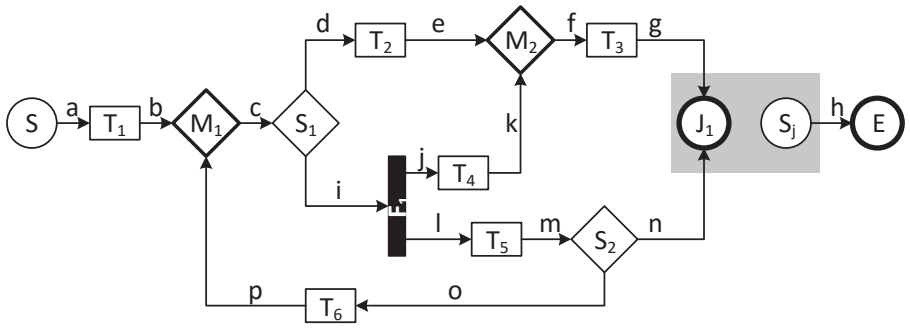


Abbildung 6.2: Trennung des Joinknotens J_1 von seiner ausgehenden Kante

es sich jedoch um Fehlerblockierungen, Fehlertäuschungen und Fehlermaskierungen. Das heißt, für die Betrachtung eines Joinknotens j ist es gut, so zu tun, als ob es *keine* anderen Fehler im Workflowgraphen gibt. Gibt es dementsprechend für keinen anderen Joinknoten eine Verklemmungsursache und ebenso keine Ursache eines Überflusses, so ist das Analyseergebnis bezüglich eines Joinknotens j korrekt.

Um nun eine möglichst ausschließliche Betrachtung von potentiellen Verklemmungen in j unabhängig von anderen Fehlern zu betrachten, wird die Annahme getroffen, dass alle Joinknoten durch einen *neuen* speziellen Knotentyp $Safe$ ersetzt werden. Dessen nicht näher spezifizierte Semantik trägt dafür Sorge, dass im Workflowgraphen keine anderen Verklemmungen auftreten. Den so modifizierten Workflowgraphen nennen wir für den Joinknoten j den *Eintrittsgraphen*:

Definition 6.2 (Eintrittsgraph)**.**

Sei ein Workflowgraph $WFG = (N_W, E_W, l_W)$ angenommen. Ein *Eintrittsgraph* eines Joinknotens $j \in N_{Join}$ mit seiner ausgehenden Kante *out* ist ein beschrifteter Graph $\mathcal{EG}(j) = (N, E, l)$. Er besitzt zu den Knoten aus WFG einen zusätzlichen Startknoten S. Der Kantenmenge von WFG wird die ausgehende Kante *out* entnommen und dafür eine Kante vom zusätzlichen Startknoten S zum Ziel der Kante *out* hinzugefügt. Die Beschriftungen der Joinknoten werden auf $Safe$ gesetzt, die des Joinknotens j auf End und die des zusätzlichen Startknotens auf $Start$. Alle anderen Beschriftungen werden übernommen.

Formal ergeben sich die folgenden Mengen:

1. Die Knotenmenge $N = N_W \cup \{S\}$
2. Die Kantenmenge $E = \big(E_W \setminus \{out\}\big) \cup \Big\{\big(S, tgt(out)\big)\Big\}$
3. Die Beschriftungen $l = l_W \setminus \Big\{(j', Join) : j' \in N_{Join}\Big\}$
   $\cup \Big\{(j', Safe) : j' \in \big(N_{Join} \setminus \{j\}\big)\Big\}$
   $\cup \Big\{(S, Start), (j, End)\Big\}$

Der Eintrittsgraph des Joinknotens J_1 unseres Beispielworkflowgraphen ist in Abbildung 6.2 zu sehen. Insgesamt sind Eintrittsgraphen sehr einfacher Natur und ähneln dabei stark dem ursprünglichen Workflowgraphen. Auch wenn ein Eintrittsgraph *nicht* der Definition eines Workflowgraphen unterliegt, gehen wir dennoch von der gewohnten Semantik der Knoten aus. Wie dann auffällt, entspricht eine Verklemmung in einem Joinknoten j nur noch der Situation, in dem ein Kontrollfluss ab einer der Startkanten mindestens eine aber nicht alle eingehenden Kanten von j mit mindestens einer Marke beliefert. Wir sprechen dann von einer *unmittelbaren* Verklemmung:

Definition 6.3 (Unmittelbare Verklemmung)**.**

Ein Joinknoten j eines Workflowgraphen $WFG = (N, E, l)$ hat eine *unmittelbare* Verklemmung in einem Kontrollfluss f_{entry} ab einer Startkante *entry* innerhalb seines Eintrittsgraphen $\mathcal{EG}(j)$, wenn mindestens eine aber nicht alle eingehenden Kanten von j im Kontrollfluss Marken erhalten:

$$j \text{ hat unmittelbare Verklemmung in } f_{entry} \tag{6.4}$$

$$\Longleftrightarrow$$

$$1 \leq |\triangleright j \cap \mathcal{E}dgesIn(f_{entry})| < |\triangleright j| \tag{6.5}$$

Beispiel 6.4 (Unmittelbare Verklemmung)**.**

Wir betrachten unseren Beispieleintrittsgraphen aus Abbildung 6.3. In diesem ist ein Kontrollfluss der Startkante durch schwarze Marken skizziert, der auf der Kante (T_3,J_1) endet. Damit stoppt der Kontrollfluss in einer *unmittelbaren* Verklemmung im Joinknoten J_1 ab der Startkante (S, T_1).

Hingegen endet der durch die grauen Marken illustrierte Kontrollfluss in einem Zustand, in dem auf der Kante (T_3, J_1) zwei Marken liegen und die Kante (S_2, J_1) eine Marke trägt. Dadurch würde nach Ausführung von J_1 im Workflowgraphen eine Marke auf dessen eingehender Kante verbleiben, die wiederum eine Verklemmung verursacht. Diese Verklemmung in J_1 resultiert dann aber aus der Ursache eines Überflusses und ist nicht unmittelbar.

Unmittelbare Verklemmungen sind unabhängig von bereits zuvor aufgetretenen Fehlern. Werden Ursachen potentieller, unmittelbarer Verklemmungen identifiziert, dann können diese tatsächlich nur dann nicht zur Ausführung in einem Workflowgraphen auftreten, wenn andere Fehler zuvor diese verhindern. So oder so kann der Workflowgraph dann jedoch nicht korrekt sein.

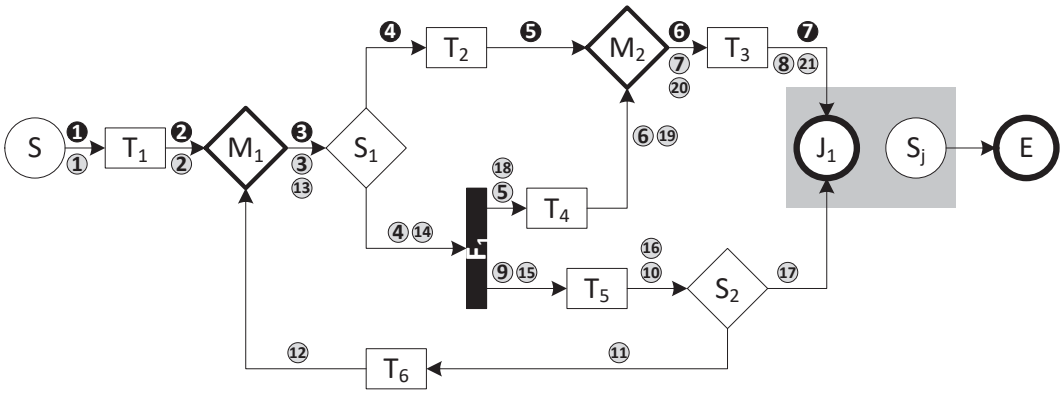


Abbildung 6.3: Eine unmittelbare (schwarze Marken) und eine durch einen Überfluss herbeigeführte Verklemmung (graue Marken)

6.2 Aktivierungskanten und Verklemmungen

Geeignete Eintrittspunkte für eine Analyse von potentiellen Verklemmungen eines Joinknotens j wurden im letzten Abschnitt heraus-, sowie unmittelbare Verklemmungen als ausreichende Fehleridentifikatoren dargestellt. Intuitiv folgt nun im Eintrittsgraphen von j *garantiert* die Ausführung von j ab einer Kante a, wenn in jedem Kontrollfluss ab a jede eingehende Kante von j eine Marke erhält. Wir nennen a dann *Aktivierungskante* der eingehenden Kanten von j und ebenfalls von j selbst. Dies hält die folgende Definition fest:

Definition 6.5 (Aktivierungsrelation und -kante)**.**

Innerhalb des Eintrittsgraphen eines Joinknotens j ist eine Kante a eine *Aktivierungskante* einer eingehenden Kante in von j, wenn es in jedem Kontrollfluss ausgehend von a einen Zustand gibt, in dem in eine Marke trägt. Dafür schreiben wir $a \rightsquigarrow in$:

$$(a, in) \in \rightsquigarrow \iff \underset{f_a \in \mathcal{F}_a}{\forall} in \in \mathcal{E}dgesIn(f_a) \tag{6.6}$$

Ist eine Kante a Aktivierungskante einer jeden eingehenden Kante von j, dann ist a eine Aktivierungskante von j, $a \overset{all}{\rightsquigarrow} j$.

Beispiel 6.6 (Aktivierungsrelation und -kante)**.**

Abbildung 6.4 zeigt das laufende Beispiel mit der hervorgehobenen Kante g, für die sowohl a, b, c, d, e, f, i, j, k als auch g Aktivierungskante sind:

$$\underset{x \in \{a,b,c,d,e,f,g,i,j,k\}}{\forall} x \rightsquigarrow g$$

In jedem Kontrollfluss ausgehend von diesen Kanten gibt es demnach einen Zustand, in dem g eine Marke trägt. Dies lässt sich einfach nachvollziehen. Denn egal ob der Splitknoten S_1 eine Marke auf die ausgehende

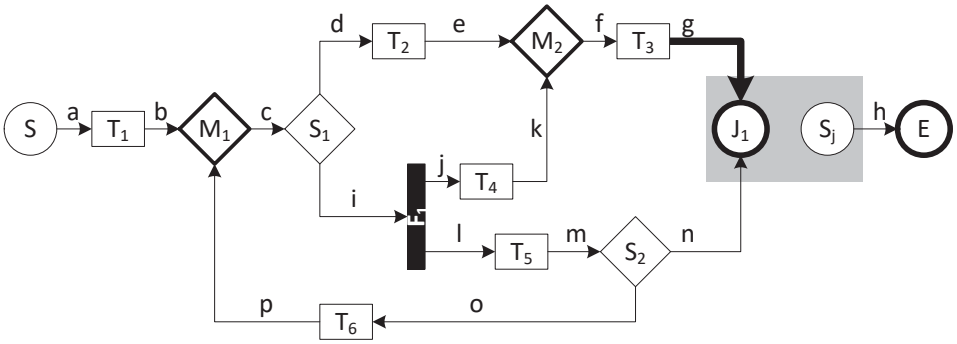


Abbildung 6.4: Die Kanten $a - g$ und $i - k$ sind Aktivierungskanten der Kante g

Kante d oder i legt, eine Marke erreicht in jedem Fall die Kante g über die Wege (d, e, f, g) und (i, j, k, f, g) (wegen des Forkknotens F_1).

In diesem Beispiel gibt es keine Aktivierungskante des Joinknotens J_1, da bis auf die Kante n selbst, keine andere Kante eine Marke auf n garantiert. Somit gibt es keine Kante, die Aktivierungskante von g und n ist.

Wenn nun auf allen Wegen von einer Startkante des Eintrittsgraphen zu j eine Aktivierungskante liegt, so kann j offensichtlich nicht unmittelbar verklemmen. Andererseits folgt aus einer unmittelbaren Verklemmung in j, dass es einen Weg ohne Aktivierungskante von einer Startkante des Eintrittsgraphen zu j gibt. Dies hält der folgende Satz fest:

Satz 6.7 (Verklemmungssatz)**.**

Ein Joinknoten j mit seinem Eintrittsgraphen $\mathcal{EG}(j)$ und eine Startkante *entry* des Eintrittsgraphen seien gegeben. Weiterhin sei $\rightsquigarrow(j)$ die Menge aller Aktivierungskanten von j, $\rightsquigarrow(j) = \{a \in E : a \overset{all}{\rightsquigarrow} j\}$. Es gilt:

$$\underset{f_{entry} \in \mathcal{F}_{entry}}{\forall} j \text{ hat keine unmittelbare Verklemmung in } f_{entry} \tag{6.7}$$

$$\Longleftrightarrow$$

$$\underset{in \in \triangleright j}{\forall} \underset{W \in \mathcal{W}_{entry \to in}}{\forall} W \cap \rightsquigarrow(j) \neq \varnothing \tag{6.8}$$

Beweis (Satz 6.7). Es werden die Rück- und Hinrichtung bewiesen.

$\Longleftarrow$ Es gilt zu beweisen: $\underset{in \in \triangleright j}{\forall} \underset{W \in \mathcal{W}_{entry \to in}}{\forall} W \cap \rightsquigarrow(j) \neq \varnothing \implies \underset{f_{entry} \in \mathcal{F}_{entry}}{\forall} j$ hat keine unmittelbare Verklemmung in f_{entry}. Dies ist ein konstruktiver Beweis. Dazu betrachten wir die zwei folgenden, vollständigen Fälle für jeden Kontrollfluss $f_{entry} \in \mathcal{F}_{entry}$ in $\mathcal{EG}(j)$:

Fall 1: f_{entry} erreicht keine Aktivierungskante aus $\rightsquigarrow(j)$. Da auf allen Wegen zu einer eingehenden Kante von j eine Aktivierungskante liegt, kann demzufolge keine Marke eine eingehende Kante von j erreichen (sonst würde eine Aktivierungskante von f_{entry} erreicht werden). Das heißt, es erreicht keine Marke eine eingehende Kante von j, weshalb j in f_{entry} keine unmittelbare Verklemmung hat. ✓

Fall 2: f_{entry} erreicht eine Aktivierungskante aus $\rightsquigarrow(j)$. Damit erreicht f_{entry} alle eingehenden Kanten von j. Somit tritt keine unmittelbare Verklemmung von j in f_e auf. ✓

$\Longrightarrow$ Beweis per Kontradiktion:

$$\forall_{f_{entry} \in \mathcal{F}_{entry}} j \text{ hat keine unmittelbare Verklemmung in } f_{entry}$$

$$\wedge \quad \exists_{in \in \triangleright j} \exists_{W \in \mathcal{W}_{entry \to in}} W \cap \rightsquigarrow(j) = \varnothing$$

Sei W so ein Weg ohne Aktivierungskante von j von *entry* zu einer eingehenden Kante *in* von j. Da alle Joinknoten im Eintrittsgraphen durch einen sicheren Knotentyp $Safe$ ersetzt wurden, kann im Eintrittsgraphen keine Verklemmung auftreten. Daraus folgt mit Hilfe des Satzes 5.7, dass es mindestens einen Kontrollfluss f_{entry} gibt, der auf jede Kante von W hintereinander eine Marke legt. Dadurch erreicht die eingehende Kante *in* als letzte Kante von W mindestens eine Marke. Da keine Aktivierungskante von j auf W liegt, gibt es die Möglichkeit, dass auf mindestens einer eingehenden Kante von j *keine* Marke landet. Demnach gibt es einen Kontrollfluss f_{entry}, in dem auf mindestens einer aber nicht auf allen eingehenden Kanten von j Marken liegen. j verklemmt unmittelbar in f_{entry}. $\lightning$

□

Beispiel 6.8.

Zur Illustration des vorangegangenen Satzes greifen wir auf die korrekte Version unseres fortlaufenden Beispielworkflowgraphen zurück. Der Eintrittsgraph des Joinknotens J_1 ist in Abbildung 6.5 zu sehen, wobei der Joinknoten J_1 in den Endknoten J_1 sowie den Startknoten S_j zerlegt wurde.

Die Kante (S, T_1) ist eine Startkante des Eintrittsgraphen von J_1. Auf allen Wegen von (S, T_1) zu (T_7, J_1) sowie zu (T_5, J_1) liegt eine Aktivierungskante beider Kanten: (S_1, F_1). Nach obigem Satz kann von (S, T_1) keine unmittelbare Verklemmung in J_1 folgen. Dies lässt sich anhand des Graphen auch gut nachvollziehen, da ein Kontrollfluss ab (S, T_1) entweder alle eingehenden Kanten von J_1 erreicht oder aber gar keine.

Im Gegensatz dazu kann der Joinknoten J_1 im inkorrekten Eintrittsgraphen aus Abbildung 6.4 unmittelbar verklemmen, da es tatsächlich keinen Kontrollfluss gibt, in dem eine Aktivierungskante von J_1 eine Marke bekommt – denn diese existiert nicht.

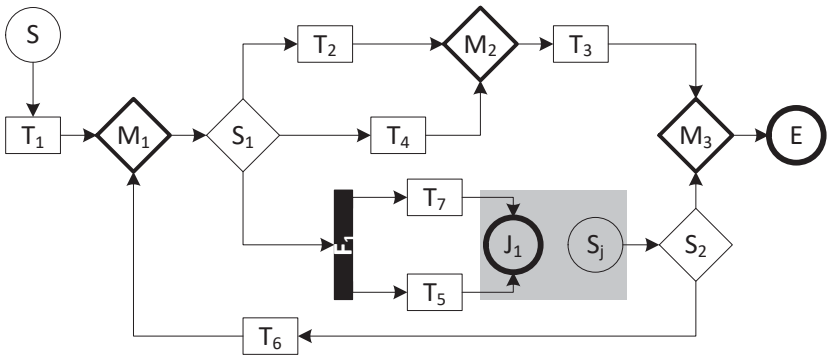


Abbildung 6.5: Der Eintrittsgraph von J_1 eines korrekten Workflowgraphen

Mit Hilfe des letzten Satzes können nun potentielle, unmittelbare Verklemmungen in Joinknoten bestimmt werden. Die *Ursache* von diesen potentiellen, unmittelbaren Verklemmungen sind strukturelle Fehler im Workflowgraphen: Joinknoten können in Berechnungen erreicht werden, ohne dass diese deren Ausführung garantieren.

6.3 Algorithmische Herleitung

Am Ende dieses Kapitels wird nun die algorithmische Herleitung zur Bestimmung der Aktivierungskanten und der unmittelbaren Verklemmungen behandelt. Dafür werden zunächst die Aktivierungskanten eines Joinknotens berechnet. Anschließend wird der Verklemmungssatz genutzt, um unmittelbare Verklemmungen zu identifizieren.

Die Berechnung der Aktivierungskanten benötigt laut Definition die Betrachtung der Kontrollflüsse. Dies entspricht jedoch einer (partiellen) Erkundung des Zustandsraums und ist ineffizient. Um dennoch eine effiziente Berechnung zu ermöglichen, hilft uns die folgende Beziehung zwischen einer Kante und ihren direkten Nachfolgern:

Bemerkung 6.9.

Ein Joinknoten j mit seinem Eintrittsgraphen $\mathcal{EG}(j)$ und eine seiner eingehenden Kanten in seien angenommen. Weiterhin sind noch eine beliebige Kante a und ihre direkten Nachfolger $B = tgt(a)^{\triangleleft}$ gegeben.
a ist Aktivierungskante von in genau dann, wenn

1. $a = in$ gilt, oder
2. $tgt(a)$ ein Task-, Fork-, Merge- oder Safeknoten (Joinknoten) ist und $\underset{b \in B}{\exists}\, b \rightsquigarrow in$ gilt, Abbildungen 6.6 *a)* bis *d)*, oder
3. $tgt(a)$ ein Splitknoten ist und $\underset{b \in B}{\forall}\, b \rightsquigarrow in$ gilt, Abb. 6.6 *e)*.

Demnach besitzt jede Aktivierungskante ($\neq in$) mindestens eine Aktivierungskante als direkten Nachfolger.

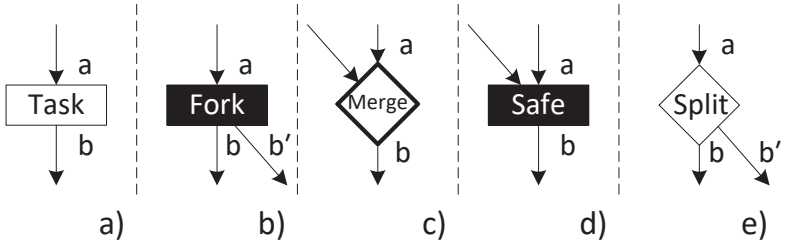


Abbildung 6.6: Eine Vorgängerkante einer Aktivierungskante ist ebenfalls Aktivierungskante, außer es handelt es sich um einen Splitknoten

Aus diesem einfachen Zusammenhang ergibt sich aus den Aktivierungskanten (inklusive der dazugehörigen Knoten) ein zusammenhängender Graph [53, S. 547]. Damit sind für eine eingehende Kante in eines Joinknotens j alle Aktivierungskanten auch Vorgänger. Die Menge der Vorgänger bildet damit eine Übermenge der Aktivierungskanten. Diese lässt sich durch eine inverse Tiefensuche[1] ausgehend von in auf den Kanten des Eintrittsgraphen bestimmen. Unser algorithmischer Ansatz ist es, ausgehend von dieser Übermenge, genannt $Activation(in)$, *iterativ* die Aktivierungskanten zu bestimmen. In jeder Iteration wird dabei $Activation(in)$ aktualisiert. Dabei werden diejenigen Kanten herausgenommen, die nicht Aktivierungskanten von in sind. Findet keine Aktualisierung mehr statt, terminiert der Algorithmus. $Activation(in)$ entspricht dann der Menge der Aktivierungskanten von in.

Nun hat jede Kante ($\neq in$) aus $Activation(in)$ innerhalb dieser Menge mindestens einen direkten Nachfolger. Demnach können eingehende Kanten von Task-, Fork-, Merge- und Safeknoten erst aus $Activation(in)$ herausgenommen werden, wenn bereits all ihre direkten Nachfolger entfernt wurden. Hingegen kann die eingehende Kante eines Splitknotens bereits herausgenommen werden, wenn mindestens ein direkter Nachfolger nicht in $Activation(in)$ ist.

Es ergeben sich daraus zwei Schritte für jede Iteration: (1) Das Untersuchen und Herausnehmen von eingehenden Kanten von Splitknoten; und (2) das Untersuchen und Herausnehmen von eingehenden Kanten von Task-, Fork-, Merge- und Safeknoten. Da letztere, herauszunehmende Kanten keinen direkten Nachfolger mehr in $Activation(in)$ besitzen, sind diese auch nicht mehr von in erreichbar. Die aktualisierte Menge $Activation(in)$ *nach* Schritt (2) besteht demnach aus den Kanten, die noch von in erreichbar sind. Anstelle nun im Schritt (2) die zu entfernenden Kanten zu bestimmen und $Activation(in)$ zu aktualisieren, kann mit Hilfe einer inversen Tiefensuche ab in die aktuelle Menge $Activation(in)$ direkt bestimmt werden.

Der gesamte Algorithmus zur Bestimmung der Aktivierungsrelation kann in Algorithmus 1 nachvollzogen werden. Abbildung 6.7 illustriert die Schritte des Algorithmus für die eingehende Kante g des Joinknotens J_1 unseres Beispielworkflow-

[1] In einer inversen Tiefensuche werden die Kanten des Graphen in Rückwärtsrichtung besucht.

Algorithmus 1 Bestimmung der Aktivierungskanten

Require: Eintrittsgraph $EG = \mathcal{EG}(join)$ eines Joinknotens $join$.

Ensure: Für jede eingehende Kante in von $join$ sowie für $join$ die Menge der Aktivierungskanten $Activation(in)$ von in bzw. $ActivationJoin$ von $join$.

```
ActivationJoin ← E(EG)
for all in ∈ ▷join do
    /** Initialisierung. Ausgang: Alle Vorgänger von in bilden Activation(in). Bestimmung aller
        Vorgänger mit inverser Tiefensuche ausgehend von in auf Kanten von E(EG). **/
    Activation(in) ← InverseDepthFirstSearch(E(EG), in)
    repeat
        OldActivation ← Activation(in)
        /** Entferne eingehende Kante von Splitknoten aus Activation(in), wenn mindestens
            eine seiner ausgehenden Kanten die Kante in nicht aktiviert. **/
        for all split ∈ N_Split do
            if split◁ ∉ Activation(in) then
                // Entferne eingehende Kante aus Activation(in).
                Activation(in) ← Activation(in) ∖ ▷split
        /** Inverse Tiefensuche ausgehend von in auf Kanten von Activation(in).
            Gefundene Kanten bilden aktualisiertes Activation(in). **/
        Activation(in) ← InverseDepthFirstSearch(Activation(in), in)
        /** Wiederhole bis Activation(in) sich nicht mehr ändert. **/
    until Activation(in) = OldActivation
    ActivationJoin ← ActivationJoin ∩ Activation(in)
```

graphen. In der Teilabbildung 6.7 *a)* ist der Eintrittsgraph von J_1 zu sehen. In ihm sind alle durch die inverse Tiefensuche ab g bestimmten Vorgänger durch dickere Linien markiert. Sie bilden die Ausgangsmenge $Activation(g)$. Aus den Kanten von $Activation(g)$ ergibt sich der Graph aus Abb. *b)*. Jetzt können die Splitknoten identifiziert werden, die eine ausgehende Kante außerhalb von $Activation(g)$ besitzen. Diese Eigenschaft erfüllt im Graph nur der Knoten S_2, dessen eingehende Kante m daraufhin eliminiert wird, *c)*. In *c)* sind wieder jene Kanten dick markiert, welche durch eine inverse Tiefensuche ab g erreicht werden. Die Kante l ist demzufolge nicht mehr in $Activation(g)$, *d)*. Anschließend terminiert der Algorithmus, da keine Kante mehr entfernt werden kann. Wie gut zu erkennen ist, garantiert jede der verbliebenen Kanten eine Marke auf g, sobald die Kante selbst eine Marke trägt.

Das asymptotische Laufzeitverhalten des Algorithmus 1 wird für eine eingehende Kante in eines Joinknotens im Wesentlichen durch zwei Schleifen bestimmt: Die äußere Repeat-Until-Schleife und die innere For-All-Schleife. Die Repeat-Until-Schleife wird maximal sooft ausgeführt, wie Kanten entfernt werden. Dies entspricht $O(E)$.

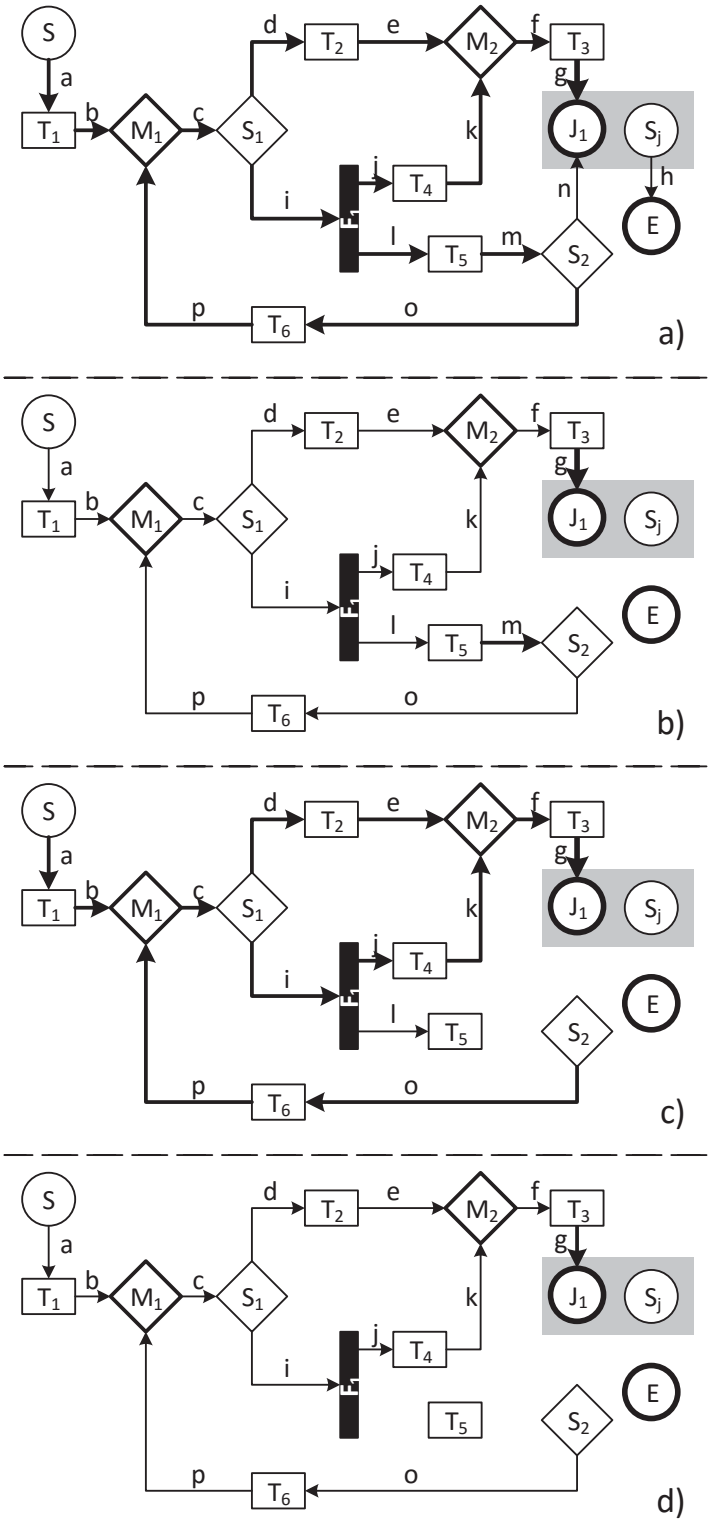


Abbildung 6.7: Anwendung des Algorithmus auf den Eintrittsgraphen von J_1 für die eingehende Kante g.

Zeitintensive Schritte der Repeat-Until-Schleife sind die inverse Tiefensuche ausgehend von in und die innere For-All-Schleife. Die inverse Tiefensuche ist in $O(E)$ möglich und zudem noch aufgrund der schwindenden Kantenanzahl in jeder Iteration schneller. Die For-All-Schleife betrachtet jeden Splitknoten. Dies sind maximal $O(N_{Split})$ oder abgeschätzt $O(E)$. Demzufolge wird jede Iteration der Repeat-Until-Schleife in $O(E)$ und die Schleife insgesamt in $O(E^2)$ berechnet. Da der Algorithmus für jede eingehende Kante von allen Joinknoten ausgeführt werden muss, ist die Laufzeit insgesamt kubisch, $O(E^3)$. Insgesamt wurde damit ein effizienter Algorithmus konstruiert, der die Aktivierungskanten bestimmt.

Zum Abschluss müssen mit Hilfe der berechneten Aktivierungskanten noch die unmittelbaren Verklemmungen in einem Joinknoten j bestimmt werden. Dazu hilft uns direkt der Verklemmungssatz 6.7, in dem wir für jede Startkante des Eintrittsgraphen von j alle Wege zu j untersuchen. Wird dabei ein Weg ohne Aktivierungskante von j gefunden, kann in j eine unmittelbare Verklemmung auftreten.

Für die Untersuchung der Wege kann wieder eine inverse Suche ab den eingehenden Kanten von j durchgeführt werden. Erreicht diese Suche eine Aktivierungskante von j, dann hört die Suche in die Tiefe an dieser Stelle auf. Erreicht die Suche jedoch eine der Startkanten des Eintrittsgraphen, so gibt es einen Weg von dieser Startkante zu mindestens einer eingehenden Kante von j ohne Aktivierungskante.

Der vollständige Algorithmus ist in Algorithmus 2 zu sehen. Das asymptotische Laufzeitverhalten des Algorithmus wird im Wesentlichen durch drei Schritte bestimmt: (1) Die Konstruktion der Eintrittsgraphen, (2) die Bestimmung der Aktivierungskanten und (3) die Berechnung der inversen Tiefensuche inkl. Bestimmung unmittelbarer Verklemmungen. Die Konstruktion der Eintrittsgraphen, (1), ist für einen Joinknoten in linearer Zeit möglich, $O(E)$. Dafür müssen lediglich die Joinknoten mit dem neuen Label $Safe$ versehen und der betrachtete Joinknoten geteilt werden. Insgesamt verbringt der Algorithmus somit quadratisch viele Schritte für die Berechnung der Eintrittsgraphen aller Joinknoten.

Die Berechnung der Aktivierungskanten (2) ist, wie bereits behandelt, in kubischer Laufzeit möglich. Die Berechnung der inversen Tiefensuche und die Bestimmung einer unmittelbaren Verklemmung (3) wird genau einmal für alle eingehenden Kanten der Joinknoten durchgeführt. Eine inverse Tiefensuche hat eine lineare Laufzeit, $O(E)$. Die Bestimmung der unmittelbaren Verklemmung ist in konstanter Zeit möglich. Der Schritt (3) geschieht demnach in quadratisch asymptotischer Laufzeit hinsichtlich der Anzahl der Kanten. Insgesamt sind wir somit zur Bestimmung aller unmittelbaren Verklemmungen in Besitz eines Algorithmus mit kubischem Laufzeitverhalten.

Algorithmus 2 Bestimmung unmittelbarer Verklemmungen

Require: Workflowgraph $WFG = (N, E, l)$

Ensure: Menge $\mathcal{V} \subseteq N_{Join}$ aller Joinknoten mit unmittelbarer Verklemmung

```
Berechne Eintrittsgraphen für alle Joinknoten mit Startkanten Start(join)
Bestimme Aktivierungskanten für alle Joinknoten
for all join ∈ N_Join do
    for all in ∈ ▷join do
        /** Festlegung der Kanten, bei denen die Suche aufhören soll.
            Dies sind die Aktivierungskanten von join. **/
        StopSearch ← {a ∈ E : a ↝^all join}
        /** Erstelle leere Menge. Sie speichert die durch die Tiefensuche besuchten Kanten. **/
        Visited ← ∅
        /** Führe inverse Tiefensuche ab in durch. **/
        InverseStoppableDeepSearch(in, StopSearch, Visited)
        /** Bestimme die unmittelbaren Verklemmungen. **/
        if (Start(join) ⊆ Visited) then
            V ← V ∪ {join}
procedure InverseStoppableDeepSearch(current, StopSearch, Visited)
    Visited ← Visited ∪ {current}
    for all pred ∈ (▷src(current) ∖ (StopSearch ∪ Visited)) do
        InverseStoppableDeepSearch(pred, StopSearch, Visited)
```

Kapitel 7

Ursachen von Überflüssen

Die im letzten Kapitel behandelten Verklemmungen sind in Workflowgraphen stets an Joinknoten geknüpft. Hingegen können Überflüsse auf einer Vielzahl unterschiedlicher Kanten in verschiedener Anzahl an Marken auftreten. Alleine durch eine andere Ausführungsreihenfolge der Knoten kann ein Überfluss bereits auf einer früheren oder erst späteren, wenn nicht sogar auf keiner Kante auftreten. Zum Beispiel kann der in Abbildung 7.1 illustrierte Überflusszustand in unserem Beispielworkflowgraphen aufgrund einer anderen Ausführungsreihenfolge anstelle auf der Kante f auch erst auf g zustandekommen. Dadurch manifestieren sich in vielen Workflowgraphen praktisch beliebig, sogar endlos viele Überflusszustände. Eine Findung all dieser Überflusszustände beispielsweise durch eine Zustandsraumerkundung oder partielle Analyse ist praktisch meist nicht möglich und wenn sehr zeitintensiv.

Wie bei den Verklemmungen ist es unser Ziel, zeitnah Ergebnisse zu liefern. Dies erreichen wir durch den Verzicht auf die Bestimmung aller erreichbaren Überflusszustände vom Startzustand. Stattdessen nehmen wir die *Ursachen* der Überflüsse in den Fokus, denn aus der Ursache eines Überflusses können viele tatsächlich zur Laufzeit auftretende Überflüsse abgeleitet werden. Außerdem ist zur Vermeidung von Fehlern zur Laufzeit offenbar die Kenntnis über die Ursache des Fehlers unabdingbar.

Die Ursache von Überflüssen beschreibt der in der Literatur geläufige Begriff *Lack of Synchronization* (fehlende Synchronisierung). Aufgrund einer solchen fehlenden Synchronisierung zweier paralleler Kontrollflüsse gelangen zwei Marken auf die selbe Kante. Die Grundidee unseres Ansatzes ist nun sehr einfach: Parallele Kontrollflüsse werden durch Forkknoten erzeugt und „starten“ von verschiedenen ausgehenden Kanten des Forkknotens. *Erstmals* können sich diese Kontrollflüsse in der selben Kante *treffen*, zu der es disjunkte Wege gibt. Geht diese Kante aus einem Joinknoten aus, dann werden die Kontrollflüsse synchronisiert. Anderenfalls ist potentiell ein Überfluss möglich. Diesen Ansatz beschreiben wir in diesem Kapitel ausführlich. Wir zeigen außerdem, wie die Ursachen von Überflüssen effizient in Workflowgraphen gefunden werden können.

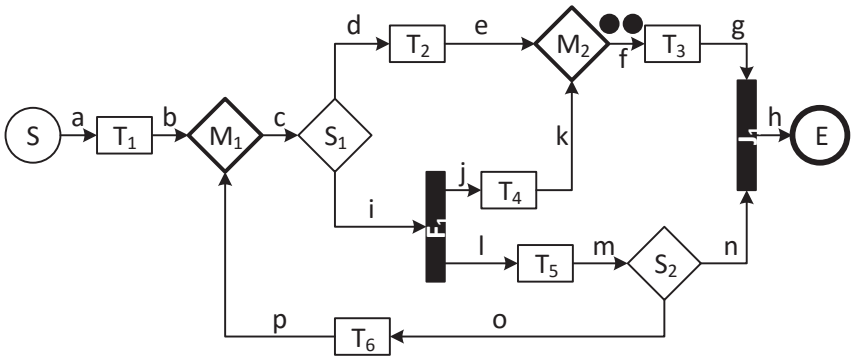


Abbildung 7.1: Ein Überflusszustand durch zwei Marken auf f

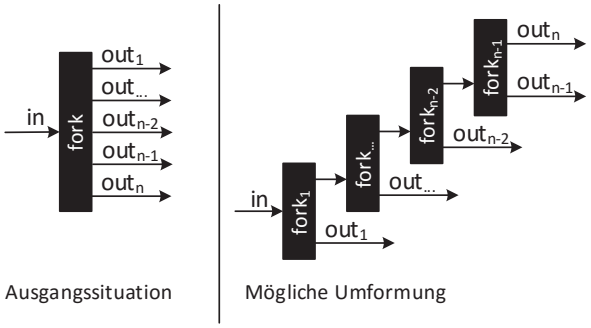


Abbildung 7.2: Umformung von Forkknoten mit mehr als zwei ausgehenden Kanten

Hinweis: Wir gehen im Rest dieses Kapitels davon aus, dass jeder Forkknoten genau zwei ausgehende Kanten besitzt. Auch wenn diese Voraussetzung natürlich für gewöhnliche Workflowgraphen nicht erfüllt ist, kann diese durch das Hinzufügen zusätzlicher Forkknoten eingehalten werden, ohne die Semantik des Workflowgraphen zu verändern (vgl. Abbildung 7.2). Für jeden Forkknoten $fork$ mit $|fork\triangleleft| > 2$ sind dazu maximal $|fork\triangleleft| - 1$ zusätzliche Forkknoten notwendig. Die Anzahl der Kanten kann sich unterdessen (wie gut in Abb. 7.2 zu sehen ist) maximal verdoppeln. Da die Kantenanzahl im Normalfall größer ist als die Anzahl der Knoten, vergrößert sich der Graph hinsichtlich der Kanten nur linear.

7.1 Treffpunkte von Kontrollflüssen

Der erste Schritt zur Bestimmung der Ursachen von Überflüssen ist die Untersuchung eines vom Startzustand aufgetretenen Überflusszustands. In diesem Überflusszustand liegen laut Definition auf mindestens einer Kante mindestens zwei Marken. Da mindestens eine Kante mindestens zwei Marken trägt, muss zuvor ein Forkknoten ausgeführt worden sein, der diese Parallelität ermöglicht.

Um diese Idee zu illustrieren, betrachten wir nochmals das Beispiel aus Abbildung

7.1. Zur Erinnerung: Die Kante f trägt zwei Marken, der Workflowgraph befindet sich in einem Überflusszustand. Alle Möglichkeiten, in denen auf f zwei Marken gelangen, erfordern die vorherige Ausführung des einzigen Forkknoten F_1 – anderenfalls gibt es immer nur maximal eine Marke im Workflowgraphen.

Wenn Überflusszustände stets nur aus der Ausführung mindestens eines Forkknotens resultieren können, bilden die eingehenden Kanten von Forkknoten ausgezeichnete Eintrittspunkte für intensivere Analysen. Betrachten wir aus diesem Grund die Forkknoten etwas genauer. Wir möchten an dieser Stelle darauf hinweisen, dass bei gleichzeitiger Ausführbarkeit mindestens zweier Forkknoten mindestens ein weiterer Forkknoten existieren muss, der diese gleichzeitige Ausführbarkeit ermöglicht. Da wir *alle* Forkknoten betrachten, wird solch ein Fall natürlich mit untersucht.

Nach der Ausführung eines Forkknotens $fork$ liegt auf jeder seiner ausgehenden Kanten mindestens eine Marke. Diese Marken bewegen sich daraufhin relativ unabhängig von Kante zu Kante. Dabei passieren sie Stellen, in denen sie *erstmals* mit anderen Marken desselben Forkknotens $fork$ aufeinandertreffen können. In anderen Worten könnten an diesen Stellen *erstmals* zwei von $fork$ produzierte Marken gleichzeitig auf einer Kante landen und einen Überfluss erzeugen. Formal lässt sich solch eine Stelle t über Wege ausgehend von den zwei ausgehenden Kanten von $fork$ definieren: Gibt es von den zwei ausgehenden Kanten von $fork$ mindestens zwei disjunkte Wege zu t, so nennen wir t einen *Treffpunkt* von $fork$.

Definition 7.1 (Treffpunkte)**.**

Sei ein Forkknoten $fork$ eines Workflowgraphen (N, E, l) gegeben. Für dessen ausgehende Kanten $a, b \in fork^{\triangleleft}$ nennen wir eine Kante $t \in E$ *Treffpunkt*, wenn es einen Weg W_a von a und einen Weg W_b von b nach t gibt und $W_a \cap W_b = \{t\}$ gilt:

$$t \text{ ist Treffpunkt von } a \text{ und } b \iff \exists_{\substack{W_a \in \mathcal{W}_{a\to t} \\ W_b \in \mathcal{W}_{b\to t}}} W_a \cap W_b = \{t\} \tag{7.1}$$

Die Wege W_a und W_b bezeichnen wir als *Routen* von a und b nach t. Die Menge aller Paare solcher Routen wird durch $\bigwedge_{a\ b}^{t}$ beschrieben:

$$\bigwedge_{a\ b}^{t} = \left\{(W_a, W_b) \in \mathcal{W}_{a\to t} \times \mathcal{W}_{b\to t} \colon \quad W_a \cap W_b = \{t\}\right\} \tag{7.2}$$

t ist außerdem ein Treffpunkt von $fork$. Die Menge aller Treffpunkte von $fork$ beschreiben wir mit $人(fork)$.

In unserem Beispiel aus Abbildung 7.1 ist die Kante f ein solcher Treffpunkt. Über die zwei disjunkten Wege (j, k, f) und (l, m, o, p, c, d, e, f) ist er von den zwei ausgehenden Kanten j und l des Forkknotens erreichbar.

Besitzen nun alle Treffpunkte aller Forkknoten als Quelle einen Joinknoten, dann treffen sich alle Marken *immer erstmals* in einem Joinknoten – und werden dort synchronisiert. Ein Überflusszustand ist offensichtlich im gesamten Workflowgraphen *nicht* mehr möglich.

Bemerkung 7.2 (Ausschluss von Überflüssen)**.**
Sei ein Workflowgraph (N, E, l) gegeben. Es gilt offenbar:

$$\bigvee_{fork \in N_{Fork}} \bigvee_{t \in \curlywedge(fork)} src(t) \in N_{Join} \tag{7.3}$$

$$\Longrightarrow$$

$$\bigvee_{e \in E} \bigvee_{\substack{Z \in \mathcal{Z}(E) \\ [\![e]\!] \rightarrow^* Z}} Z \text{ ist kein Überflusszustand} \tag{7.4}$$

Bilden wir die *Kontraposition* $(a \rightarrow b \Leftrightarrow \neg b \rightarrow \neg a)$ aus Bemerkung 7.2, erhalten wir die notwendige Bedingung für jeden Überfluss:

Bemerkung 7.3 (Bedingung für Überflüsse)**.**
Sei ein Workflowgraph (N, E, l) gegeben. Aus der Kontraposition der Bemerkung 7.2 ergibt sich:

$$\bigwedge_{e \in E} \bigwedge_{\substack{Z \in \mathcal{Z}(E) \\ [\![e]\!] \rightarrow^* Z}} Z \text{ ist Überflusszustand} \tag{7.5}$$

$$\Longrightarrow$$

$$\bigwedge_{fork \in N_{Fork}} \bigwedge_{t \in \curlywedge(fork)} src(t) \notin N_{Join} \tag{7.6}$$

Betrachten wir nochmals unser Beispiel aus Abbildung 7.1, so haben wir bereits gezeigt, dass die Kante f ein Treffpunkt des Forkknotens F_1 ist. Offensichtlich ist die Quelle der Kante (der Mergeknoten M_2) *kein* Joinknoten.

7.2 Überflüssige Treffpunkte

Zuletzt haben wir festgestellt, dass bei Auftreten eines Überflusses immer auch mindestens ein Treffpunkt eines Forkknotens existiert, dessen Quelle *kein* Joinknoten ist. Nun müssen wir für eine eindeutige Ursachenidentifikation von Überflüssen überprüfen, ob diese notwendige Bedingung auch hinreichend ist, das heißt, ob die Rückrichtung von Bemerkung 7.3 ebenso Gültigkeit besitzt. Dazu nehmen wir im Folgenden einen Forkknoten $fork$ mit einem seiner Treffpunkte $t \in \curlywedge(fork)$ an.

Die grundlegende Idee für die Überprüfung ist, einen Nachweis zu finden, ob und wann zwei Kontrollflüsse (im Sinne zweier Marken) gleichzeitig ausgehend von $fork$ bei der Quelle von t ankommen können. Dabei sei per Annahme ausgeschlossen, dass

vorher ein Überfluss auftritt. Solch ein Überfluss kann durch die Betrachtung der anderen Treffpunkte und Forkknoten identifiziert werden.

Können nun zwei Marken über zwei Kontrollflüsse gleichzeitig bei der Quelle von t ankommen und diese Quelle ist ein Joinknoten, so werden die beiden Marken korrekt synchronisiert. Ist die Quelle von t jedoch in diesem Fall *kein* Joinknoten, so liegt zwangsläufig ein potentieller Überfluss vor. Für die Ursachenfindung von Überflüssen sind also jene Treffpunkte der Forkknoten interessant, die *keine* ausgehenden Kanten von Joinknoten sind. Wir betrachten daher im Nachfolgenden einen Treffpunkt t mit $src(t) \notin N_{Join}$. Zeigen müssen wir demzufolge noch, ob bei t gleichzeitig ausgehend von $fork$ zwei Marken ankommen können. Ist dies *nicht* möglich, so ist der Treffpunkt t hinsichtlich $fork$ *überflüssig*.

Definition 7.4 (Überflüssige Treffpunkte)**.**

> Ein Treffpunkt t mit $src(t) \notin N_{Join}$ eines Forkknotens $fork$ heißt *überflüssig*, wenn direkt von $fork$ *niemals* zwei Marken gleichzeitig auf t ankommen.

Schaffen wir es, die überflüssigen Treffpunkte von den wichtigen zu trennen, so sind wir präzise. Als Grundlage dieser Präzisierung nutzen wir Satz 5.7. Zur Erinnerung, dieser Satz sagt aus, dass bei Existenz eines Weges von einer Kante a zu einer Kante t alle Kanten dieses Weges nacheinander von einer Marke besucht werden können – außer es gibt eine Verklemmung.

Nun gibt es laut Definition der Treffpunkte zu t ausgehend von $fork$ mindestens zwei Routen $(W_a, W_b) \in \overset{t}{\underset{a}{\wedge}}\,\underset{b}{}$ von den ausgehenden Kanten $a, b \in fork^{\triangleleft}$. Zur Erinnerung, W_a und W_b sind zwei bis auf die Kante t disjunkte Wege. Schätzen wir mit Hilfe dieser beiden Routen zwei Kontrollflüsse nach Satz 5.7 ab, so können auf Basis dieser Abschätzung zwei Marken gleichzeitig bei t ankommen. Es ist offensichtlich ein Überfluss auf der Kante t möglich.

Betrachten wir unser Beispiel aus Abbildung 7.3, so sehen wir den Forkknoten F_1 direkt nach seiner Ausführung. Weiterhin sind in der Abbildung auch die beiden Routen von j und l zum Treffpunkt t zu sehen. Wie leicht zu überprüfen ist, stimmt unsere Annahme zweier Kontrollflüsse durch die beiden Routen für dieses Beispiel.

Natürlich handelt es sich dabei zunächst nur um eine *Abschätzung*. Ist diese nicht präzise, so ist sie offensichtlich eine *Über*abschätzung, da wir zwei parallele Kontrollflüsse annehmen, die vielleicht so während der Ausführung niemals auftreten können. Dadurch könnte immer nur maximal ein Kontrollfluss den Treffpunkt t erreichen. Somit wäre t als Treffpunkt obsolet.

Um solche überabgeschätzten Treffpunkte zu eliminieren, müssen wir eine Aussage über die Überflüssigkeit von Treffpunkten in beliebigen Workflowgraphen finden. Um die Überflüssigkeit eines Treffpunkts zu bestimmen, gehen wir für den Moment von einem *korrekten* Workflowgraphen aus. In einem korrekten Workflowgraphen kann

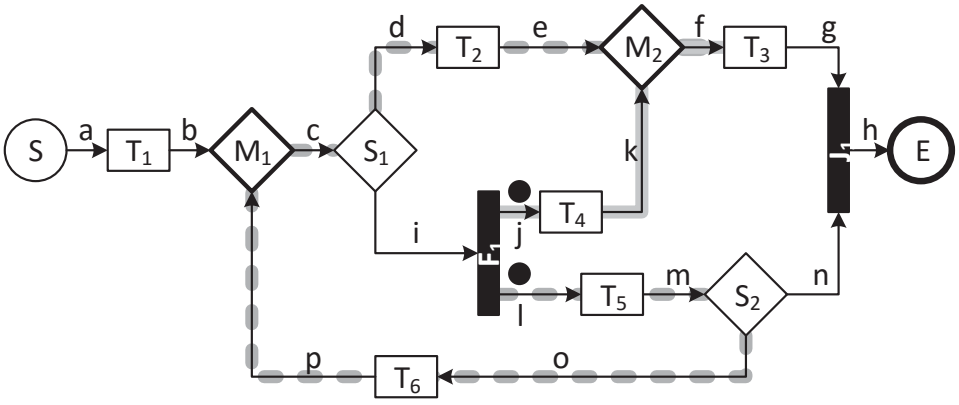


Abbildung 7.3: Routen zum Treffpunkt f

ein (überabgeschätzter) Treffpunkt nur genau dann keinen Joinknoten als Quelle besitzen, wenn er überflüssig ist. Damit erhalten wir eine für korrekte Workflowgraphen abschließende Aussage darüber, ob ein Treffpunkt überflüssig ist oder nicht. Zusammengefasst nehmen wir für den Moment einen korrekten Workflowgraphen an, in dem es einen Forkknoten $fork$ mit einem Treffpunkt t gibt, wobei t *keinen* Joinknoten als Quelle hat.

Da der Workflowgraph korrekt ist, gibt es nun mindestens einen vom Startzustand erreichbaren Zustand, in dem $fork$ ausgeführt wird. Nach der Ausführung von $fork$ können mindestens zwei Marken *ohne* Verklemmung über mindestens zwei Routen W_a und W_b zum Treffpunkt t gelangen (Satz 5.7). Demzufolge kann das schrittweise Ablaufen der beiden Routen W_a und W_b von den Marken *nicht* verhindert werden. Früher oder später gelangen also *beide* Marken *garantiert* auf den Treffpunkt t. Sie dürfen dies aber niemals *gleichzeitig* tun. Es muss also eine Reihenfolge geben, in der die Marken auf t ankommen. Das heißt, solange bereits eine Marke auf t liegt, darf die andere Marke niemals ebenfalls auf t gelangen.

Gehen wir o.B.d.A. davon aus, dass die Marke, welche die Route W_a abschreitet, bereits beim Treffpunkt t angekommen ist. Damit die Marke von W_b nun nicht ebenso auf t angelangt, muss sie irgendwo vor einem Knoten D auf der Route W_b hängen bleiben. Dies muss sie mindestens solange, bis die Marke auf t die Kante t verlässt. Im Falle dieses Verlassens muss der Knoten D dann eine Art „Signal“ bekommen, dass er ausgeführt werden kann. Da die Semantik in Workflowgraphen nur durch Marken bestimmt wird, muss demnach das „Signal“ in Form einer Marke ausgehend von t bei D ankommen. Ohne dieses Signal in Form einer Marke von t sollte D nicht ausgeführt werden können. Anderenfalls wäre nicht ausgeschlossen, dass D bereits zu früh ausgeführt wird. Wenn D also unbedingt eine Marke über t zur Ausführung erhalten muss, so ist D *abhängig* von einer Marke auf t bzgl. $fork$.

Definition 7.5 (Ausführungsabhängigkeit)**.**

Sei ein Workflowgraph (N, E, l) und ein Forkknoten $fork$ mit seiner eingehenden Kante in gegeben.

Eine Kante d, zu der in einen Weg besitzt, heißt bzgl. $fork$ *abhängig* von einer Kante $t \neq d$, wenn *nach* der Löschung der Kante t die Kante d in *keinem* Kontrollfluss ab der eingehenden Kante in von $fork$ mehr eine Marke erhält. Anderenfalls heißt d *unabhängig* von t.

Ein Knoten D heißt *abhängig* von t, sobald mindestens eine ausgehende Kante von D abhängig von t ist.

Im weiteren Verlauf verzichten wir aus Gründen der Lesbarkeit auf die Angabe, bezüglich welches Forkknotens eine Kante oder ein Knoten von einer anderen Kante abhängt, wenn dieser Bezug aus dem Kontext ersichtlich ist. Sei also D solch ein Knoten, der bzgl. $fork$ abhängig von t ist und das Erreichen einer Marke über W_b verzögert. Dessen ausgehende Kante d muss ein Treffpunkt der Kanten a und b und somit des Forkknotens $fork$ sein, da in D zwei Marken von a und b erstmals aufeinanderstoßen. Das heißt, auf W_b gibt es einen anderen Treffpunkt d, der *abhängig* von t ist und dessen Quelle D die Marken von a und b auf den Routen W_a und W_b stets vor t synchronisiert.

Sei nun angenommen, es liegt für alle zwei Routen (W_a, W_b) von $fork$ zu einem Treffpunkt t o.B.d.A. auf W_b ein von t abhängiger anderer Treffpunkt d. Dann wissen wir nach der vorangegangenen Argumentation, dass keine Marke über W_b bei t ankommt, bevor nicht die Quelle des Treffpunkts d die Marken beider Routen W_a und W_b synchronisiert hat. Demzufolge können *niemals* ausgehend von der Ausführung von $fork$ bei t gleichzeitig zwei Marken ankommen: Im Zusammenhang mit $fork$ ist der Treffpunkt t in korrekten Workflowgraphen *überflüssig*. Wir zeigen nun, dass diese Aussage auch in beliebigen Workflowgraphen Gültigkeit besitzt:

Satz 7.6 (Überflüssige Treffpunkte)**.**

Sei t mit $src(t) \notin N_{Join}$ ein Treffpunkt eines Forkknotens $fork$ mit seinen ausgehenden Kanten a und b in einem beliebigen Workflowgraphen $WFG = (N, E, l)$. Sei D die Menge der von t abhängigen Treffpunkte. Dann gilt:

$$\forall_{(W_a, W_b) \in \overset{t}{\underset{a}{\wedge}}_{b}} (W_a \cup W_b) \cap D \neq \varnothing \tag{7.7}$$

$$\Longrightarrow$$

$$t \text{ ist überflüssig (hinsichtlich } fork) \tag{7.8}$$

Beweis (Satz 7.6). Konstruktiver Beweis mittels vollständiger Fallunterscheidung:

Fall 1: WFG ist korrekt. Da $src(t) \notin N_{Join}$, muss t überflüssig sein. ✓

Fall 2: *Sonst:* WFG ist nicht korrekt. Es gibt genau zwei Fälle:

Fall a: Ausgehend von der Startkante erreicht *kein* Kontrollfluss den Forkknoten $fork$. Da t von $fork$ niemals Marken bekommt, ist t bzgl. $fork$ überflüssig. ✓

Fall b: *Sonst:* Ausgehend von der Startkante erreicht mindestens ein Kontrollfluss den Forkknoten $fork$. Nach Ausführung von $fork$ liegt auf dessen beiden ausgehenden Kanten a und b jeweils eine Marke. Es gibt wieder genau zwei Fälle:

Fall I: Auf jedem Paar an Routen W_a und W_b von a und b zu t liegen von t abhängige Treffpunkte d_a (auf W_a) und d_b (auf W_b). Die Quellen von d_a und d_b werden erst ausgeführt, wenn auf t eine Marke liegt. Da somit jedoch jede Marke von $fork$ zu t bei den Quellen von d_a und d_b hängen bleibt, erreicht *keine* Marke t direkt. t ist überflüssig. ✓

Fall II: *Sonst:* Es gibt zwei Routen W_a und W_b von a und b nach t, wobei o.B.d.A. auf der Route W_a ein von t abhängiger Treffpunkt liegt und auf W_b nicht. Es gibt abschließend wieder genau zwei Fälle:

Fall A: Beide Routen können ausgehend vom aktuellen Zustand *nicht* gemeinsam nach Satz 5.7 von den Marken abgelaufen werden. Demzufolge erreicht t wenn überhaupt maximal eine Marke ausgehend von $fork$. t ist überflüssig. ✓

Fall B: *Sonst:* Beide Routen können ausgehend vom aktuellen Zustand gemeinsam von Marken abgelaufen werden. Da auf W_a ein von t abhängiger Treffpunkt d liegt, erreicht die Marke auf W_b die Kante t *immer vor* der Marke auf W_a. Außerdem bekommt d *immer* erst eine Marke *nachdem* die Marke von t wieder herunter ist. Somit liegen auf t hinsichtlich $fork$ niemals zwei Marken gleichzeitig. t ist überflüssig. ✓

□

Alle nicht-überflüssigen Treffpunkte können in einem *korrekten* Workflowgraphen zwei Kontrollflüsse gleichzeitig erreichen. Aus diesem Grund müssen die Quellen dieser Treffpunkte immer Joinknoten sein. Dies hält die folgende Bemerkung fest:

Bemerkung 7.7.
Sei $WFG = (N, E, l)$ ein Workflowgraph.

$$WFG \text{ ist korrekt} \tag{7.9}$$

$$\Longrightarrow$$

$$\forall_{fork \in N_{Fork}} \; \forall_{t \in \curlywedge(fork)} \; src(t) \in N_{Join} \vee t \text{ ist überflüssig} \tag{7.10}$$

Durch die Kontraposition erhalten wir die endgültige Bemerkung, um die *Ursachen* aller Überflüsse in Workflowgraphen zu identifizieren:

Bemerkung 7.8 (Erkennung der Ursachen von Überflüssen)**.**
Sei $WFG = (N, E, l)$ ein Workflowgraph.

$$\exists_{fork \in N_{Fork}} \exists_{t \in \lambda(fork)} src(t) \notin N_{Join} \wedge t \text{ ist nicht überflüssig} \tag{7.11}$$

$$\Longrightarrow$$

$$WFG \text{ ist nicht korrekt} \tag{7.12}$$

Mit Hilfe dieser Bemerkung können wir die *potentiellen* Überflüsse in Workflowgraphen abschätzen. Diese Abschätzung ist präzise, sobald der Workflowgraph korrekt ist. Das heißt, in korrekten Workflowgraphen finden wir *keine* Ursachen von Überflüssen. In inkorrekten Workflowgraphen hingegen finden wir solche potentiellen Ursachen. Es stellt sich nun die Frage nach der Vollständigkeit dieses Ansatzes.

Da wir nicht die Rückrichtung von Satz 7.6 geführt haben, kann es durchaus in inkorrekten Workflowgraphen dazu kommen, dass wir *nicht* alle überflüssigen Treffpunkte erkennen. Demzufolge überabschätzen wir die *wichtigen* Treffpunkte und können auf potentielle Ursachen von Fehlern stoßen, ohne dass dort jemals zur Laufzeit ein Überfluss auftreten kann. Da in diesem Fall der Workflowgraph bereits inkorrekt ist, ist diese Überabschätzung nicht schön, aber auch nicht gefährlich.

Finden wir in einem beliebigen Workflowgraphen auf Basis der Bemerkung 7.8 die Ursache eines potentiellen Überflusses, so muss der Workflowgraph *inkorrekt* sein. Das heißt, wir identifizieren in jedem Fall zuverlässig inkorrekte und korrekte Workflowgraphen hinsichtlich der Ursachen von Überflüssen. Wiederum auf Basis dieser Bemerkung können wir jedoch *nicht* garantieren, dass jemals ein Überfluss an der gefundenen Stelle während der Ausführung auftritt. Das tatsächliche Auftreten von Überflüssen zur Laufzeit kann beispielsweise durch Verklemmungen verhindert werden. Außerdem können in inkorrekten Workflowgraphen Überflüsse an Stellen auftreten, an denen wir sie nicht vorhersagen. Dies kommt dadurch, dass wir nicht alle Überflüsse sondern deren *Ursachen* betrachten. Dadurch finden wir immer nur die ersten möglichen Stellen von Überflüssen; ein späteres Manifestieren zur Laufzeit ist aber möglich.

Zusammengefasst können wir jedoch garantieren, dass der Workflowgraph *inkorrekt* ist, sobald wir die potentielle Ursache eines Überflusses gefunden haben. Außerdem finden wir mindestens eine Ursache eines Überflusses sobald in einem Workflowgraphen ein Überfluss möglich ist. Wir *sind* demnach hinsichtlich der Korrektheit *vollständig*. Wie sich später in der Evaluation zeigen wird, sind die gefundenen potentiellen Ursachen sehr präzise und die Verhinderung eines Überflusses durch Verklemmungen gut nachvollziehbar.

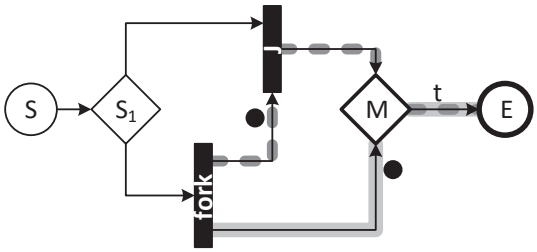


Abbildung 7.4: Eine Verklemmung verhindert einen Überfluss

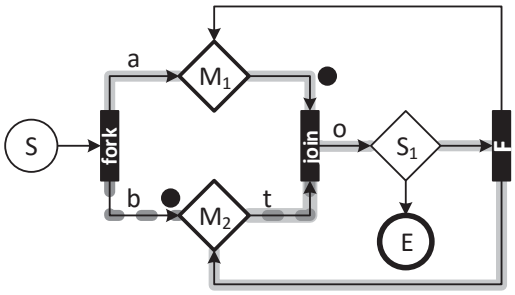


Abbildung 7.5: Beispiel eines überflüssigen Treffpunkts t

Betrachten wir dazu ein simples Beispiel: In Abbildung 7.4 ist ein einfacher Workflowgraph illustriert. Für diesen ist die Kante t ein Treffpunkt des Forkknotens $fork$, denn ab der oberen und unteren ausgehenden Kante von $fork$ existieren zwei Routen zu t (dunkelgrau gestrichelt und grau hinterlegt). t ist offenbar laut Definition nicht überflüssig. Nach der letzten Bemerkung ist bei t ein Überfluss möglich, da die Quelle von t kein Joinknoten ist. Nun liegt jedoch auf der oberen gestrichelten Route der Joinknoten J. Dieser kann offenbar in keiner Berechnung ab dem Startzustand ausgeführt werden. Demzufolge gerät der Workflowgraph immer in eine Verklemmung in J. Aus diesem Grund tritt zur Laufzeit niemals ein Überfluss auf. Dennoch wissen wir in diesem Fall, dass der Workflowgraph inkorrekt ist, auch wenn sich diese Inkorrektheit nicht durch einen Überfluss äußert.

In Abbildung 7.5 ist ein korrekter Workflowgraph abgebildet, in dem ein überflüssiger Treffpunkt existiert. Die Kante t des Graphen ist ein über die Routen W_a (grau solide) und W_b (dunkelgrau gestrichelt) erreichbarer (überabgeschätzter) Treffpunkt des Forkknotens $fork$. Und dieser Treffpunkt t besitzt offensichtlich einen Mergeknoten als Quelle. Er ist aber *überflüssig*, da der Joinknoten $join$ die Marke von t zur Ausführung benötigt ($join$ ist abhängig von t). Die ausgehende Kante o von $join$ ist ebenso ein Treffpunkt und $join$ synchronisiert die Kontrollflüsse des Forkknotens stets korrekt, *bevor* zwei Marken t erreichen können. Dadurch erreicht t zu jedem Zeitpunkt immer nur maximal eine Marke.

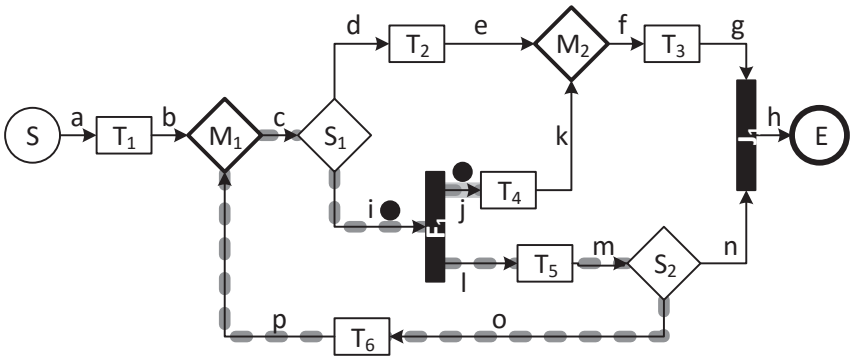


Abbildung 7.6: Die ausgehende Kante j von F_1 ist ein Treffpunkt

Bei genauer Betrachtung der Definition der (nicht-überflüssigen) Treffpunkte gibt es genau einen Sonderfall, sobald eine ausgehende Kante t eines Forkknotens selbst ein Treffpunkt darstellt. Dabei besteht eine der Routen nur aus t. Die andere Route hingegen besteht aus einem Zyklus, der wieder zum Forkknoten führt. Damit treffen sich die beiden Routen erstmals in der ausgehenden Kante t.

Ist eine ausgehende Kante eines Forkknotens selbst ein Treffpunkt des Forkknotens, so ist der Workflowgraph laut Bemerkung 7.8 inkorrekt. Er ist inkorrekt, da offensichtlich der Forkknoten nochmals ausgeführt werden kann, *bevor* all seine Marken synchronisiert wurden. Dadurch ist eine beliebige, wenn nicht sogar endlose Zahl an Marken möglich.

Einen solchen Fall beinhaltet auch unser Beispielworkflowgraph aus Abbildung 7.6. In diesem sind die zwei Routen zum Treffpunkt j eingezeichnet. Offenbar können über diese beiden Routen zwei Marken gleichzeitig auf j gelangen.

7.3 Bestimmung von Überflüssen

Nachdem wir bisher in diesem Kapitel die Ursachen von Überflüssen untersucht haben, behandeln wir am Ende dieses Kapitels deren algorithmische Bestimmung. Nach Bemerkung 7.8 sind dazu drei Schritte nötig: (1) Die Bestimmung der Treffpunkte der Forkknoten, (2) die Bestimmung der abhängigen Treffpunkte für jeden Treffpunkt, und (3) die Eliminierung der überflüssigen Treffpunkte.

Diese drei Schritte untersuchen wir in den folgenden drei Abschnitten.

7.3.1 Bestimmung der Treffpunkte

Das Interessante an dem Problem der Bestimmung der Treffpunkte ist die Äquivalenz zu einem klassischen Problem aus dem Übersetzerbau: Dem Setzen der ϕ-Funktionen in der minimalen Static Single Assignment Form (SSA-Form) [2][66].

Bei der Platzierung von ϕ-Funktionen geht es um die Übersetzung eines beliebigen Kontrollflussgraphen mit Variablen und Anweisungen in die SSA-Form. Die Besonderheit in der SSA-Form liegt in der (statisch) einmaligen Wertzuweisung einer jeden Variablen. Wenn also ein Kontrollflussgraph in die SSA-Form übersetzt wird, wird für jede Zuweisung einer Variablen eine neue Variable (Definition) eingefügt. Wenn nun an einer Stelle im Kontrollflussgraphen zwei unterschiedliche Definitionen der selben Variablen aufeinandertreffen, werden ϕ-Funktionen eingesetzt. Diese wählen abhängig vom tatsächlich zur Laufzeit stattgefundenen Kontrollfluss die passende Definition. Eine ϕ-Funktion hat dabei den Aufbau $d_n = \phi(d_1, \ldots, d_m)$, $m \geq 2$, wobei die verschiedenen Definitionen $d_1, \ldots, d_m$ in dieser ϕ-Funktion zusammenlaufen und die von der ϕ-Funktion zur Laufzeit gewählte gültige Definition in d_n abgelegt wird.

Letztendlich können für alle Definitionen an allen Zusammenführungspunkten in *Kontrollflussgraphen* ϕ-Funktionen gesetzt werden; mit dem Resultat, dass überflüssige ϕ-Funktionen entstehen. In der sogenannten *minimalen* SSA-Form werden ϕ-Funktionen nur an den Stellen platziert, an denen sie (statisch) gesehen[1], notwendig sind [14]. Cytron et al. beschreiben Positionen für ϕ-Funktionen als Knoten, in denen zwei Wege von zwei unterschiedlichen Definitionen der gleichen Variablen v erstmals zusammentreffen [14]. Bei Übertragung dieser Definition auf Kanten entspricht sie gerade der Definition unserer Treffpunkte.

Cytron et al. haben für die Erzeugung der minimalen SSA-Form einen Algorithmus entworfen. Für alle von zwei beliebigen, unterschiedlichen Knoten startenden Wege ermittelt dieser die ersten konvergierenden Knoten. Dieser Algorithmus kann äquivalent auch auf Kanten angewandt werden.

Für die Bestimmung der Treffpunkte eines Forkknotens $fork$ nehmen wir aus diesem Grund auf all seinen ausgehenden Kanten eine Zuweisung einer für $fork$ spezifischen Variablen v in der Form $v \leftarrow \phi(x)$ an, mit x sei ein beliebiger Wert (vgl. Abbildung 7.7). Für diese Variable v nutzen wir den Algorithmus von Cytron et al. [14] und berechnen die Positionen für die weiteren ϕ-Funktionen, die für v minimal notwendig sind. Dabei kann auf die Algorithmen in der erwähnten Arbeit oder auch auf schnellere Umsetzungen zurückgegriffen werden (bspw. Lengauer und Tarjan [44] und Cooper et al. [12]). Bei der Anwendung der Algorithmen muss nur darauf geachtet werden, dass viele davon von einer Variablenzuweisung auf der Startkante des Workflowgraphen ausgehen, was in unserem Fall jedoch keine Relevanz besitzt. Als Ergebnis platzieren die Algorithmen ϕ-Funktionen im Workflowgraphen in Bezug auf den Knoten $fork$ und dessen spezifische Variable v (vgl. die Positionen der ϕ-Funktionen in Abbildung 7.8). Jede ϕ-Funktion die mindestens zwei Parameter besitzt, ist demnach Schnittpunkt zweier bis dahin disjunkter Wege und somit auch

[1] Bei dem statischen Setzen von ϕ-Funktionen wird davon ausgegangen, dass die Fairness gilt; das heißt, jeder Weg im Kontrollflussgraphen kann von einem Kontrollfluss abgearbeitet werden.

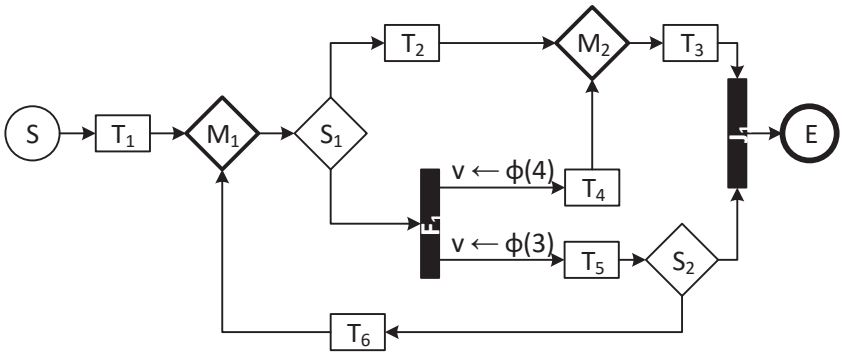


Abbildung 7.7: Zwei Variablenzuweisungen für eine virtuelle Variable v

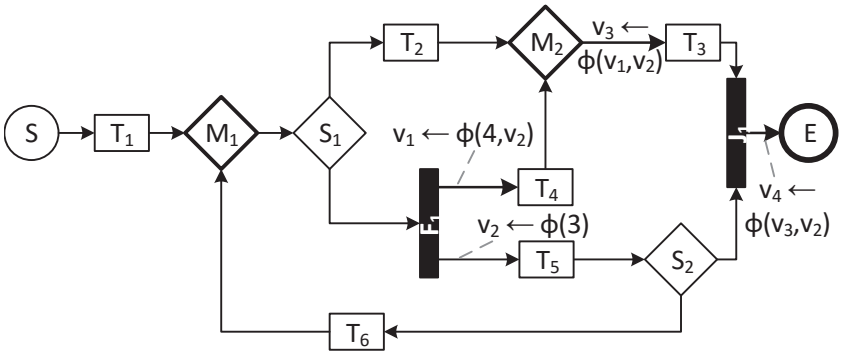


Abbildung 7.8: Platzierung der ϕ-Funktion für die virtuelle Variable v

Treffpunkt des Forkknotens $fork$. Die Bestimmung von Treffpunkten ist gelöst.

Bei genauer Betrachtung von Abbildung 7.8 fällt auf, dass die obere ausgehende Kante des Forkknotens ein Treffpunkt des Forkknotens sein muss. Dies liegt daran, dass es für diese Kante zwei Wege – der Weg bestehend aus der Kante selbst und der Weg von der unteren ausgehenden Kante des Forkknotens zu dieser Kante – gibt, die sich erstmals in dieser Kante treffen. Während der Ausführung kann es in solch einem Fall dazu kommen, dass die Marke auf der Kante verbleibt und die Marke der anderen ausgehenden Kante diese erreicht und einen Überfluss verursacht.

Algorithmus 3 gibt einen Überblick über eine mögliche Umsetzung zur Bestimmung aller Treffpunkte der Forkknoten. Die asymptotische Laufzeit des Algorithmus hängt im Wesentlichen von der Laufzeit der Konstruktion der minimalen SSA-Form ab. Diese ist nach Cytron et al. [14] kubisch, $O(X^3)$, in Bezug auf das Maximum X aus der Anzahl der Knoten, Kanten und Variablenzuweisungen. Da die Anzahl der Knoten in der Regel kleiner ist als die Anzahl der Kanten und ebenso die Anzahl der ausgehenden Kanten (mit Variablenzuweisungen) von Forkknoten kleiner ist als die Gesamtanzahl an Kanten, folgt $X = E$. Damit ist in unserem Fall die Worst-Case-Laufzeit $O(E^3)$.

Algorithmus 3 Bestimmung aller Treffpunkte aller Forkknoten

Require: Workflowgraph $WFG = (N, E, l)$

Ensure: Menge $\curlywedge(fork)$ aller Treffpunkte eines jeden Forkknotens $fork$

```
/** Initialisierung: Einfügen einer Zuweisung für alle ausgehenden Kanten der Forkknoten **/
for all fork ∈ N_Fork do
    for all out ∈ fork◁ do
        Füge φ-Funktion out ← φ(x) ein, x sei ein beliebiger Wert
/** Berechnung der minimalen SSA-Form **/
Berechne minimale SSA-Form (bspw. nach Cytron et al.)
/** Bestimmung der Treffpunkte **/
for all fork ∈ N_Fork do
    for all φ-Funktionen φ von fork do
        Sei t die Kante, auf der φ liegt
        if Parameteranzahl von φ ≥ 2 then
            ⋏(fork) ← ⋏(fork) ∪ {t}
```

7.3.2 Bestimmung der abhängigen Treffpunkte

Nachdem im letzten Abschnitt die Treffpunkte der Forkknoten bestimmt wurden, muss nach Bemerkung 7.8 für jeden Treffpunkt entschieden werden können, ob er überflüssig oder wichtig ist. Ein Treffpunkt t ist genau dann wichtig, wenn es von $fork$ zu t mindestens zwei Routen gibt, auf denen *kein* von t abhängiger Treffpunkt bzgl. $fork$ liegt. Wollen wir also feststellen, ob ein Treffpunkt wichtig ist, so müssen wir für ihn seine abhängigen Treffpunkte bestimmen.

Laut Definition hängt eine Kante d bzgl. $fork$ von einer Kante t ab, wenn es von der eingehenden Kante in von $fork$ zur Kante d einen Weg gibt und *nach* der Löschung von t kein Kontrollfluss ab in mehr d erreicht. Anderenfalls ist d *unabhängig* von t. Die Betrachtung der Kontrollflüsse entspricht jedoch einer Erkundung des Zustandsraums und macht die Berechnung ineffizient.

Die Möglichkeit einer effizienten Berechnung bietet uns die Bestimmung der von t *unabhängigen* Kanten. Denn haben wir diese bestimmt, so sind alle anderen Kanten, also das Komplement, abhängig von t.

Ein Großteil der unabhängigen Kanten von t ergibt sich bereits durch die Untersuchung, ob es überhaupt einen Weg von der eingehenden Kante in von $fork$ zu einer zu betrachtenden Kante d gibt. Gibt es einen solchen Weg nicht, so gibt es auch keinen Weg von t nach d (da es einen Weg von in nach t gibt, gäbe es sonst auch einen Weg zu d). d ist also unabhängig vom Forkknoten $fork$ bzw. dessen eingehender Kante in und somit auch von t. Die Berechnung dieser Menge unabhängiger Kante, genannt *Unreachable*, kann bspw. durch das Komplement der von in erreichbaren

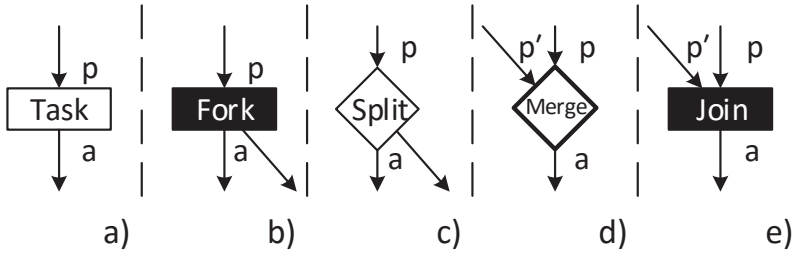


Abbildung 7.9: Ein direkter Nachfolger einer unabhängigen Kante ist ebenfalls unabhängig, außer es handelt es sich um einen Joinknoten

Kanten erfolgen. Die von *in* erreichbaren Kanten lassen sich wiederum durch eine einfache Tiefensuche bestimmen.

Für die Berechnung der übrigen unabhängigen Kanten von t gehen wir nun davon aus, dass es einen Weg von *in* zu ihnen gibt. Dann hilft uns der folgende Zusammenhang zwischen einer beliebigen Kante und ihren direkten Vorgängern:

Bemerkung 7.9.

Zwei beliebige Kanten a und t seien angenommen. Weiterhin sind noch die direkten Vorgänger $P = \triangleright src(a)$ von a gegeben.

a ist *unabhängig* von t genau dann, wenn

1. $src(a)$ ein Task-, Fork-, Split- oder Mergeknoten ist und mind. ein direkter Vorgänger $p \in P$ nicht von t abhängt ($\underset{p \in P}{\exists} p \not\gg t$), Abbildungen 7.9 *a)* bis *d)*, oder

2. $src(a)$ ein Joinknoten ist und alle direkten Vorgänger $p \in P$ unabhängig von t sind ($\underset{p \in P}{\forall} p \not\gg t$), Abb. 7.9 *e)*.

Demnach besitzt jede unabhängige Kante (bis auf die Startkante) mindestens eine unabhängige Kante als direkten Vorgänger.

Aufgrund der letzten Bemerkung ergibt sich aus den unabhängigen Kanten einer Kante t (inklusive der dazugehörigen Knoten) ein zusammenhängender Graph [53, S. 547]. Da die noch zu betrachtenden unabhängigen Kanten von t von *in* erreichbar sind, sind sie offenbar alle Nachfolger von *in*. Die Menge der Nachfolger von *in* bildet somit eine Obermenge der noch zu bestimmenden unabhängigen Kanten von t. Diese Obermenge lässt sich durch eine Tiefensuche ausgehend von *in* bestimmen, wenn die zu untersuchende Kante t aus dem Graphen entfernt wird.

Der algorithmische Ansatz ist, ausgehend von dieser Obermenge *Independent*, *iterativ* die tatsächlichen, unabhängigen Kanten von t zu bestimmen. In jeder Iteration wird dabei *Independent* aktualisiert, in dem diejenigen Kanten herausgenommen werden, die abhängig von t sind. Erreichen wir einen Fixpunkt, das heißt, es findet

keine Aktualisierung mehr statt, dann terminiert der Algorithmus. *Independent* entspricht dann der Menge der unabhängigen Kanten von t, zu denen die eingehende Kante *in* von $fork$ einen Weg besitzt.

Jede Kante aus *Independent* (bis auf *in*) besitzt *immer* einen direkten Vorgänger in dieser Menge. Demnach können nach Bemerkung 7.9 ausgehende Kanten von Task-, Fork-, Split- und Mergeknoten erst aus *Independent* herausgenommen werden, wenn bereits all ihre direkten Vorgänger entfernt wurden. Hingegen kann die ausgehende Kante eines Joinknotens bereits herausgenommen werden, wenn mindestens ein direkter Vorgänger der Kante nicht in *Independent* liegt.

Es ergeben sich daraus zwei Schritte für jede Iteration: (1) Das Untersuchen und Herausnehmen von ausgehenden Kanten von Task-, Fork-, Split- und Mergeknoten; und (2) das Untersuchen und Herausnehmen von ausgehenden Kanten von Joinknoten. Da erstere, herauszunehmende Kanten keine direkten Vorgänger mehr in *Independent* besitzen, sind diese auch nicht mehr von der eingehenden Kante *in* von $fork$ erreichbar. *Independent* besteht also *nach* Schritt (1) nur noch aus den Kanten, die von *in* über die Kanten aus *Independent* erreicht werden können. Anstelle also im Schritt (1) die zu entfernenden Kanten zu bestimmen, kann mit Hilfe einer einfachen Tiefensuche ab *in* die aktuelle Menge *Independent* direkt bestimmt werden. Die ausgehenden Kanten der Joinknoten können im Schritt (2) anhand der Menge *Independent* überprüft und bei Abhängigkeit herausgenommen werden.

Erreicht der Algorithmus einen Fixpunkt für die Kante t, so kennen wir die Menge aller unabhängigen Kanten von t. Dann ergibt sich die Menge der abhängigen Kanten $Dependent(fork, t)$ bzgl. des Treffpunkts t und des Forkknotens $fork$ durch $Dependent(fork, t) = E \setminus (Independent \cup Unreachable \cup \{t\})$.

Der vollständige Algorithmus zur Bestimmung der Menge der *abhängigen* Kanten der Treffpunkte eines Forkknotens kann in Algorithmus 4 nachvollzogen werden. Abbildung 7.10 veranschaulicht die Schritte des Algorithmus am Beispiel der Kante l und des Forkknotens F_1 unseres Beispielworkflowgraphen. In der Teilabbildung *a)* ist der Workflowgraph zu sehen. Die zu untersuchende Kante l ist durch eine gestrichelte, dickere Linie markiert, gehört aber nicht mehr zur Menge *Independent*. Die dünnen gestrichelten Linien markieren die Menge der von der eingehenden Kante i von F_1 nicht erreichbaren Kanten *Unreachable*. In Abb. *b)* wird auf diesem Graphen eine Tiefensuche beginnend ab der eingehenden Kante i durchgeführt. Alle erreichten Kanten (dicke Linien) bleiben in *Independent*, alle anderen wurden entfernt (illustriert durch entfernte Kanten im Beispiel). Danach findet, wie in Algorithmus 4 beschrieben, eine Überprüfung der Joinknoten statt. Im Beispiel gibt es nur den Joinknoten J_1. Für diesen ist nur noch die eingehende Kante g in *Independent*. Daher ist die ausgehende Kante h nicht unabhängig von l und kann aus *Independent* entfernt werden, Abb. *c)*. Wie sich leicht überprüfen lässt, bringt

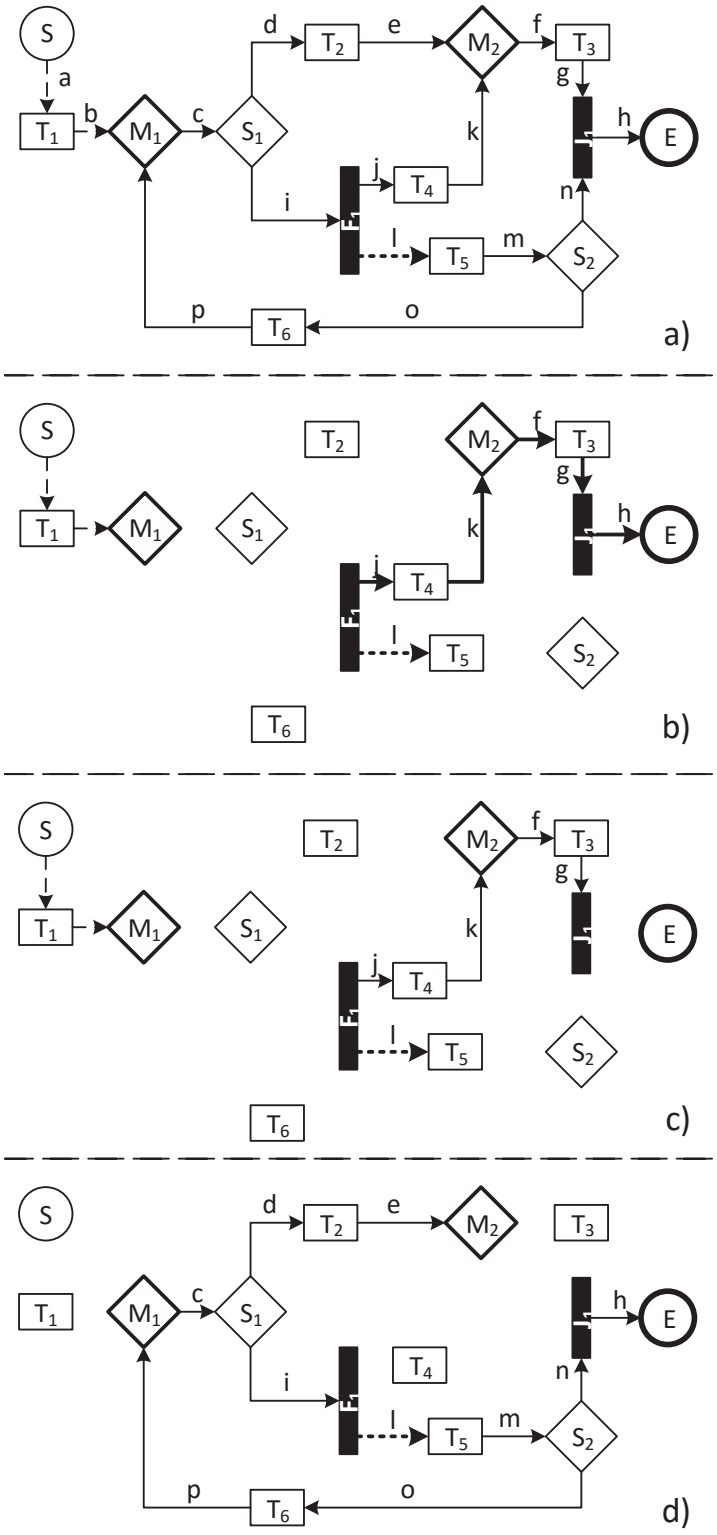


Abbildung 7.10: Anwendung des Algorithmus auf den Workflowgraphen am Beispiel der Kante l.

Algorithmus 4 Bestimmung der abhängigen Kanten

Require: Workflowgraph $WFG = (N, E, l)$ mit dem Forkknoten $fork$ und seiner eingehenden Kante in.

Ensure: Für jeden Treffpunkt $t \in \curlywedge(fork)$ von $fork$ die Menge der von t abhängigen Kanten $Dependent(fork, t)$.

```
/** Bestimme die von in erreichbaren Kanten mittels Tiefensuche über der Kantenmenge E. **/
Reachable ← depthFirstSearch(E, in)
Unreachable ← E ∖ Reachable
for all t ∈ ⋏(fork) do
    /** Initialisierung. Alle von in erreichbaren Kanten (bis auf t) sind unabhängig von t. **/
    Independent_Old ← Independent ← Reachable ∖ {t}
    repeat
        Independent_Old ← Independent
        /** Tiefensuche ausgehend von in auf Kanten aus Independent.
            Gefundene Kanten bilden aktualisiertes Independent. **/
        Independent ← depthFirstSearch(Independent, in)
        /** Joinknoten, deren eingeh. Kanten nicht vollständig in Independent liegen,
            sind abhängig von edge. Deren ausgeh. Kanten sind nicht unabhängig von t. **/
        for all join ∈ N_Join do
            if ▷join ∉ Independent then
                Independent ← Independent ∖ join◁
        /** Wiederhole bis Independent sich nicht mehr ändert. **/
    until Independent_Old = Independent
    /** Die Menge der von t abhg. Kanten umfasst alle Kanten ohne die unabhg. **/
    Dependent(fork, t) ← E ∖ (Independent ∪ Unreachable ∪ {t})
```

eine erneute Tiefensuche keine Änderungen an *Independent*. Damit terminiert die Repeat-Until-Schleife des Algorithmus. Die abhängigen Kanten von l ergeben sich nun durch $E \setminus (Independent \cup Unreachable \cup \{l\})$, vgl. Abb. *d)*.

Das asymptotische Laufzeitverhalten des Algorithmus 4 wird für einen Forkknoten $fork$ durch drei Schleifen bestimmt: (1) Die äußere For-All-Schleife, die (2) Repeat-Until-Schleife und (3) die innere For-All-Schleife. Die äußere For-All-Schleife (1) iteriert einmalig über die Menge der Treffpunkte von $fork$. Dies entspricht im schlechtesten Fall $O(E)$. Die dazugehörige Tiefensuche für die Bestimmung der Menge *Unreachable* ist linear, $O(E)$. Die innere For-All-Schleife (3) hingegen untersucht jeden Joinknoten, das heißt, $O(N_{Join})$. Die Repeat-Until-Schleife (2) wird maximal sooft ausgeführt, wie Kanten entfernt werden können, das heißt, $O(E)$. Durch die innerhalb der Repeat-Until-Schleife durchgeführte Tiefensuche, $O(E)$, und die innere For-To-Schleife (3), ergibt sich eine Gesamtlaufzeit von $O(E(E + N_{Join}))$. Schätzen

wir N_{Join} im Worst-Case mit E ab, erhalten wir eine asymptotisch quadratische Laufzeit in Abhängigkeit der Kantenmenge für die Repeat-Until-Schleife und damit eine kubische Laufzeit $O(E^3)$ für den Algorithmus 4. Untersuchen wir die abhängigen Kanten für jeden Forkknoten, so ergibt sich demnach im schlechtesten Fall eine biquadratische Laufzeit $O(E^4)$.

In der Literatur wird jedoch stark vermutet, dass die Gesamtanzahl der ϕ-Funktionen bei der Konstruktion der SSA-Form und somit die Gesamtanzahl aller Treffpunkte aller Forkknoten im Workflowgraphen linear zur Größe des Graphen ist[2] [5, S. 49 ff.][6, S. 399][14][90]. Dies zeigt sich durch jede bisherige Untersuchung praktisch relevanter Programme und wird in der Literatur mitunter als gegeben angenommen (bspw. von Surendran et al. [73]). Nehmen wir die Gesamtanzahl aller Treffpunkte aller Forkknoten abschätzend durch $O(E)$ an, so wäre die Iteration über alle Forkknoten und die dazugehörigen Treffpunkte nicht mehr quadratisch. Die Gesamtlaufzeit reduzierte sich dann im schlechtesten Fall auf $O(E^3)$ – sie wäre kubisch. Aus Gründen der formalen Korrektheit geben wir die asymptotische Laufzeit dieses Algorithmus im schlechtesten Fall dennoch als biquadratisch an, wohl wissend, dass sie in fast ausschließlich allen Fällen kubisch ist.

Auf Basis der Menge der abhängigen Kanten für jeden Treffpunkt können abschließend für sie einfach deren abhängige Treffpunkte berechnet werden.

7.3.3 Bestimmung der wichtigen Treffpunkte

Für die Bestimmung, ob ein Treffpunkt t eines Forkknotens $fork$ überflüssig oder wichtig ist, müssen wir überprüfen, ob es zwei Routen von $fork$ zu t gibt, auf denen kein von t abhängiger Treffpunkt bzgl. $fork$ liegt.

Für die Lösung dieses Problems greifen wir auf die Theorie der *Flussnetzwerke* [13, S. 722] zurück. Ein Flussnetzwerk ist ein beliebiger Digraph $G = (N, E)$ mit genau einer ausgezeichneten *Flussquelle* $source \in N(G)$ und genau einer ausgezeichneten *Flusssenke* $sink \in N(G)$. Die Flussquelle besitzt dabei keine eingehende Kante. Zudem besitzt jede Kante e des Flussnetzwerks eine ganzzahlige *Maximalkapazität* $cap(e)$ an möglichen *Flüssen*, die sie nutzen können. In Abb. 7.11 *a)* ist ein kleines Flussnetzwerk zu sehen.

Das bekannteste Problem auf Flussnetzwerken ist die Berechnung des *maximalen Flusses* (Max-Flow-Problem) [13, S. 721 ff.][25]. Dabei sollten von der Flussquelle größtmöglich viele Flüsse starten, ohne dabei die Kapazitäten der einzelnen Kanten zu überschreiten (vgl. Abb. 7.11 *b)*). Die obere Schranke entspricht dabei der Summe der Kapazitäten aller eingehenden Kanten in die Flusssenke, $\sum_{e \in \triangleright sink} cap(e)$.

[2] Der interessierte Leser findet im Anhang D die Auswertung der Anzahl der Treffpunkte der in der Evaluation (siehe Kapitel 9) verwendeten, praxisnahen Prozessbibliothek. Diese bestätigt die Annahme einer zu E linearen Anzahl an Treffpunkten aller Forkknoten.

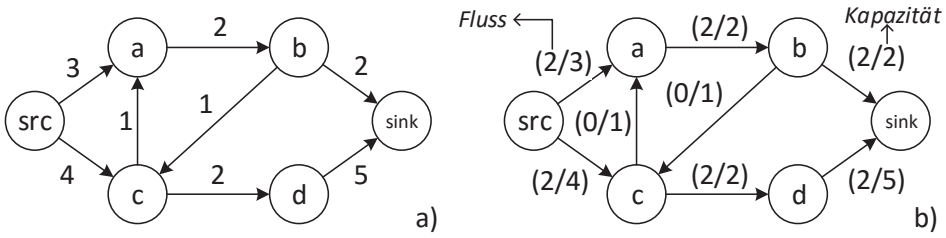


Abbildung 7.11: Ein einfaches Flussnetzwerk, *a)*, und sein max. Fluss, *b)*

Das Problem des maximalen Flusses kann zur Überprüfung genutzt werden, ob ein Treffpunkt t eines Forkknotens $fork$ wichtig ist. Das heißt, es lässt sich mit ihm prüfen, ob es mindestens zwei bis auf t disjunkte Wege von $fork$ zu t gibt, die keinen von t abhängigen Treffpunkt passieren. Dazu formulieren wir unser Problem in das des maximalen Flusses um, indem wir den Workflowgraphen bzgl. t und $fork$ in ein Flussnetzwerk übertragen.

Ein Fluss entspricht in etwa einem Kontrollfluss bzw. genauer der „Wanderung“ einer Marke von einer ausgehenden Kante von $fork$ zur Kante t. Dabei kann über jede Kante maximal ein Fluss / eine Marke wandern. Würden über eine Kante zwei Flüsse laufen, so wären die gefundenen Wege nicht disjunkt. Das heißt, wir nehmen zunächst die Kapazität einer jeden Kante e mit 1 an, $cap(e) = 1$. Als Flussquelle *source* legen wir den Forkknoten $fork$ fest, da er im Workflowgraphen die Kontrollflüsse erzeugt. Die Flusssenke *sink* ist die Quelle der Kante t, $src(t)$.

Im vorangegangenen Abschnitt haben wir die von t abhängigen Treffpunkte bzgl. $fork$ bestimmt. Um zu vermeiden, dass ein Fluss eine solche Kante passiert, setzen wir einfach die Kapazität all dieser abhängigen Treffpunkte auf 0. Dadurch fließt kein Fluss durch diese Kanten.

Zu guter Letzt müssen wir noch die eingehende Kante *in* des Forkknotens $fork$ eliminieren, um der Definition eines Flussnetzwerks zu genügen. Dadurch würden wir jedoch potentielle Wege von einer ausgehenden Kante von $fork$ nach t zerstören. Um dies zu vermeiden, eliminieren wir die Kante *in nicht*. Anstatt dessen lösen wir die Kante *in* von $fork$ und führen einen weiteren Knoten $fork'$ ein, den wir anstelle von $fork$ mit *in* verbinden (vgl. Abbildung 7.12, Schritt 1). Im nächsten Schritt lösen wir jede ausgehende Kante *out* von $fork$ und fügen für jede einen neuen Mergeknoten M_{out} ein, den wir mit *out* verbinden (vgl. Abbildung 7.12, Schritt 2). Danach fügen wir für jede dieser ausgehenden Kanten *out* sowohl eine Kante von $fork$ als auch eine Kante von $fork'$ zu M_{out} ein. Diese neuen Kanten bekommen alle die Kapazität 1, außer *out* ist ein abhängiger Treffpunkt von t, dann bekommen sie die Kapazität 0. Das Resultat ist in Abbildung 7.12, Schritt 3, zu sehen. Wie leicht zu überprüfen ist, kann dennoch über jede ursprüngliche ausgehende Kante von $fork$ maximal ein Fluss laufen.

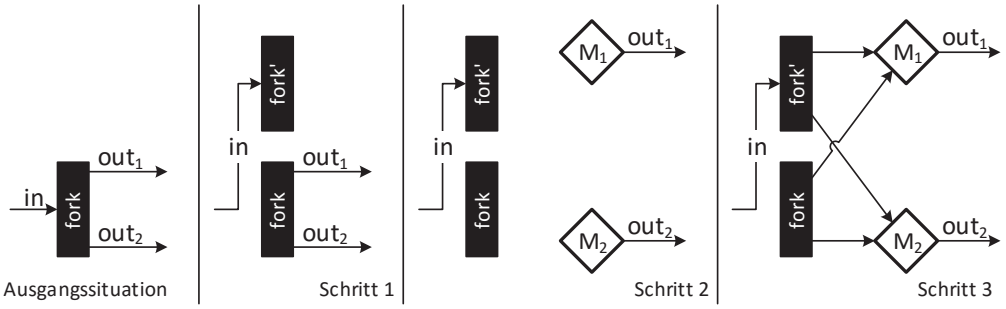


Abbildung 7.12: Schritte zur Duplizierung eines Forkknotens zur Beibehaltung der Wege über die Kante *in*

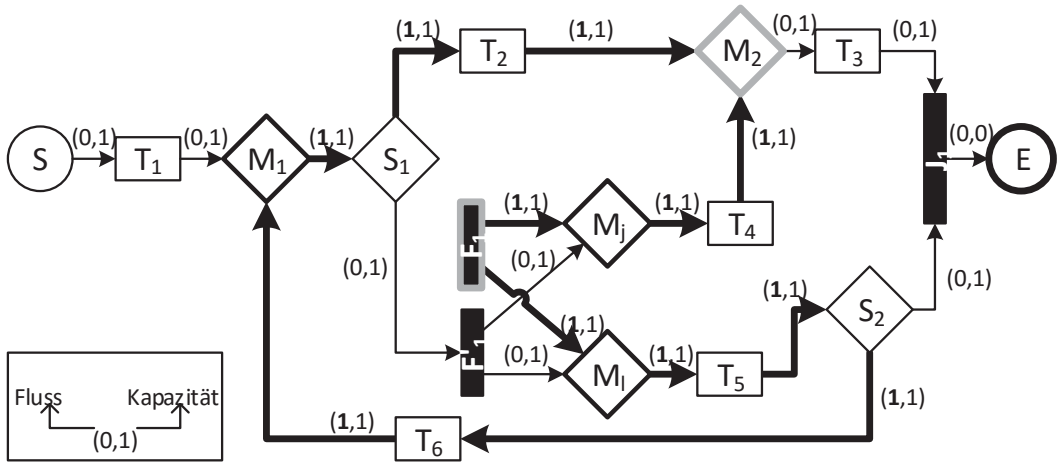


Abbildung 7.13: Das Flussnetzwerk bzgl. des Forkknotens F_1 und der Kante (M_2, T_3)

Damit ist die Transformation des Workflowgraphen bzgl. $fork$ und t in ein Flussnetzwerk abgeschlossen. Unser transformierter Beispielworkflowgraph ist bzgl. des Forkknotens F_1 und dem Treffpunkt (M_2, T_3) in Abbildung 7.13 illustriert.

Auf dem erzeugten Flussnetzwerk wird nun der maximale Fluss ausgehend von $fork$ berechnet (vgl. Abbildung 7.13). Ist der in der Flusssenke (also bei $src(t)$, M_2 im Beispiel) ankommende Fluss größer gleich 2, dann gibt es zwei disjunkte Wege W_1 und W_2 von $fork$ zu t. Diese laufen durch die 0-Kapazität auch nicht über einen abhängigen Treffpunkt von t. Damit ist t ein wichtiger, nicht überflüssiger Treffpunkt von $fork$. Im Beispiel ist (M_2, T_3) ein solcher wichtiger Treffpunkt von F_1, da ihn über (T_2, M_2) und (T_4, M_2) zwei Flüsse erreichen.

Anderenfalls, also wenn der ankommende Fluss kleiner gleich 1 ist, ist t überflüssig. t kann bzgl. des Forkknotens $fork$ als Treffpunkt eliminiert werden.

In Abbildung 7.14 ist das Flussnetzwerk des bereits schon vorher betrachteten, korrekten Workflowgraphen aus Abbildung 7.5, S. 70, zu sehen. Zur Erinnerung, in diesem besitzt der Forkknoten $fork$ einen überflüssigen Treffpunkt $(M_2, join)$ und

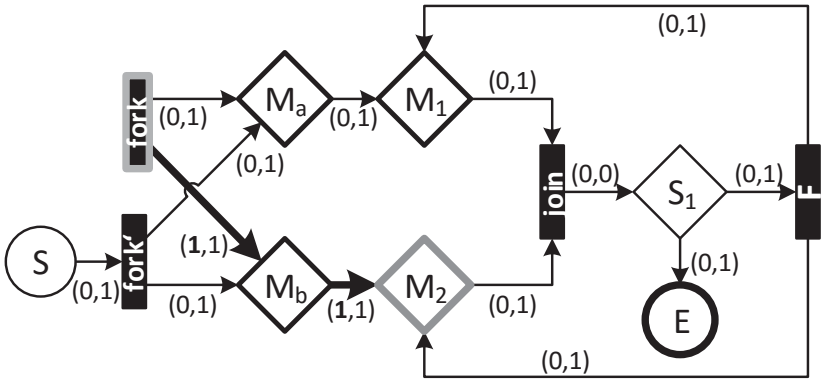


Abbildung 7.14: Das Flussnetzwerk bzgl. des Forkknotens $fork$ mit dem überflüssigen Treffpunkt $(M_2, join)$

die Kante $(join, S_1)$ ist von diesem abhängig. Wie gut in der Abbildung zu sehen ist, ist der bei $(M_2, join)$ ankommende maximale Fluss gleich 1, da über $(join, S_1)$ aufgrund der Kapazität 0 kein Fluss fließen kann – $(M_2, join)$ ist ein überflüssiger Treffpunkt von $fork$. Das heißt, unsere Analyse erkennt diesen korrekt.

Unser Problem der Überprüfung, ob ein Treffpunkt überflüssig oder wichtig ist, können wir demnach erfolgreich durch die Transformation in ein Flussnetzwerk und der Berechnung des maximalen Flusses lösen. Die Berechnung des maximalen Flusses kann beispielsweise über den Algorithmus von Ford und Fulkerson [13, S. 727 ff.][46] geschehen. Dieser berechnet den maximalen Fluss $|f|$ in einer asymptotischen Laufzeit von $O(|f|E)$. Der maximale Fluss $|f|$ entspricht höchstens der Anzahl an Parametern der zu t gehörenden ϕ-Funktion. Dadurch kann der maximale Fluss im Normalfall als Konstante interpretiert werden. Die Berechnung des größtmöglichen Flusses ist somit in linearer Zeit $O(E)$ möglich.

Die Transformation des Workflowgraphen bzgl. eines Forkknotens $fork$ und einer potentiellen Synchronisierungskante t ist offenbar ebenso in linearer Zeit möglich $O(E)$. Da wir im Worst-Case sowohl die Anzahl der Forkknoten N_{Fork} als auch die Anzahl der für einen Forkknoten zu untersuchenden Treffpunkte mit E abschätzen können, erreichen wir für die Überprüfung aller Treffpunkte eine kubische Gesamtlaufzeit $O(E^3)$.

Abschließend ist der gesamte Algorithmus zur Ursachenbestimmung von Überflüssen in Algorithmus 5 zusammengefasst. Dieser besitzt aufgrund der Bestimmung der abhängigen Treffpunkte eine biquadratische Gesamtlaufzeit, die für praxisnahe Prozesse kubisch ist.

Algorithmus 5 Bestimmung der Ursachen von Überflüssen

Require: Workflowgraph $WFG = (N, E, l)$

Ensure: Menge $Causes \subseteq N_{Fork} \times E$ aller Ursachen von Überflüssen

```
/** Initialisierung **/
Bestimme die zu jedem Forkknoten fork gehörenden Treffpunkte 人(fork)
Bestimme die zu jedem Treffpunkt gehörenden abhängigen Treffpunkte
/** Untersuchung aller Treffpunkte, die keinen Joinknoten als Quelle besitzen **/
for all fork ∈ N_Fork do
    for all t ∈ 人(fork) do
        if src(t) ∉ N_Join then
            /** Überprüfung, ob t ein überflüssiger Treffpunkt ist **/
            Transformiere WFG bzgl. fork und t in Flussnetzwerk
            Ermittle maximalen Fluss maxFlow des Flussnetzwerks
            if maxFlow >= 2 then
                Causes ← Causes ∪ {(fork, t)}
```

Kapitel 8

Das Werkzeug Mojo

Um die in dieser Arbeit entstandenen Algorithmen und Techniken zu evaluieren, wurde ein Werkzeug in Java mit dem Namen *Mojo* entwickelt. Mojo integriert sich in unser Konzept des in Abbildung 8.1 illustrierten Systems zur Ausführung von Arbeitsprozessen [60]. Es bildet im derzeitigen Entwicklungsstand einen Teil der im Bild dargestellen *Produzentenseite* ab.

Auf der Produzentenseite, dem Übersetzer, wird ein Arbeitsprozess eingelesen (Parser). Während des Einlesens findet eine Überprüfung der Struktur statt. Außerdem wird der Prozess in die Zwischencoderepräsentation (ZCR) überführt (Transformierer). Als ZCR benutzt Mojo die in dieser Arbeit eingeführten Workflowgraphen. Diese abstrahieren von den speziellen Eigenschaften der Eingabesprachen, das heißt, von den benutzten Modellierungssprachen der Arbeitsprozesse.

Nachdem die ZCR erstellt wurde, wird der Prozess semantischen Überprüfungen unterzogen. Als Beispiel wäre die in dieser Arbeit vorgestellte Korrektheitsverifikation genannt. Werden dabei Fehler festgestellt, können diese in Rücksprache mit dem Entwickler des Prozesses beseitigt werden. Ist die ZCR jedoch bereits semantisch korrekt, so wird sie kodiert und in einer Datei bzw. Prozessablage gespeichert.

Neben der Produzentenseite unterteilt sich das System noch in eine *Konsumentenseite*. Die Konsumentenseite ist eine virtuelle Maschine. Sie liest aus der Prozessablage oder Datei die ZCR ein und verifiziert sie nochmals durch eine semantische Überprüfung. Diese Überprüfung kann durch Annotationen der ZCR auf der Produzentenseite beschleunigt werden. Sie ist notwendig, um sicherzustellen, dass die ZCR nicht manipuliert wurde. Anschließend führt die virtuelle Maschine den Prozess aus und kontrolliert ihn ggf. durch Laufzeitanalysen.

Genauere Informationen über das Gesamtsystem und die darin enthaltenen Konzepte können in Prinz et al. [60] nachgelesen werden. Eine Übersicht über den Übersetzer und die virtuelle Maschine bieten die Arbeiten [56] und [59].

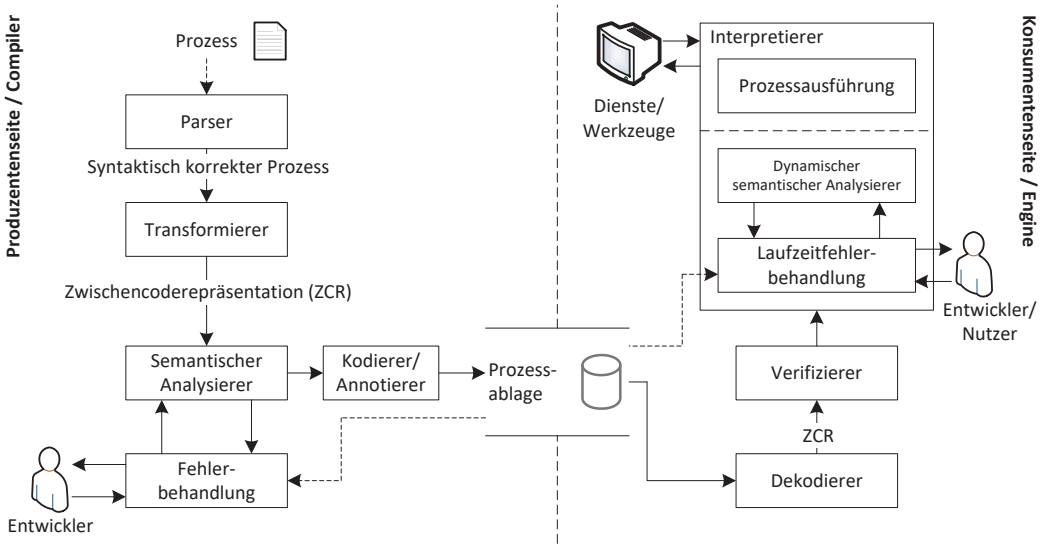


Abbildung 8.1: Überblick über ein System zur Ausführung von Arbeitsprozessen

8.1 Konzeptioneller Aufbau

Unser Übersetzer Mojo entspricht, wir bereits erwähnt, einer ersten Version einer möglichen Produzentenseite unseres vorgeschlagenen Gesamtsystems. Es deckt eine Vielzahl der Schritte der Produzentenseite ab.

Konzeptionell ist Mojo ein erweiterbares System. Durch Erweiterungspunkte kann es beispielsweise um weitere Eingabesprachen als auch um weitere Analysen ergänzt werden. Damit steht es auch anderen Forschungen im Arbeitsprozessbereich offen.

Der Ablauf der Analysen wird in Mojo durch *Analysepläne* geregelt. Ein Analyseplan definiert die notwendigen Phasen, um die Analysen korrekt durchzuführen. Dabei kann eine solche Phase selbst wieder ein komplexer Analyseplan sein und weitere Phasen beinhalten. Damit folgt Mojo einem klassischen Übersetzer.

Einen groben Überblick von Mojo zeigt Abbildung 8.2. In der aktuell vorliegenden Version sind Eingaben über Dateien der Beschreibungssprachen Petri Net Markup Language (PNML) und BPMN sowie über die direkte programmatische Eingabe von Workflowgraphen möglich. Für PNML und BPMN gibt es entsprechend vordefinierte Plugins. Sie bestehen aus einem Parser, der die Dateien in ihre Bestandteile zerlegt, und einem Transformierer, der die Arbeitsprozesse auf Basis der semantischen Eigenheiten der Eingangssprache in semantisch äquivalente Workflowgraphen überführt. So werden zum Beispiel mehrere Startknoten zu einem einzigen verbunden und mehrere Endknoten mit Kiepuszewksis Algorithmus [38] zu einem vereint.

Die entstandenen Workflowgraphen können anschließend durch die zur Verfügung stehenden Analysepläne untersucht werden. Typische Bestandteile eines solchen Analyseplans sind Dominator- und Postdominatoranalysen sowie die Bestimmung der

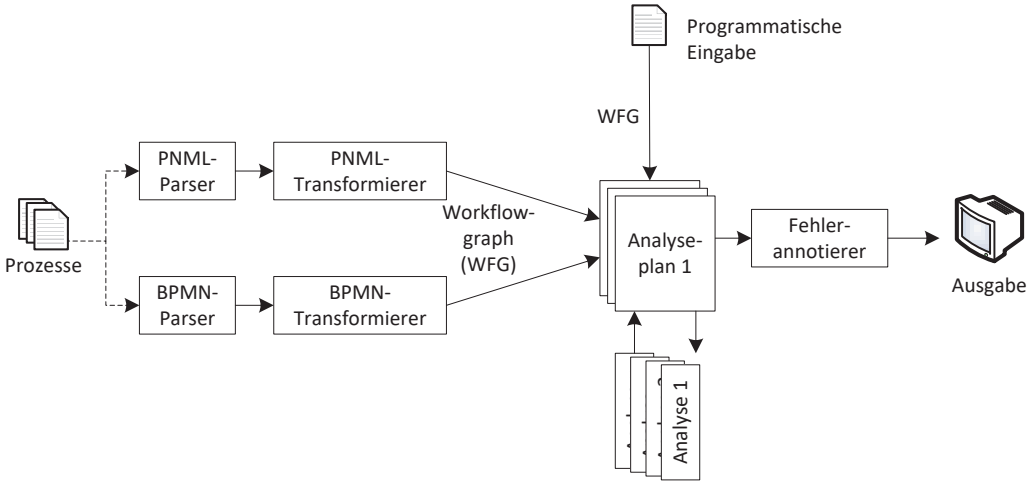


Abbildung 8.2: Aufbau des Übersetzers Mojo

Ursachen von Verklemmungen und Überflüssen. Der jeweilige Analyseplan wird bei Nutzung des Werkzeugs festgelegt. Da jeder Analyseplan eine eindeutige Nummer besitzt, ist ein präzises Auswählen unkompliziert.

Der Analyseplan mit der Nummer 0 wird zur Evaluierung unserer Algorithmen eingesetzt. Er ist ein allgemeiner Plan zur Untersuchung von Workflowgraphen und besteht aus den folgenden Phasen: (1) Die Vorbereitung, (2) die Ermittlung der (Post-)Dominanzgrenzen und (Post-)Dominatorrelationen und (3) die Ursachenanalysen von Überflüssen und Verklemmungen.

In der ersten Phase, (1) Vorbereitung, wird der Workflowgraph für die weiteren Schritte modifiziert. Es werden weitere Taskknoten eingefügt, um die Eindeutigkeit von Mehrfachkanten zwischen Knoten zu gewährleisten. Zudem werden die Kanten des Workflowgraphen extrahiert. Diese Kanten sind notwendig, um im Schritt (2) die (Post-)Dominanzgrenzen und (Post-)Dominatorrelationen zu ermitteln.

Basierend auf den gesammelten Informationen werden im letzten Schritt (3), der Ursachenanalyse, parallel die Analysen zur Erkennung der Gründe von Verklemmungen und Überflüssen gestartet. Diese basieren auf den in dieser Arbeit vorgestellten Algorithmen. Alle dabei festgestellten Fehlerursachen werden registriert und an den Workflowgraphen annotiert. Sie können entweder in textueller Form ausgegeben oder aber durch grafische Visualisierung direkt in einem Entwicklungswerkzeug hervorgehoben werden.

Eine solche Integration in ein Entwicklungswerkzeug wurde durchgeführt. Dazu wurde der *Activiti BPMN 2.0 Designer*[1] angepasst und mit Mojo verbunden. Dies ist möglich, da Mojo sowohl als eigenständige Java-Bibliothek als auch als Eclipse-Plugin genutzt werden kann.

[1] `http://activiti.org/`

Abbildung 8.3 zeigt den Einsatz von Mojo in dem Prozessdesigner *Activiti*. Dabei wird in jedem Konstruktionsschritt eines Prozesses ohne merkbare Latenz eine Analyse durch Mojo durchgeführt. Um dies zu gewährleisten, transformiert Mojo das interne Prozessmodell von *Activiti* in ein bis mehrere Workflowgraphen. Danach wird ein entsprechender Analyseplan ausgeführt. Die gewonnenen Analyseergebnisse werden innerhalb unserer *Activiti*-Erweiterungen grafisch aufbereitet und dem Nutzer visualisiert. Der Nutzer kann die Fehler in zwei unterschiedlichen Modi begutachten. Im *Überblicksmodus* sind alle gefundenen Fehler mit reduzierten Informationen zu sehen, um ein schnelles Bild über die Korrektheit des Prozesses zu bekommen. Im *Detailmodus* hingegen wird vom Nutzer ein Fehler ausgewählt, für den alle Diagnoseinformationen visualisiert werden. So ist die Fehlerursache gut nachvollziehbar und der Nutzer kann den Prozess entsprechend korrigieren.

Neben der Nutzung von Mojo als Analysebibliothek bietet es den Einsatz als selbstständiges Werkzeug. Dazu wird es direkt über die Kommandozeile benutzt. Mit Hilfe des Befehls

```
java -classpath core-all-<CORE_VERSION>.jar de.jena.uni.mojo.Mojo
```

wird Mojo gestartet. Durch die Erweiterung des Klassenpfades ist es zudem durch Plugins ergänzbar. Mit verschiedenen Parametern kann das Verhalten von Mojo außerdem noch gesteuert werden. Diese werden vollständig im Anhang A erklärt.

8.2 Verwendete Algorithmen

Mojo verwendet für die Findung der Ursachen potentieller Verklemmungen und Überflüsse die in dieser Arbeit eingeführten Algorithmen. Die für diese Algorithmen notwendigen Dominator- und Postdominatorrelationen und die dazugehörigen Dominanz- und Postdominanzgrenzen werden mit Hilfe der Algorithmen von Cooper et al. [12] bestimmt. Die Algorithmen von Cooper et al. sind beispielsweise auch in der *LLVM Compiler Infrastructure*[2] implementiert und besitzen den Vorteil, dass sie im Gegensatz zum derzeit besten Algorithmus (hinsichtlich seiner asymptotischen Laufzeit) von Lengauer und Tarjan [44] recht einfacher Natur sind [12]. Auch wenn die Algorithmen ein quadratisches Laufzeitverhalten in der Theorie besitzen, werden sie in der Praxis von Cooper et al. aber als mindestens genauso effizient eingestuft, da sich die Nutzung des komplexeren Algorithmus von Lengauer und Tarjan erst überhaupt ab einer Knoten-/Kantenanzahl von über 30.000 rentiert [12].

[2] `http://llvm.org/`

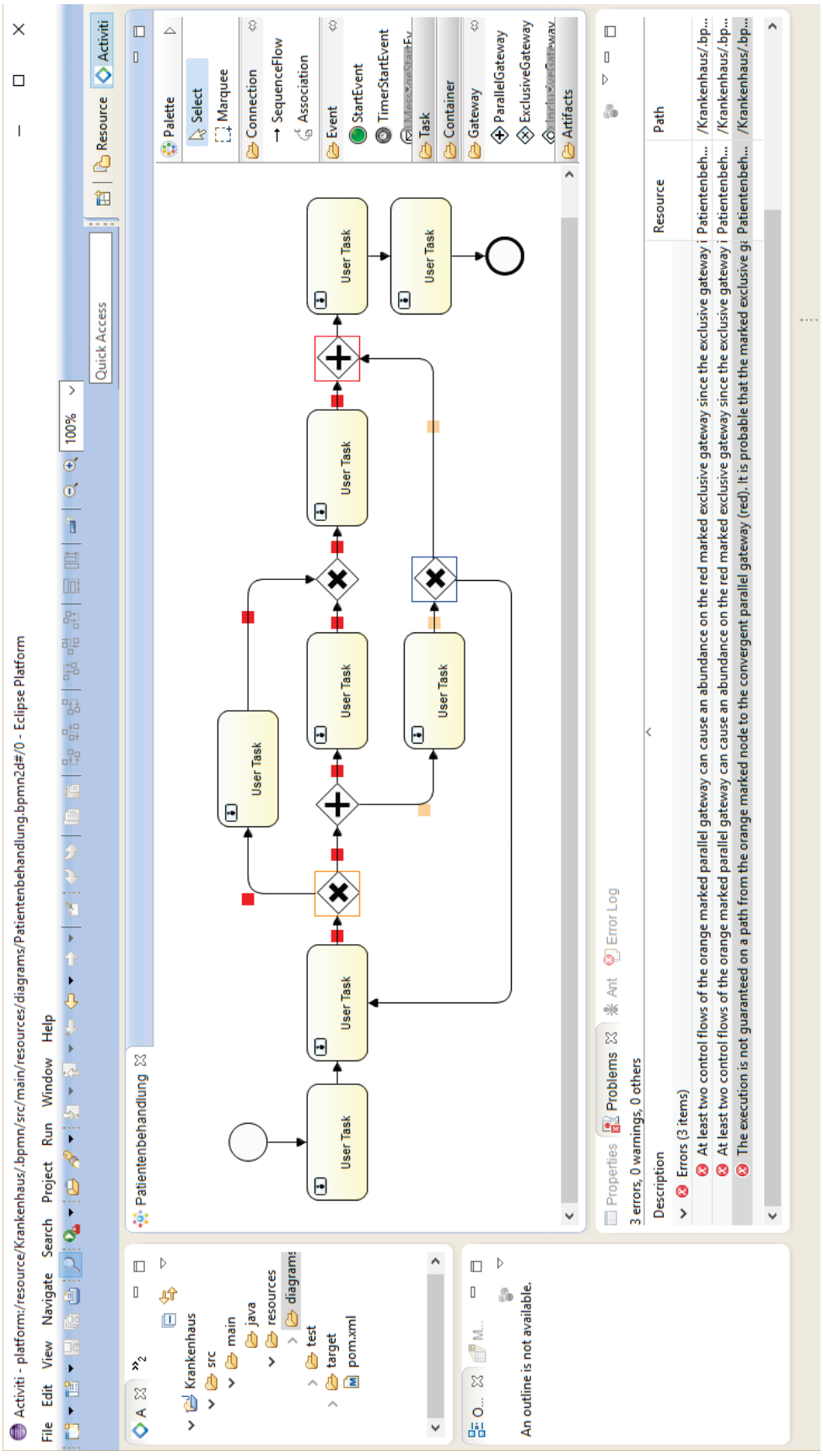


Abbildung 8.3: Integration von Mojo in den Activiti BPMN 2.0 Designer

Kapitel 9

Evaluation

In den letzten Kapiteln wurden die Grundlagen gelegt, aus denen dann Algorithmen zur Bestimmung der Ursachen von potentiellen Verklemmungen und Überflüssen hergeleitet werden konnten. In diesem Kapitel wollen wir nun diese Algorithmen hinsichtlich der Qualität und Quantität der gefundenen Fehlerursachen sowie hinsichtlich des Laufzeitverhaltens in realen Anwendungsfällen evaluieren.

Zu diesem Zweck nutzen wir das im letzten Kapitel vorgestellte Werkzeug *Mojo*. Mit diesem überprüfen wir eine Bibliothek mit über 1.000 realen Arbeitsprozessen auf Korrektheit. Für diese Überprüfung wird kurz in die Testumgebung eingeführt. Im Anschluss folgt die Evaluation anhand eines Vergleichs unseres Ansatzes mit zwei weiteren Ansätzen zum Korrektheitsnachweis. Dies ermöglicht uns die Güte der gefundenen Fehlerursachen, die dazugehörigen diagnostischen Informationen und die Anzahl der gefundenen Fehler sowie die Laufzeit der Algorithmen zu vergleichen.

9.1 Testumgebung

Die Testumgebung für Mojo wurde auf einem handelsüblichen Rechner mit einem 64-Bit *Debian GNU/Linux 9.0 (stretch)* Betriebssystem installiert. Die Version des Linux-Kernels hieß *4.8.0-2-amd64 x86_64*. Der Rechner besaß als Ausstattung einen *4-Kern Intel©Core™ i5-4570* Prozessor mit jeweils *3,2* GHz Rechengeschwindigkeit und *8 GB* Arbeitsspeicher. Mojo wurde mit der Java-Version *1.8.0_111* der OpenJDK Laufzeitumgebung ausgeführt. Der für Mojo zugesicherte Heapspeicher während der Evaluationen betrug 2.048 MB.

Für quantitative Aussagen von Evaluationen wird eine große Anzahl an Testfällen benötigt. Dadurch wird die Wirkung verschiedener nicht relevanter Einflussgrößen minimiert. Im Bereich der Korrektheitüberprüfung von Arbeitsprozessen wird in der Literatur häufig eine Bibliothek von realen Prozessmodellen des *IBM WebSphere*

*Business Modeler*s[1] genutzt. Diese Bibliothek umfasst 1.368 Prozesse. Unterteilt sind diese in fünf Benchmarks mit den Namen A (282 Prozesse), $B1$ (288 Prozesse), $B2$ (363 Prozesse), $B3$ (421 Prozesse) und C (32 Prozesse). $B1$ bis $B3$ beschreiben dabei kontinuierlich weiterentwickelte Prozesse.

Die Bibliothek stand kostenlos im Internet[2] von IBM Zürich im Original XML-Format des WebSphere Modelers zur Verfügung. Ein einfacheres Einlesen in einem standardisierten Format ist aber in dem PNML-Format [89] möglich. PNML beschreibt Petrinetze in einer einfachen Syntax mit Transitionen, Plätzen und den verbindenden Kanten. Erhältlich sind die Prozesse der Bibliothek im PNML-Format im Zusammenhang mit der Arbeit von Fahland et al., in der verschiedene Korrektheitsprüfer miteinander verglichen wurden [18].[3] Das PNML-Format der Bibliothek wurde auch für die Evaluation in dieser Arbeit verwendet.

Insgesamt bildet die Bibliothek einen schönen Querschnitt über Arbeitsprozesse verschiedener Größe, Struktur und Semantik. Einen Überblick über die Prozessbibliothek findet der interessierte Leser im Anhang B.

Die Prozessbibliothek wurde mit drei verschiedenen Werkzeugen untersucht: (1) Mit unserem Werkzeug Mojo [61], (2) mit dem Werkzeug *LoLA* [69] und (3) mit dem Werkzeug *jBPT* [55]. Das Werkzeug LoLA benutzten wir, um zu verifizieren, dass Mojo hinsichtlich des einfachen Korrektheitsnachweises richtige Ergebnisse liefert. Für LoLa werden die Petrinetze in einem speziellen, proprietären Format verlangt. Die dazu notwendigen Dateien können im Internet inklusive Anleitung zur Verwendung heruntergeladen werden.[4]

Das andere verwendete Werkzeug, *jBPT*, stellt das Projekt *Business Process Technologies 4 Java* zur Verfügung und ist ebenso im Internet als Open-Source erhältlich.[5] Es bietet die Möglichkeit die Algorithmen der Arbeiten von Vanhatalo et al. [84][85] zu nutzen (die SESE-Dekomposition); darunter auch die Konstruktion des Prozessstrukturbaums. Auf Basis des Prozessstrukturbaums bauten wir anhand der einfachen Regeln von Vanhatalo et al. [85] einen Korrektheitsnachweis für wohlstrukturierte Fragmente der Prozesse. Wohlstrukturierte Fragmente eines Prozesses sind sogenannte Verbände, die bei der Konstruktion des Prozessstrukturbaums entstehen [85]. Ein solcher Verband bildet eine typische Struktur aus einem öffnenden und einem schließenden Knoten, wie sie aus der Programmierung bekannt sind (bspw. If-Then-Else-Konstrukte).

Der von uns auf den wohlstrukturierten Fragmenten durchgeführte Korrektheitsnachweis untersucht die Passung der öffnenden auf die schließenden Knoten. Passen

[1] `http://www-03.ibm.com/software/products/de/modeler-basic` (Mai 2017)
[2] ehemals `http://www.zurich.ibm.com/csc/bit/downloads.html`
[3] `http://www.service-technology.org/publications/fahlandfjklvw_2009_bpm/` (Mai 2017)
[4] `http://www.service-technology.org/publications/fahlandfjklvw_2009_bpm/` (Mai 2017)
[5] `https://code.google.com/archive/p/jbpt/` (Mai 2017)

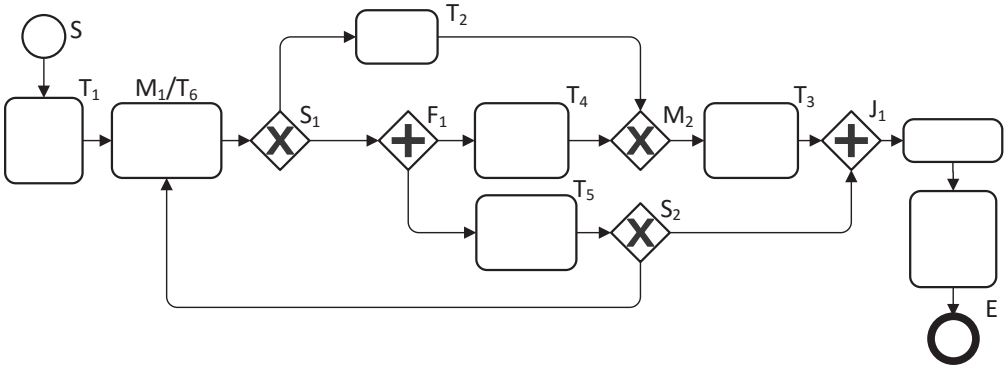


Abbildung 9.1: Beispielprozess zum Vergleich der Analyseergebnisse

der öffnende und schließende Knoten nicht zusammen (beispielsweise wird ein Splitknoten von einem Joinknoten geschlossen), so registriert der Algorithmus einen Fehler. Die heuristischen Regeln von Vanhatalo et al. wurden von dem Algorithmus *nicht* überprüft. Da dieser Algorithmus auf der SESE-Dekomposition basiert, bezeichnen wir die Analyse mit dem Werkzeug *jBPT* im Folgenden kurz als *SESE*.

Der Einsatz des Algorithmus von Vanhatalo et al. erfolgte, da er nach Fahland et al. die effizienteste Technik darstellt [18] (Stand 2011). Auch wenn der dazugehörige Ansatz nachgewiesen unvollständig ist, würde sich eine Vorbehandlung der strukturierten Prozessteile mit Hilfe dieses Ansatzes womöglich lohnen, so dass andere Techniken nur für die unstrukturierten Fragmente Anwendung finden müssten.

9.2 Direktvergleich der Analysetechniken

Im Zuge der Evaluation haben wir uns mit einem Direktvergleich unseres Ansatzes mit den verschiedenen Ansätzen der Zustandsraumerkundung und der SESE-Dekomposition befasst. Dazu wählten wir unseren Beispielprozess, der in der Darstellung eines BPMN-Prozesses in Abbildung 9.1 zu sehen ist.

Durch die Anwendung unserer Analysetechniken kann eine ähnliche Übersicht des Prozesses berechnet werden, wie Abbildung 9.2 zeigt. In dieser Übersicht bekommt ein Entwickler eines Prozesses einen Überblick über die gefundenen Ursachen von potentiellen Verklemmungen und Überflüssen. Dazu findet er nützliche Kurzhinweise zu jeder Fehlerursache. In diesem Fall sind es drei: (1) Eine potentielle Verklemmung in einem Joinknoten, (2) einen möglichen Überfluss durch eine fehlerhafte Synchronisierung und (3) eine potentielle Erzeugung beliebig vieler Kontrollflüsse und somit auch eines Überflusses. Die potentielle Erzeugung beliebig vieler Kontrollflüsse können wir genau dann bestimmen, wenn in einem Sonderfall eine ausgehende Kante eines Forknotens selbst ein (wichtiger) Treffpunkt ist. Es hat sich herausgestellt, dass dann

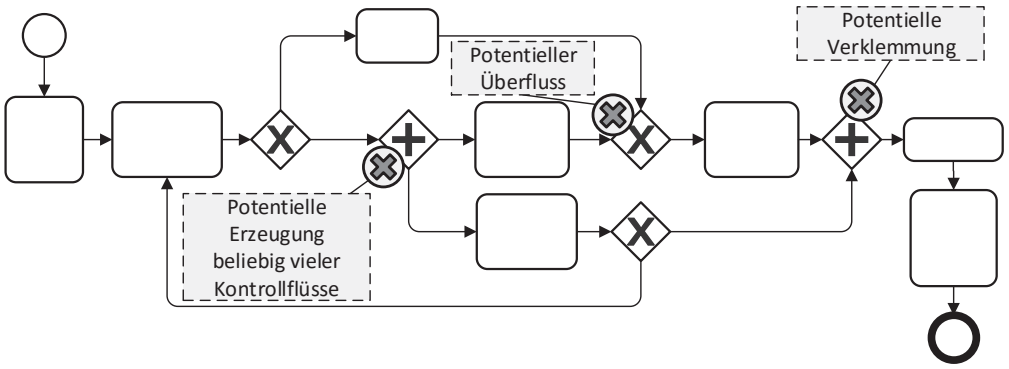


Abbildung 9.2: Gefundene Fehlerursachen durch unsere Techniken

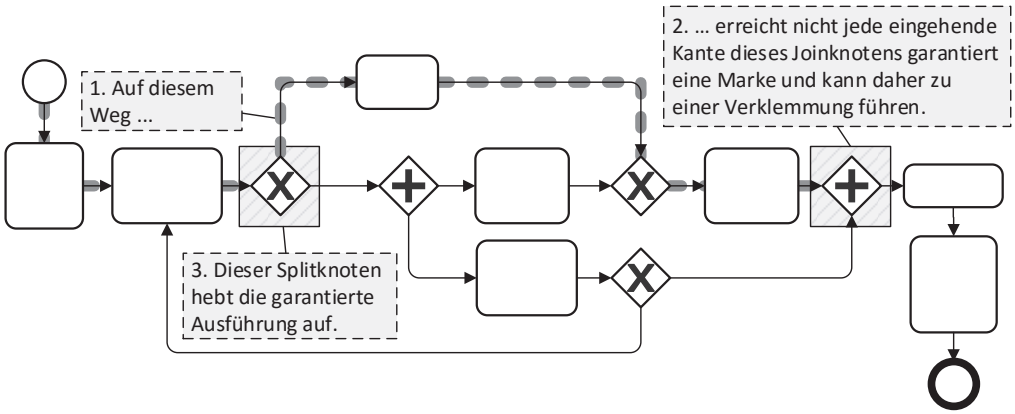


Abbildung 9.3: Detaillierte Fehleranalyse der Ursache der potentiellen Verklemmung im Joinknoten J_1

die Kontrollflüsse des Forkknotens vor erneuter Ausführung nicht synchronisiert werden. Dadurch können beliebig viele Kontrollflüsse entstehen. Im Folgenden nennen wir dies einen Überfluss in einer *Schleife*.

Jede Fehlerursache kann nochmals genauer vom Entwickler untersucht werden. Dazu werden zusätzliche, einfache Techniken angewandt, um detailliertere Diagnoseinformationen zu erhalten. Letztendlich laufen diese Techniken meist auf eine einfache Tiefensuche hinaus, deren Laufzeitverhalten mindestens genauso effizient ist, wie das unserer hier beschriebenen Ansätze. Beispielsweise zeigt Abbildung 9.3 eine Detailansicht der Ursache der potentiellen Verklemmung im Prozess. Die für diese Verklemmung abgeleiteten Informationen wurden durch eine inverse Tiefensuche gewonnen. Weiterhin wurde überprüft, an welcher Stelle es erstmals keine Ausführungskante einer eingehenden Kante des Joinknotens gibt.

Mit Hilfe dieser Informationen sieht der Entwickler deutlich, auf welchem Weg der Joinknoten nicht garantiert ausgeführt wird und welcher Knoten eine solche

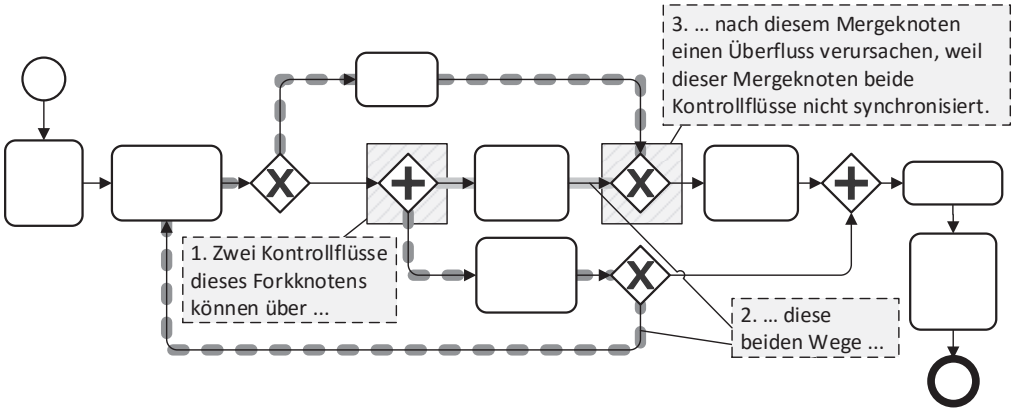


Abbildung 9.4: Detaillierte Analyse des Überflusses im Mergeknoten M_2

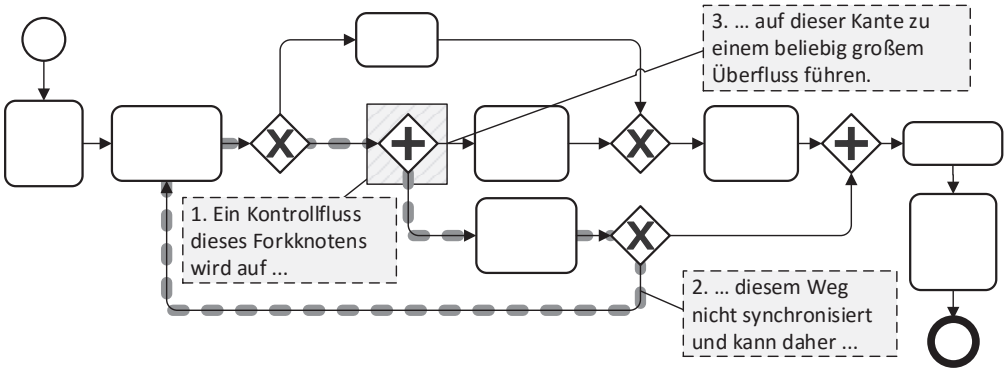


Abbildung 9.5: Der Forkknoten F_1 kann beliebig viele Kontrollflüsse erzeugen

garantierte Ausführung aufhebt. Da der Entwickler im Idealfall am besten weiß, wann und wie ein Joinknoten ausgeführt werden sollte, kann er den Prozess anhand dieser Informationen reparieren.

Ein ähnliches Feindiagnosebild liefert Abbildung 9.4 für den potentiellen Überfluss im Mergeknoten durch eine fehlende Synchronisierung. In diesem bekommt der Entwickler die möglichen Wege zweier Kontrollflüsse angezeigt und wie bzw. wo sie sich erstmals, fehlerhaft treffen können. Die dazu notwendigen Informationen können direkt aus dem maximalen Fluss abgelesen werden, der für jede fehlende Synchronisierung berechnet wird.

Auch für die letzte Fehlerursache, der Überflussschleife des Forkknotens F_1, können vom Entwickler weitere diagnostische Informationen angefordert werden (vgl. Abbildung 9.5). Diese umfasst bspw. den Weg über den keine Synchronisierung stattfindet, so dass der dazugehörige Forkknoten mehrfach ausgeführt werden kann. Dieser Weg kann ebenfalls über den maximalen Fluss abgelesen werden.

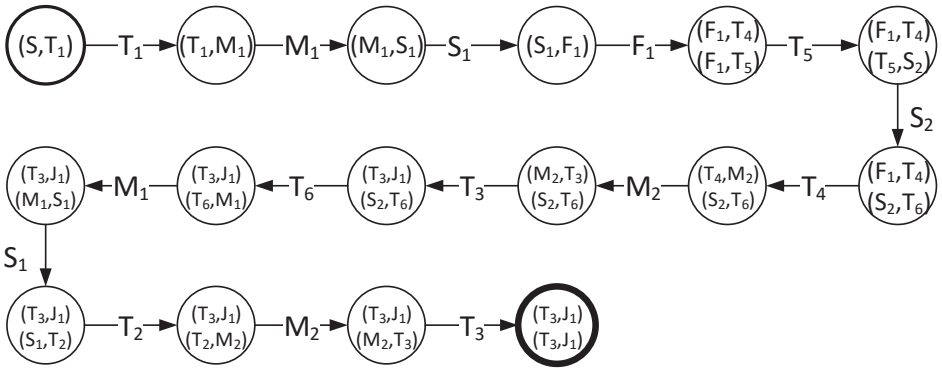


Abbildung 9.6: Fehlerspur einer Verklemmung bzw. Überflusses im Prozess

Zusammengefasst bieten unsere Techniken und damit auch Mojo eine sehr gute Diagnostik hinsichtlich der Ursachen von Fehlern. Diese können zahlreiche Hinweise und weitere Visualisierungen umfassen.

Nachdem wir nun die Qualität der gefundenen Fehlerursachen unseres Ansatzes betrachtet haben, möchten wir diese jetzt mit den zustandsraumbasierten Techniken vergleichen. Das Ergebnis einer Analyse des Prozesses durch eine Zustandsraumerkundung bietet dem Entwickler genau *eine* Fehler*wirkung*, beispielsweise eine Verklemmung. Zusätzlich zu der reinen Information über die Art der Fehlerwirkung bekommt der Entwickler eine sogenannte *Fehlerspur* (vgl. Abbildung 9.6). Eine solche Fehlerspur beinhaltet einen (Teil-)Kontrollfluss ausgehend von der Startkante inklusive der dabei durchlaufenen Zustände bis zum Fehler. Die vermutlich beste Visualisierung einer solchen Fehlerspur ist die Simulation, in dem der Entwickler schrittweise im Prozess angezeigt bekommt, wie der Fehler zustande kommt.

Positiv daran ist, dass der Entwickler durch die Zustandsraumerkundung einen tatsächlich zur Laufzeit auftretenden Fehler angezeigt bekommt und genau sieht, wie dieser zustande kommt. Negativ ist, dass die Analyse dem Entwickler nur einen einzigen Fehler liefert; denn eine vollständige Erkundung des Zustandsraums endet häufig in einer Zustandsraumexplosion, wie bereits in dieser Arbeit ausführlich erörtert wurde. Schwerwiegender wirken jedoch die Fehlerblockierung, -täuschung und -maskierung [23][32, S. 31][57]. Sie machen es dem Entwickler sehr schwer, die Ursache des gefundenen Fehlers zu erkennen und zu beheben [57]. Das Ergebnis einer Zustandsraumerkundung ist zwar recht passabel, zeigt aber noch deutliches Verbesserungspotential für die Unterstützung eines Entwicklers.

Zum Abschluss der Untersuchung der Qualität der gefundenen Fehler betrachten wir noch den Ansatz einer SESE-Dekomposition. Das Analyseergebnis der SESE-Dekomposition unseres Beispielprozesses (vgl. Abbildung 9.7) kann als solches nicht bezeichnet werden: Der größte Teil des Prozesses, welcher zu dem noch alle Fehlerursachen beinhaltet, ist ein unstrukturierter Teilgraph des Prozesses. Für unstruk-

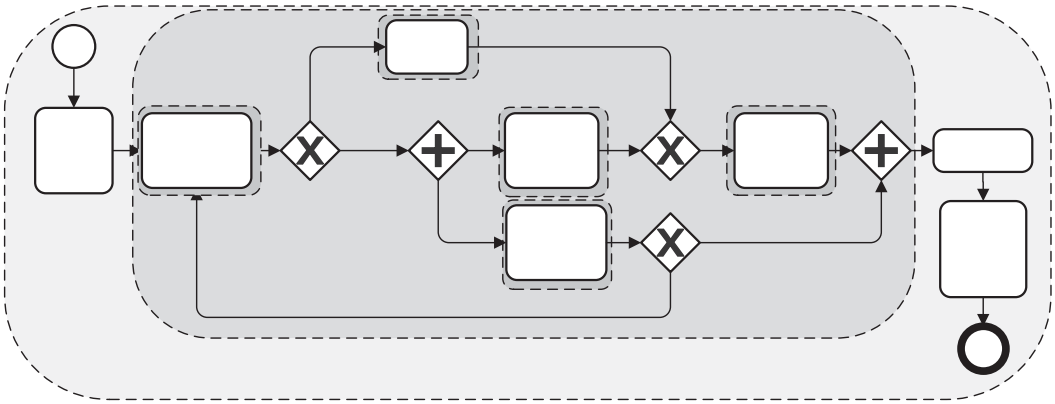


Abbildung 9.7: SESE-Dekomposition des Beispielprozesses

turierte Teilgraphen können jedoch nur Heuristiken angewandt werden, um Fehler zu erkennen. Diese liefern jedoch nur wenig diagnostische Informationen. Zumal hält nach Vanhatalo et al. [85] für dieses komplexe Fragment auch keine der bekannten Heuristiken. Dementsprechend ist bei Anwendung der SESE-Dekomposition für den Prozessentwickler unklar, ob der Prozess einen Fehler beinhaltet oder nicht.

Zusammengefasst bietet unser Ansatz den besten Überblick und die besten diagnostischen Details für die zur Behebung notwendigen Ursachen von Verklemmungen und Überflüssen für den hier gewählten Beispielprozess. Die Anwendung der Zustandsraumerkundung bietet zwar bei Auftreten eines Fehlers eine entsprechende Simulation als Fehlerdiagnostik, jedoch bleibt die Ursache des Fehlers wegen einer ungewissen Blockierung, Täuschung und Maskierung einer anderen Fehlerwirkung unklar. Zudem fehlt auch der Gesamtüberblick der im Prozess enthaltenen Fehler. Unklar bleibt auch die Erkennung von Fehlern bei Anwendung der SESE-Dekomposition bei Prozessen mit komplexen Strukturen. Für einfache, wohlgeformte Strukturen eignet sich hingegen die SESE-Dekomposition recht gut, da dort die Fehlerursache hinsichtlich eines falsch gewählten öffnenden bzw. schließenden Knotens einer solchen Struktur recht eingänglich zu erkennen ist.

9.3 Gefundene Fehlerursachen

Das Finden möglichst vieler Ursachen von Fehlern in einem Prozess sowie dessen Korrektheitsnachweis ist die Hauptaufgabe der verwendeten Werkzeuge in unserem Kontext. Aus diesem Grund haben wir untersucht, welche Prozesse die Werkzeuge als korrekt bzw. inkorrekt einstufen und zusätzlich, wie viele und welche Fehlerursachen die Werkzeuge finden. Da die zustandsraumbasierten Techniken keine Ursachen liefern, nutzen wir als Vergleichswert die gefundenen Laufzeitfehler.

	Verklemmungen	Überflüsse
A	140	170
B1	273	720
B2	326	948
B3	289	1.056
C	24	61
Summe	1.052	2.955
Insgesamt		4.007

Tabelle 9.1: Anzahl der Ursachen von Verklemmungen und Überflüssen (mit Mojo)

Wir beginnen zunächst mit der Analyse der gefundenen Fehlerursachen mit unserem Werkzeug Mojo. Dafür zeigt Abbildung 9.11, S. 107, eine Grafik, in der alle 1.386 Prozesse der Bibliothek durchnummeriert wurden und verschiedene Farben besitzen. Ist ein Prozess *weiß* gefärbt, so konnte Mojo in diesem *keine* Ursachen von Fehlern feststellen. *Grüne* Prozesse besitzen nach Mojo lediglich Ursachen für mögliche Verklemmungen. Überflussursachen sind demnach nicht feststellbar. Bei *blau* gefärbten Prozessen hingegen sind nur die Ursachen möglicher Überflüsse für Mojo erkennbar. Ist ein Prozess jedoch *orange* eingefärbt, so sind sowohl Ursachen für Verklemmungen als auch von Überflüssen im Prozess vorhanden.

Insgesamt identifiziert Mojo 742 Prozesse als inkorrekt und nur 644 als korrekt. In Tabelle 9.4, S. 101, sind die Ergebnisse zusammengefasst (Spalte *Mojo*).

Die Gesamtanzahl an gefundenen Fehlerursachen ist mit 4.007 sehr hoch. Damit enthält jeder Prozess durchschnittlich ca. 2 bis 3 Fehlerpotentiale. Tabelle 9.1 zeigt die Arten gefundener Fehler in den verschiedenen Benchmarks. Sie zeigt, dass die Ursachen für Verklemmungen weniger häufig zu denen von Überflüssen auftreten.

Nachdem wir nun die Anzahl der gefundenen Ursachen von Verklemmungen und Überflüssen für Mojo betrachtet haben, vergleichen wir diese Ergebnisse nun mit denen von LoLA. Tabelle 9.2 fasst die Anzahl gefundener Verklemmungen und Überflüsse von LoLA in den verschiedenen Benchmarks zusammen. Es ist deutlich zu sehen, dass LoLA im Vergleich zu Mojo nur ca. ein Viertel der Anzahl an Fehlern (Fehlerwirkungen!) findet. An dieser Stelle ist sehr interessant, welche Prozesse von Mojo und LoLA unterschiedlich erkannt werden. LoLA verwendet für seine Korrektheitsüberprüfung eine Zustandsraumerkundung und bricht beim ersten Fehlerzustand ab. Dabei wird eine separate Suche für Verklemmungen und für Überflüsse durchgeführt. Aus diesem Grund kann LoLA maximal 2 Laufzeitfehler pro Prozess erkennen.

Einen Vergleich zwischen den beiden Werkzeugen bietet Abbildung 9.12, S. 108. Die abgebildeten Kästchen sind mit unterschiedlichen Farben gefüllt. Wir benutzen weiße Kästchen, um Prozesse zu markieren, für die keine Unterschiede zwischen bei-

	Verklemmungen	Überflüsse
A	97	68
B1	81	200
B2	84	238
B3	83	262
C	10	14
Summe	355	782
Insgesamt		1.137

Tabelle 9.2: Anzahl der Verklemmungen und Überflüssen (mit LoLA)

den Werkzeugen festgestellt werden konnten. Das heißt, entweder (1) hat der Prozess weder in Mojo noch in LoLA einen Fehler, (2) beide Werkzeuge erkennen mindestens eine (Ursache einer) Verklemmung, (3) beide erkennen mindestens einen Überfluss oder (4) beide erkennen sowohl mindestens eine Verklemmung und einen Überfluss bzw. deren Ursachen. Wir konnten weitergehend fünf *Unterschiede* zwischen beiden Werkzeugen entdecken, die durch die Farben Rot, Lila, Grün, Orange und Blau markiert sind.

Der schlechteste Fall, dass LoLA einen Überflusszustand ableiten kann, Mojo den Prozess aber korrekt einschätzt (rot), taucht zwei Mal auf – in den Prozessen mit den Nummern 1.382 und 1.383. Um die Ursache dieses Fehlverhaltens aufzuklären, haben wir die beiden Prozesse genau untersucht. Beide Prozesse sind nicht zusammenhängend; das heißt, das beschriebene Petrinetz ist kein zusammenhängender Graph[6] [53, S. 547]. Mojo ist jedoch als Werkzeug entwickelt worden, dass die Konstruktion von Arbeitsprozessen bereits während der Erstellung unterstützt. Dadurch muss Mojo mit nicht-zusammenhängenden Prozessen umgehen können. Dafür konstruiert es automatisch nach den semantischen Regeln von BPMN einen zusammenhängenden Graphen aus den Teilgraphen des Prozesses. Nach intensiver Analyse dieser konstruierten Graphen konnten wir keine Ursachen von Fehlern feststellen. Da wir nicht wissen, wie LoLA mit diesen zerstückelten Teilprozessen umgeht, nehmen wir an, dass dort die Ursache der Divergenz liegt.

Ähnlich verhält es sich mit dem einzigen lila markierten Prozess 1.381, in dem LoLA einen Überfluss erkennt, Mojo jedoch eine Ursache für eine Verklemmung. Offenbar kann aus der Ursache einer Verklemmung kein Überflusszustand resultieren, wenn es keine Ursache für einen Überfluss gibt. Auch hier handelt es sich um einen unzusammenhängenden Prozess, der bei genauer Betrachtung offensichtlich eine Ursache für eine Verklemmung beinhaltet. LoLA erkennt aber aufgrund seiner anderen Herangehensweise mit zerstückelten Prozessen einen Überflusszustand.

[6] In der ungerichteten Version des Graphen besitzt jede Kante einen Weg zu jeder anderen Kante.

Tatsächlich handelt es sich bei diesen drei Prozessen um die einzigen Prozesse der gesamten Bibliothek, die nicht zusammenhängend sind. Aus diesem Grund sind die anderen Prozesse spannender. Es gibt 3 Prozesse (grün), in denen LoLA sowohl einen Überfluss als auch eine Verklemmung erreicht. In diesen findet Mojo aber nur Ursachen für Überflüsse. Für diese Prozesse wird deutlich, dass die Verklemmungszustände von LoLA aus einem Überflusszustand resultieren. Dieses Verhalten wird auch in anderen Prozessen häufiger der Fall sein, fällt jedoch aufgrund zuvor erkannter Fehler nicht auf.

Für die orange gefärbten Prozesse gilt, dass LoLA einen Überflusszustand erreicht, Mojo zusätzlich aber auch Ursachen von Verklemmungen findet. Dabei handelt es sich insgesamt um 171 Prozesse. In diesen Fällen erreicht LoLA immer zuerst einen Überflusszustand und bricht sowohl bei der Erkennung von Überflüssen als auch bei der Findung von Verklemmungen ab.

Zum Schluss gibt es auch 205 Prozesse (blau), in denen Mojo zusätzlich zu Ursachen von Verklemmungen auch die für Überflüsse erkennt. LoLA erreicht aber keinen Überflusszustand, da vorher bereits immer eine Verklemmung auftritt.

Insgesamt können wir als Resultat festhalten, dass Mojo allein von der Anzahl gefundener Fehler ein detaillierteres Fehlerbild der Prozesse liefert als LoLA. Außerdem gibt es noch viele Prozesse, in denen LoLA überhaupt entweder keinen Überfluss oder keine Verklemmung entdeckt. Bis auf die drei erwähnten Prozesse, erkennen sonst LoLA und Mojo die gleichen korrekten und inkorrekten Prozesse.

Zum Abschluss der Untersuchung der Fehler betrachten wir noch den SESE-Ansatz. Für SESE sind tatsächlich nur die „wohlgeformten“ Prozessstrukturen interessant. Für diese kann festgestellt werden, ob der schließende Knoten der Struktur mit dem Öffnenden zusammenpasst, sprich, ob ein Split- von einem Mergeknoten und ein Fork- mit einem Joinknoten geschlossen wird. Dies kann auch für einfache Schleifen untersucht werden. Es handelt sich dabei ähnlich zu Mojo um die Ursachen potentieller Fehler.

Tabelle 9.3 zeigt die Ergebnisse unserer Untersuchungen. Für die Benchmark *A* versagt SESE komplett und findet keine fehlerhafte Strukturen; das heißt, alle Fehler befinden sich in unstrukturierten Teilgraphen. Auch für die anderen Benchmarks ist die Fehlerausbeute eher mäßig. Auf den ersten Blick war dieses Ergebnis für uns überraschend. Wir hatten erwartet, dass SESE mehr Fehlerursachen identifiziert als LoLA Laufzeitfehler, da SESE ähnlich zu unserem Ansatz Ursachen von Fehlern in der Struktur findet. Dadurch kann SESE auch Fehlerursachen hinter Fehlern finden. Anscheinend kommen aber Fehler in unstrukturierten Teilgraphen der Prozesse deutlich häufiger vor als in strukturierten. Dadurch kann unsere Implementierung von SESE diese nicht finden.

	Verklemmungen	Überflüsse
A	0	0
B1	30	98
B2	36	113
B3	18	118
C	8	1
Summe	92	330
Insgesamt		422

Tabelle 9.3: Anzahl der Ursachen von Verklemmungen und Überflüssen (mit SESE)

		Mojo		LoLA		SESE		
	Gesamt	K	F	K	F	K	F	U
A	282	152	130	152	130	65	0	217
B1	288	107	181	107	181	75	92	121
B2	363	161	202	161	202	121	103	139
B3	421	207	214	207	214	144	107	170
C	32	17	15	15	17	9	7	16
Summe	1386	644	742	642	744	414	309	663

Tabelle 9.4: Anzahl der bestimmten korrekten (K), inkorrekten (F) und ungewissen (U) Prozesse mit den verschiedenen Werkzeugen.

Tabelle 9.4 macht die Unterschiede zwischen Mojo, LoLA und SESE deutlich. Der Buchstabe K steht für die Anzahl korrekter und der Buchstabe F für die Anzahl fehlerhafter Prozesse. Für SESE haben wir geschaut, welche Prozesse keine unstrukturierten Teilgraphen besitzen. Wurde für diese kein Fehler von SESE erkannt, so ist der Prozess korrekt. Wurde in einem beliebigen Prozess ein Fehler von SESE gefunden, so ist er inkorrekt, egal ob er unstrukturierte Bereiche besitzt oder nicht. Die Differenz zwischen der Gesamtanzahl der Prozesse zu den eindeutig korrekten und inkorrekten sind dann jene Prozesse, die nicht eingeschätzt werden können. Diese werden dann als *ungewiss* U bezeichnet.

Zusammengefasst besitzt Mojo in der Summe die beste Technologie zum Finden von Ursachen von Kontrollflussfehlern in Prozessen. Im letzten Teil der Evaluation überprüfen wir, ob sich Mojo diese detaillierte Fehlererkennung durch zeitintensivere Algorithmen erkauft.

9.4 Zeitverhalten

Für eine praktische Anwendung der Werkzeuge zum Finden von (Ursachen von) Fehlern in Arbeitsprozessen ist die dafür benötige Zeit ein entscheidendes Kriterium. Noch im Jahr 2009 galt eine Zeit von etwa 5 Sekunden als schnell für einen Prozess [91]. Für eine Anwendung direkt während der Konstruktion eines Prozesses ist eine Latenz von 5 Sekunden aber fast zu langsam, gerade wenn diese Zeit bei zunehmender Knoten- und Kantenzahl stark steigt.

Für die Untersuchung des Zeitverhaltens der verschiedenen Techniken mussten zunächst verlässliche Zeitreihen aufgenommen werden. Im Anhang C wird die Aufnahme dieser Messreihen sowie dabei auftretende Schwierigkeiten beschrieben.

Abbildung 9.8 zeigt jeweils für Mojo (oben) als auch für LoLA (unten) ein Diagramm über die Verteilung der Prozesse hinsichtlich der für sie benötigten Analysezeit. Wie gut in der oberen Grafik zu sehen ist, benötigt der Großteil der Prozesse (ca. 87%) unter einer und sogar 95% unter zwei Millisekunden zur Analyse in Mojo. Damit ist Mojo hinsichtlich der Analysezeit sehr schnell und kann offenbar ohne spürbare Latenz direkt während der Konstruktion eines Prozesses verwendet werden. Minimal benötigt Mojo für einen Prozess $0,03[ms]$, im Median $0,25[ms]$ und maximal $23,25[ms]$.

LoLA benötigt etwas mehr Zeit für die Analyse der Prozesse (unteres Diagramm). Etwa 67% der Prozesse benötigen im Schnitt 5 und ca. 99% der Prozesse brauchen zwischen 5 und 10 Millisekunden zur Analyse. Dies ist deutlich langsamer als mit Mojo aber immer noch eine sehr gute Zeit. Minimal brauchte LoLA $5,26[ms]$, im Median $5,78[ms]$ und maximal $17,97[ms]$ zur Analyse eines einzelnen Prozesses.

Die Analysezeiten für SESE sind weiter gestreut und deswegen für eine Grafik weniger gut geeignet. In ca. 94% der Fälle benötigt SESE unter $50[ms]$ für einen Prozess. Dies überrascht, da diese Technik in der Arbeit von Fahland et al. [18] das schnellste Werkzeug im Vergleich darstellte. Noch verwunderlicher ist dies, da der Algorithmus von Vanhatalo et al. [84] lediglich eine Abwandlung des Linearzeitalgorithmus von Tarjan zum Parsen von gerichteten Graphen und der Ableitung der darin enthaltenen Strukturen ist [74]. Dieser Algorithmus basiert wiederum auf einem anderen Algorithmus von Hopcraft und Tarjan, zum Unterteilen eines gerichten Graphen in 3-zusammenhängende Komponenten [30]. Da beide Algorithmen aufgrund ihres linearen Laufzeitverhaltens äußerst effizient arbeiten, muss zwangsläufig der Grund für die schlechte Laufzeit gegenüber LoLA und Mojo in der Implementierung dieser Algorithmen in *jBPT* liegen. Minimal beträgt die Analysezeit von SESE $0,87[ms]$, im Median $8,23[ms]$ und maximal $260,90[ms]$. Damit ist SESE sogar im Median langsamer als LoLA und findet dazu (im Gegensatz zu den beiden anderen Werkzeugen) nicht in jedem Fall heraus, ob ein Prozess korrekt ist.

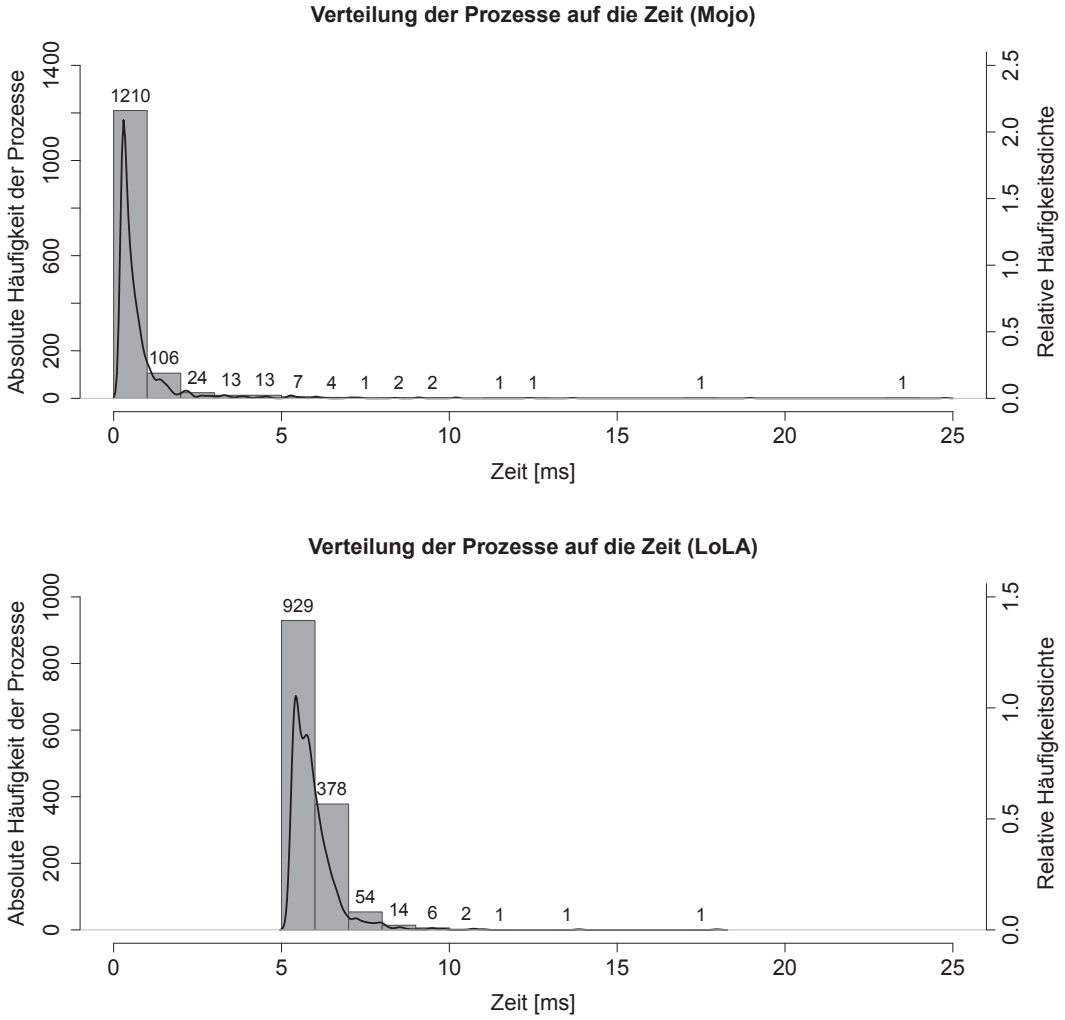


Abbildung 9.8: Verteilung der Analysezeiten für Mojo (oben) und für LoLA (unten)

Die unterschiedlichen Analysezeiten der untersuchten Werkzeuge spiegeln sich auch in der Gesamtlaufzeit zur Untersuchung der Prozessbibliothek wider. Die Gesamtlaufzeit der verschiedenen Analyseschritte von Mojo kann in Tabelle 9.5 nachvollzogen werden. In Mojo laufen die Verklemmungs- und Überflussanalyse parallel zueinander. Demzufolge ist ca. die Hälfte der Gesamtzeit pro Benchmark für andere Schritte, wie das Einlesen des Prozesses, das Transformieren in einen Workflowgraph, die Dominanz- und Postdominanzanalyse, etc. verantwortlich. Insgesamt braucht Mojo nur $817,68[ms]$ zur Analyse aller 1.386 Prozesse.

Die Tabellen 9.6 und 9.7 geben die Laufzeiten der einzelnen Analyseschritte der untersuchten Werkzeuge LoLA und SESE wieder. Wie bereits erwähnt, muss LoLA zwei separate Durchläufe für jeden Prozess berechnen, um in einem Durchlauf Verklemmungen und im nächsten Durchlauf Überflüsse zu finden. Die Findung von

	Verklemmung *[ms]*	Überfluss *[ms]*	Insgesamt *[ms]*
A	75,55	89,73	169,53
B1	70,62	92,92	165,92
B2	79,46	93,45	196,60
B3	129,10	157,13	258,04
C	8,53	12,28	27,60
Summe	363,27	445,50	817,68

Tabelle 9.5: Benötigte Zeit zur Analyse der Bibliothek (mit Mojo)

	Verklemmung *[ms]*	Überfluss *[ms]*	Insgesamt *[ms]*
A	838,84	816,41	1.655,25
B1	859,00	840,28	1.699,29
B2	1.108,54	1.092,29	2.200,83
B3	1.262,41	1.228,95	2.491,36
C	97,34	94,34	191,68
Summe	4.166,14	4.072,27	8.238,40

Tabelle 9.6: Benötigte Zeit zur Analyse der Bibliothek (mit LoLA)

Verklemmungen dauert in der Regel etwas länger als die Findung der Überflüsse. Insgesamt benötigt LoLA $8,24$ Sekunden zur Auswertung aller Prozesse der Bibliothek. Damit ist LoLA ca. 10 mal langsamer als Mojo.

Die Laufzeiten für die Erstellung des Prozessstrukturbaums und der anschließenden Fehlerfindung in SESE aus Tabelle 9.7 sind ebenso deutlich langsamer als die von Mojo. Insgesamt benötigt SESE ca. $22,59$ Sekunden zur Analyse aller Prozesse der Bibliothek. Damit ist SESE fast um das 27-fache langsamer als Mojo.

Neben der reinen Ausführungszeit ist ebenso die asymptotische Laufzeit beim Anstieg der Größe der Graphen wichtig. Diese wurde ausschließlich für unser Werkzeug Mojo untersucht.

Als wichtigster Kennwert zur Beschreibung der Größe eines Graphens erachten wir die Anzahl der Kanten, $|E|$. Abbildungen 9.9 und 9.10 untersuchen die Laufzeit in Abhängigkeit der Anzahl der Kanten. Dabei wird zum einen die tatsächliche Laufzeit in Millisekunden (Abb. 9.9) als auch das asymptotische Laufzeitverhalten mit Vergleich zur Anzahl untersuchter Kanten (Abb. 9.10) betrachtet. Die asymptotische Laufzeit ist durch den Exponenten e von $|E|^e$ gegeben. Um einen ruhigeren Verlauf der Kurven und somit einen besseren Überblick über das Verhalten zu bekommen, sind die abgebildeten Verläufe interpoliert.

Es ist deutlich zu sehen, dass die Laufzeit bei zunehmender Anzahl der Kanten ebenso zunimmt (Abb. 9.9). Hingegen nähert sich das asymptotische Laufzeitverhal-

	Prozessstruktur-baum *[ms]*	Fehlerfindung *[ms]*	Insgesamt *[ms]*
A	3.499,87	346,55	4.026,52
B1	4.446,43	455,19	5.114,04
B2	5.741,47	597,14	6.607,07
B3	5.268,10	515,95	6.054,44
C	679,21	77,33	785,56
Summe	19.635,08	1.992,15	22.587,63

Tabelle 9.7: Benötigte Zeit zur Analyse der Bibliothek (mit SESE)

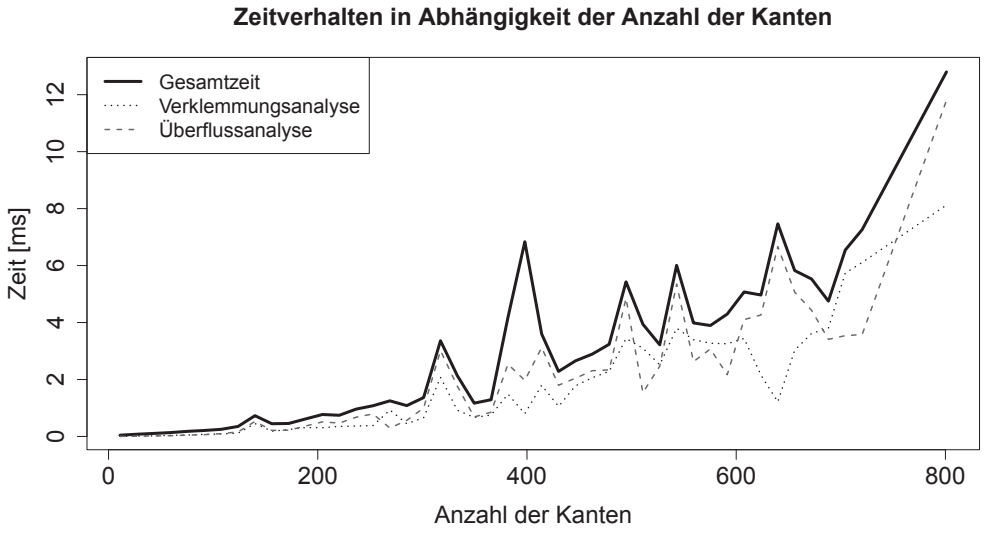


Abbildung 9.9: Zeitverlauf der verschiedenen Analysen (Mojo)

ten immer stabiler einem Wert von 2,0 an (Abb. 9.10). In der Praxis besitzen unsere Techniken demnach ein maximal quadratisches Laufzeitverhalten $O(E^2)$.

Weiterhin werden in beiden Abbildungen 9.9 und 9.10 die Laufzeitverhalten der wesentlichen, durchgeführten Analysen betrachtet: Die Überfluss- und die Verklemmungsanalyse. Im asymptotischen Laufzeitverhalten sind beide Analysen ähnlich.

Alles in allem kann aus den Daten gewonnen werden, dass Mojo im Vergleich zu LoLA und SESE in den meisten Fällen eine bessere Laufzeit besitzt. Das heißt, die detailliertere Findung von Ursachen von Fehlern wird nicht durch eine höhere Laufzeitkomplexität erkauft, denn diese ist praktisch quadratisch.

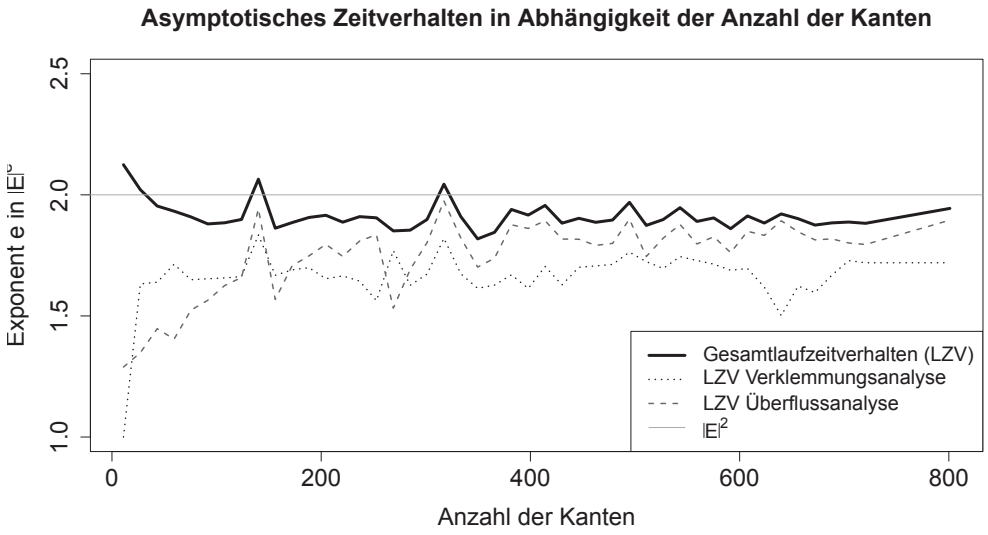


Abbildung 9.10: Asympt. Laufzeitverhalten der verschiedenen Analysen (Mojo)

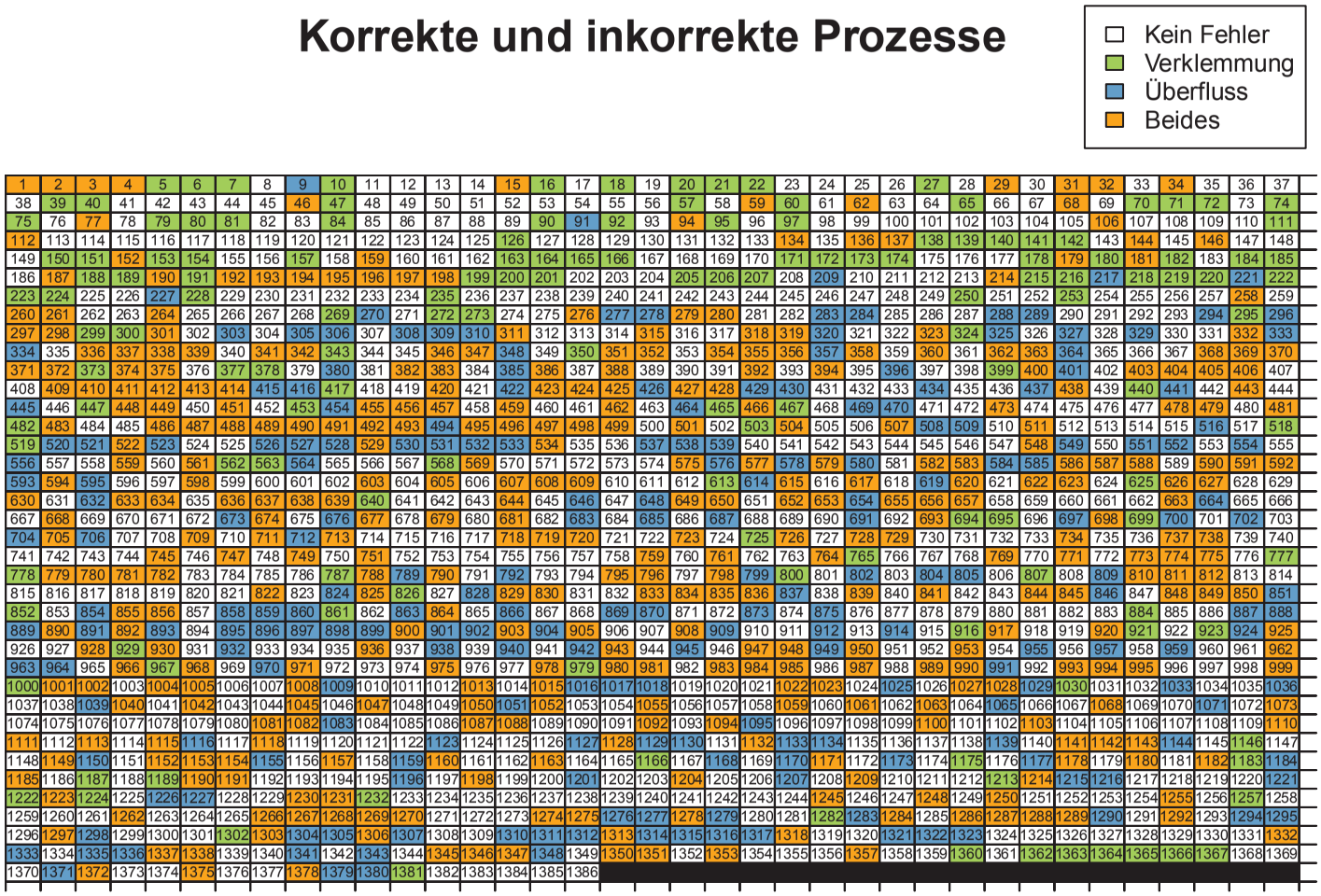


Abbildung 9.11: Korrekte und inkorrekte Prozesse der Prozessbibliothek

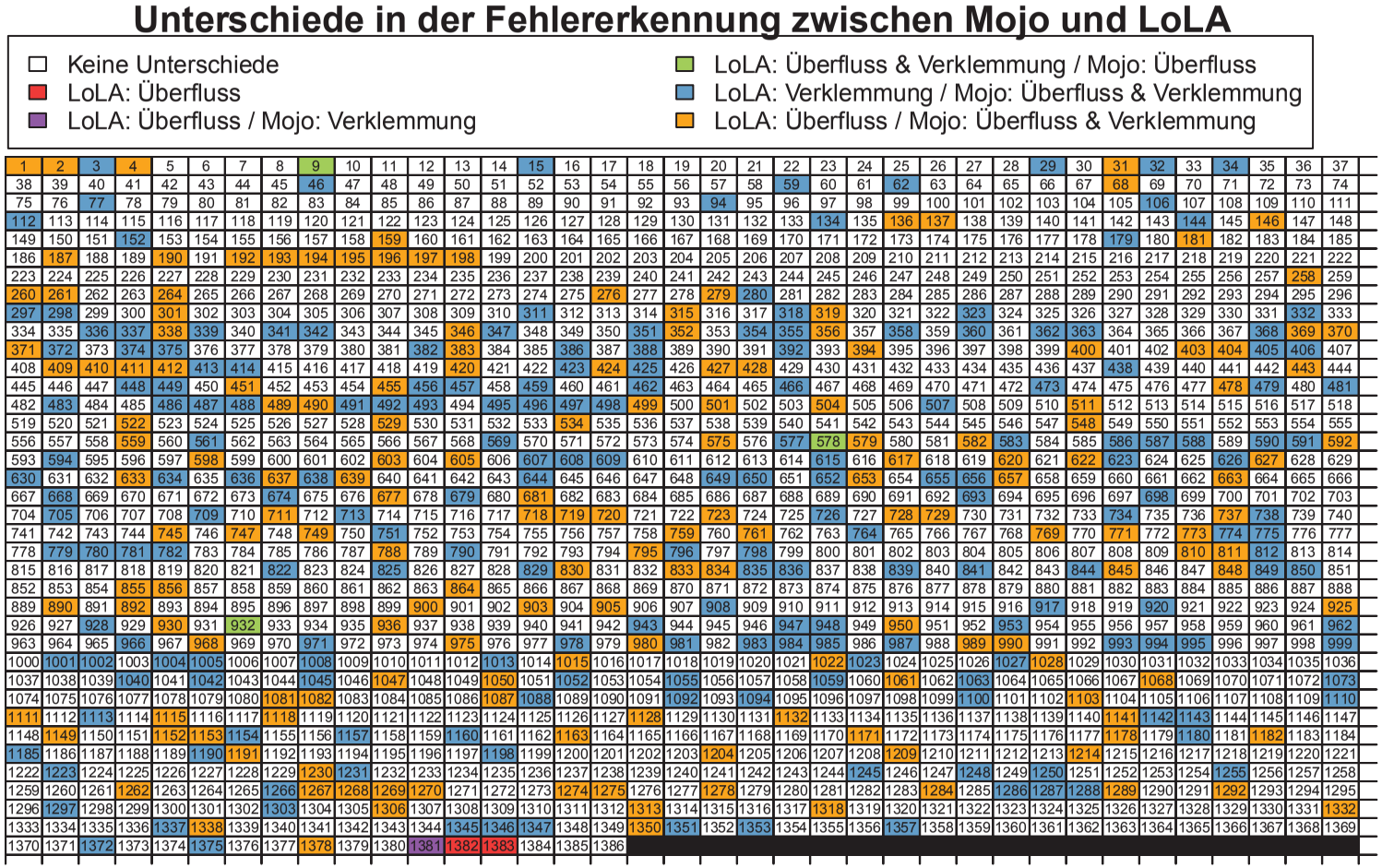


Abbildung 9.12: Differenzen zwischen gefundenen Fehlern in LoLA und Mojo

Kapitel 10

Diskussion und Zusammenfassung

Auf der Grundlage der Evaluationsergebnisse und der in dieser Arbeit aufgestellten Theorien diskutieren wir zum Abschluss unsere Ansätze hinsichtlich der in Kapitel 4 aufgestellten Thesen. Im Anschluss daran geben wir eine kurze Zusammenfassung der Schrift und einen Ausblick auf potentielle, zukünftige Arbeiten.

10.1 Diskussion

In diesem Abschnitt führen wir eine Diskussion der Techniken und Evaluationsergebnisse hinsichtlich der in Kapitel 4, Abschnitt 4.3, aufgestellten Thesen.

Die erste These dieser Arbeit behauptet, dass die Korrektheit über die Abwesenheit der Ursachen von Verklemmungen und Überflüssen charakterisiert werden kann. Eine solche Charakterisierung fand in dieser Arbeit in Form des Verklemmungssatzes 6.7 und der Bemerkung 7.8 über die Erkennung von Überflüssen statt. Auch wenn aus einer Ursache einer Verklemmung bzw. eines Überflusses nicht notwendigerweise eine Verklemmung bzw. ein Überfluss resultieren muss, so haben wir dennoch festgestellt, dass bei Anwesenheit einer Fehlerursache der Workflowgraph inkorrekt sein muss. Zusammengefasst können wir die anschließende Folgerung aufstellen, die damit auch die erste These der Arbeit als gültig erklärt:

Folgerung 10.1 (Korrektheit über Fehlerursachen)**.**
Ein Workflowgraph heißt genau dann korrekt, wenn er weder Ursachen unmittelbarer Verklemmungen noch von Überflüssen besitzt.

Aufgrund der letzten Folgerung eignen sich unsere Techniken zur Korrektheitsverifikation von Workflowgraphen. Da unsere Techniken ihren Ursprung im Übersetzerbau besitzen (Dominator- und Postdominatorrelation und SSA-Form), haben wir in dieser Arbeit bereits gezeigt, dass ein Korrektheitsnachweis von Workflowgraphen auch mit Techniken des Übersetzerbaus anstelle petrinetzbasierter Ansätze möglich

ist. Die vorangegangene Evaluation hat außerdem gezeigt, dass diese übersetzerbasierten Ansätze qualitativ und quantitativ besser sind als die bisherigen Petrinetztechniken: Unsere Techniken finden mehr und detailliertere Fehler; finden Fehlerursachen ungeachtet anderer Fehler; und sind deutlich schneller (besitzen eine theoretisch biquadratische Worst-Case Laufzeitkomplexität, die jedoch akkurater mit einer theoretisch kubischen Laufzeit abgeschätzt werden sollte und in der Praxis sogar quadratisch ist). Damit ist die zweite These dieser Arbeit aus Abschnitt 4.3 ebenso gültig. Ihr Wortlaut war: „Typische Begriffe und Algorithmen aus dem Übersetzerbau eignen sich qualitativ und quantitativ mindestens genauso gut zur Kontrollflussanalyse von Workflowgraphen, wie typischerweise eingesetzte petrinetzbasierte Ansätze."

Petrinetzbasierte Ansätze nutzen meist eine Zustandsraumerkundung zur Analyse von Workflownetzen/-graphen. Wir haben durch die 3. These behauptet, dass neben der klassischen Zustandsraumanalyse ab dem Startzustand auch eine partielle Zustandsraumanalyse ab einer beliebigen Kante möglich ist und zum Korrektheitsnachweis eingesetzt werden kann. In Kapitel 5 führten wir diese partielle Analyse ein, die nachfolgend ab dem genannten Kapitel in die Techniken zur Findung der Ursachen von Verklemmungen und Überflüssen eingeflossen ist. Aus diesem Grund ist diese These ebenso gültig.

Die 4. These befasst sich mit der Existenz zweier für die Ursachenanalyse geeigneter Algorithmen, mit den qualitativen Eigenschaften, dass sie (1) eine polynomielle, maximal biquadratische Laufzeit besitzen, (2) exakte diagnostische Informationen liefern, und (3) die Ursachen von Verklemmungen und Überflüssen hinter anderen Fehlern entdecken. Diese These geht mit der Diskussion der These 2 einher und ist somit gültig. Das Laufzeitverhalten der Algorithmen ist im Normalfall und somit für praktisch relevante Prozesse sogar kubisch, wie bereits in Kapitel 7 diskutiert. Die praktische Evaluation hat vielmehr noch ein quadratisches Laufzeitverhalten ergeben. Aus praktischer Sicht wurden somit zwei Algorithmen mit maximal kubischem asymptotischen Laufzeitverhalten gefunden.

10.2 Zusammenfassung

Der Korrektheitsbegriff wurde bisher über Fehlerwirkungen, wie der Verklemmung und den Überfluss, definiert. Dies spiegelt sich auch in den gängigen Techniken zum Nachweis der Korrektheitseigenschaft wider, in denen ein Prozess nach den Wirkungen anstelle der Ursachen von Fehlern untersucht wird. Dadurch entsteht jedoch eine Kluft in Form einer hohen Entfernung zwischen der eigentlichen Ursache und der Wirkung des Fehlers. Diese Kluft wird schier unüberwindbar für einen Prozessmodellierer, wenn Fehlerursachen blockiert, maskiert oder vorgetäuscht werden [57]. Unsere Motivation für diese Arbeit war es demnach, neue Techniken zu entwickeln,

welche direkt die Ursachen von Verklemmungen und Überflüssen in Prozessen bestimmen anstelle nur deren Auswirkungen.

Dazu führten wir durch grundlegende Definitionen in das Thema der Prozessverifikation ein. Die Prozessverifikation in Bezug auf den Korrektheitsbegriff wurde anschließend im Stand der Forschung noch einmal genauer hinsichtlich der bereits existierenden Techniken untersucht. Im Stand der Forschung konnten wir lediglich zwei Arbeiten identifizieren, die am ehesten nach den Ursachen von Verklemmungen und Überflüssen suchen: Die Dekomposition des Prozesses in sogenannte SESE-Fragmente [85] und die Nutzung von Antipatterns [19]. Beide Techniken haben jedoch ihre Schwächen. So ist die Dekomposition in SESE-Fragmente nachweislich unvollständig. Die derzeitige Nutzung von Antipatterns wiederum ist zum einen langsam (quintisches Laufzeitverhalten) und zum anderen schwer auf neue Kontrollstrukturen, wie Or-Joinknoten, übertragbar. Außerdem können beide Techniken nicht ausreichend die *Ursachen* innerhalb des Prozesses finden und detailliert beschreiben.

Aus diesem Grund haben wir offene Probleme aus dem Stand der Forschung extrahiert und in unseren wissenschaftlichen Beitrag und die daraus gewonnenen Thesen einfließen lassen. In diesen Thesen wurde festgehalten, dass der Korrektheitsbegriff auch über *Ursachen* von Verklemmungen und Überflüssen definiert werden kann. Diese Definition kann dazu genutzt werden, dass mit übersetzerbasierten Ansätzen qualitativ und quantitativ genauso gute Ergebnisse bei der Findung von Kontrollflussfehlern erzielt werden können wie durch petrinetzbasierte Techniken. Weiter noch führen diese Ansätze sogar zu Algorithmen mit kubischen Laufzeitverhalten für praktisch relevante Prozesse (biquadratisch im schlechtesten Fall), die exakte diagnostische Informationen für alle gefundenen Ursachen von Verklemmungen und Überflüssen liefern. Sie sind tatsächlich auch dazu geeignet, Just-in-Time (JIT) während der Konstruktion eines Prozesses Anwendung zu finden. Außerdem sind sie auf andere Kontrollstrukturen erweiterbar.

Die Erfüllung dieser Thesen basiert auf der Einführung einer partiellen Analyse von Prozessen (bzw. Workflowgraphen). Diese partielle Analyse untersucht den Zustandsraum eines Workflowgraphen nicht mehr nur vom Startzustand sondern über Kontrollflüsse von jeder beliebigen Kante. Speziell ausgewählte Kanten, die Eintrittspunkte, sind dazu geeignet, Aussagen über das Verhalten eines Teils des Prozesses (eben partiell) zu treffen.

Aufbauend auf der partiellen Analyse konnten die Ursachen von Verklemmungen gefunden werden. Verklemmungen kommen offenbar immer dann zustande, wenn mindestens eine aber nicht alle eingehenden Kanten eines Joinknoten Marken erreichen. Damit nun garantiert alle eingehenden Kanten eines Joinknotens Marken erreichen, muss auf jedem Weg zu diesen Kanten eine Ausführungskante liegen. Diese Ausführungskante garantiert die Ausführung des Joinknotens. Existiert auf einem

Weg zum Joinknoten *keine* Ausführungskante, so kann er zwar durch eine Marke erreicht werden, es ist aber nicht garantiert, dass er auch ausgeführt wird – der Prozess kann verklemmen. Für die Suche nach diesen Ursachen von Verklemmungen haben wir einen effizienten Algorithmus mit kubischer Laufzeitkomplexität hinsichtlich der Anzahl der Kanten entworfen.

Einen von der Literatur als kubisch eingestuften Algorithmus konnten wir auch für die Bestimmung der Ursachen von Überflüssen finden (im Ausnahmefall ist die Laufzeit biquadratisch). Die Ursache eines Überflusses ist das Fehlen einer Synchronisierung zweier Kontrollflüsse. Die Synchronisierung sollte immer dann vorgenommen werden, wenn zwei parallele Kontrollflüsse erstmals aufeinander treffen können. Diese Stellen bezeichneten wir als Treffpunkte. Damit die Ursachen von Überflüssen bestimmt werden können, muss überprüft werden, ob ein Treffpunkt eines Forkknotens (unter Ausschluss von Verklemmungen) von zwei Kontrollflüssen gleichzeitig erreicht werden kann. Dies kann genau dann geschehen, wenn es mindestens zwei disjunkte Wege vom Forkknoten zum Treffpunkt gibt, die unabhängig von zwei Kontrollflüssen abgelaufen werden können. Die Unabhängigkeit ist gewährleistet, wenn auf mindestens zwei solcher Wege kein vom Treffpunkt abhängiger Joinknoten liegt. Der resultierende Algorithmus zur Bestimmung der Ursachen von Überflüssen basiert auf der SSA-Form, der Bestimmung von abhängigen Kanten und der Berechnung des maximalen Flusses.

Da die im Worst-Case biquadratische Laufzeitkomplexität der Algorithmen zur Findung der Ursachen von Verklemmungen als auch von Überflüssen theoretischer Natur ist, führten wir im Zuge dieser Arbeit eine Evaluierung der Algorithmen hinsichtlich des Zeitverhaltens und der Anzahl und Qualität gefundener Fehlerursachen durch. Dafür stellten wir zunächst unsere Implementierung, das Werkzeug Mojo, als auch das angestrebte Gesamtsystem vor. Die anschließende Evaluation auf Basis dieses Werkzeugs hat ergeben, dass unsere Technologien deutlich mehr Fehler(ursachen) ermitteln als die Zustandsraumerkundung und die Dekomposition. Diese gefundenen Fehler sind qualitativ außerdem besser für die Visualisierung und Beschreibung für einen Prozessmodellierer geeignet als die, der anderen Werkzeuge. Zudem ergab sich, dass Mojo in der Praxis ein maximal quadratisches Laufzeitverhalten hinsichtlich der untersuchten Kanten aufweist. Insgesamt stellen unsere Techniken zur Findung der Ursachen von Verklemmungen und Überflüssen derzeit die besten dar – sowohl hinsichtlich der Fehlerdiagnostik als auch des Laufzeitverhaltens.

Ausblick auf zukünftige Arbeiten Zum Abschluss dieser Arbeit werden Möglichkeiten zur Weiterführung der in dieser Arbeit behandelten Themen vorgestellt.

Als ein sehr wichtiges Thema erachten wir die Anwendung und ggf. Erweiterung unserer Techniken auf Workflowgraphen mit Or-Joinknoten auf Basis der von uns in [58] eingeführten Semantik. Auf Grundlage dieser Semantik sollte der Nachweis mit

denen in dieser Arbeit eingeführten Techniken auch auf zyklische Workflowgraphen inklusive Or-Joinknoten angewandt werden können.

Ein weiteres Thema für weitergehende Arbeiten ist das in Kapitel 8 eingeführte Gesamtsystem zur Entwicklung, Speicherung und Ausführung von Arbeitsprozessen [60]. Bisher akzeptiert Mojo lediglich die Struktur eines Prozesses – Prozessdaten werden außer Acht gelassen. Ein Ziel einer zukünftigen Arbeit sollte es sein, ein sauberes Zwischencodeformat für Prozesse zu entwickeln, auf dem sowohl Kontrollfluss- als auch Datenflussanalysen durchgeführt werden können. Eine Möglichkeit eines solchen Zwischencodeformats könnten erweiterte Workflowgraphen sein [3]. Erweiterte Workflowgraphen sind Workflowgraphen, die zusätzlich zu den Kontrollflussstrukturen auch Datenflussstrukturen in der Concurrent Static Single Assignment Form (CSSA-Form) [43] aufweisen. In der CSSA-Form gibt es neben den ϕ-Funktionen der SSA-Form zusätzliche π-Funktionen. π-Funktionen werden immer genau dann benötigt, wenn auf ein und dieselbe Variable konkurrierend zugegriffen werden kann. Eine Weiterentwicklung der erweiterten Workflowgraphen sind die Ausklappgraphen (Foldoutgraph), die wir in [59] eingeführt haben und die auf den Konzepten von SafeTSA [4] basieren. In Foldoutgraphen werden strukturierte Elemente des erweiterten Workflowgraphen hierarchisch mit Hilfe des Prozessstrukturbaums [84] angeordnet. Dadurch sind wesentlich effizientere Analysen als auch eine komprimierte Abspeicherung möglich [4].

Einhergehend mit einer solchen Entwicklung eines Zwischenformats basierend auf der CSSA-Form müssen Techniken entwickelt werden, um eine minimale CSSA-Form für unstrukturierte Prozesse zu erzeugen. Die in Lee et al. [43] gezeigten Algorithmen funktionieren nur, wenn die Ausführungsreihenfolge der Knoten im Graphen effizient bestimmt werden kann. Um diese Ausführungsreihenfolge auch in unstrukturierten Prozessen zu finden, könnte auf der Überflusserkennung aufgebaut werden.

Neben der Entwicklung eines solchen Zwischenformats für Prozesse müssten unsere Techniken auch erweitert werden, um nun auch die in dem Prozess enthaltenen Daten in den Korrektheitsnachweis oder in einem zusätzlichen Analyseschritt mit einzubeziehen. So ist es denkbar, im Anschluss der von Heinze et al. beschriebenen Techniken zur Umstrukturierung und Entfaltung eines Prozesses [26][27][28][29] unsere Techniken anstelle der vorgeschlagenen petrinetzbasierten Ansätze einzusetzen.

Kehren wir wieder zum Gesamtsystem zur Entwicklung, Speicherung und Ausführung von Arbeitsprozessen zurück, so bergen die Ideen dieser Arbeit auch großes Potential während der Ausführung von Prozessen. Die Ausführung dieser Arbeitsprozesse erfolgt in unserem Vorschlag in einer virtuellen Maschine, die den Prozess einliest, verifiziert und ausführt [56]. Die Verifikation kann beispielsweise die Korrektheitseigenschaft des Prozesses erneut mit Hilfe unserer Techniken überprüfen. Diese Überprüfung könnte auch beschleunigt werden, wenn unsere Analysen ihre ge-

fundenen Informationen an den Prozess annotieren würden. Die annotierten Daten müssten dann während der Verifikation nur auf Stimmigkeit überprüft werden.

Diese Annotationen können aber auch später während der Ausführung des Prozesses Anwendung finden [60]. So ist denkbar, die Korrektur eines Prozesses durch eine Simulation vorzunehmen. Dabei wird der Prozess ausgeführt und aufgrund der von unseren Algorithmen produzierten Informationen vorherbestimmt, ob der Prozess gerade in eine Verklemmung oder in einen Überfluss geraten kann und auch unter welchen Bedingungen dies möglich ist. Daraufhin wird der Modellierer unterrichtet. Dieser kann entscheiden, ob diese Bedingungen eintreten können oder ob sie ausgeschlossen sind. Somit ist der Modellierer selber in der Lage einzuschätzen, ob eine gefundene Fehlerursache für den Prozess relevant ist oder nicht. Dies wäre somit auch eine Alternative zur Reduzierung der Überabschätzung unserer Techniken aufgrund der missachteten Dateninformationen.

Insgesamt votieren wir für die Umsetzung eines solchen leichtgewichtigen Gesamtsystems zur Entwicklung und Ausführung von Arbeitsprozessen, dass vor allen Dingen auf bewährten Übersetzerbautechniken aufbaut und effiziente JIT-Analysen zulässt. In der Literatur wurde unser Systemansatz beispielsweise schon diskutiert [45][71][72]. Solch ein System ließe sich auch einfacher für die normale Anwendungsprogrammierung als auch für Lehrzwecke einsetzen. Puzzlebasierte und graphische Programmiersprachen, wie *Scratch*[1] oder der *MIT App Inventor*[2] zeigen dies.

Zu guter Letzt ist die Integration von *Mojo* in das Werkzeug *BPMNspector*[3] [24] geplant. *BPMNspector* ist ein statisches Analysewerkzeug für die Überprüfung der BPMN-Compliance.

[1] `https://scratch.mit.edu/` (Mai 2017)
[2] `http://appinventor.mit.edu/explore/` (Mai 2017)
[3] `http://bpmnspector.org/` (Mai 2017)

Anhang A

Parameter des Werkzeugs *Mojo*

-p / -path Legt den Pfad fest, in dem Mojo *rekursiv* nach Dateien mit den Endungen *.pnml* und *.bpmn.xml* suchen soll. Ohne diese Angabe sucht Mojo im aktuellen Verzeichnis nach Prozessen.

-e / -export Legt den Pfad fest, in dem Mojo entstehende Dateien abspeichern soll. Dieser ist standardmäßig auf den Pfad festgelegt, in dem Mojo Prozesse sucht.

-v / -verbose Liefert Debuginformationen.

-t / -times Legt die Anzahl fest, wie oft Mojo Analysepläne auf dem selben Prozess ausführt. Dies eignet sich insbesondere für valide Zeitmessungen, wie später im Anhang C gezeigt.

-d / -dot Erzeugt für jeden gefundenen Prozess eine Datei mit der Beschreibung des Workflowgraphen in der DOT-Sprache[1]. Ein Graph in DOT-Sprache kann mit dem Werkzeug *Graphviz*[2] visualisiert werden.

-no Unterdrückt textuelle Fehlerausgaben. Eignet sich für Performancemessungen.

-h / -help Gibt eine kurze Hilfe aus.

-f / -file Ermöglicht die Angabe eines einzelnen Prozesses, dessen Dateiname im aktuell eingestellten Suchpfad von Mojo liegen muss.

-ap / -analysisPlan Legt den zu verwendeten Analyseplan von Mojo fest. Für diese Arbeit sind die Analysepläne 0 und 2 wichtig. Der Analyseplan 0 führt die in dieser Arbeit beschriebenen Analysen und der Analyseplan 2 eine SESE-Dekomposition durch.

-c / -csv Gibt Analyseinformationen in einer CSV-Datei aus.

Alle anderen Parameter, die nicht zu dieser Liste gehören, werden jeweils als Dateiname eines Prozesses verstanden.

[1] `http://www.graphviz.org/content/dot-language` (Mai 2017)
[2] `http://www.graphviz.org/` (Mai 2017)

Anhang B

Aufbau der Prozessbibliothek

Die Bibliothek mit 1.368 Arbeitsprozessen des *IBM WebSphere Business Modeler*s bildet einen schönen Querschnitt über Arbeitsprozesse verschiedener Größe, Struktur und Semantik. Tabelle B.1 fasst die Beschaffenheit der Prozesse zusammen. So bestehen die Prozesse im Median aus 67 Knoten mit 81 Kanten; d.h., 50% der Prozesse haben weniger als 67 Knoten, 50% haben mehr. Der kleinste Prozess besitzt gerade 10 und der größte Prozess sogar 549 Knoten bzw. 11 und 639 Kanten.

Den größten Anteil an Knoten bilden die Taskknoten: Im Median sind in einem beliebigen Prozess 49% davon enthalten. Nach den Taskknoten sind Fork- neben den Joinknoten am häufigsten in Prozessen zu finden. Das bedeutet, dass Parallelitäten in der realen Anwendung eine höhere Bedeutung als Entscheidungen (Split- und Mergeknoten) besitzen.

Neben dieser einfachen Anzahl an verschiedenen Knotentypen ist auch die strukturelle Zusammensetzung der Prozesse interessant und von Bedeutung. Unter Strukturen verstehen wir in diesem Zusammenhang spezielle Teilgraphen eines Prozesses. Mit den *Prozessstrukturbäumen* von Vanhatalo et al. [84] konnten für Prozesse eindeutige Strukturen gefunden werden. Diese Strukturen entstehen, wenn der dem Arbeitsprozess zugrundeliegende Workflowgraph in verschiedene Teilgraphen zerlegt wird. Im Grunde werden größtmögliche Teilgraphen im Workflowgraphen gesucht,

	Task	Fork	Join	Split	Merge	Knoten	Kanten
Minimum	8	0	0	0	0	10	11
Median	49	8	7	1	1	67	81
Mittelwert	62	12	11	2	1	88	108
Maximum	500	80	79	24	20	549	639

Tabelle B.1: Anzahl Knoten verschiedener Arten in den Prozessen der Testbibliothek

	Triviale	Verbände	Polygone	Rigids
A	30.650	3.695	11.355	217
B1	28.550	2.987	9.382	167
B2	37.608	3.862	12.907	209
B3	48.299	5.853	16.195	245
C	4.567	116	600	22
Summe	149.674	16.513	50.439	860

Tabelle B.2: Vorhandene Strukturen in der Testbibliothek

die durch genau eine Kante ersetzt werden können.[1] Aus dieser schrittweisen Zerlegung entstehen genau vier verschiedene Arten von Strukturen: die *Trivialen*, die *Verbände*, die *Polygone* und die *Rigids*.

Eine triviale Struktur besteht aus genau einer Kante und bildet damit die kleinste Struktur, die in jeder anderen Struktur enthalten ist; zum Beispiel in Verbänden. Verbände bestehen im Gegensatz zu Trivialen aus genau zwei Knoten und mindestens zwei Kanten zwischen ihnen. Sie repräsentieren typische Strukturen mit einem öffnenden und einem schließenden Knoten, wie einem Split- und einem Mergeknoten. Sequenzen von Knoten mit Kanten heißen Polygone. Alle anderen Verbindungen von Knoten und Kanten stellen Rigids dar. Sie sind unstrukturierte Teilgraphen [84].

Wie aus dem Übersetzerbau bekannt, existieren meist performantere Algorithmen für strukturierte Graphen. Beispielsweise wurde in den Arbeiten von Vanhatalo et al. [84][85] gezeigt, dass für strukturierte Prozesse eine vollständige und schnelle Korrektheitsanalyse stattfinden kann. Für unstrukturierte Prozesse können diese Techniken leider nicht angewandt werden.

Ein Prozess der verwendeten Prozessbibliothek besitzt maximal 3 Rigids, also 3 unstrukturierte Bereiche. Besitzt er eine solche Struktur, so umfasst sie minimal 12, im Median 86 und im Maximum 634 Kanten. Somit stellen Rigids meist große Strukturen innerhalb von Prozessen dar. Tabelle B.2 fasst die Gesamtanzahl der enthaltenen Strukturen in den einzelnen Benchmarks der Prozessbibliothek zusammen.

[1] Eine Beschreibung dieser Zerlegungsschritte kann in Vanhatalo et al. [84] nachgelesen werden.

Anhang C

Aufnahme der Messreihen

Wichtig für eine qualitativ hochwertige Evaluation ist die Aufnahme verlässlicher und valider Zeitmessungen. In diesem Anhang wird gezeigt, wie diese für die Evaluation gewonnen worden.

Für unser Werkzeug Mojo übernimmt Mojo selbst die Zeitmessung, in dem es für jede getätigte Analyse Performanzmessungen durchführt. Dabei wird es für Zeitmessungen und einen einfachen Korrektheitsnachweis am besten mit dem folgenden Kommando ausgeführt:

```
java -classpath core-all-<CORE_VERSION>.jar;
                mojo.reader.pnml-all-<READER_VERSION>.jar
                de.jena.uni.mojo.Mojo
                -csv -p "<PATH>" -no -ap 0 -times <TIMES>
```

Die Parameter haben die folgenden Bedeutungen: `-csv` veranlasst Mojo alle aufgenommenen Daten als Comma-Separated Values (CSV)-Datei (Komma-separierte Werte) zu speichern. Durch `-p` kann der Pfad `<PATH>` (am besten in Anführungszeichen) angegeben werden, in dem die PNML- oder BPMN-Dateien liegen. Mit `-no` wird jede unnötige Ausgabe auf die Konsole unterdrückt. Der Befehl `-ap` legt den Analyseplan fest. Hier verwenden wir den Analyseplan mit der Nummer `0`, der Kontrollflussanalysen durchführt. Mittels des Kommandos `-times` führt Mojo den Analyseplan für alle gefundenen Prozesse `<TIMES>`-mal aus.

Die Sinnhaftigkeit des letzten Parameters wird offensichtlich, wenn wir das Zeitverhalten des Korrektheitsnachweis der Prozesse in den verschiedenen Wiederholungen miteinander vergleichen. Wir haben uns für 3 Durchläufe entschieden und die verschiedenen Durchläufe im Diagramm der Abbildung C.1 notiert. In diesem Diagramm bildet die x-Achse die Nummer des Prozesses in Analysereihenfolge und die y-Achse die benötigte Zeit in Millisekunden ab. Für eine bessere Übersichtlichkeit haben wir in diesem Diagramm auf eine interpolierte Darstellung des Zeitverhaltens zurückgegriffen, um die Unterschiede deutlicher sichtbar zu machen.

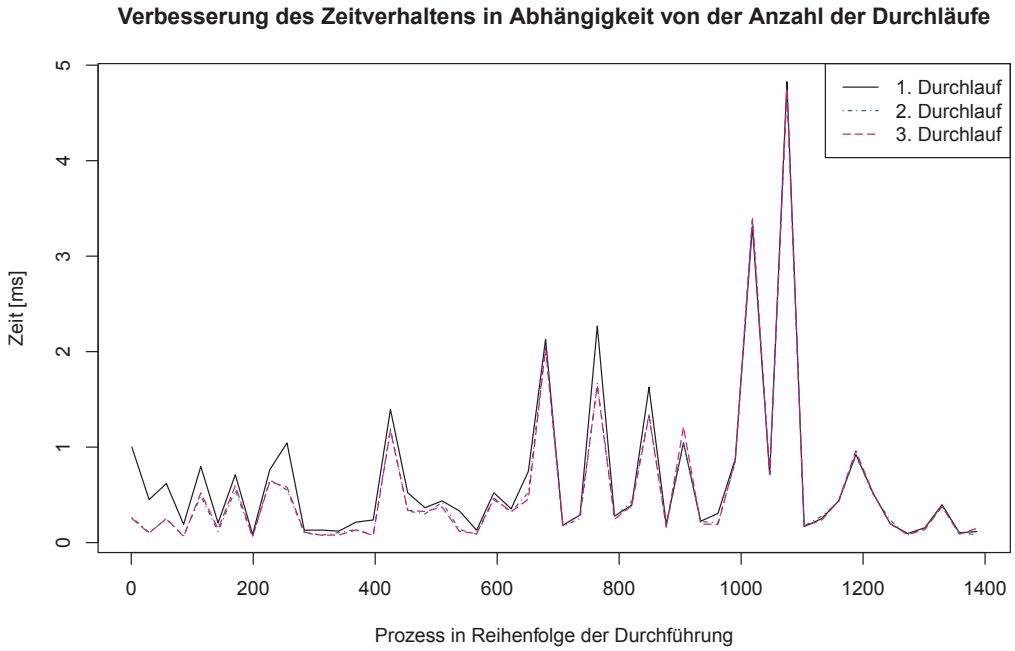


Abbildung C.1: Approx. Zeitverhalten nach drei Durchläufen der Prozessbibliothek

Wie in diesem Diagramm gut zu sehen, gibt es zwischen dem 1. und 2. sowie dem 1. und 3. Durchlauf zu Beginn deutliche Abweichungen im Zeitverhalten bis zum ca. 1.000. analysierten Prozess. Danach verhalten sich die Verläufe annähernd identisch. Die Ursache dieses Verhaltens liegt in der Natur der Java Virtuellen Maschine (JVM) [47]. Die JVM besitzt als Hauptmerkmal den bekannten HotSpot™-Compiler [41]. Dieser erkennt automatisch Methoden, die häufig ausgeführt werden oder lange Zeit benötigen. Auf diesen führt der HotSpot einige (mitunter auch *spekulative* [1]) Optimierungen aus. In dieser Phase der Optimierung geschehen im Grunde zwei Dinge im Sinne des Zeitverhaltens: Zunächst bricht die Laufzeit kurz ein, da der Übersetzer einige Ressourcen benötigt, die nun dem ausgeführten Programm nicht mehr zur Verfügung stehen. Danach jedoch wird das Programm sprunghaft schneller und das Zeitverhalten verbessert sich drastisch.

Für Performanzmessungen ist solch ein Verhalten natürlich unglücklich. Leider kann bei der JVM dieses Verhalten nicht unterdrückt werden. Aus diesem Grund haben wir entschieden den dritten der Durchläufe zu wählen, da dort die meisten wichtigen und zeitintensiven Methoden bereits optimiert wurden und es dadurch valide Zeitmessungen geben sollte. Dies zeigt auch Abbildung C.1, da sich der 2. und 3. Durchlauf nicht mehr unterscheiden lassen.

Durch das Computersystem selbst können natürlich dennoch sogenannte „Ausreißer“ in den Zeitmessungen auftreten. Um diese so gut wie möglich zu reduzieren,

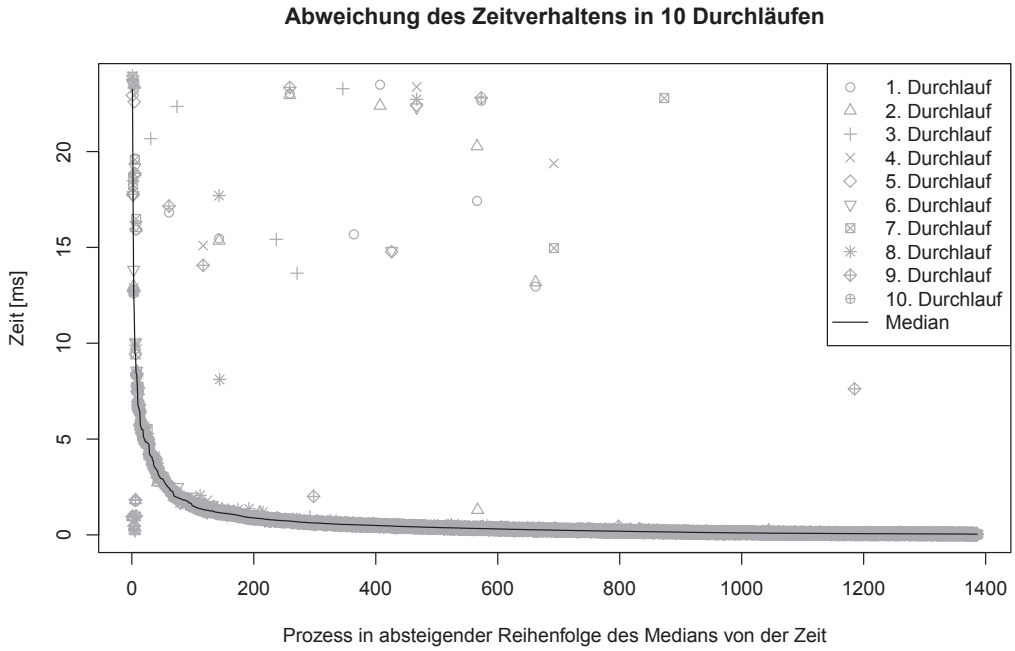


Abbildung C.2: Der Median als Zeitmessreihe

haben wir Mojo 10 mal hintereinander mit 3 Durchläufen ausgeführt. Die Messergebnisse der 10 Durchläufe sind in Abbildung C.2 durch verschiedene Symbole je Durchlauf illustriert. Auf der x-Achse werden die Prozesse aufgelistet, diesmal jedoch in absteigender Reihenfolge bezüglich des Medians über die Zeit. Das heißt, an Position 1 steht der Prozess mit der größten Ausführungsdauer (im Median) und an der Position 1.368 der Prozess mit der geringsten Ausführungsdauer (im Median).

Zwischen den Zeitverläufen sind wenig Differenzen zu erkennen. Durch eine schwarze Linie haben wir den Median zwischen den Durchläufen eingezeichnet. Wie sehr schön in der Grafik zu erkennen ist, eignet sich der Median optimal als valide Zeitmessreihe. Aus diesem Grund verwenden wir für Mojo nachfolgend für alle Zeitmessungen den Median aus 10 Durchläufen aus der 3. Wiederholung der Analysen pro Prozess.

Für das Werkzeug jBPT, im Folgenden SESE im Sinne der Zerlegung in Strukturen mit einer eingehenden und einer ausgehenden Kante genannt, sind wir analog zu Mojo vorgegangen, da die Algorithmen auch in unserem Werkzeug Mojo implementiert wurden. Mit Angabe des Analyseplans 2 vollzieht Mojo die SESE-Analyse.

Analog sind wir bei dem Werkzeug LoLA vorgegangen. Dafür wurden alle Prozesse der Bibliothek durch ein Shell-Skript analysiert. Da LoLA als Maschinenprogramm vorliegt, sind Differenzen in der Anzahl der Wiederholungen wie bei Java ausgeschlossen. Leider bietet LoLA selbst jedoch keine Möglichkeit zur Messung der Zeiten. Aus

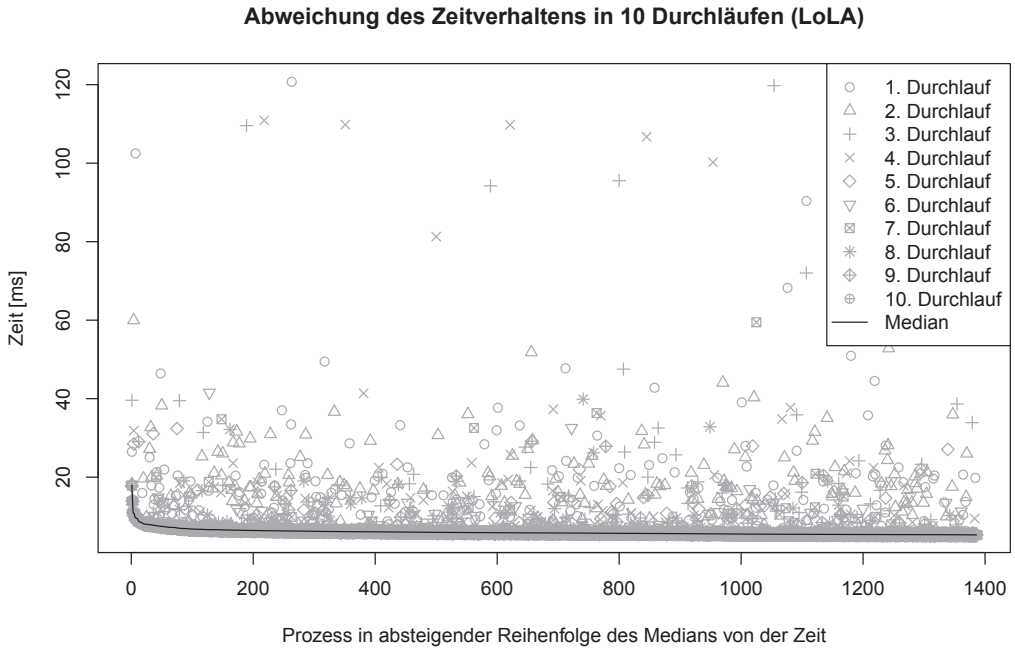


Abbildung C.3: Der Median als Zeitmessreihe für LoLA

diesem Grund haben wir die Ausführungsdauer des Programms mit Systembefehlen gemessen. Da es dennoch zu Ausreißern während der Ausführung durch das System selbst kommen kann, haben wir auch für LoLA 10 Durchläufe untersucht (Abbildung C.3). Wie in der Abbildung gut zu sehen, ist die Varianz zwischen den Durchläufen größer als bei Mojo. Dies zeigt sich durch höhere Zeitunterschiede zwischen den einzelnen Durchläufen. Aber auch hier ist zu sehen, dass der Median sehr gut als verlässliche und valide Zeitreihe geeignet ist.

Anhang D

Anzahl der Treffpunkte in praktischen Prozessen

In Kapitel 7, Abschnitt 7.3.2 (S. 74), haben wir über die Anzahl entstehender ϕ-Funktionen bei der Transformation eines Programms in die SSA-Form und damit über die Gesamtanzahl an Treffpunkten in einem Workflowgraphen diskutiert. Um diese Diskussion mit empirischen Daten zu untermauern, haben wir für die während der Evaluation (vgl. Kapitel 9) verwendeten Prozessbibliothek praxisnaher Prozesse die Anzahl entstehender Treffpunkte untersucht. Dabei wurde während der Analyse der Ursachen von Überflüssen für jeden Prozess die Summe aller für diesen Prozess ermittelten Treffpunkte bestimmt. Abbildung D.1 zeigt die gemessenen Summen dieser Treffpunkte in Abhängigkeit der Kantenanzahl. Diese Summen wurden anschließend durch die Anzahl der in den jeweiligen Prozessen befindlichen Kanten geteilt, um das Verhältnis zwischen der Anzahl an Treffpunkten und der Anzahl an Kanten zu erhalten. Diese Untersuchung ergab, dass im Median 57% Treffpunkte in einem Prozess enthalten sind. Der Mittelwert liegt bei ca. 60%. Minimal enthält ein Prozess keinen Treffpunkt und maximal 176%.

Da in der erwähnten Diskussion für einen linearen Zusammenhang zwischen der Anzahl an Treffpunkten aller Forkknoten und der Anzahl an Kanten im Worst-Case argumentiert wird, haben wir noch den Exponent x der Gleichung

$$\sum_{fork \in N_{Fork}} 人(fork) = |E|^x$$

für jeden Prozess bestimmt. Diese Untersuchung ergab einen minimalen Exponenten x von 0.24; im Median liegt er bei 0.89 und im schlechtesten Fall bei 1.08.

Zusammengefasst bestätigen die Untersuchungen einen linearen Zusammenhang zwischen der Anzahl an Treffpunkten aller Forkknoten und der Anzahl an Kanten. Für Algorithmen, die über die Anzahl aller Treffpunkte iterieren, kann demnach in der Praxis von einer asymptotischen Laufzeit $O(E)$ ausgegangen werden.

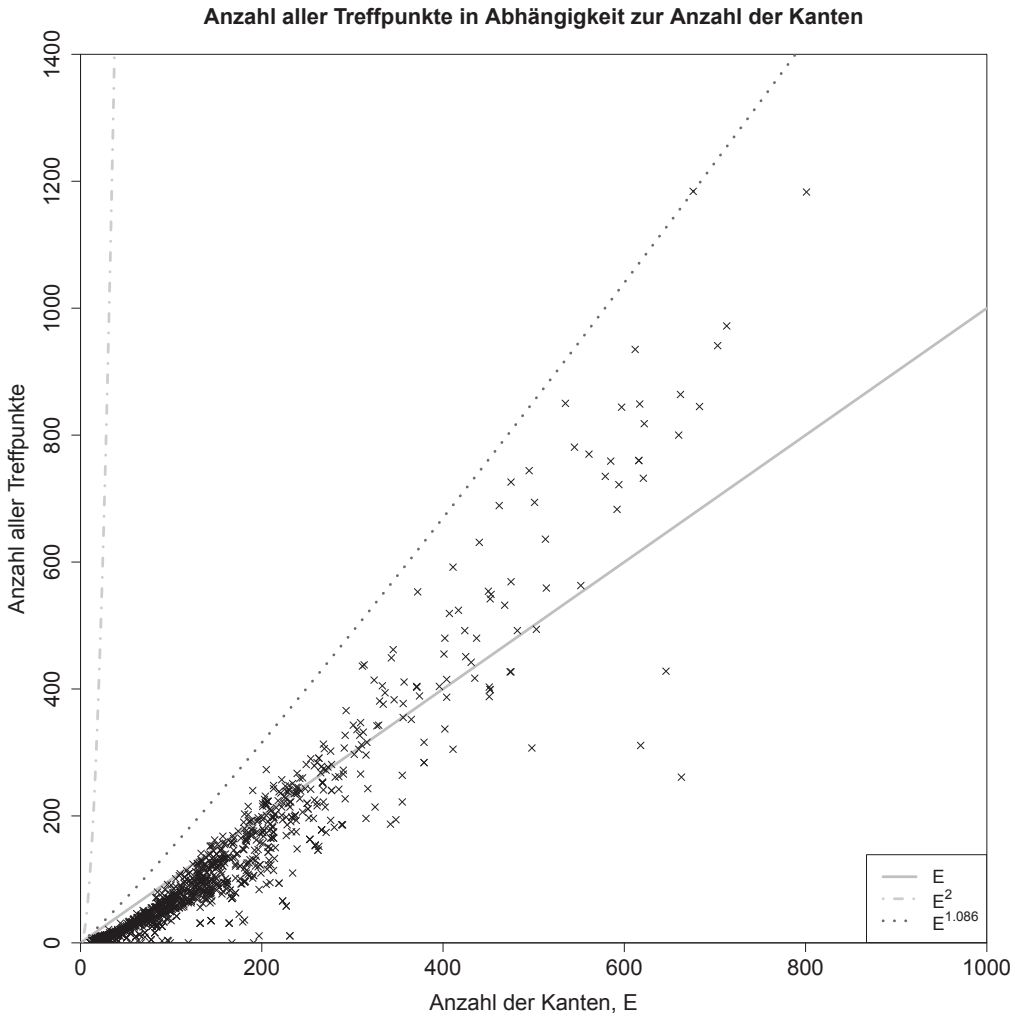


Abbildung D.1: Verhältnis der Anzahl aller Treffpunkte zur Anzahl der Kanten

Literaturverzeichnis

[1] Adler, Philipp ; Amme, Wolfram: Speculative optimizations for interpreting environments. In: *Software: Practice and Experience (SPE)* 44 (2014), Nr. 10, S. 1223–1249

[2] Alpern, Bowen ; Wegman, Mark N. ; Zadeck, F. K.: Detecting Equality of Variables in Programs. In: Ferrante, Jeanne (Hrsg.) ; Mager, P. (Hrsg.) ; Association for Computing Machinery (ACM) (Veranst.): *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1998*. Association for Computing Machinery (ACM), ACM Press, S. 1–11

[3] Amme, Wolfram ; Martens, Axel ; Moser, Simon: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: *International Journal of Business Process Integration and Management (IJBPIM)* 4 (2009), Nr. 1, S. 47–59

[4] Amme, Wolfram ; Ronne, Jeffery von ; Franz, Michael: SSA-based mobile code: Implementation and empirical evaluation. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 4 (2007), Nr. 2. – Article No. 13

[5] Ananian, C. S.: The Static Single Information Form / Massachusetts Institute of Technology (MIT). Massachusetts, September 1999 (MIT-LCS-TR-801). – Technical Report. – Available from http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-801.pdf

[6] Appel, Andrew W. ; Palsberg, Jens (Hrsg.): *Modern Compiler Implementation in Java*. Second Edition. New York, NY, USA : Cambridge University Press, 2002

[7] Apt, Krzysztof R. ; Francez, Nissim ; Katz, Shmuel: Appraising Fairness in Languages for Distributed Programming. In: *Distributed Computing* 2 (1988), Nr. 4, S. 226–241

[8] Bossert, Martin ; Breitbach, Markus ; Fliege, Norbert (Hrsg.) ; Bossert, Martin (Hrsg.): *Digitale Netze*. 1. Stuttgart, Leipzig : Teubner Verlag, 1999 (Informationstechnik)

[9] Chartrand, Gary ; Zhang, Ping: *Discrete Mathematics*. 1. Long Grove, Illinois, USA : Waveland Press, Inc., 2011

[10] Cheng, Allan ; Esparza, Javier ; Palsberg, Jens: Complexity Results for 1-Safe Nets. In: *Theoretical Computer Science* 147 (1995), Nr. 1&2, S. 117–136

[11] Chrząstowski-Wachtel, Piotr ; Benatallah, Boualem ; Hamadi, Rachid ; O'Dell, Milton ; Susanto, Adi: A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. In: van der Aalst, Wil M. P. (Hrsg.) ; ter Hofstede, Arthur H. M. (Hrsg.) ; Weske, Mathias (Hrsg.): *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings* Bd. 2678, Springer (Lecture Notes in Computer Science), S. 336–353

[12] Cooper, Keith D. ; Harvey, Timothy J. ; Kennedy, Ken: A Simple, Fast Dominance Algorithm / Department of Computer Science, Rice University. Version: 2001. `https://www.cs.rice.edu/~keith/EMBED/dom.pdf`. Houston, Texas, USA : Department of Computer Science, Rice University, 2001 (TR-06-33870). – Technical Report. – Online erhältlich

[13] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald ; Stein, Clifford ; Molitor, Paul (Hrsg.): *Algorithmen - Eine Einführung.* 4., durchgesehene und korrigierte Auflage. München : Oldenbourg Verlag, 2013

[14] Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), Nr. 4, S. 451–490

[15] Davenport, Thomas H.: *Process Innovation: Reengineering Work Through Information Technology.* Boston, MA, USA : Harvard Business School Press, 1993

[16] Desel, Jörg ; Esparza, Javiar ; Rijsbergen, C.J. van (Hrsg.) ; Abramsky, S. (Hrsg.) ; Aczel, P. H. (Hrsg.) ; Bakker, J. W. (Hrsg.) ; Goguen, J. A. (Hrsg.) ; Gurevich, Y. (Hrsg.) ; Tucker, J. V. (Hrsg.): *Free Choice Petri Nets.* Cambridge, Great Britain : Cambridge University Press, 1995 (Cambridge Tracts in Theoretical Computer Science 40)

[17] Eshuis, Rik ; Kumar, Akhil: An integer programming based approach for verification and diagnosis of workflows. In: *Data & Knowledge Engineering* 69 (2010), Nr. 8, S. 816–835

[18] Fahland, Dirk ; Favre, Cédric ; Koehler, Jana ; Lohmann, Niels ; Völzer, Hagen ; Wolf, Karsten: Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models. In: *Data & Knowledge Engineering* 70 (2011), Nr. 5, S. 448–466

[19] Favre, Cédric: *Detecting, Understanding, and Fixing Control-Flow Errors in Business Process Models.* Zürich, Switzerland, ETH Zürich, Diss., 2014. – DISS. ETH NO 22266

[20] Favre, Cédric ; Fahland, Dirk ; Völzer, Hagen: The relationship between workflow graphs and free-choice workflow nets. In: *Inf. Syst.* 47 (2015), S. 197–219

[21] Favre, Cédric ; Völzer, Hagen: Symbolic Execution of Acyclic Workflow Graphs. In: [31], S. 260–275

[22] Favre, Cédric ; Völzer, Hagen ; Müller, Peter: Diagnostic Information for Control-Flow Analysis of Workflow Graphs (a.k.a. Free-Choice Workflow Nets). In: Chechik, Marsha (Hrsg.) ; Raskin, Jean-François (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings* Bd. 9636, Springer (Lecture Notes in Computer Science), S. 463–479

[23] Friedman, Michael A. ; Voas, Jeffrey M.: *New Dimensions In Engineering Series*. Bd. Book 16: *Software Assessment: Reliability, Safety, Testability*. 1. New York, NY, USA : John Wiley & Sons, Inc., 1995

[24] Geiger, Matthias ; Neugebauer, Philipp ; Vorndran, Andreas: Automatic Standard Compliance Assessment of BPMN 2.0 Process Models. In: Kopp, Oliver (Hrsg.) ; Lenhard, Jörg (Hrsg.) ; Pautasso, Cesare (Hrsg.): *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS 2017), Lugano, Switzerland, February 13-14, 2017.* Bd. 1826, CEUR-WS.org (CEUR Workshop Proceedings), S. 4–10

[25] Harris, T. E. ; Ross, F. S.: *Fundamentals of a Method for Evaluating Rail Net Capacities*. 1. Santa Monica, California, USA : Armed Services Technical Information Agency, 1955. – RM-1573. A report prepared for United States Air Force Project RAND

[26] Heinze, Thomas S.: *Eine Methode zur kontrollierten Kontrollflussentfaltung und ihre Anwendung zur Präzisierung petrinetzbasierter Verifikationsmodelle*, Friedrich Schiller University of Jena, Diss., November 2013. `http://d-nb.info/104689952X`

[27] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: A Restructuring Method for WS-BPEL Business Processes Based on Extended Workflow Graphs. In: Dayal, Umeshwar (Hrsg.) ; Eder, Johann (Hrsg.) ; Koehler, Jana (Hrsg.) ; Reijers, Hajo A. (Hrsg.): *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings* Bd. 5701, Springer (Lecture Notes in Computer Science), S. 211–228

[28] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: Compiling More Precise Petri Net Models for an Improved Verification of Service Implementations. In: *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*, IEEE Computer Society, S. 25–32

[29] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: Process Restructuring in the Presence of Message-Dependent Variables. In: Maximilien, E. M. (Hrsg.) ; Rossi, Gustavo (Hrsg.) ; Yuan, Soe-Tsyr (Hrsg.) ; Ludwig, Heiko (Hrsg.) ; Fantinato, Marcelo (Hrsg.): *Service-Oriented Computing - ICSOC*

2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG, San Francisco, CA, USA, December 7-10, 2010, Revised Selected Papers Bd. 6568 (Lecture Notes in Computer Science), S. 121–132

[30] Hopcroft, John E. ; Tarjan, Robert E.: Dividing a Graph into Triconnected Components. In: *SIAM Journal on Computing* 2 (1973), Nr. 3, S. 135–158

[31] Hull, Richard (Hrsg.) ; Mendling, Jan (Hrsg.) ; Tai, Stefan (Hrsg.): *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings.* Bd. *6336.* Springer (Lecture Notes in Computer Science)

[32] IEEE Computer Society: IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Standard 610.12-1990* (1990), S. 1–84

[33] Johnson, R. ; Pearson, D. ; Pingali, K.: Finding regions fast: Single entry single exit and control regions in linear time. / Cornell University. Ithaca, NY, USA, Juli 1993 (TR 93-1365). – Forschungsbericht. – Technical Report

[34] Johnson, Richard ; Pearson, David ; Pingali, Keshav: The Program Structure Tree: Computing Control Regions in Linear Time. In: Sarkar, Vivek (Hrsg.) ; Ryder, Barbara G. (Hrsg.) ; Soffa, Mary L. (Hrsg.): *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, ACM, S. 171–185

[35] Juric, Matjaz B. ; Mathew, Benny K. ; Sarang, Poornachandra ; Little, Mark (Hrsg.) ; Shaffer, Dave (Hrsg.): *Business Process Execution Language for Web Services: An Architects and Developers Guide to BPEL and BPEL4WS.* Second edition. Birmingham, UK : Packt Publishing Ltd., 2006 (From Technologies to Solutions)

[36] Kemper, Peter ; Bause, Falko: An Efficient Polynomial-Time Algorithm to Decide Liveness and Boundedness of Free-Choice Nets. In: Jensen, Kurt (Hrsg.): *Application and Theory of Petri Nets 1992, 13th International Conference, Sheffield, UK, June 22-26, 1992, Proceedings* Bd. 616, Springer (Lecture Notes in Computer Science), S. 263–278

[37] Khachiyan, L. G.: Polynomial algorithms in linear programming. In: *USSR Computational Mathematics and Mathematical Physics* 20 (1980), Nr. 1, S. 53–72

[38] Kiepuszewski, Bartek ; ter Hofstede, Arthur H. M. ; van der Aalst, Wil M. P.: Fundamentals of Control Flow in Workflows. In: *Acta Informatica* 39 (2003), Nr. 3, S. 143–209

[39] Kindler, Ekkart ; van der Aalst, Wil M. P.: Liveness, Fairness, and Recurrence in Petri Nets. In: *Information Processing Letters* 70 (1999), Nr. 6, S. 269–274

[40] Kirk, Wolfgang: *Die öffentliche Verwaltung der Bundesrepublik Deutschland auf dem Weg zum Verwaltungsbetrieb*. Bd. 8: *Public Management: Gestaltung von Dienstleistungen im allgemeinen Interesse - Prozessmanagement*. 1. Norderstedt, Germany : Books on Demand, 2010

[41] Kotzmann, Thomas ; Wimmer, Christian ; Mössenböck, Hanspeter ; Rodriguez, Thomas ; Russell, Kenneth ; Cox, David: Design of the Java HotSpot™ client compiler for Java 6. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5 (2008), Nr. 1, S. 7:1–7:32. – Artikel 7

[42] Kühne, Stefan ; Kern, Heiko ; Gruhn, Volker ; Laue, Ralf: Business process modeling with continuous validation. In: *Journal of Software Maintenance* 22 (2010), Nr. 6-7, S. 547–566

[43] Lee, Jaejin ; Midkiff, Samuel P. ; Padua, David A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: Li, Zhiyuan (Hrsg.) ; Yew, Pen-Chung (Hrsg.) ; Chatterjee, Siddhartha (Hrsg.) ; Huang, Chua-Huang (Hrsg.) ; Sadayappan, P. (Hrsg.) ; Sehr, David C. (Hrsg.): *Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota, USA, August 7-9, 1997, Proceedings* Bd. 1366, Springer (Lecture Notes in Computer Science), S. 114–130

[44] Lengauer, Thomas ; Tarjan, Robert E.: A Fast Algorithm for Finding Dominators in a Flowgraph. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1 (1979), Nr. 1, S. 121–141

[45] Lenhard, Jörg: *Portability of Process-Aware and Service-Oriented Software: Evidence and Metrics*, University of Bamberg, Diss., Januar 2016. `https://opus4.kobv.de/opus4-bamberg/frontdoor/index/index/docId/46252`

[46] Lestor R. Ford, Jr. ; Fulkerson, D. R.: *Flows in Networks*. 1. Santa Monica, California, USA : Princeton University Press, 1962. – R-375-PR. A report prepared for United States Air Force Project RAND

[47] Lindholm, Tim ; Yellin, Frank ; Bracha, Gilad ; Buckley, Alex: *The Java Virtual Machine Specification, Java SE 7 Edition*. 1st. Crawfordsville, Indiana, USA : Addison-Wesley Professional, 2013

[48] Lohmann, Niels ; Fahland, Dirk: Where Did I Go Wrong? - Explaining Errors in Business Process Models. In: Sadiq, Shazia W. (Hrsg.) ; Soffer, Pnina (Hrsg.) ; Völzer, Hagen (Hrsg.): *Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings* Bd. 8659, Springer (Lecture Notes in Computer Science), S. 283–300

[49] Lowry, Edward S. ; Medlock, C. W.: Object Code Optimization. In: *Communications of the ACM* 12 (1969), Nr. 1, S. 13–22

[50] Marick, Brian ; Becker, Paul (Hrsg.): *The Craft of Software Testing: Subsystem Testing. Including Object-Based and Object-Oriented Testing*. 1th Edition. New Jersey, USA : Prentice Hall PTR, 1995

[51] Murata, Tadao: Petri Nets: Properties, Analysis and Applications. In: *Proceedings of the IEEE* 77 (1989), Nr. 4, S. 541–580

[52] Object Management Group (OMG): *Business Process Model and Notation (BPMN) Version 2.0*. `http://www.omg.org/spec/BPMN/2.0`. Version: Januar 2011. – Standard

[53] Pahl, Peter J. ; Damrath, Rudolf ; Pahl, Felix (Hrsg.): *Mathematical Foundations of Computational Engineering: A Handbook*. 1. Auflage. Berlin, Germany : Springer-Verlag, 2001

[54] Petri, Carl A.: *Kommunikation mit Automaten*. Bonn, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Diss., Juli 1962

[55] Polyvyanyy, Artem ; Weidlich, Matthias: Towards a Compendium of Process Technologies - The jBPT Library for Process Model Analysis. In: Deneckère, Rébecca (Hrsg.) ; Proper, Henderik A. (Hrsg.): *Proceedings of the CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering (CAiSE), Valencia, Spain, June 20th, 2013. Proceedings* Bd. 998, CEUR-WS.org (CEUR Workshop Proceedings), S. 106–113

[56] Prinz, Thomas M.: Proposals for a Virtual Machine for Business Processes. In: Heinze, Thomas S. (Hrsg.) ; Prinz, Thomas M. (Hrsg.): *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015*. Bd. 1360, CEUR-WS.org (CEUR Workshop Proceedings), S. 10–17

[57] Prinz, Thomas M. ; Amme, Wolfram: Why We Need Advanced Analyses of Service Compositions. In: Barros, Marcelo de (Hrsg.) ; Klink, Janusz (Hrsg.) ; Uhl, Tadeus (Hrsg.) ; Prinz, Thomas M. (Hrsg.) ; IARIA Conference (Veranst.): *SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing, Athens, Greece, February 19–23, 2017. Proceedings* Bd. 8 IARIA Conference, ThinkMind Digital Library, S. 48–54

[58] Prinz, Thomas M. ; Amme, Wolfram: A Complete and the Most Liberal Semantics for Converging OR Gateways in Sound Processes. In: *Complex Systems Informatics and Modeling Quarterly (CSIMQ)* 4 (2015), S. 32–49

[59] Prinz, Thomas M. ; Charrondière, Raphaël ; Amme, Wolfram: Geschäftsprozesse kompiliert - Wichtige Unterstützung für die Modellierung. In: Knoop, Jena (Hrsg.) ; Ertl, M. A. (Hrsg.): *Proceedings 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach am Wörthersee, Austria, October 5–7, 2015*, Institut für Computersprachen, Technische Universität Wien, S. 476–491

[60] Prinz, Thomas M. ; Heinze, Thomas S. ; Amme, Wolfram ; Kretzschmar, Johannes ; Beckstein, Clemens: Towards a Compiler for Business Processes - A Research Agenda. In: Barros, Marcelo de (Hrsg.) ; Rückemann, Claus-Peter (Hrsg.) ; IARIA Conference (Veranst.): *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, Nice,*

France, March 22–27, 2015. Proceedings Bd. 6 IARIA Conference, ThinkMind Digital Library, S. 49–54

[61] Prinz, Thomas M. ; Spiess, Norbert ; Amme, Wolfram: A First Step towards a Compiler for Business Processes. In: Cohen, Albert (Hrsg.): *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings* Bd. 8409, Springer (Lecture Notes in Computer Science), S. 238–243

[62] Prosser, Reese T.: Applications of Boolean Matrices to the Analysis of Flow Diagrams. In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference.* New York, NY, USA : ACM (IRE-AIEE-ACM '59 (Eastern)), S. 133–138

[63] Puhlmann, Frank: Soundness Verification of Business Processes Specified in the Pi-Calculus. In: Meersman, Robert (Hrsg.) ; Tari, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I* Bd. 4803, Springer (Lecture Notes in Computer Science), S. 6–23

[64] Reisig, Wolfgang: The Linear Theory of Multiset Based Dynamic Systems. In: Calude, Cristian (Hrsg.) ; Paun, Gheorghe (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Salomaa, Arto (Hrsg.): *Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View [Workshop on Multiset Processing, WMP 2000, Curtea de Arges, Romania, August 21-25, 2000]* Bd. 2235, Springer (Lecture Notes in Computer Science), S. 287–298

[65] Reisig, Wolfgang: *Petrinetze: Eine Einführung.* 2., überarb. u. erw. Aufl. Berlin; Heidelberg : Springer-Verlag, 1986 (Studienreihe Informatik)

[66] Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Global Value Numbers and Redundant Computations. In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988.* Association for Computing Machinery (ACM), ACM Press, S. 12–27

[67] Rosen, Kenneth H.: *Discrete Mathematics and Its Applications.* 7th revised edition. New York, USA : Mcgraw-Hill Education Ltd, 2012

[68] Sadiq, Wasim ; Orlowska, Maria E.: Analyzing Process Models Using Graph Reduction Techniques. In: *Information Systems* 25 (2000), Nr. 2, S. 117–134

[69] Schmidt, Karsten: LoLA: A Low Level Analyser. In: Nielsen, Mogens (Hrsg.) ; Simpson, Dan (Hrsg.): *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26–30, 2000*, Springer, S. 465–474

[70] Sidorova, Natalia ; Stahl, Christian ; Trcka, Nikola: Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. In: *Information Systems* 36 (2011), Nr. 7, S. 1026–1043

[71] Singer, Robert: Agent-Based Business Process Modeling and Execution: Steps Towards a Compiler-Virtual Machine Architecture. In: Sanz, Jorge L. (Hrsg.): *Proceedings of the 8th International Conference on Subject-oriented Business Process Management, S-BPM ONE 2016, Erlangen, Germany, April 7-8, 2016*, ACM, S. 8:1–8:10

[72] Singer, Robert: Business Process Modeling and Execution - A Compiler for Distributed Microservices. In: *Computing Research Repository (CoRR)* abs/1601.05976 (2016)

[73] Surendran, Rishi ; Barik, Rajkishore ; Zhao, Jisheng ; Sarkar, Vivek: Inter-iteration Scalar Replacement Using Array SSA Form. In: Cohen, Albert (Hrsg.): *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings* Bd. 8409, Springer (Lecture Notes in Computer Science), S. 40–60

[74] Tarjan, Robert E.: Prime Subprogram Parsing of a Program. In: Abrahams, Paul W. (Hrsg.) ; Lipton, Richard J. (Hrsg.) ; Bourne, Stephen R. (Hrsg.): *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, ACM Press, S. 95–105

[75] Valmari, Antti: The State Explosion Problem. In: Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, Wadern, Germany, September 1996.* Bd. 1491, Springer (Lecture Notes in Computer Science), S. 429–528

[76] van der Aalst, Wil M. P.: Interval Timed Coloured Petri Nets and their Analysis. In: Marsan, Marco A. (Hrsg.): *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings* Bd. 691, Springer (Lecture Notes in Computer Science), S. 453–472

[77] van der Aalst, Wil M. P.: Verification of Workflow Nets. In: Azéma, Pierre (Hrsg.) ; Balbo, Gianfranco (Hrsg.): *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings* Bd. 1248, Springer (Lecture Notes in Computer Science), S. 407–426

[78] van der Aalst, Wil M. P.: A class of Petri nets for modeling and analyzing business processes / Eindhoven University of Technology. Eindhoven, Netherlands, 1995 (95/26). – Computing Science Reports. – Technical Report

[79] van der Aalst, Wil M. P.: The Application of Petri Nets to Workflow Management. In: *Journal of Circuits, Systems, and Computers* 8 (1998), Nr. 1, S. 21–66

[80] van der Aalst, Wil M. P. ; Hirnschall, Alexander ; Verbeek, H. M. W. E.: An Alternative Way to Analyze Workflow Graphs. In: Pidduck, Anne B. (Hrsg.) ; Mylopoulos, John (Hrsg.) ; Woo, Carson C. (Hrsg.) ; Özsu, M. T.

(Hrsg.): *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings* Bd. 2348, Springer (Lecture Notes in Computer Science), S. 535–552

[81] van der Aalst, Wil M. P. ; ter Hofstede, Arthur H. M.: YAWL: Yet Another Workflow Language. In: *Information Systems* 30 (2005), Nr. 4, S. 245–275

[82] van der Aalst, Wil M. P. ; van Hee, Kees M. ; ter Hofstede, Arthur H. M. ; Sidorova, Natalia ; Verbeek, H. M. W. ; Voorhoeve, Marc ; Wynn, Moe T.: Soundness of workflow nets: classification, decidability, and analysis. In: *Formal Aspects of Computing* 23 (2011), Nr. 3, S. 333–363

[83] van Dongen, Boudewijn F. ; Mendling, Jan ; van der Aalst, Wil M. P.: Structural Patterns for Soundness of Business Process Models. In: *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), October 16-20, 2006, Hong Kong, China*, IEEE Computer Society, S. 116–128

[84] Vanhatalo, Jussi ; Völzer, Hagen ; Koehler, Jana: The Refined Process Structure Tree. In: *Data & Knowledge Engineering* 68 (2009), Nr. 9, S. 793–818

[85] Vanhatalo, Jussi ; Völzer, Hagen ; Leymann, Frank: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, Bernd J. (Hrsg.) ; Lin, Kwei-Jay (Hrsg.) ; Narasimhan, Priya (Hrsg.): *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007. Proceedings* Bd. 4749, Springer (Lecture Notes in Computer Science), S. 43–55

[86] Vanhatalo, Jussi ; Völzer, Hagen ; Leymann, Frank ; Moser, Simon: Automatic Workflow Graph Refactoring and Completion. In: Bouguettaya, Athman (Hrsg.) ; Krüger, Ingolf (Hrsg.) ; Margaria, Tiziana (Hrsg.): *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings* Bd. 5364 (Lecture Notes in Computer Science), S. 100–115

[87] Verbeek, H. M. W. E. ; Basten, Twan ; van der Aalst, Wil M. P.: Diagnosing Workflow Processes using Woflan. In: *The Computer Journal* 44 (2001), Nr. 4, S. 246–279

[88] Völzer, Hagen: A New Semantics for the Inclusive Converging Gateway in Safe Processes. In: [31], S. 294–309

[89] Weber, Michael ; Kindler, Ekkart: The Petri Net Markup Language. In: Ehrig, Hartmut (Hrsg.) ; Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Weber, Herbert (Hrsg.): *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets, 2003. Proceedings* Bd. 2472, Springer (Lecture Notes in Computer Science), S. 124–144

[90] Wegman, Mark N. ; Zadeck, F. K.: Constant Propagation with Conditional Branches. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Nr. 2, S. 181–210

[91] Wynn, Moe T. ; Verbeek, H. M. W. ; van der Aalst, Wil M. P. ; ter Hofstede, Arthur H. M. ; Edmond, David: Business process verification - finally a reality! In: *Business Process Management Journal* 15 (2009), Nr. 1, S. 74–92

[92] Zima, Hans ; Chapman, Barbara: *Supercompilers for Parallel and Vector Computers*. New York, NY, USA : ACM Press Frontier Series, 1991

Abkürzungsverzeichnis

BPEL	Business Process Execution Language	11
BPMN	Business Process Model and Notation	1
CSSA-Form	Concurrent Static Single Assignment Form	113
CSV	Comma-Separated Values	119
JIT	Just-in-Time	111
JVM	Java Virtuellen Maschine	120
PNML	Petri Net Markup Language	86
SESE	Single-Entry-Single-Exit	23
SSA-Form	Static Single Assignment Form	71
YAWL	Yet Another Workflow Language	11
ZCR	Zwischencoderepräsentation	85

Abbildungsverzeichnis

Tabellenverzeichnis

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass

- mir die Promotionsordnung der Fakultät bekannt ist,
- ich die Dissertation selbst angefertigt habe,
- ich keine Textabschnitte oder Ergebnisse eines Dritten oder eigene Prüfungsarbeiten ohne Kennzeichnung übernommen habe,
- alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben wurden,
- ich keine Hilfe eines Promotionsberaters in Anspruch genommen habe,
- Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhielten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung einreichte.

Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts haben mich folgende Personen unterstützt:

- Prof. Dr. Wolfram Amme

Ich habe weder die gleiche, noch eine in wesentlichen Teilen ähnliche bzw. andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht.

....

Ort, Datum Unterschrift

Veröffentlichungen

Editorentätigkeit

[1] de Barros, Marcelo ; Klink, Janusz ; Uhl, Tadeus ; Prinz, Thomas M.: *Proceedings of the SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing.* Athens, Greece, February 19–23, 2017, IARIA Conference, Band 8.

[2] Heinze, Thomas S. ; Prinz, Thomas M.: *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015.* Jena, Germany, February 19–20, 2015, `CEUR-WS.org` Nr. 1360

Artikel in Zeitschriften

[1] Prinz, Thomas M. ; Amme, Wolfram: A Complete and the Most Liberal Semantics for Converging OR Gateways in Sound Processes. In: *Complex Systems Informatics and Modeling Quarterly (CSIMQ)* 4 (2015), S. 32–49

Beiträge in Tagungsbänden

[1] Prinz, Thomas M. ; Amme, Wolfram: Why We Need Advanced Analyses of Service Compositions. In: de Barros, Marcelo (Hrsg.) ; Klink, Janusz (Hrsg.) ; Uhl, Tadeus (Hrsg.) ; Prinz, Thomas M. (Hrsg.) ; IARIA Conference (Veranst.): *SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing, Athens, Greece, February 19–23, 2017. Proceedings* Bd. 8 IARIA Conference, ThinkMind Digital Library, S. 48–54

[2] Prinz, Thomas M.: Advanced Analysis of Service Compositions. *Editorial to SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing.* Athens, Greece, February 19–23, 2017

[3] Prinz, Thomas M. ; Gebhardt, Kai ; Meyer, Manuela ; Mundhenk, Martin: MINTividual: Ein Konzept für ein individualisiertes Studium in den Naturwissenschaften. *DisQspace auf der 45. Jahrestagung der Deutschen Gesellschaft für Hochschuldidaktik, dghd 2016.* Bochum, Germany, September 21–23, 2016

[4] Gräfe, Linda ; Prinz, Thomas M. ; Plötner, Jan ; Heßler, Christoph ; Vetterlein, Anja: Ihre Evaluation im sicheren Hafen: Das Evaluationssystem coa.st. In: *Tagungsband des 50. Kongress der Deutschen Gesellschaft für Psychologie, 5zig Leipsych 2016.* Leipzig, Germany, September 18–22, 2016, Postervorstellung

[5] Prinz, Thomas M. ; Kretzschmar, Johannes ; Hempel, Paul ; Schau, Volkmar: A Knowledge Base for Electric Vehicles in Inner-City Logistics. In: Oberhauser, Roy (Hrsg.) ; Lavazza, Luigi (Hrsg.) ; Mannaert, Herwig (Hrsg.) ; Clyde, Stephen (Hrsg.) ; IARIA Conference (Veranst.): *ICSEA 2015: The Tenth International Conference on Software Engineering Advances, Barcelona, Spain, November 15–20, 2015. Proceedings* Bd. 10 IARIA Conference ThinkMind Digital Library, S. 257–260. **Best Paper Award**

[6] Prinz, Thomas M. ; Charrondière, Raphaël ; Amme, Wolfram: Geschäftsprozesse kompiliert - Wichtige Unterstützung für die Modellierung. In: Knoop, Jena (Hrsg.) ; Ertl, M. A. (Hrsg.): *Proceedings 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach am Wörthersee, Austria, October 5–7, 2015*, Institut für Computersprachen, Technische Universität Wien, S. 476–491

[7] Prinz, Thomas M. ; Heinze, Thomas S. ; Amme, Wolfram ; Kretzschmar, Johannes ; Beckstein, Clemens: Towards a Compiler for Business Processes - A Research Agenda. In: de Barros, Marcelo (Hrsg.) ; Rückemann, Claus-Peter (Hrsg.) ; IARIA Conference (Veranst.): *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, Nice, France, March 22–27, 2015. Proceedings* Bd. 6 IARIA Conference, ThinkMind Digital Library, S. 49–54

[8] Prinz, Thomas M.: Proposals for a Virtual Machine for Business Processes. In: Heinze, Thomas S. (Hrsg.) ; Prinz, Thomas M. (Hrsg.): *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19–20, 2015.* Bd. 1360, CEUR-WS.org (CEUR Workshop Proceedings), S. 10–17

[9] Apel, Sebastian ; Prinz, Thomas M. ; Schau, Volkmar: Challenging Service Extensions for Electric Vehicles in Massively Heterogenic System Landscapes. In: Heinze, Thomas S. (Hrsg.) ; Prinz, Thomas M. (Hrsg.): *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19–20, 2015.* Bd. 1360, CEUR-WS.org (CEUR Workshop Proceedings), S. 44–50

[10] Prinz, Thomas M. ; Spieß, Norbert ; Amme, Wolfram: A First Step towards a Compiler for Business Processes. In: Cohen, Albert (Hrsg.): *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings* Bd. 8409, Springer (Lecture Notes in Computer Science), S. 238–243

[11] Prinz, Thomas M. ; Amme, Wolfram: Practical Compiler-Based User Support during the Development of Business Processes. In: Lomuscio, Alessio (Hrsg.) ; Nepal, Surya (Hrsg.) ; Patrizi, Fabio (Hrsg.) ; Benatallah, Boualem (Hrsg.) ; Brandic, Ivona (Hrsg.): *Service-Oriented Computing - ICSOC 2013, Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2–5, 2013. Revised Selected Papers* Bd. 8377, Springer (Lecture Notes in Computer Science), S. 40–53

[12] Schau, Volkmar ; Apel, Sebastian ; Gebhardt, Kai ; Prinz, Thomas M. ; Späthe, Steffen ; Nagel, Katharina ; Rossak, Wilhem R.: SmartCityLogistik (SCL) Erfurt: Deriving the main factors that influence vehicle range during short-distance freight transport when using fully electric vehicles. In: Schau, Volkmar (Hrsg.) ; Eichler, Gerald (Hrsg.) ; Roth, Jörg (Hrsg.): *Tagungsband des 10. GI/KuVS-Fachgespräch Ortsbezogene Anwendungen und Dienste Bd. 10, Jena, Germany, November 16–17, 2013.* Logos Berlin, S. 101–108

[13] Prinz, Thomas M.: Fast Soundness Verification of Workflow Graphs. In: Kopp, Oliver (Hrsg.) ; Lohmann, Niels (Hrsg.): *Proceedings of the 5th Central-European Workshop on Services and their Composition, ZEUS 2013, Rostock, Germany, February 21–22, 2013*, S. 32–40

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Zusammenfassung

Oft kann die Untersuchung, Erkennung und Niederschrift von impliziten Abläufen von großem Nutzen sein. Dadurch können Unternehmen ihre Ziele erreichen und sogar Menschenleben gerettet werden. Eine formale Niederschrift eines solchen Ablaufs heißt *Arbeitsprozess* (Workflow).

Arbeitsprozesse werden sehr häufig in visuellen Notationen als Graphen aus Knoten und Kanten formalisiert. Dabei gibt es verschiedene Arten von Knoten mit unterschiedlichen Semantiken. Durch diese können einzelne Aufgaben beschrieben sowie Entscheidungen und Parallelitäten modelliert werden.

Der Vorgang von der Untersuchung eines Ablaufs bis zur Formalisierung zu einem Arbeitsprozess ist schwierig und wird deswegen von geschulten Experten übernommen. Leider treten dabei dennoch häufig Fehler auf. Diese Fehler sind gerade in umfangreichen Arbeitsprozessen sehr schwierig zu identifizieren und führen zu einem inkorrekten Prozessverhalten. Ein inkorrektes Verhalten kann zum Nichterreichen von Unternehmenszielen und sogar im schlimmsten Fall zum Verlust von Menschenleben führen. Kurzum: Fehlverhalten sind teuer und gefährlich! Die vorliegende Arbeit untersucht solches Fehlverhalten und deckt deren *Ursachen* auf, *bevor* sie Schaden verursachen können.

Als Grundlage zur Feststellung einer Ursache eines fehlerhaften Verhaltens dient der Korrektheitsbegriff (Soundness). Er schließt das Auftreten von *Verklemmungen* und *Überflüssen* (Lacks of Synchronization) aufgrund des Kontrollflusses aus. Um die *Ursachen* dieser Fehler*wirkungen* aufzudecken, wird eine partielle Analyse eingeführt. Diese ermöglicht die Untersuchung ausgehend von verschiedenen Punkten innerhalb eines Arbeitsprozesses. Dadurch können *potentielle* Fehlerursachen hinter anderen Fehlern entdeckt werden.

Darauf aufbauend präsentiert diese Arbeit zwei neue Techniken: Die erste Technik findet die Ursachen aller unmittelbar auftretenden, potentiellen Verklemmungen; die zweite ermittelt die Ursachen aller potentiellen Überflüsse. Beide Techniken besitzen eine *kubisch* asymptotische Laufzeitkomplexität für realitätsnahe Prozesse hinsichtlich der Anzahl der Kanten. Zudem liefern sie noch detaillierte diagnostische Informationen, die hervorragend zur Visualisierung und Erklärung der gefundenen Fehlerursachen dienen können. Beide Eigenschaften der Algorithmen zertifizieren sie für den Einsatz als unterstützendes Analyseverfahren, wie deren Anwendung in jedem Konstruktionsschritt.

Als Nachweis der Effizienz und Qualität dieser Techniken wird zudem ein Werkzeug, *Mojo*, eingeführt, welches diese implementiert. Dieses Werkzeug wird außerdem für eine Evaluation mit über 1.300 Arbeitsprozessen aus der Praxis genutzt und mit zwei anderen Techniken des aktuellen Stands der Forschung verglichen.

DISSERTATIONSSCHRIFT