HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Formale Instanzverifikation zertifizierender verteilter Algorithmen:
Fallstudie Zweifärbbarkeit & alternative Konsistenzprüfung

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker (Dipl.-Inf.)

eingereicht von: Alexander Boll
geboren am: 21.04.1987
geboren in: Bernau

Gutachter/innen: Prof. Dr. Wolfgang Reisig
Prof. Dr. Nicole Schweikardt

eingereicht am: .... verteidigt am: ....

Inhaltsverzeichnis

1 Einleitung

In fast allen Bereichen unseres täglichen Lebens verlassen wir uns auf Computer-Programme. Daher ist es wichtig, dass Programme beziehungsweise deren zugrunde liegenden Algorithmen korrekt arbeiten. Um die Korrektheit von einem Algorithmus zu beweisen, wird üblicherweise ein Korrektheitsbeweis auf Papier durch den Entwickler des Algorithmus' geliefert. Für den implementierten Algorithmus – das Programm – ist die formale Verifikation eine bekannte Technik die Korrektheit sicher zu stellen. Formale Verifikation verlangt häufig das Beweisen nicht-trivialer mathematischer Theoreme (vgl. [ABMR11]) und ist damit sehr anspruchsvoll.
Eine in der Industrie häufig genutzte Alternative stellt das Testen dar. Da beim Testen jedoch nur die Korrektheit einiger Eingabe/Ausgabe-Paare überprüft wird, ist diese Methode in sicherheitskritischen Domänen nicht ausreichend. Alle nicht getesteten Eingabe/Ausgabe-Paare sind nicht vertrauenswürdig.

Eine besondere Art von Algorithmus ist ein *verteilter Algorithmus*. Ein verteilter Algorithmus läuft auf einem verteilten System, beispielsweise einem *Netzwerk*. Ein Netzwerk besteht aus Komponenten und Kommunikationskanälen zwischen den Komponenten. Die Komponenten können eigene Berechnungen ausführen und dabei Nachrichten mit anderen Komponenten austauschen. Die empfangenen Nachrichten können Komponenten wieder in die eigene Berechnung einfließen lassen.
Es ist besonders herausfordernd, verteilte Algorithmen korrekt zu entwickeln, zu implementieren und formal zu verifizieren (siehe [WWP$^+$15], [Ong14], [FZWK17]). In dieser Diplomarbeit wollen wir daher die Verifikation verteilter Algorithmen vereinfachen.

Alkassar et al. entwickelten in [ABMR14] ein Verfahren, um Eingabe/Ausgabe-Paare sequentieller Algorithmen zur Laufzeit zu verifizieren. Das Verfahren beruht auf *zertifizierenden Algorithmen*: ein zertifizierender Algorithmus berechnet zusätzlich zu seiner normalen Ausgabe einen *Zeugen* (oder Zertifikat). Mit Hilfe eines sogenannten *Zeugenprädikates* kann die Ausgabe verifiziert werden: gilt das Zeugenprädikat für Eingabe, Ausgabe und den Zeugen, so ist das Eingabe/Ausgabe-Paar korrekt. Dieser Zusammenhang heißt *Zeugeneigenschaft*. Das Zeugenprädikat wird von einem weiteren Programm, dem *Checker* entschieden. Für geschickt gewählte Zeugen ist der Checker einfacher aufgebaut als das eigentliche verteilte Programm. Die Verifikation des Checkers ist deswegen oft sogar dann möglich, wenn die Verifikation des verteilten Programms zu aufwändig ist.
Mit der Kombination von zertifizierenden Algorithmen mit formaler Verifikation des Checkers erreichen wir *formale Instanzverifikation* (vgl. [ABMR14]). Für eine konkrete Instanz, also ein Eingabe/Ausgabe-Paar, erhalten wir zur Laufzeit mittels formaler Instanzverifikation *formale Instanzkorrektheit* – einen maschinen-geprüften Beweis für die Korrektheit des Paares.

Im Rahmen ihrer Doktorarbeit entwickelte Kim Völlinger ein Framework (vgl. [VA18]) für formale Instanzkorrektheit auf Grundlage von *zertifizierenden verteilten Algorithmen* (ZVA). Das Framework von Völlinger bildet die Basis dieser Diplomarbeit. Wir vereinfachen das Framework, implementieren Teile davon und wenden es auf ein neues Fallbeispiel an.

Methode dieser Arbeit

In dieser Arbeit, so wie allen bisherigen Arbeiten, die sich mit dem Framework beschäftigten, führen wir alle Beweise mit dem Beweisassistenten Coq. Nur so können wir formale Instanzkorrektheit gewährleisten.

Menschen-geführte und menschen-geprüfte Beweise „auf Papier“ sind nicht vertrauenswürdig: Menschen übersehen Flüchtigkeitsfehler, bedenken Fälle von Fallunterscheidungen nicht etc. Daher entwickeln wir für unsere Aussagen und Hypothesen maschinen-geprüfte Beweise. Maschinen-geprüfte Beweise sind vertrauenswürdig aber schwierig zu führen, da sie kleinschrittig und sehr genau sind. Die Vertrauensbasis ist nur der Beweisassistent selbst – auf dessen Vertraulichkeit gehen wir in Abschnitt 2.3 ein.

Mit einem Beweisassistenten spannen wir eine Beweiskette, angefangen von Definitionen von Typen, Strukturen und Funktionen über Lemmata hin zu einem Schluss-Theorem. Für Außenstehende, ist dabei nur der Start- und Endpunkt der Beweiskette wichtig. Alle Schritte dazwischen werden maschinell geprüft und sind daher vertrauenswürdig. Wichtig ist vor allem, dass das jeweilige Schluss-Theorem und alle dafür wichtigen Definitionen sinnhaft sind. Aus diesem Grund gehen wir in dieser Arbeit auf Zwischen-Lemmata weniger detailliert ein.

Bisherige Arbeiten

Bis zum Erscheinen dieser Diplomarbeit gibt es kaum Arbeiten zu zertifizierenden verteilten Algorithmen. Völlinger et al. schrieben bis jetzt folgende Arbeiten zu dem Thema: Völlinger und Reisig übertrugen das Konzept der Verifikation zertifizierender Algorithmen in [VR15] erstmals auf verteilte Algorithmen. In Ashers Diplomarbeit [Ash16] wurde die Zeugeneigenschaft des zertifizierenden verteilten Bellman-Ford-Algorithmus zu großen Teilen maschinen-geprüft im Sinne der formalen Instanzverifikation bewiesen. Mit Leader Election wurde ein weiteres Fallbeispiel in [VA17] betrachtet. Ebenfalls wurde dort eine Methode vorgestellt, wie formale Instanzkorrektheit für ZVAs gezeigt werden kann. Die Methode wurde in [Völ17] erweitert, um auch komplexere Fallbeispiele, wie Bipartition abzudecken.

Bei ZVAs besteht der Zeuge aus *Teil-Zeugen*, die jeweils auf die Komponenten des Netzwerks verteilt sind. Damit die Zeugeneigenschaft mithilfe der Teil-Zeugen bewiesen werden kann, dürfen sich die Teil-Zeugen nicht gegenseitig widersprechen. [VA18] und [Aki18] behandelten diese Widerspruchsfreiheit der Teil-Zeugen und erstellten ein Framework für den Beweisassistenten Coq, um formale Instanzkorrektheit zu zeigen. In dem Framework gibt es verschiedene Beweisverpflichtungen, die teilweise nur ein einziges Mal und teilweise für jeden ZVA einzeln zu führen sind.

1.1 Ziele der Arbeit

Wir leisten in dieser Arbeit die folgenden vier Beiträge:

Framework vereinfachen. Bisher müssen für die Nutzung des Frameworks für jeden ZVA zusätzliche Beweise erbracht werden. Wir wollen die Anzahl dieser zusätzlichen Beweise reduzieren. Damit sinkt der Zusatzaufwand der Anwendung des Frameworks für Entwickelnde eines ZVA.

Fallbeispiel Zweifärbbarkeit. Die vorangegangenen Arbeiten haben das Framework auf folgende Fallbeispiele angewendet: den zertifizierenden verteilten Bellman-Ford-Algorithmus (siehe [Ash16]) und der Leader-Election (siehe [VA17]). In beiden Arbeiten fehlen jedoch noch einige notwendige maschinen-geprüfte Beweise. Wir wollen das Framework auf ein komplexeres Beispiel anwenden und eine lückenlose maschinen-geprüfte Beweiskette der Beweisverpflichtungen liefern. Wir demonstrieren auch, dass das Framework für komplexere Zeugenprädikate angewendet werden kann. Gleichzeitig untersuchen wir, ob der modulare Aufbau des Frameworks die Wiederverwendung von Beweisen aus den Vorgänger-Arbeiten unterstützt.

Konsistenzprüfung implementieren. Wenn die Teil-Zeugen in sich widersprüchlich sind, können keine Aussagen aus ihnen hergeleitet werden. Aus diesem Grund ist eine Konsistenzprüfung der Teil-Zeugen für das Framework unentbehrlich. Die Arbeit von Akili (siehe [Aki18]) stellte eine Möglichkeit der Konsistenzprüfung bereit. Die Konsistenzprüfung beschränkt sich jedoch auf spezielle Zeugen. Wir entwickeln einen alternativen verteilten Algorithmus zur Konsistenzprüfung, sodass die Teil-Zeugen auf Konsistenz geprüft werden können. Weiter beweisen wir zu großen Teilen die Korrektheit der Konsistenzprüfung mittels eines maschinen-geprüften Beweises.

Beweislücken schließen. In der Arbeit von Asher [Ash16] war die maschinen-geprüfte Beweisführung der Zeugeneigenschaft des Bellman-Ford-ZVA unvollständig. Ein schwieriger Beweis konnte auch mit Hilfe einer Coq-Experten-Mailingliste (siehe [Coq16]) nicht geführt werden. Asher spekulierte in seiner Arbeit, dass der Beweisassistent Coq nicht reichen könne, um das Problem zu lösen. Wir schließen in dieser Arbeit alle Lücken seiner Arbeit, um zu zeigen, dass Coq als alleiniges Werkzeug für das Framework ausreicht. Als alleiniges Werkzeug ist die benötigte Vertrauensbasis des Frameworks geringer.

1.2 Aufbau der Arbeit

In Kapitel 2 führen wir die wichtigsten Grundlagen für das Verständnis der Arbeit ein. Zuerst erklären wir das Konzept der zertifizierenden verteilten Algorithmen (ZVA) und unser Framework für formale Instanz-Verifikation für ZVAs. Danach geben wir einen

Überblick über den von uns genutzten Beweisassistenten Coq und dessen in dieser Arbeit genutzten Bibliotheken.

Kapitel 3 wendet unser Framework auf das Fallbeispiel der Zweifärbbarkeits-Prüfung an. Wir skizzieren einen ZVA, der Zweifärbbarkeit entscheidet und implementieren einen Checker, der das Ergebnis des ZVA prüft. Des Weiteren beweisen wir maschinen-geprüft die Korrektheit des Checkers.

Wir stellen unsere Konsistenzprüfung des Zeugen in Kapitel 4 vor. Weiter geben wir einen Überblick über die bisherige Beweiskette der Korrektheit und der Terminierung der Konsistenzprüfung.

Die Beweislücken aus [Ash16] schließen wir in Kapitel 5.

Schließlich fassen wir die Ergebnisse und Erkenntnisse unserer Arbeit in Kapitel 6 zusammen und geben eine Übersicht, wie zukünftige Arbeiten auf dieser aufbauen können.

1.3 Danksagung

Ich möchte allen Menschen danken, die mir auf dem Weg zu dieser Arbeit zur Seite standen – insbesondere:

- Prof. Dr. Wolfgang Reisig und Prof. Dr. Nicole Schweikardt, die diese Arbeit ermöglichten.
- Kim Völlinger und Samira Akili – Pionierinnen auf dem Gebiet der ZVA. Auf ihren Arbeiten und ihrem fachlichen Rat baut meine Diplomarbeit auf.
- Nina Löwen, die für alle Probleme ein offenes Ohr fand.
- Franziska Boll und Marian Boll, die jede Formulierung abrunden konnten.
- Arite Scholwin-Boll, die jedes, Komma und Apostroph an die richtige Stelle setzte.

In der Arbeit nutze ich durchgängig die „wir“-Form – einerseits, um die Unterstützung dieser Personen zu honorieren – andererseits, da diese Form im Englischen verbreitet ist.

2 Grundlagen

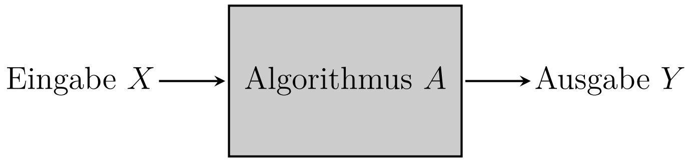
In diesem Kapitel werden die Grundlagen für das Verständnis der späteren Kapitel erläutert. Zuerst führen wir zertifizierende verteilte Algorithmen (ZVA) in Abschnitt 2.1 ein. Wir erklären dafür alle nötigen Eingaben und Ausgaben solcher Algorithmen und wie ein Eingabe/Ausgabe-Paar auf Korrektheit geprüft wird. Völlinger und Akili (siehe [VA18] und [Aki18]) erstellten ein Framework, um zertifizierende verteilte Algorithmen zu verifizieren. Eine alternative Version des Frameworks stellen wir in Abschnitt 2.2 vor. Für alle nötigen Beweisführungen dieser Arbeit nutzen wir den Beweisassistenten Coq. In Abschnitt 2.3 geben wir einen Abriss über Coq und dessen wichtigste Funktionen und Bibliotheken, die für uns relevant sind.

2.1 Zertifizierende verteilte Algorithmen (ZVAs)

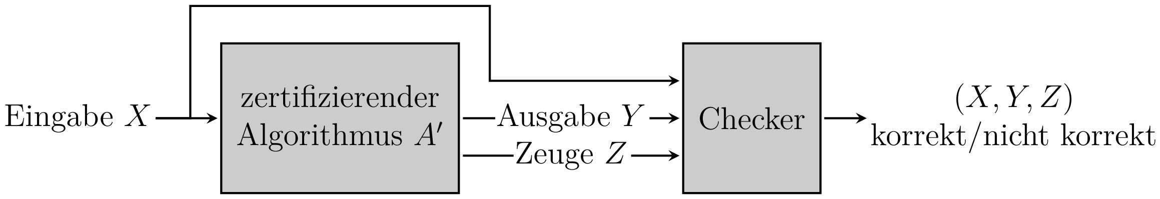
2.1.1 Zertifizierende Algorithmen

Klassische Algorithmen überführen eine Eingabe X nach dem Abbarbeiten von wohldefinierten Schritten in eine Ausgabe Y (vgl. [CLRS09]), siehe Abbildung 1a. Dabei nehmen wir an, dass die Eingaben X die *Vorbedingung* $\varphi(X)$ erfüllen. Die Ausgabe von Algorithmus A soll die *Nachbedingung* $\psi(X, Y)$ erfüllen, falls $\varphi(X)$ gilt. Zusammen ergeben die Vorbedingung und die Nachbedingung eines Algorithmus' eine *Spezifikation*. Wir sprechen dann von einem *korrekten* oder korrekt arbeitenden Algorithmus bezüglich einer Spezifikation, wenn der Algorithmus die Spezifikation erfüllt. Verkürzend sprechen wir auch von einer *korrekten Ausgabe* des Algorithmus' (bezüglich einer impliziten Eingabe).

Die Nutzenden eines Algorithmus' wissen nicht immer, ob ein von ihnen genutzter Algorithmus korrekt ist. Eventuell ist die Arbeitsweise des Algorithmus' für sie völlig unbekannt – z.B. ein Algorithmus eines obfuskierten Programms, dessen Dekompilierung schwierig ist (siehe [LD03] und [AC11]). Das Einzige, was die Nutzenden über den Algorithmus wissen, ist, dass der Algorithmus bezüglich einer Spezifikation arbeiten *soll*. Damit die Überprüfung der Ausgabe eines klassischen Algorithmus A vereinfacht wird, überführen wir A in einen *zertifizierenden Algorithmus* A' (vgl. [MN98] und [MMNS11]), der zusätzlich zur Ausgabe Y auch noch ein mathematisches Artefakt Z ausgibt. Mithilfe von Z soll beweisbar sein, dass das Eingabe/Ausgabe-Paar korrekt ist. Ist das Eingabe/Ausgabe-Paar korrekt und dies ist mit Z beweisbar, sprechen wir von einem *Zeugen* Z (eine formale Definition folgt später) für das Eingabe/Ausgabe-Paar. Die Idee von zertifizierenden Algorithmen ist es, der Berechnung von A' einen *Checker* C nachzustellen, siehe Abbildung 1b. Der Checker ist ein (möglichst einfaches) Programm mit den Eingaben X, Y und Z und Ausgaben „(X, Y, Z) korrekt" und „(X, Y, Z) nicht korrekt." Der Checker prüft das Tripel (X, Y, Z) und akzeptiert, wenn Z beweist, dass Y die korrekte Ausgabe für die Eingabe X ist. Ansonsten lehnt der Checker das Tripel ab. Die Korrektheit des Checkers wird formal verifiziert. Deshalb können die Nutzenden



(a) Das Eingabe/Ausgabe-Verhalten eines klassischen Algorithmus.



(b) Das Eingabe/Ausgabe-Verhalten eines zertifizierenden Algorithmus mit Checker. Der Checker nimmt die Eingabe, Ausgabe und Zeugen von A' und berechnet, ob für die Eingabe X der Zeuge Z die Korrektheit der Ausgabe Y beweist.

Abb. 1: Das Eingabe/Ausgabe-Verhalten von klassischem Algorithmus und zertifizierendem Algorithmus mit Checker (Eigene Darstellung auf Basis von Abbildungen aus [Riz15]).

der Ausgabe Y vertrauen, wenn der Checker das Tripel (X, Y, Z) akzeptiert. Lehnt der Checker (X, Y, Z) ab, ist entweder Y die falsche Ausgabe oder Z kann nicht die Korrektheit von Y bei Eingabe X bezeugen.

Beispiel: Zertifizierender Primzahltest Wir nehmen nun einen zertifizierenden Algorithmus an, der das Problem „Ist die natürliche Zahl X prim?“ zertifizierend löst. Die Vorbedingung ist $\varphi(X) =$ „X ist eine natürliche Zahl“. Die Nachbedingung ist $\psi(X, \text{„ja“}) \Leftrightarrow$ „X ist eine Primzahl“ und $\psi(X, \text{„nein“}) \Leftrightarrow$ „X ist keine Primzahl“. Im folgenden sehen wir zwei Ausgaben des Algorithmus’:

Ausgabe 1: $(X = 213, Y = \text{„nein“}, Z = (3, 71))$

Ausgabe 2: $(X = 211, Y = \text{„ja“}, Z = (2, 3, 5, 7, 11, 13))$

Der Checker C erhält diese Ausgaben als eigene Eingabe. Er überprüft, dass bei der ersten Ausgabe $3 \cdot 71 = 213$ gilt. Bei der zweiten Ausgabe prüft er, dass alle Primzahlen $\leq \sqrt{211}$ aufgelistet wurden und keine davon 211 teilt. In beiden Fällen beweist Z, dass Y die korrekte Ausgabe ist und der Checker akzeptiert die Ausgaben jeweils.

Wir definieren den Begriff *Zeugen* und die damit zusammenhängenden Begriffe *Zeugeneigenschaft* und *Zeugenprädikat* nun formal. Seien Eingaben, Ausgaben und Zeugen jeweils aus der gemeinsamen Menge G.

Definition 1 - Zeugeneigenschaft, Zeugenprädikat und Zeuge.

Ein Prädikat $\Gamma \subseteq G^3$, mit der *Zeugeneigenschaft*

$$\forall(X, Y, Z) \in G^3 \colon \varphi(X) \wedge \Gamma(X, Y, Z) \rightarrow \psi(X, Y) \tag{1}$$

ist ein *Zeugenprädikat* für die Spezifikation (φ, ψ). Für $(X, Y, Z) \in \Gamma$ ist Z ein *Zeuge* für die Korrektheit des Eingabe/Ausgabe-Paares (X, Y) bezüglich der Spezifikation (φ, ψ).

Beispiel: Zeugenprädikat und Zeuge für den Primzahltest (s.o.) Das Zeugenprädikat Γ setzt sich zusammen aus den beiden Prädikaten $\Gamma_{\text{„ja“}}$ und $\Gamma_{\text{„nein“}}$ für die beiden Fälle „ist eine Primzahl“ und „ist keine Primzahl.“ Das Prädikat $\Gamma_{\text{„nein“}}$ lässt sich folgendermaßen beschreiben:

$$\begin{aligned}\Gamma_{\text{„nein“}} = &\{(0, \text{„nein“}, 0)\} \cup \{(1, \text{„nein“}, 1)\} \cup \\ &\{(X, \text{„nein“}, (Z_1, Z_2)) \mid (X, Z_1, Z_2) \in \mathbb{N}^3 \wedge X \text{ ist nicht prim} \wedge Z_1 \cdot Z_2 = X \wedge \\ &(1 \neq Z_1 \neq X)\}\end{aligned}$$

Die Zahlen 0 und 1 werden zuerst als gesonderte Spezialfälle behandelt, die keinen wirklichen Zeugen benötigen. Für alle anderen Zahlen, die nicht prim sind, gibt es ein Paar (Z_1, Z_2), dessen Produkt die zusammen gesetzte Zahl X ergibt. Z_1 (und implizit auch Z_2) dürfen dabei weder 1 noch X selbst sein. Wenn diese Voraussetzungen erfüllt sind, ist das Tripel $(X, \text{„nein“}(Z_1, Z_2))$ ein Element von $\Gamma_{\text{„nein“}}$. Bei Ausgabe 1 aus dem obigen Beispiel ist $(X = 213, Y = \text{„nein“}, Z = (3, 71)) \in \Gamma_{\text{„nein“}}$ und $(3, 71)$ damit ein Zeuge für $(X = 213, Y = \text{„nein“})$.

Alkassar et al. entwickelten in [ABMR14] eine Methode, um die Korrektheit eines Eingabe/Ausgabe-Paares zu beweisen: Für eine gegebene Spezifikation und einen zertifizierenden Algorithmus wird maschinell die Zeugeneigenschaft bewiesen und zusätzlich verifiziert, dass der Checker Γ korrekt entscheidet. Dann gilt für jedes vom Checker akzeptierte Tripel (X, Y, Z) die *formale Instanzkorrektheit* für das Eingabe/Ausgabe-Paar (X, Y) – das heißt, es ist maschinell verifiziert, dass Y die korrekte Ausgabe für die Eingabe X ist. Formale Instanzkorrektheit zeigt zur Laufzeit, dass ein gegebenes Eingabe/Ausgabe-Paar korrekt ist. Programmverifikation hingegen zeigt unabhängig von der Ausführung des Algorithmus, dass *alle* Eingabe/Ausgabe-Paare korrekt sind. Die Aussagen von formaler Instanzkorrektheit sind also spezifischer und je nach Problemstellung einfacher zu zeigen.

Da nur der Checker verifiziert und die Zeugeneigenschaft bewiesen wird, kann der Algorithmus durchaus inkorrekt sein. In der Tat wird der Algorithmus A' nur als Black Box betrachtet, der Ausgaben und Zeugen bestimmter Form produziert. A' kann also durch einen Algorithmus B' der gleichen Form, aber anderen Arbeitsweise oder mit anderen Fehlern, ersetzt werden und es besteht weiter die Gewissheit von formaler Instanzkorrektheit, wenn der Checker akzeptiert.

Beispiel: Verifikation des zertifizierenden Primzahltest (s.o.) Welche Beweise müssen wir führen, um zu verifizieren, dass der Checker C im „nein“-Fall nur korrekte Ausgaben $(X, \text{„nein“}, Z)$ des Algorithmus' akzeptiert? Zuerst beweisen wir maschinell die Zeugeneigenschaft, dass eine Primzahl nie das Produkt zweier natürlicher Zahlen ist, die nicht 1 oder sie selbst sind. Weiterhin beweisen wir, dass der Checker im „nein“-Fall nur akzeptiert, wenn $Z = (Z_1, Z_2) \in \mathbb{N}^2$ und $Z_1 \cdot Z_2 = X$, sowie $1 \neq Z_1 \neq X$ gilt. Nachdem diese Beweise getätigt wurden, können wir den Ausgaben des zertifizierenden Algorithmus mit Checker im „nein“-Fall vertrauen.
An dem Beispiel der Primzahlprüfung werden sowohl Vorteile als auch Nachteile des Prinzips der Zertifizierung sichtbar: Im „nein“-Fall ist der Zeuge stets klein und schnell prüfbar und damit die Ausgabe verifiziert. Im „ja“-Fall wächst der Zeuge und der Prüfaufwand mit der Eingabe mit (ähnliche Fälle sind in [HK07] zu finden). Wenn der zeitliche Prüfaufwand zu sehr wächst, lohnt sich der Einsatz eines Checkers nicht. Gleichfalls ist ein Checker mit zu kompliziertem Prüfaufwand eventuell schwierig zu verifizieren.
Für jeden Algorithmus gibt es einen Zeugen: das Programm kann sich selbst und jeden eigenen Rechenschritt als Zeugen ausgeben. Wenn wir nun den Checker verifizieren, verifizieren wir das Programm gleich mit. Dadurch ist die Nutzung solcher Zeugen nicht sinnvoll, weil wir stattdessen das Programm selbst verifizieren könnten. Aus diesem Grund probieren wir geschickte Zeugen für ein Problem zu finden, die möglichst schnell und einfach prüfbar sind.

2.1.2 Verteilte Algorithmen

Ein *verteilter Algorithmus* ist ein Algorithmus, der auf mehreren verbundenen Komponenten ausgeführt wird. Die Komponenten können unabhängig voneinander rechnen. Jede Komponente bekommt dafür ihren eigenen Teil-Algorithmus. Damit trotz des verteilten Rechnens eine Zusammenarbeit der Komponenten möglich ist, können die Komponenten mit ihren Nachbarn Nachrichten teilen. Empfangene Nachrichten wiederum können den Arbeitsfluss einer Komponente beeinflussen. In dieser Arbeit betrachten wir nur verteilte Komponenten, die über bidirektionale Kanäle miteinander Nachrichten austauschen. Den Verbund unserer Komponenten bezeichnen wir als *Netzwerk*. Wir beschränken uns auf statische Netzwerke, bei denen weder Komponenten noch Kanäle des Netzwerks hinzukommen, ausgetauscht oder entfernt werden. Zusätzlich nehmen wir an, dass jede Komponente seinen eigenen Identifikator (*ID*) und die Identifikatoren aller ihrer Nachbarn im Netzwerk kennt. Schließlich betrachten wir hier *asynchrone* Netzwerke, d.h. Nachrichten kommen nicht nach einer fest definierten Zeit an. Dadurch kann die Reihenfolge von Nachrichten nicht-deterministisch vertauscht werden.
Wir unterscheiden die *Eingabe* eines verteilten Algorithmus von den *Teil-Eingaben*.

Jeder Komponente wird eine eigene Teil-Eingabe zugeordnet – für das gesamte Netzwerk gibt es die Eingabe des verteilten Algorithmus'. Nach der Terminierung der Berechnung der Komponente gibt die Komponente entsprechend eine *Teil-Ausgabe* aus. Als Eingabe des gesamten Netzwerks verstehen wir entweder die Gesamtheit aller Teil-Eingaben oder z.B. eine Eingabe an die Wurzel, die von dort aus an alle Komponenten als deren Teil-Eingaben verteilt wird.

Ein terminierender verteilter Algorithmus besteht häufig aus den folgenden drei Schritten:

Schritt 1: Verteilung: Die Eingabe X des Algorithmus' wird an die Komponenten verteilt. Jede Komponente v bekommt eine Teil-Eingabe x_v.

Schritt 2: Eigentlicher Algorithmus: Die Komponenten berechnen verteilt und können dabei miteinander kommunizieren. Jede Komponente v berechnet aus ihrer Teil-Eingabe zusammen mit allen erhaltenen Nachrichten eine Teil-Ausgabe y_v.

Schritt 3: Konsensbildung: Die Teil-Ausgaben werden zu einer gemeinsamen Ausgabe Y des verteilten Algorithmus' zusammengeführt.

Schritt eins und drei können je nach verteiltem Algorithmus entfallen. Der erste Schritt wird von uns nicht näher betrachtet. Wir nehmen eine fertige Verteilung der Eingabe an die Komponenten an: jede Komponente hat zu Beginn ihre Teil-Eingabe. Dazu mehr auf Seite 16.

Häufig haben Netzwerke eine besondere Komponente: einen *Leader*. Die Leader-Komponente kann zum Beispiel die Kommunikation des Netzwerks regulieren. Dafür wird die Leader-Komponente als Wurzel eines Spannbaums des Netzwerks gesehen. Die Kommunikation findet z.B. per Broadcast ausgehend von der Wurzel entlang der Kanten des Spannbaums statt. Die Komponenten können sich, für Schritt drei, per Convergecast Richtung Wurzel auf ein Ergebnis einigen. Die Wurzel, also die Leader-Komponente, kann dann die Ausgabe des verteilten Algorithmus' erzeugen.

Um Netzwerke zu modellieren, nutzen wir zusammenhängende ungerichtete Graphen. Jedem Knoten des Graphen entspricht eine Komponente des Netzwerks. Jedem Kommunikations-Kanal entspricht eine Kante des Graphen. Die Knotenmenge bezeichnen wir im weiteren als V (engl. **v**ertices) und einen einzelnen Knoten daraus als v.

2.1.3 Eine Klasse zertifizierender verteilter Algorithmen

Zertifizierende verteilte Algorithmen (ZVA) kombinieren die Ideen von zertifizierenden Algorithmen und verteilten Algorithmen. Das Konzept der ZVA wurde von Völlinger et al. in [VR15], [Völ17], [VA17] und [VA18] eingeführt. Ein ZVA ist ein terminierender Algorithmus, bei der zu jeder verteilten Eingabe eine verteilte Ausgabe sowie ein verteilter Zeuge berechnet wird. Danach kann verteilt geprüft werden, ob der verteilte

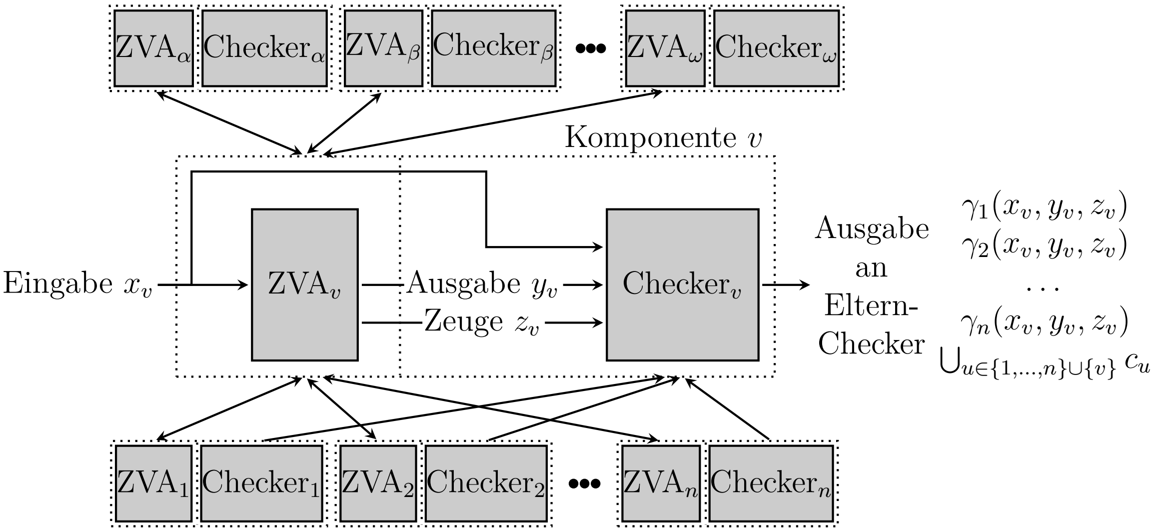


Abb. 2: Das Eingabe/Ausgabe-Verhalten innerhalb einer Komponente v eines ZVA, sowie die Kommunikation mit Nachbarkomponenten von v. (Eigene Darstellung, vgl. Abbildung aus [VA18])

Zeuge die Korrektheit des Eingabe/Ausgabe-Paares beweist.

Für die verteilte Prüfung wird auf jeder Komponente nach der Berechnung des Teil-Algorithmus ein Teil-Checker gestartet. Die Teil-Checker entscheiden zusammen über die Korrektheit der Ausgabe. Damit ist der Checker selbst ein verteilter Algorithmus. Genau wie bei zertifizierenden Algorithmen (siehe Abschnitt 2.1.1) soll auch bei ZVAs eine positive Prüfung des Checkers Instanzkorrektheit implizieren. Dafür wenden wir wieder maschinelles Theorembeweisen an, um die Zeugeneigenschaft zu beweisen und den Checker zu verifizieren.

In Abbildung 2 ist das Eingabe/Ausgabe-Verhalten einer Komponente v eines ZVA zu sehen. Die Nachbarkomponenten von v sind jeweils oben und unten kleiner eingezeichnet. Auf Komponente v rechnet zuerst der Teil-Algorithmus ZVA$_v$ und gibt seine Ausgabe an Checker$_v$. ZVA$_v$ kann dabei mit allen Nachbarkomponenten bidirektional kommunizieren. Checker$_v$ erhält die Ausgabe von ZVA$_v$ und zusätzlich die Ausgaben der Kinderkomponenten eines Spannbaums des Netzwerks (in Abbildung 2 unten eingezeichnet). Die Ausgabe von Checker$_v$ wird an den Checker der Elternkomponente von v übergeben. Somit fließen die Ausgaben der Teil-Checker in Richtung der Wurzel des Spannbaums. Darauf wird auf Seite 16 und in Kapitel 4 näher eingegangen.

Eingabe, Ausgabe und Zeuge Jede Komponente erhält eine Teil-Eingabe und berechnet daraus (eventuell mit zusätzlicher Kommunikation mit ihren Nachbarn) ihre Teil-Ausgabe und ihren Teil-Zeugen. Alle Teil-Zeugen zusammen bilden den verteilten Zeugen. Formal sind die Teil-Eingaben eine Menge von Variablen mit zugehörigen Werten. Sei eine Menge g von Variablen Var und eine Menge von Werten Val, dann ist für eine Komponente v ihre Teil-Eingabe $x_v \in g \subseteq (Var \times Val)$. Teil-Ausgaben y_v und

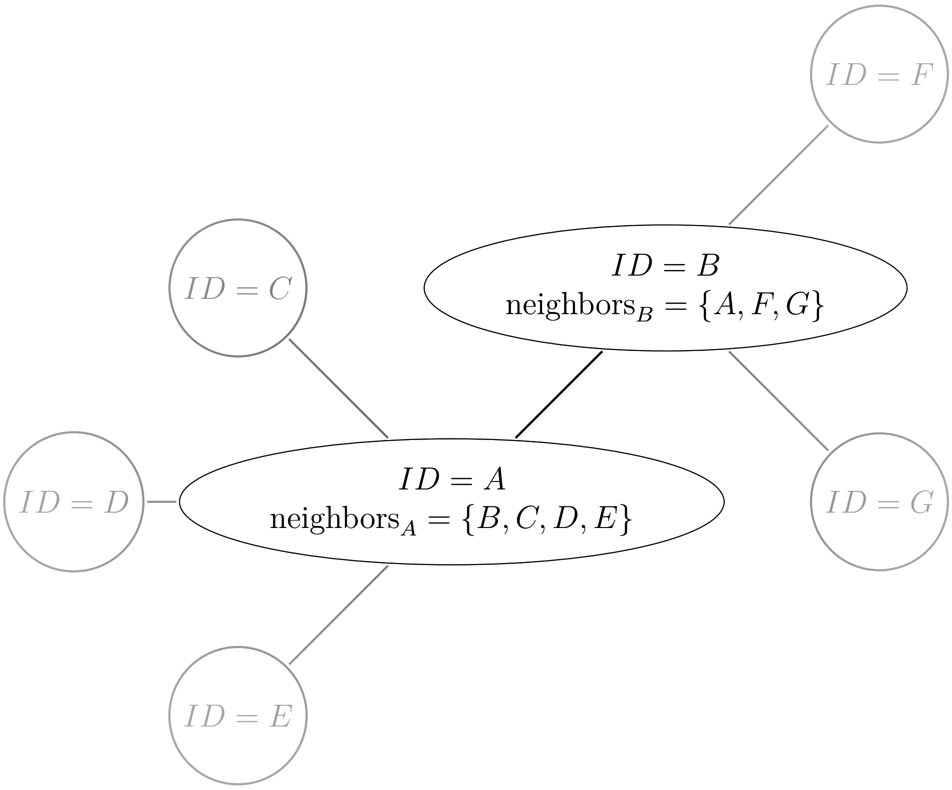


Abb. 3: Minimaleingabe für zwei Komponenten eines Netzwerkes. (Eigene Darstellung)

Teil-Zeugen z_v werden ebenso definiert. Teil-Eingaben, Teil-Ausgaben und Teil-Zeugen zusammen bezeichnen wir als *Kombination* $c_v = x_v \cup y_v \cup z_v$, wie in Abbildung 2 unten genutzt. Dort schicken alle Kinder im Spannbaum ihre Belegungen nach oben zu Checker_v. Checker_v kombiniert seine eigene Belegung mit den Erhaltenen und schickt diese Belegung weiter an den eigenen Eltern-Checker. Das Prinzip wird in Kapitel 4 im Detail erläutert.
Jede Komponente bekommt als Minimaleingabe ihre ID und die ID aller ihrer Nachbarn. Für zwei Komponenten A und B sind die Eingaben in Abbildung 3 eingezeichnet.

Verteilung des Zeugenprädikats Damit der Checker den Zeugen verteilt prüfen kann, führen wir *Teil-Zeugenprädikate* ein. Jeder Teil-Checker prüft sein Teil-Zeugenprädikat. Nachdem die Teil-Prüfungen abgeschlossen sind, einigen sich die Teil-Checker auf ein gemeinsames Ergebnis der Prüfung des Zeugenprädikats. Ein Zeugenprädikat kann universell oder existentiell auf die Teil-Checker verteilt werden:

Definition 2 - Verteilbares Zeugenprädikat.

Ein Zeugenprädikat $\Gamma \subseteq G^3$ ist verteilbar, wenn für alle Eingaben $(X, Y, Z) \in G^3$ gilt:

- Γ ist universell verteilbar, wenn ein Teil-Prädikat $\gamma \subseteq g^3$ existiert, sodass:

  $$\forall v \in V, (x_v, y_v, z_v) \in g^3 \colon \gamma(x_v, y_v, z_v) \rightarrow \Gamma(X, Y, Z). \tag{2}$$

- Γ ist existentiell verteilbar, wenn ein Teil-Prädikat $\gamma \subseteq g^3$ existiert, sodass:

  $$\exists v \in V, (x_v, y_v, z_v) \in g^3 \colon \gamma(x_v, y_v, z_v) \rightarrow \Gamma(X, Y, Z). \tag{3}$$

  Im Unterschied zur universellen Verteilbarkeit reicht es hier also, wenn für mindestens eine Komponente ihr Teil-Prädikat erfüllt ist.

- Es existieren die Prädikate Γ_1 und Γ_2, mit:

  $$(\Gamma_1(X, Y, Z) \land \Gamma_2(X, Y, Z)) \rightarrow \Gamma(X, Y, Z). \tag{4}$$

- Es existieren die Prädikate Γ_1 und Γ_2, mit:

  $$(\Gamma_1(X, Y, Z) \lor \Gamma_2(X, Y, Z)) \rightarrow \Gamma(X, Y, Z). \tag{5}$$

- Es existiert ein Prädikat Γ_1, mit:

  $$(\neg\Gamma_1(X, Y, Z)) \rightarrow \Gamma(X, Y, Z). \tag{6}$$

Die Prädikate Γ_1 und Γ_2 können wiederum selbst entsprechend der Definition 2 auf Unter-Prädikate verteilbar sein. Durch die induktive Definition ist ein komplexer Aufbau der Verteilung eines Zeugenprädikats möglich.
Beispiel: Seien Γ_1 ein universelles und Γ_2 ein existentielles Prädikat und gelte für ein Zeugenprädikat Γ: $\Gamma_1(X, Y, Z) \land \Gamma_2(X, Y, Z) \rightarrow \Gamma(X, Y, Z)$. Wenn alle Teil-Checker das Teil-Prädikat von Γ_1 akzeptieren und mindestens ein Teil-Checker das Teil-Prädikat von Γ_2, dann kann der Checker das Prädikat $\Gamma(X, Y, Z)$ akzeptieren.

Prüfung des Checkers und der Konsistenz Der Checker prüft das Zeugenprädikat durch Teil-Checker in jeder Komponente. Nachdem der Teil-Algorithmus seine Teil-Ausgabe produziert hat, kann der Teil-Checker sein Teil-Zeugenprädikat prüfen. Der Teil-Checker muss also keine Verteilung der Ausgabe abwarten. Daher betrachten wir nur Schritt zwei und drei von Seite 13. In Schritt drei schicken die Teil-Checker in einem Convergecast ihre Teil-Ausgaben Richtung Leader-Komponente, siehe Abbildung 2. Die Leader-Komponente wertet dann je nach Verteilung der Zeugenprädikate die Teil-Zeugenprädikate aus. Sei z.B. $\Gamma = \Gamma1 \lor \Gamma2$ mit $\Gamma1$ universell und $\Gamma2$ existentiell verteilt. Dann kann die Leader-Komponente den Zeugen akzeptieren, wenn alle Komponenten Teil-Prädikat $\Gamma1$ oder ein Teil-Prädikat von $\Gamma2$ akzeptiert wurde.

Der Checker muss noch ein weiteres Problem lösen: damit die Teil-Zeugen keine Widersprüche enthalten, müssen die Teil-Checker noch die *Konsistenz* aller Teil-Zeugen prüfen.

Definition 3 - Konsistenz des Zeugen.

Der Zeuge ist konsistent, wenn alle Teil-Zeugen konsistent belegt sind:

$$\forall u, v \in V : \forall (var, val_1) \in z_u : \forall (var, val_2) \in c_v : var \neq ID \rightarrow val_1 = val_2. \tag{7}$$

Trifft ein Teil-Zeuge Aussagen über Variablen, die in (anderen) Teil-Eingaben, -Ausgaben oder -Zeugen genutzt werden, so müssen diese konsistent belegt werden. Eine Folge unserer Definition ist, dass eine unterschiedliche Mehrfachbelegung einer Variable, innerhalb einer Komponente, ausgeschlossen ist. Als Ausnahme dürfen die Identitäten der Komponenten unterschiedlich sein, da jeder Komponente eine einzigartige ID zugeordnet ist.

Diese Definition der Konsistenz ist wesentlich umfassender, im Vergleich zur Definition der Konsistenz in [VA18], weil bei uns auch die Teil-Eingaben und Teil-Ausgaben des ZVA auf Konsistenz geprüft werden. Wir stellen diese umfassendere Forderung an die Konsistenz, um mehrere Beweisverpflichtungen des Frameworks aus Abschnitt 2.2.1 entfernen zu können.

Nachdem der Checker festgestellt hat, dass der Zeuge konsistent ist und die richtigen Teil-Zeugenprädikate, entsprechend der Verteilung, erfüllt sind, kann er den Zeugen akzeptieren. Ansonsten verwirft er ihn. Wie Teil-Checker verteilt die Konsistenz prüfen können, beschreiben wir in Kapitel 4.

Beispiel: In Abbildung 4 sind nicht konsistente Zeugen in den Komponenten A und B vorhanden. Die Variablen n_A und n_B geben die Anzahl der Nachbarn der Komponente A respektive B an. Im Zeugen von Komponente A ist jedoch eine falsche Nachbaranzahl für die Komponente B eingetragen. Eigentlich hat B nur 3 Nachbarn. Wenn die Nachbaranzahl für die Beweisführung des Checkers nötig ist, so können die Teil-Checker mit diesen inkonsistenten Teil-Zeugen keinen Beweis führen. Eine Instanzverifikation ist ohne geprüfte Zeugen also unmöglich.

2.2 Framework zur Verifikation zertifizierender verteilter Algorithmen

Völlinger und Akili stellen in [VA18] bzw. [Aki18] ein Framework zur formalen Instanz-Verifikation von ZVAs vor. Unser darauf aufbauendes Framework beschreiben wir in Abschnitt 2.2.1. Wenn alle Bausteine unseres Frameworks zusammengeführt sind, so kann man damit formal Instanzen verifizieren, siehe Abschnitt 2.1.1. Auf die Unter-

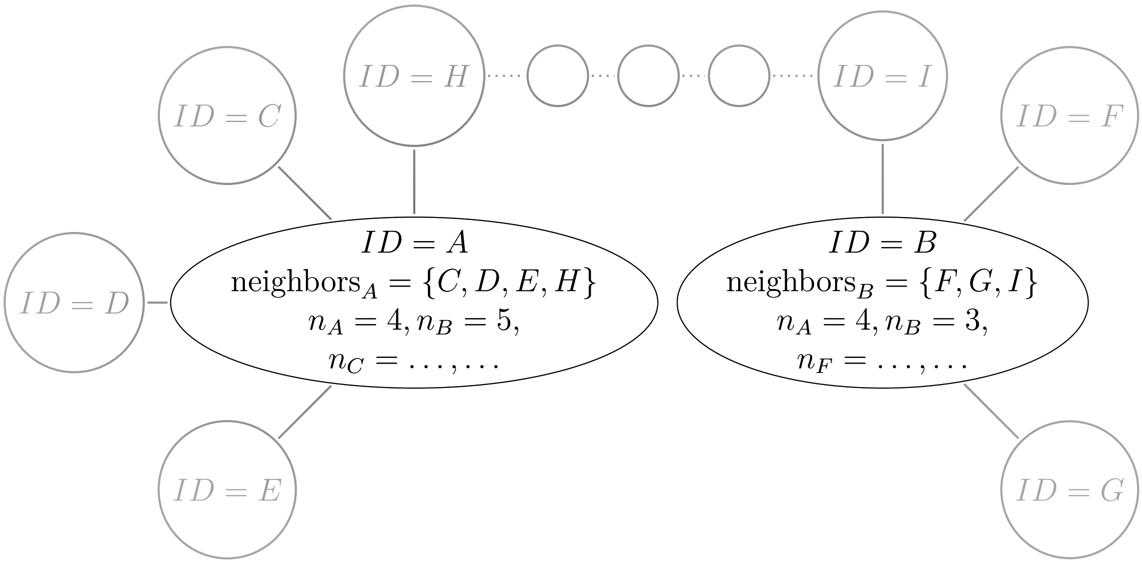


Abb. 4: Nicht konsistente Zeugen in den Komponenten A und B: B hat nur drei Nachbarn. In A wird jedoch eine Nachbaranzahl von 5 für B behauptet. (Eigene Darstellung)

schiede zwischen unserer Variante des Frameworks und der aus [Aki18] gehen wir in Abschnitt 2.2.2 ein.
Wir übernehmen die Nummerierung der Beweisverpflichtungen (BV) aus [Aki18] für eine bessere Vergleichbarkeit.

2.2.1 Übersicht über das Framework

In Abbildung 5 ist unser Framework dargestellt. Auf der linken Seite stehen alle Modellierungsaufgaben, beziehungsweise Beweisverpflichtungen in Coq (in der Abbildung mit BV abgekürzt). Auf der rechten Seite sind die extrahierten Anteile und des Checkers in OCaml zu sehen. Jede Box entspricht dabei entweder einer Modellierungsaufgabe, einer Beweisverpflichtung oder einem extrahierten Programm. Modellierungsaufgaben sind Coq-Definitionen des Modells. Beweisverpflichtungen sind Beweise, die Eigenschaften des Modells, des Zeugen oder des Checkers beweisen. Manche Beweisverpflichtungen sind nur einmalig auszuführen, andere müssen für jeden ZVA einzeln gezeigt werden. Letztere sind mit einem Pfeil von links markiert. Wenn alle notwendigen Aufgaben erfüllt sind, können wir den OCaml-Code entsprechend Abbildung 1b an den ZVA anschließen. Wenn wir den OCaml-Code des Checkers ausführen, gelten die Garantien der Beweise aller Beweisverpflichtungen für den Code. Insbesondere die formale Instanzkorrektheit bleibt durch die Extraktion gewährt und wir können den Ausgaben des ZVAs vertrauen.[1]

---

[1] Wir nehmen hier an, dass die Extraktion nach OCaml fehlerfrei abläuft. Derzeit gibt es noch keine vollständig verifizierte Extraktion.

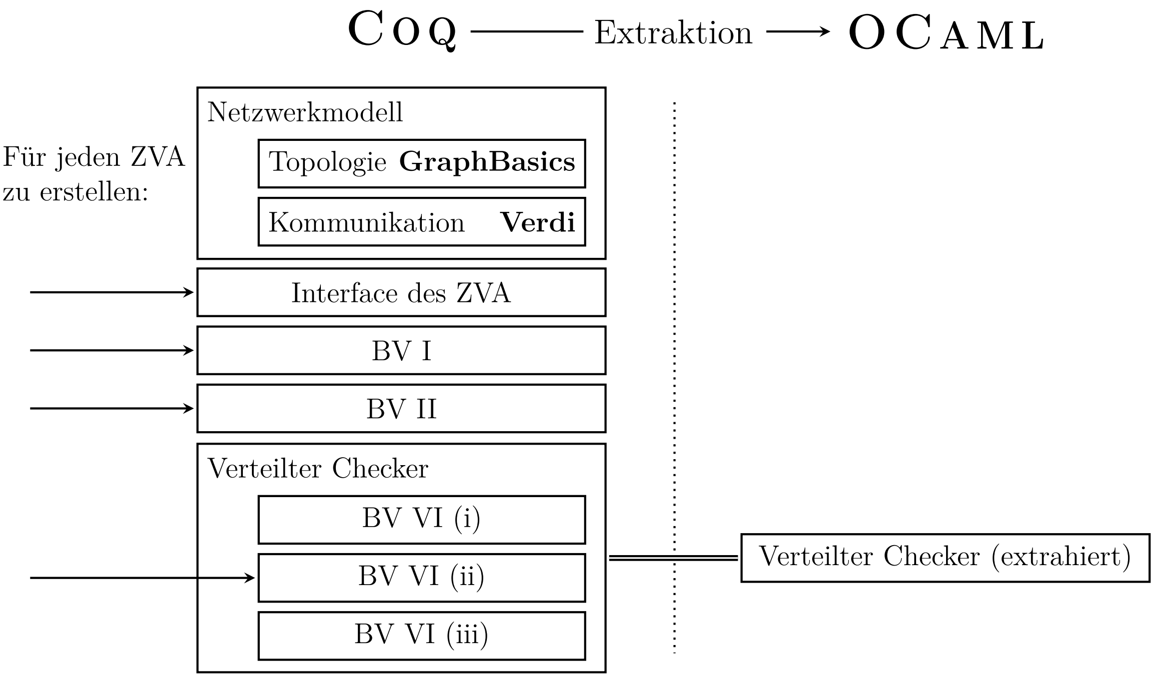


Abb. 5: Unser Framework für formale Instanzkorrektheit für verteilte Algorithmen. (Eigene Darstellung, vgl. Abbildung aus [Aki18], insbesondere die Bezeichnungen der Beweisverpflichtungen sind übernommen.)

Wir beschreiben nun die nötigen Modellschritte und Beweisverpflichtungen des Frameworks:

Netzwerkmodell Um Beweise zum Netzwerk und einem verteilten Algorithmus zu führen, brauchen wir ein Modell. Dafür modellieren wir die Topologie und Kommunikation des Netzwerks in Coq. Da wir jedes Netzwerk so modellieren, brauchen wir diesen Schritt nur einmal für alle ZVAs machen.

Topologie Wir modellieren die Struktur des Netzwerks als zusammenhängenden Graphen mit der Coq-Bibliothek GraphBasics, siehe Abschnitt 2.3.3.

Kommunikation Die Kommunikation der Netzwerk-Komponenten modellieren wir mit der Coq-Bibliothek Verdi, siehe Abschnitt 2.3.4.

Interface des ZVA Hier modellieren wir die Form der Eingabe, der Ausgabe, des Zeugen, sowie der Vorbedingung und Nachbedingung des ZVA. Weiterhin modellieren wir das Zeugenprädikat Γ. Da jeder ZVA eine andere Eingabe, Ausgabe etc. hat, muss dieser Schritt für jeden ZVA erfolgen.

BV I Wir beweisen, dass Γ die Zeugeneigenschaft hat. Dies muss für jeden ZVA eigenständig durchgeführt werden, da die Zeugeneigenschaft sehr individuell von der Form des Zeugen abhängt. Dadurch verändern sich die Beweise je nach Problemstellung beziehungsweise Zeugenform so sehr, dass kein allgemeiner

Beweis für die Zeugeneigenschaft möglich ist.

BV II Wir beweisen, dass Γ verteilbar ist, siehe Seite 15. Ähnlich wie bei dem Beweis der Zeugeneigenschaft, ist hier kein allgemeiner Beweis möglich und BV II muss für jeden ZVA einzeln bewiesen werden.

Verteilter Checker Wir implementieren den verteilten Checker mit der COQ-Bibliothek VERDI. Der Checker kann dann nach OCAML extrahiert werden, damit er ausgeführt werden kann. Mit den folgenden Beweisverpflichtungen zeigen wir die Korrektheit der Implementierung.

BV VI (i) Beweis für die verteilte Prüfung der Konsistenz durch den verteilten Checker. Weitere Ausführungen dazu folgen in Kapitel 4.

BV VI (ii) Beweis dafür, dass jeder Teil-Checker seine Teil-Prädikate korrekt entscheidet.

BV VI (iii) Beweis für die korrekte Auswertung der Teil-Prädikate, entsprechend der Verteilung des Zeugenprädikats.

2.2.2 Vergleich unseres Frameworks mit dem Framework aus [Aki18]

In der Arbeit [Aki18] erstellte Akili ein Framework mit einer anderen Konsistenzprüfung als der unsrigen. Die vier Beweisverpflichtungen (BV III, BV IV, BV V und BV VI (i)) aus ihrer Arbeit werden bei uns alle in einer einzigen neuen Konsistenzprüfung (BV VI (i)) gebündelt.
Unsere Definition der Konsistenz umfasst die Definition der Vollständigkeit aus [Aki18] – dadurch entfällt bei uns die Prüfung der Vollständigkeit (BV V). In [Aki18] wird die Konsistenz in mehreren Teilen geprüft: einerseits die Konsistenz von Nachbarn im Graphen und andererseits der Zusammenhang von Zeugen (BV VI (i) und BV IV). Dafür benötigte das Framework aus [Aki18] einen Beweis, dass dieses zweigeteilte Verfahren korrekt die Konsistenz prüft (BV III, genannt Theorem I). Unser Ansatz hingegen prüft in einem Verfahren definitionsnah die Konsistenz aller Belegungen des Graphen (siehe Abschnitt 4). Aus diesem Grund entfallen bei uns auch die Beweisverpflichtungen des Zusammenhangs des Zeugen und des Theorems I. Da bei unserem Ansatz weder die Vollständigkeit, noch der Zusammenhang geprüft werden, unterliegt der Zeuge bei uns auch diesen beiden Einschränkungen nicht.
Wir schlagen in dieser Arbeit also ein Framework vor, welches anstatt sechs noch vier Beweisverpflichtungen für jeden ZVA einzeln hat. Dies sehen wir als Vorteil an, da die Entwickelnden des ZVA weniger Eigenschaften des ZVA beweisen müssen. Der ZVA kann dadurch mehr als Black Box behandelt werden.

Während der Konsistenzprüfung werden in unserem Verfahren die Variablen-Belegungen des gesamten Netzwerks mit der Wurzel geteilt. Im Ansatz von [Aki18] hingegen werden die Belegungen nur mit den Nachbar-Komponenten geteilt. Die Entwickelnden eines ZVA müssen entscheiden, ob diese Lokalität der Konsistenz-Prüfung für sie relevant ist. Ist dies der Fall, empfehlen wir den Ansatz von Akili.

2.3 Kurze Einführung für den Beweisassistenten Coq

In diesem Abschnitt stellen wir den Beweisassistenten Coq und zwei von uns genutzte Bibliotheken vor und begründen kurz die Wahl von Coq für diese Arbeit. Online sind viele Einführungen zu Coq zu finden, wir empfehlen insbesondere das offizielle Tutorial für Coq: [HKPM16], sowie das sehr ausführliche Online-Lehrbuch [PAC+18]. In letzterem sind viele praktische Übungen enthalten. Schließlich sind Details zu den benutzten Sprachen und dem Umgang mit Coq in dem offiziellen Manual [Tea18] nachschlagbar und viele weitere Dokumentationen in [Coq18] aufgelistet.

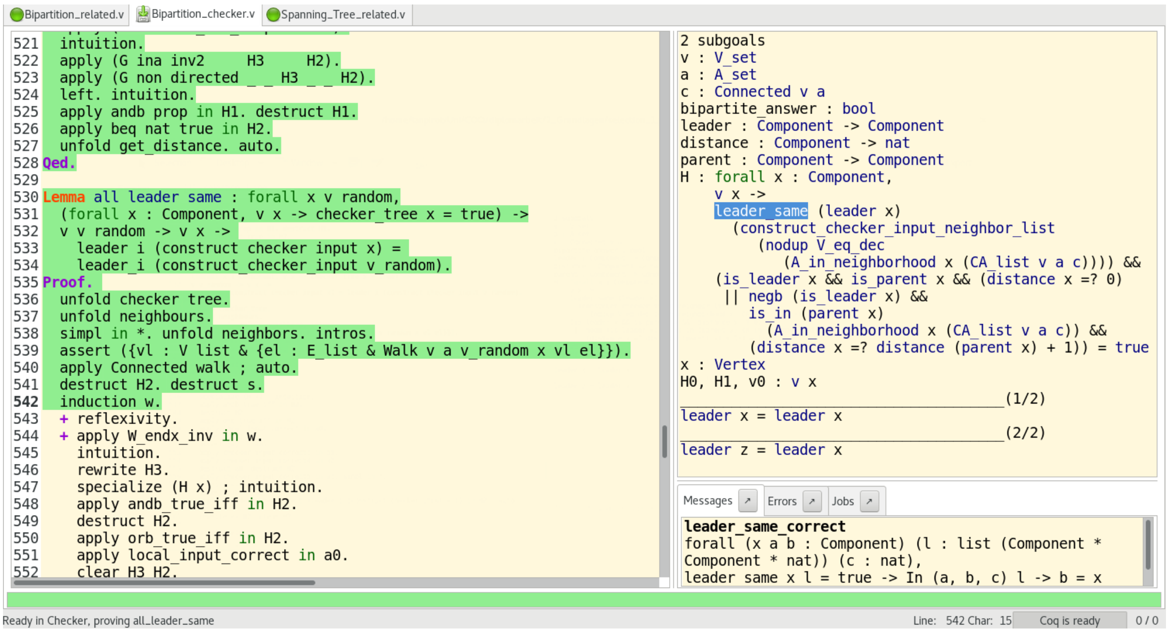
Im Folgenden legen wir dar, warum wir Coq als Beweisassistenten für diese Arbeit wählen: Das Framework, welches wir in dieser Arbeit anpassen (siehe Abschnitt 2.2.1), wurde bisher mit Coq umgesetzt. In dieser Arbeit liefern wir maschinen-geprüfte Beweise für die Zeugeneigenschaft und die Checker-Korrektheit für Zweifärbbarkeit in Kapitel 3. Damit die Beweise aus Kapitel 3 interfacelos in das Framework passen, bietet sich Coq an. Auch das bisher offene Problem aus [Ash16], welches wir in Kapitel 5 beweisen, nutzt Coq.

Mit Coq lassen sich die für diese Arbeit relevanten Theoreme und nötigen Begriffe formalisieren und maschinen-geprüft beweisen. Des weiteren können wir in Coq programmieren und Beweise zu den Eigenschaften der Programme erstellen. Schließlich können wir Programme in andere Sprachen, wie z.B. OCaml oder Haskell extrahieren, um sie auszuführen. Um Programme zu schreiben, verifizieren und lauffähig zu extrahieren, benötigen wir damit nur ein einziges Werkzeug, dem wir vertrauen müssen. Wir können den Beweisen von Coq vertrauen, da Coq das de Bruijn-Kriterium erfüllt. Das de Bruijn-Kriterium besagt, dass Beweise in einer Kernel-Sprache geprüft werden, die extrem klein und damit für Menschen nachprüfbar ist (siehe [Wie03]).

Coq wurde vielseitig eingesetzt. Einige erfolgreiche Projekte sind z.B. die Erstellung eines formal verifizierten C-Compilers (siehe [Ler18]), eines formal verifizierten statischen Analyse-Tools (siehe [Jou16]), das Vier-Farben-Theorem (siehe [Gon08]), die Verifizierung einer großen Haskell-Bibliothek (siehe [BSZL+18]) und ein formal verifizierter Web-Browser (siehe [JTL12]).

2.3.1 Schematischer Arbeitsablauf mit Coq

Für eine beispielhafte Ansicht der CoqIDE enthält Abbildung 6 ein Bildschirmfoto. Das Arbeitsfenster der CoqIDE ist dreigeteilt. Im linken Fenster geben wir die Definitionen, Theoreme und Beweisschritte ein. Der letzte korrekte und berechnete Schritt ist grün markiert. Im oberen rechten Fenster wird der zum letzten berechneten



```
521   intuition.
522   apply (G ina inv2      H3      H2).
523   apply (G non directed _ _ H3 _ _ H2).
524   left. intuition.
525   apply andb prop in H1. destruct H1.
526   apply beq nat true in H2.
527   unfold get_distance. auto.
528 Qed.
529
530 Lemma all leader same : forall x v random,
531   (forall x : Component, v x -> checker_tree x = true) ->
532   v v random -> v x ->
533     leader i (construct checker input x) =
534     leader_i (construct_checker_input v_random).
535 Proof.
536   unfold checker tree.
537   unfold neighbours.
538   simpl in *. unfold neighbors. intros.
539   assert ({vl : V list & {el : E_list & Walk v a v_random x vl el}}).
540   apply Connected walk ; auto.
541   destruct H2. destruct s.
542   induction w.
543   + reflexivity.
544   + apply W_endx_inv in w.
545     intuition.
546     rewrite H3.
547     specialize (H x) ; intuition.
548     apply andb_true_iff in H2.
549     destruct H2.
550     apply orb_true_iff in H2.
551     apply local_input_correct in a0.
552     clear H3 H2.
```

```
2 subgoals
v : V_set
a : A_set
c : Connected v a
bipartite_answer : bool
leader : Component -> Component
distance : Component -> nat
parent : Component -> Component
H : forall x : Component,
    v x ->
    leader_same (leader x)
      (construct_checker_input_neighbor_list
         (nodup V_eq_dec
            (A_in_neighborhood x (CA_list v a c)))) &&
    (is_leader x && is_parent x && (distance x =? 0)
     || negb (is_leader x) &&
        is_in (parent x)
          (A_in_neighborhood x (CA_list v a c)) &&
        (distance x =? distance (parent x) + 1)) = true
x : Vertex
H0, H1, v0 : v x
____________________________________(1/2)
leader x = leader x
____________________________________(2/2)
leader z = leader x
```

```
leader_same_correct
forall (x a b : Component) (l : list (Component *
Component * nat)) (c : nat),
leader same x l = true -> In (a, b, c) l -> b = x
```

Abb. 6: Ein Bildschirmfoto der CoqIDE. (Eigene Darstellung)

Schritt passende *Beweiskontext* und die *Beweisziele* angezeigt (diese werden weiter unten beschrieben). Oberhalb der beiden horizontalen Linien ist der Beweiskontext. Die Beweisziele stehen unter den horizontalen Linien; in Abbildung 6 zwei Beweisziele. Im unteren rechten Fenster werden Fehlermeldungen oder sonstige Zusatzinformationen ausgegeben; in Abbildung 6 wird im unteren rechten Fenster das Interface eines Lemmas angezeigt.

Wir arbeiten mit Coq, indem wir zuerst das Problem formalisieren. Dafür fangen wir bei atomaren Strukturen wie z.B. Knoten eines Graphen an. Danach definieren wir aufbauend auf bisherigen Definitionen komplexere Zusammenhänge, wie z.B. Wege in einem Graphen als Listen von Knoten. Weiterhin können wir auch Funktionen definieren, wie z.B. die Länge eines Weges als natürliche Zahl.

Mit den geschaffenen Definitionen können wir nun Aussagen über Eigenschaften der definierten Strukturen und deren Zusammenhänge formulieren und beweisen. In Abbildung 6 wird in Zeile 530 das Theorem behauptet: „Wenn für alle Knoten eines Graphen der Spannbaum lokal korrekt ist, so ist für zwei beliebige Knoten des Graphen die gewählte Wurzel des Spannbaums identisch.“ Mit dem Schlüsselwort `Proof` (Zeile 535) beginnt der Beweis der Aussage unseres Theorems. Die meisten Theoreme sind Implikationen. Die Prämissen der Implikationen werden als Beweiskontext aufgenommen. Die Konklusion wird das Beweisziel.

Ab nun beweisen wir schrittweise das Beweisziel, indem wir Aussagen des Beweiskontextes in Richtung des Beweiszieles umformen oder das Beweisziel in Richtung einer Aussage des Beweiszieles umformen. Für Umformungen können wir *Taktiken* anwenden. Taktiken sind atomar korrekte Schritte, wie z.B. „Tausche die Seiten einer

Gleichung.“ Auch alle bisher bewiesenen Lemmata sind für Umformungen einsetzbar.[2] Des weiteren gibt es einige automatische Taktiken mit denen einfache und viel-schrittige Umformungen überspringbar sind. Wenn das Beweisziel durch die Umformungen im Beweiskontext auftritt oder bei einer Gleichung die linke und rechte Seite identisch sind, so ist das Beweisziel erfüllt. Sind alle Beweisziele erfüllt, so ist der Beweis abgeschlossen; in Abbildung 6 in Zeile 528 steht der Beweisabschluss eines vorherigen Beweises mit dem Schlüsselwort `Qed`.

2.3.2 Beispiel-Arbeitsablauf mit COQ

Wir wollen für die Lesenden einen interaktiven Beweis zeigen, damit der eben skizzierte Arbeitsablauf besser verständlich wird. Dafür definieren wir Knoten, Kanten und eine Funktion. Danach beweisen wir eine Eigenschaft der Funktion. Schließlich extrahieren wir die Funktion in Haskell.
In COQ werden einige Typen schon vordefiniert. Für unser Beispiel die natürlichen Zahlen `nat` und Booleans `bool`. Die Definition von `nat` und `bool` stehen im HASKELL-Codeblock 12. Auf `nat` aufbauend werden in Codeblock 1 Knoten als induktiver Typ definiert. Mit dem Schlüsselwort `Inductive` beginnt die Definition, der Name `Vertex` von unserem Typ wird danach festgelegt. In der zweiten Zeile wird definiert, dass der Konstruktor `vertex` (diesmal klein geschrieben) eine Funktion ist, die als Eingabe eine natürliche Zahl erhält und dadurch einen Knoten vom Typ `Vertex` erstellt. Eine Instantiierung eines Knotenswäre z.B. `vertex 2`.

Codeblock 1: Definition von Knoten

```
Inductive Vertex :=
  vertex : nat -> Vertex.
```

In Codeblock 2 definieren wir gerichtete Kanten. Wieder ist in der zweiten Zeile `arc` der Konstruktor des Typen `Arc`. Der Konstruktor `arc` erhält zwei Knoten und erstellt daraus eine gerichtete Kante. Wir können Kanten folgendermaßen instantiieren: z.B. durch `arc (vertex 7) (vertex 6)` oder bei gegebenem Knoten `Variable x : Vertex.` eine Schlaufe als `arc x x`.

Codeblock 2: Definition von Kanten

```
Inductive Arc :=
  arc : Vertex -> Vertex -> Arc.
```

Codeblock 3 zeigt die Definition einer Funktion. Eingeleitet mit dem Schlüsselwort `Function` definieren wir die Funktion `get_end_vertex`. Die Funktion erhält zu einer

---

[2] Das sind alle bewiesenen Lemmata, die COQ in diesem Moment bekannt sind. Wir steuern selbst, welche Bibliotheken von Definitionen und bewiesenen Theoremen COQ lädt.

Kante noch einen Boolean-Wert `b`. Wenn `b = true` gilt, so wird das vordere Ende der Kante ausgegeben, sonst das hintere Ende.

Codeblock 3: Eine Funktion über Kanten

```
Function get_end_vertex (a : Arc) (b : bool) : Vertex :=
 match a with
 | arc x y => match b with
       | true => x
       | false => y
       end
 end.
```

Es sind auch rekursive Funktionen oder Definitionen von Strukturen möglich. Diese müssen aber stets terminieren. Das wird dadurch sicher gestellt, dass beim Rekursionsaufruf nur strukturell kleinere Parameter übergeben werden dürfen.

In Codeblock 4 sehen wir eine Aussage über die in Codeblock 3 definierte Funktion. In Lemma `get_end_vertex_correct` wird behauptet, dass für eine beliebige Kante `arc x y` und beliebige Boolean-Werte `b` durch `get_end_vertex` eines der Enden der Kante ausgegeben wird: `x` oder `y`. Der interaktive Beweis des Lemmas wird mit dem Schlüsselwort `Proof` begonnen. Wir erklären den Beweis im folgenden zeilenweise, wobei in jeder Zeile nur eine Taktik angewendet wurde.

Codeblock 4: Ein Lemma über die Funktion `get_end_vertex`

```
Lemma get_end_vertex_correct : forall (x y : Vertex) (b : bool) (a : Arc),
  a = (arc x y) ->
  get_end_vertex a b = x \/ get_end_vertex a b = y.
Proof.
  intros x y b a H.
  rewrite H.
  destruct b.
  left.
  simpl.
  reflexivity.
  right.
  simpl.
  reflexivity.
Qed.
```

Nach dem Start des Beweises erhalten wir einen leeren Beweiskontext (nichts ist über der Linie) und das Beweisziel (unter der Linie, die Aussage des Lemmas).

Codeblock 5: Beweisbeginn.

```
______________________________________(1/1)
forall (x y : Vertex) (b : bool) (a : Arc),
a = arc x y -> get_end_vertex a b = x \/ get_end_vertex a b = y
```

Wir nutzen die Taktik `intros`, um die Prämissen der Implikation des Beweiszieles in den Beweis-Kontext zu führen. Universell quantifizierte Variablen werden durch beliebige, aber feste Elemente der Variablen-Typen im Beweis-Kontext ersetzt. Prämissen des Beweiszieles werden im Beweiskontext zu bewiesenen Hypothesen. Um jedem Element des Beweiskontextes einen Namen zu geben, benennen wir die Elemente in der Reihenfolge, wie sie im Beweisziel auftreten: `intros x y b a H`.

Codeblock 6: Nach Taktik `intros`.

```
x, y : Vertex
b : bool
a : Arc
H : a = arc x y
______________________________________(1/1)
get_end_vertex a b = x \/ get_end_vertex a b = y
```

Die Variable `a` ist in der Hypothese `H` des Beweiskontextes genauer definiert. Mit der Taktik `rewrite H` können wir alle Vorkommnisse von `a` im Beweisziel ersetzen.

Codeblock 7: Nach Taktik `rewrite`.

```
x, y : Vertex
b : bool
a : Arc
H : a = arc x y
______________________________________(1/1)
get_end_vertex (arc x y) b = x \/ get_end_vertex (arc x y) b = y
```

Die Variable `b` kann nur die Werte `true` und `false` annehmen. In diesem Fall lohnt eine Fallunterscheidung, die durch die Taktik `destruct b` ausgelöst wird. Durch die Fallunterscheidung erhalten wir zwei Beweisziele. Die Variable `b` taucht nun nicht mehr im Beweiskontext auf, da sie in dem einen Fall durch `true` und in dem anderen durch `false` ersetzt wurde.

Codeblock 8: Nach Taktik `destruct`.

```
x, y : Vertex
a : Arc
H : a = arc x y
______________________________________(1/2)
get_end_vertex (arc x y) true = x \/ get_end_vertex (arc x y) true = y
______________________________________(2/2)
get_end_vertex (arc x y) false = x \/ get_end_vertex (arc x y) false = y
```

Bei dem ersten Beweisziel gilt laut Definition nur der linke Fall der Disjunktion. Durch die Taktik `left` wird nur der linke Teil der Disjunktion übernommen. Nur das obere Beweisziel (das aktuell bearbeitete) wird von dieser Taktik verändert.

Codeblock 9: Nach Taktik `left`.

```
x, y : Vertex
a : Arc
H : a = arc x y
______________________________________(1/2)
get_end_vertex (arc x y) true = x
______________________________________(2/2)
get_end_vertex (arc x y) false = x \/ get_end_vertex (arc x y) false = y
```

Mit der Taktik `simpl` versucht Coq komplexere Terme durch einfachere zu ersetzen. Dafür werden vor allem Definitionen eingesetzt und einfache Berechnungen gemacht.

Codeblock 10: Nach Taktik `simpl`.

```
x, y : Vertex
a : Arc
H : a = arc x y
______________________________________(1/2)
x = x
______________________________________(2/2)
get_end_vertex (arc x y) false = x \/ get_end_vertex (arc x y) false = y
```

Für das erste Beweisziel ist nur noch zu zeigen, dass eine Gleichung zweier identischer Terme `x` gilt. Das erfolgt mit der Taktik `reflexivity`. Dadurch ist nur noch das andere Beweisziel offen.

Codeblock 11: Nach Taktik `reflexivity`.

```
x, y : Vertex
a : Arc
H : a = arc x y
______________________________________(1/1)
```

```
get_end_vertex (arc x y) false = x \/ get_end_vertex (arc x y) false = y
```

Beim zweiten Beweisziel gehen wir analog vor. Dort wird die rechte Seite der Disjunktion durch die Taktik `right` genutzt. Nach der Vereinfachung durch `simpl` und Anwenden von `reflexivity` ist das zweite Beweisziel bewiesen. Wir schließen den Beweis mit dem Schlüsselwort `Qed.` ab.

Wir extrahieren als letztes die Funktion `get_end_vertex` nach HASKELL und erhalten Codeblock 12. Die in COQ bewiesenen Eigenschaften der Funktion `get_end_vertex` gelten bei jeder Benutzung der Funktion in HASKELL. In unserem Beispiel wissen wir also, dass `get_end_vertex_correct` für die Funktion `get_end_vertex` in Haskell gilt.
Die Typen `bool` und `nat` wurden in unserem Beispiel von einer COQ-Bibliothek gestellt. Im HASKELL-Code steht ihre Definition: `Bool` besteht aus den zwei Werten `True` und `False`. Ein Element aus `Nat` ist entweder `O` oder der Nachfolger eines Elementes aus `Nat`.[3] Da es nur einen Konstruktor für `Vertex` gibt, wurde der Konstruktor `vertex` bei der Extraktion automatisch entfernt. Dadurch sind im HASKELL-Code `Nat` und `Vertex` identisch. `Arc` und `get_end_vertex` entsprechen den Definitionen aus den Codeblöcken 3 und 4.

Codeblock 12: Der extrahierte Haskell-Code der Funktion `get_end_vertex`.

```
module Main where
import qualified Prelude

data Bool = True | False
data Nat  = O | S Nat

type Vertex = Nat
  -- singleton inductive, whose constructor was vertex

data Arc = arc Vertex Vertex

get_end_vertex :: Arc -> Bool -> Vertex
get_end_vertex a b =
  case a of {
   arc x y -> case b of {
                True -> x;
                False -> y}}
```

---

[3] `S` ist die Sukzessor-Funktion. Die Zahlen $0, 1, 2$ entsprechen `O`, `S(O)`, `S(S(O))` etc.

2.3.3 Coq-Bibliothek GraphBasics

Die verteilten Algorithmen, die wir in dieser Arbeit betrachten, laufen auf einem Netzwerk. Um die Struktur eines Netzwerks zu modellieren, bieten sich richtungslose zusammenhängende Graphen an. Damit nicht alle Graphen-theoretischen Definitionen und Beweise dieser Arbeit von Null aus geführt werden, nutzen wir eine Coq-Bibliothek für Graphen. Wir wählen die Graphen-Bibliothek GraphBasics von Duprat [Dup01], da GraphBasics in den bisherigen Arbeiten des Frameworks aus Abschnitt 2.2.1 genutzt wurde.
In der Bibliothek sind viele grundlegende Definitionen von Graphen enthalten: *Knoten, Kanten, (richtungslose) Graphen, zusammenhängende Graphen, Bäume, Wege, Pfade* etc. GraphBasics bietet einige bewiesene Theoreme über die definierten Strukturen – oftmals sind bei unseren Beweisführungen aber noch eigene, grundlegende oder spezielle Theoreme zu den Strukturen zu beweisen.

Für einen Eindruck in die Definitionen und Beweisführungen zeigen wir in Codeblock 13 beispielhaft die Definition von zusammenhängenden richtungslosen Graphen in GraphBasics.[4] In GraphBasics werden solche Graphen `Connected` genannt. Die Definitionen von `Vertex` und `Arc` haben wir aus Abschnitt 2.3.2 übernommen. Wir erinnern uns: Knoten entsprechen natürlichen Zahlen und Kanten sind ein Paar von Knoten. Durch die drei Konstruktoren `C_isolated`, `C_leaf` und `C_edge` ergeben sich drei Fälle, wie ein Element vom Typ `Connected` korrekt aufgebaut wird. Für die Fälle `C_leaf` und `C_edge` in Codeblock 13 nutzen wir die kleinere Struktur `c1` vom Typ `Connected`, um damit den neuen zusammenhängenden richtungslosen Graphen zu bilden. Die drei Konstruktoren werden im Folgenden erklärt.

Codeblock 13: Definition von zusammenhängenden richtungslosen Graphen

```
Inductive Connected : V_set -> A_set -> Set :=
  | C_isolated : forall (x : Vertex), Connected (V_single x) A_empty
  | C_leaf : forall (v : V_set) (a : A_set) (c1 : Connected v a) (x y : Vertex),
      v x -> ~v y -> Connected (V_union (V_single y) v) (A_union (arc x y) a)
  | C_edge : forall (v : V_set) (a : A_set) (c1 : Connected v a) (x y : Vertex),
      v x -> v y -> x <> y -> ~a (arc x y) -> ~a (arc y x) ->
      Connected v (A_union (arc x y) a).
```

`C_isolated` Ein einzelner Knoten `x` mit leerer Kantenmenge ist vom Typ `Connected`. `V_single x` entspricht der Menge $\{x\}$ und `A_empty` entspricht der leeren Menge $\emptyset$.

`C_leaf` Mit diesem Konstruktor bilden wir ein neues Element vom Typ `Connected`. Das neue Element nutzt als Ausgang ein schon bestehendes `c1` vom Typ

---

[4] Für das einfachere Verständnis wurden wesentliche Änderungen an der Definition vorgenommen. Die grundsätzlichen Ideen der Definition und der Art darüber zu beweisen, bleiben aber erhalten.

`Connected`. In `c1` existiert ein Knoten `x` an den wir einen neuen Blatt-Knoten `y` anhängen. Dafür darf `y` noch nicht in der Knotenmenge von `c1` enthalten sein. Wir erhalten ein neues Element vom Typ `Connected` mit dem Blatt-Knoten `y` und der Kante `arc x y`. `V_union` und `A_union` entsprechen Mengen-Vereinigungen über Knoten- und Kanten-Mengen.

`C_edge` Wir bilden wieder ein neues Element vom Typ `Connected` mit Ausgangsbasis `c1` vom Typ `Connected`. Dieses Mal verbinden wir zwei in `c1` enthaltene Knoten mit einer Kante. Die Knoten `x` und `y` müssen Element der Knotenmenge `c1` und verschieden sein. Zwischen den Knoten darf noch keine Kante `arc x y` oder `arc y x` in `c1` existieren. Wir erhalten ein neues Element vom Typ `Connected` mit der Kante `arc x y`.

Solche induktiven Definitionen nutzen wir für Beweise in Coq. Induktionsbeweise laufen über die induktive Definition der Struktur. Wenn wir beispielsweise einen Induktions-Beweis führen, der Aussagen über alle `Connected` trifft, so beweisen wir:

`C_isolated` den Basis-Fall für einen einzelnen Knoten,

`C_leaf` falls es sich um ein Element vom Typ `Connected` handelt, an den ein Blatt-Knoten angefügt wurde und

`C_edge` für ein Element vom Typ `Connected`, an den eine Kante angefügt wurde.

Sind die drei Fälle bewiesen, gilt die Aussage für alle Elemente vom Typ `Connected` und damit für alle von uns betrachteten Netzwerk-Graphen.

2.3.4 Coq-Framework Verdi

In Kapitel 4 verifizieren wir einen Teil des Checking-Prozesses unseres Frameworks. Es handelt sich dabei um die verteilte Konsistenzprüfung des Zeugen. Diese verteilte Prüfung nutzt einen verteilten Algorithmus im Netzwerk. In Coq können nur totale Funktionen ohne Seiteneffekte programmiert werden. Dadurch ist die Implementierung eines Programms mit Seiteneffekten, wie z.B. Eingaben und Ausgaben des Netzwerks nicht ohne weiteres möglich. Um Programme mit Seiteneffekten zu programmieren, sind uns die Coq-Erweiterung Ynot (siehe [NMS$^+$08], seit 2011 nicht mehr weiter entwickelt), das Framework Verdi (siehe [WWP$^+$15], Verifikation des Raft-Protokolls mit Verdi [WWA$^+$16]) und Chapar (siehe [LBC16]) bekannt. Verdi ist für die generelle Implementierung und Verifikation von verteilten Algorithmen gedacht. Chapar spezialisiert sich auf die Verifizierung verteilter Daten-Speicher mit kausal-konsistenten Zuständen. Wir wollen in Zukunft noch andere Teile des Checkers implementieren und verifizieren. Da wir die genaue Form der zukünftigen Implementierungen nicht kennen, können wir nicht das spezialisierte Chapar wählen. Deshalb wählen wir für unsere Verifikation des Checking-Prozesses Verdi.
In der offenen Verdi-Distribution [Vera] gibt es viele Beispiel-Implementierungen

und verifizierte Theoreme, die sonstige Dokumentation in [Verb] war für uns nicht immer verständlich und umfangreich genug.

Die verteilten Algorithmen, die wir mit VERDI beschreiben können, haben eine Modellierung von Seiten-Effekten, wie Input und Output. Von Inputs und Outputs wird ein Trace modelliert. Es kann nicht modelliert werden, wohin Outputs gesendet werden oder was sie dort bewirken. Außerdem ist nicht modellierbar, zu welchen Zeiten Inputs gesendet werden, oder von wo sie kommen. Dadurch ist aus COQ-Modellierungs-Perspektive das Programm Seiteneffekt-frei. Erst bei der Extraktion zu ausführbarem Code in OCAML werden von VERDI System-Calls o.ä. eingeführt.
Allerdings muss beachtet werden, dass VERDI bei der Extraktion nicht verifiziert bugfrei ist (siehe [FZWK17], für alte Bugs in genutzten Interfaces von VERDI). Um dieses Problem zu umgehen, könnte in Zukunft eine Weiterentwicklung des Compilers ŒUF (siehe [MPW$^+$18], [Mul18]) für eine verifizierte Kompilierung angewandt werden. Dennoch bleibt der Ansatz von VERDI der beste uns bekannte, solange lauffähige Programme nicht direkt in COQ schreibbar sind.

In idealistischen Netzwerken verläuft der Berechnungsablauf fehlerfrei – in realistischen Netzwerken kann es jedoch u.a. zu Nachrichtenausfall oder Komponentenausfall kommen. Verteilte Algorithmen fehlerfrei zu implementieren oder zu verifizieren ist kompliziert (siehe [WWP$^+$15], [Ong14], [FZWK17]). Sollen jedoch Eigenschaften eines Algorithmus' in fehlerbehafteten Netzwerken gezeigt werden, so ist die Verifikation wesentlich schwieriger (siehe [San06]). VERDI stellt dafür eine sehr nützliche Funktion bereit.
Wir können ein Programm in einem idealistischen Netzwerk implementieren, in dem alle Nachrichten korrekt geliefert werden und alle Komponenten fehlerfrei arbeiten. Danach zeigen wir gewisse Eigenschaften des Programms. Nun kann VERDI unser Programm so verifiziert transformieren, dass die gleichen Eigenschaften gelten, obwohl:

- Nachrichten ausfallen,
- Nachrichten in der Reihenfolge vertauscht werden,
- Nachrichten dupliziert werden,
- einzelne Komponenten fehlerhaft arbeiten oder
- Komponenten ausfallen.

In VERDI besteht ein Netzwerk aus Komponenten mit eindeutiger ID, die auf physischen Knoten laufen. Diese Komponenten können sich gegenseitig *externe* Nachrichten schicken. In VERDI können alle Komponenten mit allen anderen kommunizieren – wir müssen sicher stellen, dass nur Nachbarn im Netzwerk-Graphen Nachrichten austauschen können. Zusätzlich zu den externen Nachrichten können Komponenten zu parallel laufenden Programmen auf demselben physischen Knoten *interne* Nachrichten schicken. Diese internen Nachrichten modellieren den Input und Output des Netzwerks. Externe Nachrichten bestehen aus einem Sender, Empfänger und Inhalt (z.B. Werte von Variablen, Listen, Befehle für einen Zustandswechsel o.ä.). Interne Nachrichten haben nur einen Empfänger und Inhalt.

Der Zustand eines Netzwerkes in Verdi besteht aus den Zuständen aller Komponenten und einer Nachrichten-Liste. Der Zustand einer Komponente können Werte von Variablen, Listen o.ä. sein. Alle noch nicht abgearbeiteten externen Nachrichten stehen in der Nachrichten-Liste.
Das Netzwerk startet in einem Initialzustand, in dem alle Komponenten in ihrem eigenen Initialzustand sind und die Nachrichten-Liste leer ist. Das Netzwerk kommt von einem Zustand zu einem nächsten Zustand indem:

- eine Komponente eine an sie adressierte interne Nachricht (einen Input) empfängt und abarbeitet oder
- eine Komponente eine an sie adressierte externe Nachricht empfängt und abarbeitet. Die Nachricht wird aus der Nachrichten-Liste entfernt.

Im Initialzustand ist die Nachrichten-Liste leer. Daher kann das Netzwerk nur durch eine interne Nachricht in einen Folge-Zustand überführt werden.
Bei der Abarbeitung einer Nachricht wird eine Funktion ausgeführt, die der eigentliche Netzwerk-Algorithmus ist. Je nach Inhalt (und Sender) der Nachricht kann die Komponente in andere eigene Zustände übergehen und eventuell eine Liste neuer Nachrichten oder Outputs generieren. Neue Nachrichten werden in die Nachrichten-Liste aufgenommen.

Der induktive Aufbau von Netzwerkzuständen ermöglicht induktive Beweise über alle Zustände. Sogenannte Zustandsinvariablen sind Eigenschaften, die in allen Zuständen gelten. Wir beweisen Zustandsinvariablen, indem wir die Eigenschaft für den Initialzustand, so wie für alle Folgezustände von (erreichbaren) Zuständen zeigen. Ein Folgezustand eines Zustands entsteht nur durch die Abarbeitung einer internen oder externen Nachricht. Wenn wir beweisen, dass nach allen möglichen Abarbeitungen die Eigenschaft gilt, so gilt die Eigenschaft für alle (erreichbaren) Zustände.
Durch Zustände von Komponenten, können wir z.B. den Endzustand des Netzwerks definieren. Hierfür bekommt jede Komponente eine neue Boolean-Variable „fertig.“ Nun wird die Abarbeitung von Nachrichten so definiert, dass die Komponente nichts mehr verändert, sobald „fertig“ gesetzt wurde. Es können also keine Nachrichten mehr erzeugt werden und die Komponente darf nicht mehr ihren eigenen Zustand verändern. Wenn so ein Endzustand definiert ist, können wir auch Terminierungs-Eigenschaften beweisen: „Wenn das Netzwerk im Endzustand ist (in jedem Knoten „fertig“=„wahr“), gilt in jeder Komponente ...“. In Kapitel 4 ist eine Komponente in ihrem Endzustand, wenn ihre `child_todo`-Liste leer ist.

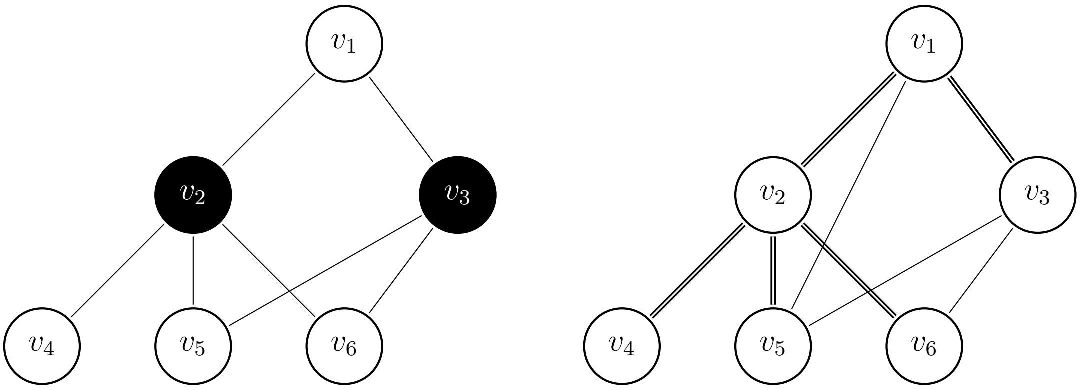
3 Fallstudie Zweifärbbarkeit

Wir zeigen hier, dass das in Kapitel 2 vorgestellte Framework auch für komplexe Probleme nützlich ist. Bisher sind Vorarbeiten für das Problem der kürzesten Pfade in [Ash16] und Leader Election in [VA17] entstanden. Hier betrachten wir das Problem der Zweifärbbarkeit, dessen Zeugenprädikat komplexer verteilt ist, als bei den beiden Arbeiten zuvor. Schließlich können wir manche Teile der Vorarbeit von [VA17] wiederverwenden, was den Vorteil des modularen Aufbaus unseres Frameworks widerspiegelt.

Wir definieren in Abschnitt 3.1 Zweifärbbarkeit und geben einen Algorithmus an, der das Entscheidungsproblem „Ist ein gegebener Graph zweifärbbar oder nicht?“ verteilt löst. Wir diskutieren Zeugen und deren Verteilung, sowohl für den Fall „Graph ist zweifärbbar“, als auch für den Fall „Graph ist nicht zweifärbbar“ in Abschnitt 3.2. In Abschnitt 3.3 zeigen wir, wie Teil-Checker ihre lokalen Zeugenprädikate prüfen. Die Abschnitte 3.2 und 3.3 wurden komplett in Coq umgesetzt und bewiesen, einige der Beweisideen geben wir auch im Text wider.
In den Abschnitten 3.2.2 und 3.3.2 nutzen wir das Konzept des Spannbaums eines Graphen. In [VA17] wurde auch das Konzept des Spannbaums genutzt. Wir können Teile der Arbeit von [VA17] hier wieder verwenden. Damit wir eine lückenlose Beweiskette für die Aussagen dieses Kapitels erhalten, beweisen wir alle offenen Lemmata zu Spannbäumen von [VA17].
Insgesamt erfüllen wir für das Problem der Zweifärbbarkeit die drei individuellen Beweisverpflichtungen BV I, BV II und BV VI (ii) unseres Frameworks (siehe Abschnitt 2.2.1). Auszüge des Coq-Codes des Falls „Graph ist zweifärbbar“ stellen wir in Abschnitt 3.4 vor.

3.1 Ein zertifizierender verteilter Algorithmus für den Zweifärbbarkeits-Test

Eine *Zweifärbung* (oder *Bipartition*) eines Graphen G ist eine Einteilung jedes Knotens von G in eine von zwei disjunkten Mengen. Dabei muss für jedes benachbarte Knotenpaar des Graphen G gelten, dass ein Knoten des Paares in der einen Menge und der andere Knoten in der zweiten Menge liegt. Seien s (schwarz) und w (weiß) zwei Farben, dann ordnet eine Färbefunktion $\mathrm{col}\colon V \to \{s, w\}$ jedem Knoten der Knotenmenge einer der beiden Farben s und w zu. Für manche Graphen existiert eine Zweifärbung, diese Graphen nennen wir *zweifärbbar*. Es gibt auch Graphen, für die keine Zweifärbung existiert, diese nennen wir *nicht zweifärbbar*. Daraus ergibt sich die *Zweifärbbarkeit* eines Graphen als entweder „der Graph ist zweifärbbar“ oder „der Graph ist nicht zweifärbbar.“
In Abbildung 7 zeigen wir einen zweifärbbaren und einen nicht zweifärbbaren Graphen. Da alle Nachbarpaare des Graphen aus Abbildung 7a unterschiedlich gefärbt sind, ist der Graph zweifärbbar. So ist beispielsweise der Knoten v_3 schwarz und alle seine Nachbarn (v_1, v_5, v_6) sind weiß gefärbt. Der Graph in Abbildung 7b ist nicht zweifärbbar.



(a) Ein zweifärbbarer Graph. Eine mögliche Zweifärbung ist eingezeichnet.

(b) Ein nicht zweifärbbarer Graph. Der Graph ist bis auf die zusätzliche Kante $\{v_1, v_5\}$ mit dem Graphen aus Abbildung 7a identisch. Die doppelt eingezeichneten Kanten sind die Kanten des Spannbaums vom Graphen.

Abb. 7: Ein zweifärbbarer und ein nicht zweifärbbarer Graph. (Eigene Darstellung)

Er gleicht dem Graphen aus Abbildung 7a, bis auf die zusätzliche Kante zwischen v_1 und v_5. Diese zusätzliche Kante erzeugt den ungeraden Zyklus (v_1, v_3, v_5, v_1) mit drei Kanten. In Abschnitt 3.2 erklären wir, warum Graphen mit ungeraden Zyklen nicht zweifärbbar sind. Auf den eingezeichneten Spannbaum in Abbildung 7b gehen wir später ein.

Das Entscheidungsproblem „Ist der Graph $G = (V, A)$ eines gegebenen Netzwerkes zweifärbbar oder nicht zweifärbbar?“ ist zertifizierend verteilt lösbar. Dabei bekommt jeder Knoten nur seine Minimaleingabe (siehe Abschnitt 2.1.3): die eigene ID (Identifikation) und die ID seiner Nachbarn. Einen zertifizierenden verteilten Algorithmus (ohne Checker) für das Zweifärbbarkeits-Entscheidungsproblem skizzieren wir [Völ17] folgend:

Phase 1 Die Knoten des Netzwerkes erzeugen einen Spannbaum des gegebenen Graphen. Ein Knoten wird als Wurzelknoten gewählt.

Phase 2 Der Spannbaum wird von der Wurzel ausgehend ebenenweise unterschiedlich gefärbt.

Phase 3 Jeder Knoten prüft, ob alle Nachbarn aus G eine andere Farbe haben. Wenn alle Nachbarn anders gefärbt sind, dann ist die Teil-Ausgabe „ja“, ansonsten „nein.“

Phase 4 Jeder Knoten gibt seine Teil-Ausgabe an die Wurzel weiter.

Phase 5 Die Wurzel prüft alle Teil-Ausgaben:

- Gibt es kein „nein“ unter den Teil-Ausgaben, dann ist die Ausgabe des

Algorithmus' „Graph ist zweifärbbar." Als Zeuge wird die Färbung gewählt, die sich aus Phase 2 ergibt.

- Gibt es ein „nein" unter den Teil-Ausgaben, dann ist die Ausgabe des Algorithmus' „Graph ist nicht zweifärbbar." Als Zeugen wählen wir den Spannbaum aus Phase 1 und die Färbung aus Phase 2. Mit diesem Zeugen kann ein Checker prüfen, ob ein ungerader Zyklus im Graphen existiert (mehr dazu in Abschnitt 3.2.2).

Für Phase 1 wird ein Spannbaum-Algorithmus (siehe [Lyn96] und [GHS83]) genutzt. Am Ende von Phase 1 weiß jeder Knoten, ob er die Wurzel ist, welches sein Eltern-Knoten und welche die Kinder-Knoten im Spannbaum sind, sowie die Distanz zum Wurzelknoten. In Phase 2 wird ein Broadcast von der Wurzel aus gestartet. Jeder Knoten gibt an seine Kinder seine eigene Farbe weiter. Beim Erhalten einer Nachricht mit einer Farbe nimmt der Knoten die jeweils andere Farbe an. In Abbildung 7b nimmt die Wurzel v_1 z.B. die Farbe „weiß" an. Dann bekommt v_2 von v_1 die Farbe „weiß" gesandt und nimmt daher selbst „schwarz" an. Die Knoten v_4, v_5 und v_6 erhalten dann von v_2 „schwarz" als Nachricht etc. Für Phase 3 befragt jeder Knoten alle seine Nachbarn nach ihren Farben und vergleicht die eigene Farbe mit diesen. Für Phase 4 nutzen wir einen Convergecast (siehe [Lyn96]) von allen Knoten zur Wurzel des Spannbaums. Schließlich wertet die Wurzel in Phase 5 alle gesammelten Teil-Ausgaben aus und gibt selbst die Ausgabe für den verteilten Algorithmus.

Im Fall „Graph ist zweifärbbar" ist der Teil-Zeuge jedes Knotenss x die in Phase 2 gewählte Farbe für x und die Farbe aller Nachbarn von x. Der Teil-Zeuge eines Knotens x für den Fall „Graph ist nicht zweifärbbar" besteht aus den in Phase 1 und Phase 2 berechneten Werten (mehr dazu in Abschnitt 3.2.2):

- von Knoten x gewählte Wurzel,
- Distanz von x zur Wurzel,
- Elternknoten von Knoten x,
- von allen Nachbarn y des Knotens x: von Knoten y gewählte Wurzel, Distanz von y zur Wurzel und
- die Farbe von Knoten x sowie die Farbe aller Nachbarn y von x.

Wir verdeutlichen die Vorgehensweise des Algorithmus' am Beispiel der Graphen aus Abbildung 7b: Die Knoten des Netzwerkes bilden den doppelt eingezeichneten Spannbaum mit Wurzel v_1. Nun wird der Spannbaum, wie in Abbildung 7a von der Wurzel ausgehend ebenenweise weiß, schwarz und weiß eingefärbt. Schließlich prüft jeder Knoten, ob in seiner Nachbarschaft ein Knoten mit derselben Farbe liegt. Bei den Knoten v_1 und v_5 ist dies der Fall, da sie beide weiß gefärbt sind. Die Knoten v_1 und v_5 versenden beim Convergecast „nein." Die Wurzel v_1 gibt deshalb in Phase 5 „Graph ist nicht zweifärbbar" aus.

Bei dem Graphen aus Abbildung 7a gibt es nach dem Einfärben in den Nachbarschaften nur ungleich farbige Knoten. Daher gibt der Algorithmus „Graph ist zweifärbbar" aus.

3.2 BV I + BV II: Zeugeneigenschaft und Verteilbarkeit des Zeugenprädikates

3.2.1 Fall „Graph ist zweifärbbar“

Wir nutzen als Zeugen eine Zweifärbung des Graphen. Laut Definition ist jeder Graph zweifärbbar für den eine Zweifärbung existiert.
Teilzeuge jedes Knotens ist die eigene Farbe und die Farbe seiner Nachbarn. Alle Teilzeugen zusammen ergeben die Färbung des gesamten Graphen. Damit der gesamte Graph zweigefärbt ist, müssen lokal gesehen alle Nachbarschaften zweigefärbt sein. Dies lässt sich lokal für jeden Knoten prüfen, siehe Abschnitt 3.3.1. Wenn jeder Teil-Checker eine zweigefärbte Nachbarschaft besitzt, dann ist der Zeuge (die Zweifärbung des gesamten Graphen) korrekt.

3.2.2 Fall „Graph ist nicht zweifärbbar“

Wir nutzen als Zeugen die Existenz eines ungeraden Zyklus' im Graphen, vgl. Satz von Brooks aus [Bro41]. Ein ungerader Zyklus als allein stehender Graph ist nicht zweifärbbar: Für den Beweis zeigen wir in Coq, dass zweigefärbte Wege ungerader Länge unterschiedlich gefärbte Start- und Endknoten haben. Bei Zyklen sind Start- und Endknoten identisch. Daher müsste man bei einer Zweifärbung den Startknoten im ungeraden Zyklus mit zwei verschiedenen Farben färben. Das ist ein Widerspruch zur Definition der Zweifärbung. Abbildung 8 veranschaulicht den Widerspruch mit einem Zyklus der Länge $2k + 1, k \in \mathbb{N}$.
Weiterhin zeigen wir, dass der Gesamt-Graph auch nicht zweifärbbar ist, wenn ein Teil-Graph eines Graphen nicht zweifärbbar ist. Daraus folgt, dass ungerade Zyklen in einem Graphen bezeugen, dass der Graph nicht zweifärbbar ist.

Um die Existenz des ungeraden Zyklus' zu beweisen, nutzen wir einen Spannbaum T vom Graphen G und eine Widerspruchs-Kante $\{x, y\}$ im Graphen G. Die Kante $\{x, y\}$ erfüllt dann unsere Kriterien, wenn x und y beide geraden oder beide ungeraden Abstand zur Wurzel w von T haben. Haben beide geraden Abstand, so existieren Pfade xw und wy im Spannbaum T, die die Längen $l_x = 2 \cdot n_x$ und $l_y = 2 \cdot n_y$, mit $n_x, n_y \in \mathbb{N}$ haben. Haben beide ungeraden Abstand, so sind die Längen $l_x = 2 \cdot n_x + 1$ und $l_y = 2 \cdot n_y + 1$, mit $n_x, n_y \in \mathbb{N}$. Die Kante $\{x, y\}$ hat die Länge eins.
Die Pfade xw, wy und die Kante $\{y, x\}$ bilden zusammen einen Zyklus mit der Länge $l_x + l_y + 1 = 2 \cdot n_x + 2 \cdot n_y + 1 = 2(n_x + n_y) + 1$ bzw. bei ungeraden Abständen zur Wurzel $l_x + l_y + 1 = 2(n_x + n_y + 1) + 1$. In beiden Fällen ist die Länge des Zyklus' (xw, wy, yx) ungerade. Der Spannbaum T und die Kante $\{x, y\}$, mit den oben genannten Kriterien, zusammen bezeugen also die Existenz eines ungeraden Zyklus' in G.
In Abbildung 7b sieht man die Kanten des Spannbaums doppelt eingezeichnet. Die Kante $\{v_1, v_5\}$ ist in der Abbildung die Widerspruchs-Kante. Der Knoten v_1 und v_5 haben geraden Abstand zur Wurzel v_1 des Spannbaums. Es entsteht der ungerade

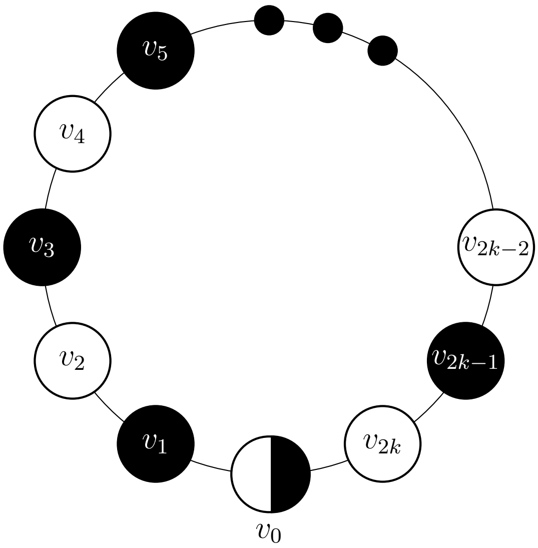


Abb. 8: Ein Zyklus ungerader Länge ist nicht zweifärbbar: Färben wir von Knoten v_0 aus die Knoten im Uhrzeigersinn abwechselnd weiß und schwarz, erhält Knoten v_0 schließlich auch die schwarze Farbe. (Eigene Darstellung)

Zyklus (v_1, v_5, v_2, v_1).

Die Verteilung des Zeugen für den Fall „Graph ist nicht zweifärbbar“ wird zweigeteilt: einen Zeugen für einen lokal korrekten Spannbaum für jeden Knoten (Universalverteilung). Einen Zeugen für eine Widerspruchs-Kante $\{x, y\}$, bei der die Distanzen zur Wurzel des Spannbaums l_x und l_y entweder beide gerade oder beide ungerade sind. Die Existenz einer Widerspruchs-Kante $\{x, y\}$ reicht für den zweiten Zeugenteil aus (Existenzialverteilung).
Den lokal korrekten Spannbaum bezeugen zusammen: die lokal gewählte Wurzel, der Elternknoten des Spannbaums und die Distanz zur Wurzel sowie von allen Nachbarn: die gewählte Wurzel und die Distanz zur Wurzel. Die Widerspruchs-Kante $\{x, y\}$ bedarf keines weiteren lokalen Zeugen, da die eigene Distanz und die Distanzen der Nachbarn schon für die Bezeugung des Spannbaums nötig sind. Wie der Checker diese lokalen Zeugen prüft, folgt im Abschnitt 3.3.2.

3.3 BV VI (ii): Checker

Die Teil-Checker in diesem Abschnitt wurden in Coq implementiert und nach Haskell extrahiert.

3.3.1 Fall „Graph ist zweifärbbar“

Jeder Teil-Checker erhält als Eingabe die eigene Farbe und die Farbe aller Nachbarn. Der Teil-Checker prüft, ob alle Nachbarn anders als er selbst gefärbt sind. Sind alle Nachbarn anders gefärbt, gibt er „Zweifärbung lokal korrekt“ aus. Ansonsten „Zweifärbung lokal nicht korrekt.“ Wenn alle Teil-Checker „Zweifärbung lokal korrekt“ ausgeben, ist die Zweifärbung ein korrekter Zeuge für den Fall „Graph ist zweifärbbar.“
In unserem Algorithmus wird bei der Implementierung durch ungerade und gerade Distanzen zur Wurzel eine implizite Zweifärbung erzeugt. Daher prüft der Teil-Checker von Knoten v in der folgenden Aussage 8 mit dem Modulo-Operator die Geradheit der Distanzen.

$$\nexists n \in \text{nachbarn}(v)\colon \text{distanz}\ (v) \equiv \text{distanz}\ (n) \pmod 2 \tag{8}$$

Wenn die Aussage für einen Knoten v gilt, so haben alle Nachbarknoten von v eine andere Farbe als v selbst. Im Fall „Graph ist zweifärbbar“ werden nur die Distanzen verglichen – es wird nicht geprüft, zu welcher Wurzel diese Distanz besteht, oder ob die gewählte Wurzel überhaupt existiert. Da aber hier die Distanz als Färbung interpretiert wird, ist die jeweils gewählte Wurzel irrelevant. Nur die Korrektheit der Zweifärbung des Graphen ist für diesen Fall wichtig.

3.3.2 Fall „Graph ist nicht zweifärbbar“

Für die Prüfung des Spannbaums in diesem Abschnitt verwenden wir die Prinzipien von Akili et al. (vgl. [VA17]). Wir passen den Quelltext von Akili et al. an und beweisen einige offene Axiome, so dass die verteilte Prüfung des Spannbaums axiomfrei ist.
Jeder Teil-Checker erhält als Eingabe die lokal gewählte Wurzel und die Distanz zur Wurzel, sowie den gewählten Elternknoten. Des Weiteren bekommt jeder Teil-Checker von all seinen Nachbarn deren gewählte Wurzel und Distanz zur Wurzel. Um zu prüfen, dass der Spannbaum lokal korrekt ist, muss jeder Teil-Checker einer Komponente v Aussage 9 prüfen.

$$\begin{aligned}(\forall n \in \text{nachbarn}\ (v)\colon \text{wurzel}\ (v) = \text{wurzel}\ (n)) \land \\ ((\text{wurzel}\ (v) = v \land \text{elter}\ (v) = v \land \text{distanz}\ (v) = 0) \lor \\ (\text{wurzel}\ (v) \neq v \land \text{elter}\ (v) \in \text{nachbarn}\ (v) \land \text{distanz}\ (v) = \text{distanz}\ (\text{elter}\ (v)) + 1))\end{aligned} \tag{9}$$

In Zeile eins von Aussage 9 wird überprüft, dass alle Knoten des Netzwerks die gleiche Wurzel gewählt haben. Zeile zwei erdet für die Wurzel die Eltern- und die Distanzfunktion. Zeile drei beschreibt alle anderen Knoten und garantiert deren Elternknoten in der Nachbarschaft (und damit Teil des Graphen). In Zeile drei wird weiterhin ein Weg über die Elternknoten zur Wurzel garantiert, der höchstens distanz(v) lang ist.
Alle Teil-Checker überprüfen zusätzlich mit Aussage 8 aus Abschnitt 3.3.1 die Zweifär-

bung ihrer Nachbarschaft.

Wenn für alle Teil-Checker ihr Zeugenprädikat 9 gilt, ist der Spannbaum korrekt. Wenn zusätzlich mindestens ein Teil-Checker ein nicht geltendes Zeugenprädikat 8 hat, so existiert ein ungerader Zyklus. Der Zeuge ist dann ein korrekter Zeuge für den Fall „Graph ist nicht zweifärbbar."

3.4 Ausschnitte aus dem Coq-Code

In diesem Abschnitt werden Code-Fragmente der vorherigen Abschnitte des Kapitels aufgeführt und erklärt. Wir führen hier nur Definitionen und Theoreme auf. Da die Beweise und alle Zwischen-Schritte komplett maschinen-geprüft sind, ist nur das Verständnis der Definitionen und Theoreme notwendig, die Zwischenschritte sind mechanisch geprüft. Wir konzentrieren uns auf den „Graph ist zweifärbbar"-Fall, da damit alle Ideen dieses Abschnitts veranschaulicht werden können. Der „Graph ist nicht zweifärbbar"-Fall folgt den unformalen Erläuterungen der vorherigen Abschnitte.
Für das bessere Verständnis haben wir einige Code-Fragmente vereinfacht. Für nicht hier aufgelistete Zwischen-Lemmata und den originalen Code verweisen wir auf unser GitHub-Repository [Coq].
Mit den hier gezeigten Theoremen und den Theoremen des „Graph ist nicht zweifärbbar"-Falls haben wir alle individuellen Beweisverpflichtungen des Frameworks erfüllt.

3.4.1 Grundlegende Definitionen und Interface des ZVA

Wir übernehmen die in Kapitel 2 genannten Definitionen für `nat`, `Vertex`, `Arc` und `Connected`. Die Notation für „ein Knoten ist Element einer Knotenmenge": $v_1 \in v$, entspricht `v v1`. Für eine Kante $(v_1, v_2) \in a$ analog `a (arc v1 v2)`. Als semantische Abstraktion haben wir in den Code-Fragmenten `Component` für `Vertex` genutzt.

Zweifärbung

Codeblock 14: `Definition bipartition`

```
Definition bipartition (a: A_set) (color : Component -> bool) : Prop :=
 forall (ar : Arc), a ar -> color (arc_tail ar) <> color (arc_head ar).
```

Für alle Kanten einer Kantenmenge gibt eine Zweifärbung dem Kanten-Anfang eine andere Farbe, als dem Kanten-Ende. Die Parameter-Funktion `color` ordnet jedem Knoten dabei eine der Farben `True` oder `False` zu. Die Eigenschaft `bipartition` gilt genau dann, wenn `color` alle Nachbar-Knoten des Graphen unterschiedlich gefärbt hat.

Codeblock 15: Definition `bipartite`

```
Definition bipartite (a: A_set) :=
  exists (color : Component -> bool), bipartition a color.
```

Eine Kantenmenge (eines Graphen) ist zweifärbbar, wenn es eine Zweifärbung für sie gibt. Hier nutzen wir das eben definierte `bipartition` und übergeben als Parameter eine Färbung.

ZVA-Interface und Checker-Eingabe

Codeblock 16: Interface des ZVA

```
Variable is_bipartite : bool.
Variable distance : Component -> nat.
Variable neighbors_distance : Component -> list (Component * nat).
```

Für das Interface des ZVA definieren wir die Signatur von drei Variablen. Diese Variablen muss der ZVA dann konkret implementieren. An dieser Stelle reicht es, die Form und einige Annahmen der Funktionen zu kennen und die Implementierung als Black Box zu betrachten. Mit der Variable `bipartite` gibt der ZVA eine (globale) Antwort auf die Frage, ob der Graph zweifärbbar ist oder nicht. Mit der variablen Funktion `distance` wird für jeden Knoten implizit festgelegt, welche Farbe der Knoten bekommt. Die Farbe ergibt sich daraus, ob die Distanz des Knotens gerade oder ungerade ist. Je nachdem bekommt der Knoten die Farbe `True` oder `False` zugeordnet. In den Codeblöcken 22 und 24 wird beispielhaft die Distanz zu einer Farbe umgerechnet. Die Funktion `neighbors_distance` ist eine Liste von Tupeln aller Nachbarn und deren Distanzen. Das Interface des ZVA ist hier nur für den Fall „Graph ist zweifärbbar“ angegeben. Für den anderen Fall gibt es noch weitere variable Funktionen, wie den gewählten Leader. Die Distanz wird in dem Fall auch als natürliche Zahl und nicht nur als Farbe ausgewertet. Dies ist für den hier betrachteten einfacheren Fall nicht nötig.
Für unseren Ansatz benötigen wir nicht die genaue Vorgangsweise, wie der ZVA die Funktion `neighbors_distance` berechnet. Dennoch haben wir bestimmte Annahmen, in welcher Form die Variablen belegt werden: Die Liste `neighbors_distance v` eines Knotens `v` soll für jeden Nachbarn von `v` exakt einen Eintrag mit einer Distanz aufweisen. Dass die eingetragene Distanz der tatsächlichen Distanz des Nachbarn entspricht, wird mit der Konsistenzprüfung aus Kapitel 4 geprüft.

Codeblock 17: Die Minimal-Eingabe jedes Teil-Checkers

```
Record minimal_input: Set := mk_minimal_input {
  i : Component;
  neighbors : list Component;
}.
```

```
Definition construct_minimal_input (x: Component) : minimal_input :=
  mk_minimal_input
    x
    (neighbors x).
```

Zuerst definieren wir mit `minimal_input`, wie die Minimal-Eingabe eines Teil-Checkers syntaktisch aussieht (siehe Abschnitt 2.1.3). Sie ist ein Zweier-Tupel bestehend aus einer ID (die ID des Knotens des Teil-Checkers) und einer Liste von IDs (die IDs aller Nachbarn). In `construct_minimal_input` konstruieren wir die Minimal-Eingabe genau wie oben in den Klammern semantisch beschrieben.

Codeblock 18: Die Eingabe für den Teil-Checker vom ZVA

```
Record checker_input : Set := mk_checker_input {
  algo_answer : bool;
  distance_i : nat;
  distance_nl : list nat
}.

Definition construct_checker_input (x : Component) : checker_input :=
  mk_checker_input
    is_bipartite
    (distance x)
    (neighbors_distance x).
```

In `checker_input` definieren wir die Form der ZVA-Eingabe an den Checker. Eine Bool-Variable (die Ausgabe des Algorithmus'), eine natürliche Zahl (die eigene Distanz/Farbe) und eine Liste von natürlichen Zahlen (die Distanzen/Farben der Nachbarn). Wir definieren, wie diese Einträge konstruiert werden in der Funktion `construct_checker_input`. Die Funktion `construct_checker_input` übernimmt die vom ZVA berechneten Werte aus Codeblock 16.

3.4.2 BV I + BV II: Zeugeneigenschaft und Verteilbarkeit des Zeugenprädikates

Wir verteilen ein Zeugenprädikat `Gamma_1` universell auf die Knoten. Jeder Knoten muss das Teil-Prädikat `gamma_1` prüfen und akzeptieren.

Codeblock 19: `Definition gamma_1`

```
Definition gamma_1 (v: V_set) (a: A_set) (c: Connected v a) (v1: Component) (color:
     Component -> bool) :=
      forall (v2 : Component), In v2 (construct_minimal_input v1).neighbors ->
          color v1 <> color v2.
```

Das Teil-Prädikat `gamma_1` für einen Knoten `v1` prüft die Farbe aller Nachbarn der Knoten von `v1`. Wenn alle Nachbarn eine andere Farbe haben, als `v1`, gilt das Prädikat und die Nachbarschaft von `v1` ist zweifärbbar. Der Parameter `color` wird wie in Codeblock 14 genutzt. In Codeblock 24 wird dem Parameter eine implizite Färbung durch die Distanzen vom ZVA zugewiesen.

Codeblock 20: `Definition Gamma_1`

```
Definition Gamma_1 (v:V_set) (a:A_set)(c: Connected v a) (color: Component ->
    bool) :=
  forall (v1 : Component), v v1 -> gamma_1 v a c v1 color.
```

Die Verteilung des Zeugenprädikats: alle Nachbarschaften aller Knoten des Graphen müssen zweigefärbt sein. Damit ist es eine universelle Verteilung. Für die Verteilung ist kein eigener Beweis nötig, da sie per Definition durch Codeblock 20 gilt.

Codeblock 21: `Theorem Gamma_1_implies_Postcondition`

```
forall (v: V_set) (a: A_set) (c: Connected v a) (color: Component -> bool),
  Gamma_1 v a c color -> bipartite a.
```

Hier beschreiben wir die Zeugeneigenschaft: wenn das Zeugenprädikat gilt (wenn alle Nachbarschaften eines Graphen zweigefärbt sind), so gilt die Nachbedingung (der Graph ist zweifärbbar).

3.4.3 BV VI (ii): Verifikation der Teil-Checker

Der Teil-Checker soll prüfen, ob die Nachbarschaft lokal zweigefärbt ist. Hierfür bekommt er die eigene Farbe und die Farbe aller Nachbar-Knoten als Zeugen. Die Farbe wird in unserem Teil-Checker aus der Distanz gewonnen, siehe Seite 39.

Codeblock 22: `Fixpoint locally_bipartite`

```
Fixpoint locally_bipartite (d: nat) (neighbor_l: list nat): bool
  match neighbor_l with
  | nil => true
  | n :: tl => if (eqb (Nat.odd d) (Nat.odd n)) then false else
      locally_bipartite d tl
  end.
```

Der Checker nutzt diese Unterfunktion, welche prüft, ob alle in der Liste `neighbor_l` gelisteten Nachbarn eine andere Farbe haben als der Knoten selbst. Die Funktion ist rekursiv aufgebaut.

Codeblock 23: `Definition checker_local_bipartition`

```
Definition checker_local_bipartition (x: Component): bool
  locally_bipartite (checker_input x).distance_i (checker_input x).
      distance_nl.
```

Das Checker-Programm des Knotens `x`: die Unterfunktion `locally_bipartite` wird mit den korrekten Startparametern aufgerufen. Es wird die eigene Farbe `(checker_input x).distance_i` und eine Liste aller Nachbarn von `x` übergeben: `(checker_input x).distance_nl`. Diese Werte werden in Codeblock 18 berechnet. Der Wert der Funktion ist die Ausgabe des Teil-Checkers von `x`.

Codeblock 24: `Theorem local_checker_correct`

```
forall (x: Component),
  v x -> checker_local_bipartition x = true ->
  gamma_1 v a c x (Nat.odd (checker_input x).distance_i).
```

Dieses Theorem beschreibt die Korrektheit der Teil-Checker: Wenn der Teil-Checker von `x` eine lokale Zweifärbung akzeptiert, so ist die Nachbarschaft von `x` tatsächlich zweigefärbt.

Codeblock 25: `Theorem checker_correct`

```
(forall (x : Component), v x -> (checker_local_bipartition x) = true) ->
    bipartite a
```

Nun werden die Theoreme `part_checker_correct`, `Gamma_1_Psi` und die Definition von `Gamma_1` für einen Beweis des folgenden Theorems kombiniert: Wenn alle Teil-Checker ihren Teil-Zeugen akzeptieren, ist der Graph zweifärbbar. Mit den hier gezeigten und in unserem Quell-Code [Coq] bewiesenen Theoremen haben wir die individuellen Beweisverpflichtungen des Frameworks erfüllt.

4 Prüfung der Zeugenkonsistenz

In diesem Kapitel stellen wir unseren neuen Ansatz vor, die Konsistenz eines Zeugen zu prüfen. Für den Ansatz aus [Aki18] und [VA18] ist die Konsistenz in der Nachbarschaft, der Zusammenhang und die Vollständigkeit zu prüfen. In unserem Ansatz werden alle dieser Schritte durch eine einzige, netzwerkweite Konsistenzprüfung ersetzt. Insbesondere die für jeden ZVA zu erfolgenden Verpflichtungen des Frameworks werden so um zwei Verpflichtungen reduziert.
Die Problemstellung dieses Kapitels wird in Abschnitt 2.1.3 erläutert. Hier stellen wir zuerst in Abschnitt 4.1 unsere Idee der Prüfung vor, die Implementierung der Prüfung folgt in Abschnitt 4.2. Darauf folgen in Abschnitt 4.3 skizzenhaft die gefundenen oder noch nötigen Beweise für die Korrektheit und Terminierung unserer Konsistenzprüfung.

4.1 Prinzip der Konsistenz-Prüfung

Wir schreiben ein verteiltes Programm für die Teil-Checker, das die Aussage 7 (Seite 17) prüft. Zur Kommunikation nutzen wir einen Spannbaum. Den Spannbaum nehmen wir als gegeben an, da er an verschiedenen Stellen des Frameworks genutzt wird.
Anfangs hat jede Komponente eine Liste von Variablen-Zuordnungen. In diese Liste werden als Initialzuordnung die Eingabe, Ausgabe und Teil-Zeuge des ZVA aufgenommen. Jeder Teil-Checker schickt, von den Blättern des Spannbaums her, seine Variablen-Zuordnungen (anfangs nur die Initialzuordnung) an seine Eltern-Komponente im Spannbaum. Bei Erhalt von Zuordnungen durch ein Kind im Spannbaum nimmt die Komponente die erhaltenen Zuordnungen zu den eigenen auf. Haben alle Kinder ihre Zuordnungen geschickt, schickt die Komponente die Zuordnungen an die Eltern-Komponente. Wenn die Wurzel alle Zuordnungen erhalten hat, prüft sie die Konsistenz der Zuordnungen und gibt das Ergebnis der Prüfung aus.

Der Algorithmus, wie hier beschrieben, ist noch nicht optimiert und der Anteil an verteilter Berechnung minimal. Da durch den hier gewählten Ansatz alle Komponenten den gleichen Ablauf haben, ist so jedoch die Verifikation wesentlich leichter. Mögliche Optimierungen, die man vornehmen könnte, listen wir hier auf:

- Duplikate gleicher Zuordnungen werden entfernt, bevor sie an die Eltern-Komponente verschickt werden.
- Wenn eine Komponente eine Inkonsistenz in ihrem Teilbaum entdeckt, verschickt sie `false` zu ihrer Eltern-Komponente (oder als Broadcast). Bei Erhalt von `false` kann jede Komponente `false` an ihre Elternkomponente schicken und ihr Teil-Programm terminieren. Die Wurzel gibt dann `false` als Antwort der Konsistenzprüfung aus.
- Statt der eigentlichen Zuordnungen werden Hashwerte verglichen. Dies erleichtert den Kommunikations- und Speicheraufwand jeder Komponente enorm, je nach

gewählter Hashfunktion (möglichst kollisionsresistent, siehe [RS04]) gibt es dafür aber eventuell falsche positive Ergebnisse.

4.2 Umsetzung der Konsistenz-Prüfung

Wir schreiben einen verteilten Algorithmus mit Hilfe von VERDI. Jeder Teil-Checker bekommt einen eigenen Teil-Algorithmus zugeordnet. Die Gemeinschaft aller Teil-Checker soll die Konsistenz der Teil-Zeugen gegenüber den Teil-Eingaben, Teil-Ausgaben und anderen Teil-Zeugen prüfen. Wir definieren in Abschnitt 4.2.1, wie die Eingaben, Ausgaben und Zeugen aufgebaut sind. Weiterhin definieren wir dort die internen Variablen eines Teil-Checkers. In Abschnitt 4.2.2 implementieren wir Hilfsprogramme, die vom Haupt-Algorithmus genutzt werden und zeigen die wichtigsten Eigenschaften von den Hilfsprogrammen. Danach stellen wir in Abschnitt 4.2.3 den verteilten Algorithmus vor, den jeder Teil-Checker ausführt, wenn die Konsistenz geprüft wird. In Abschnitt 4.3.1 listen wir wichtige Eigenschaften des Algorithmus' auf, die wir zeigen konnten, bzw. deren Beweis noch offen ist. Diese Eigenschaften werden genutzt, um die Korrektheit (Abschnitt 4.3.2) und die Terminierung (Abschnitt 4.3.3) der Konsistenzprüfung zeigen zu können.

4.2.1 Eingabe, Ausgabe und Zustand eines Teil-Checkers

Wie in Abschnitt 2.1.3 beschrieben, nutzen wir für Teil-Eingaben, Teil-Ausgaben und Teil-Zeugen jeweils die gleiche Form: eine Menge (oder Liste) von Zweier-Tupeln. Jedes Paar besteht dabei aus einem Variablen-Namen und einer dazugehörigen Belegung. Falls z.B. die Komponente v_5 die Nachbarn v_1, v_8, v_9 hat, wäre die Darstellung dafür: $(n_5, \{v_1, v_8, v_9\})$.

Codeblock 26: Definition von Belegungen

```
Variable Var: Type.
Variable Value: Type.
Inductive Assignment := assign_cons: Var -> Value -> Assignment.
Definition Certificate := list Assignment.
```

In COQ definieren wir Variablen (`Var`) und Belegungen (`Value`) nicht spezifisch. Beide sind vom allgemeinen Typ `Type`. Insbesondere die Belegungen `Value` sollen frei wählbar – z.B. Zahlen, Strings oder Mengen – sein können. Eine Zuweisung wird durch den Konstruktor `assign_cons` gebildet. Mit dem Konstruktor `assign_cons` entsteht für jede Zuweisung ein Paar aus einer Variable und einer Belegung. Eine Liste von Zuweisungen nennen wir `Certificate`.

Codeblock 27: Definition der Eingaben des Algorithmus’

```
Variable init_input : Component -> Certificate.
Variable init_output : Component -> Certificate.
Variable init_witness : Component -> Certificate.
Function init_combined (n : Component) : Certificate :=
  (init_input n) ++ (init_output n) ++ (init_witness n).
```

Jede Komponente bekommt als ihre Teil-Eingabe (`init_input`), Teil-Ausgabe (`init_output`) und ihren Teil-Zeugen (`init_witness`) vom ZVA eine Liste von Zuordnungen zugeordnet. Da bei der Konsistenzprüfung alle Zuordnungen getestet werden, führen wir die drei mit der Funktion `init_combined` zu einem gemeinsamen `Certificate` jeder Komponente zusammen. Dieses `Certificate` nutzen wir als Eingabe für unsere verteilte Konsistenzprüfung.

Codeblock 28: Eingabe und Ausgabe des Algorithmus’

```
Record Data := mkData{
  assign_list : Certificate;
  child_todo : list Component}.

Definition init_Data (me: Component) :=
  mkData (init_combined me)
         (init_children me).

Definition Output := bool.
```

Jeder Teil-Checker hat bei seiner Prüfung einen internen Zustand vom Typ `Data`. Der interne Zustand besteht aus den Werten zweier interner Variablen: einer Liste von Zuordnungen `assign_list` und einer Liste von nicht abgearbeiteten Kindern `child_todo`. Der Ausgangs-Zustand wird in `init_Data` gesetzt. Wir übernehmen das Gesamt-Zertifikat aus Code-Block 27 für die eigene Komponente. Die Liste der noch nicht abgearbeiteten Kinder wird auf alle Kinder der Komponente im Spannbaum gesetzt. Die Funktion `init_children` wird aus der Minimal-Eingabe jedes Teil-Checkers übernommen.

Im Laufe der Bearbeitung sammeln sich in der Liste `assign_list` nach und nach alle Zuordnungen, die von den Kindern und Kindeskindern an den Teil-Checker geschickt wurden. Jedes Kind, was seine Zuordnung geschickt hat, wird dann aus der Liste `child_todo` gestrichen. Wenn die Liste `child_todo` leer ist, sind alle Kinder abgearbeitet. Das ist gleichzeitig das Signal, dass die Komponente ihre Zuordnungen an die Eltern-Komponente schicken kann. Insgesamt sammeln sich so alle Zuordnungen des Teilbaums der Komponente in `assign_list` an. Insbesondere bei der Wurzel sammeln sich so alle Zuordnungen des gesamten Baumes.

Schließlich definieren wir die Ausgabe des Algorithmus’ als Boolean-Variable. Wenn die Wurzel des Spannbaums `true` ausgibt, ist der Zeuge konsistent, anderenfalls nicht.

4.2.2 Hilfsprogramme und deren wichtigste Eigenschaften

Die in diesem Abschnitt erklärten Unterprogramme werden vom verteilten Algorithmus aus Abschnitt 4.2.3 aufgerufen. Das erste Unterprogramm `assign_list_consistent` prüft, ob die aktuellen Zuordnungen konsistent sind. Das zweite Unterprogramm `remove_child` entfernt ein Kind aus der `child_todo`-Liste einer Eltern-Komponente, nachdem die Zuordnungen des Kindes von der Eltern-Komponente empfangen wurden.

Codeblock 29: Prüfung der Konsistenz einer Liste

```
Fixpoint is_always (test_case : Assignment) (vl : Certificate) : bool :=
  let (var, val) := test_case in
  match vl with
  | nil => true
  | hd :: tl => let (vl_var, vl_val) := hd in
      if (var == vl_var) then (val == vl_val) && is_always test_case tl else
          is_always test_case tl
  end.

Fixpoint assign_list_consistent (vl : Certificate) : bool :=
  match vl with
  | nil => true
  | hd :: tl => (is_always hd tl) && assign_list_consistent tl
  end.
```

Die Funktion `assign_list_consistent` prüft die Konsistenz einer Zuordnungsliste. Sie prüft dabei für jedes Element (`var`, `val`), ob `var` in der Liste immer mit `val` belegt ist. Die von `assign_list_consistent` aufgerufene Hilfsfunktion `is_always` übernimmt dabei die Aufgabe zu prüfen, ob überall die konsistente Zuordnung von `var` eingehalten wird.

Codeblock 30: Korrektheit von `assign_list_consistent`

```
Definition is_consistent (cert : Certificate) : Prop :=
  forall (var : Var) (val1 val2 : Value),
    In (var,val1) cert -> In (var,val2) cert ->
      val1 = val2.

Lemma assign_list_consistent_works : forall (cert : Certificate),
  assign_list_consistent cert = true <-> is_consistent cert.
```

Um zu zeigen, dass die Funktion `assign_list_consistent` korrekt ist, definieren wir die Eigenschaft `is_consistent` für eine Liste von Zuordnungen: sind die Zuordnungen (`var`, `val1`) und (`var`, `val2`) in der Liste enthalten, müssen `val1` und `val2` gleich sein. Das in Coq bewiesene Lemma `assign_list_consistent_works` bezeugt, dass unsere Funktion `assign_list_consistent` korrekt arbeitet.

Codeblock 31: Entfernen aus einer Komponentenliste

```
Fixpoint remove_child (src : Component) (child_list : list Component) : list
    Component :=
  match child_list with
  | []  => []
  | hd ::  tl => if src == hd then (remove_child src tl) else
          hd ::  (remove_child src tl)
  end.
```

Die Funktion `remove_child` entfernt alle Instanzen einer Komponente aus einer Komponentenliste. Wir nutzen die Funktion im verteilten Algorithmus, um ein bearbeitetes Kind aus der Kinderliste `child_todo` zu entfernen.

Codeblock 32: Korrektheit von `remove_child`

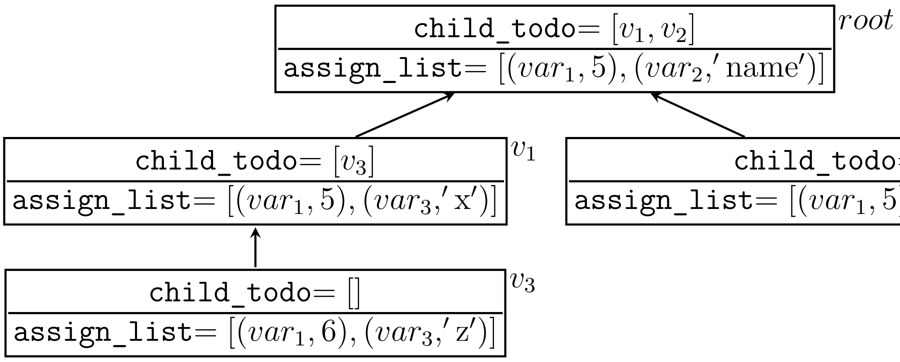
```
Lemma remove_if_not_in : forall (c : Component) (cl : list Component),
  ~In c cl -> remove_child c cl = cl.

Lemma remove_removes_one : forall (c : Component) (cl : list Component),
  In c cl -> NoDup cl -> Permutation (c :: remove_child c cl) cl.
```

Wir listen hier zwei wichtige Eigenschaften der Funktion `remove_child` auf. Das Lemma `remove_if_not_in` zeigt, dass `remove_child` die Liste unverändert lässt, wenn die gesuchte Komponente nicht in ihr enthalten ist. Das zweite Lemma `remove_removes_one` zeigt, dass `remove_child` korrekt arbeitet, wenn die gesuchte Komponente in der Liste enthalten ist und keine Duplikate enthält: die Komponentenliste `cl` ist dann eine Permutation von `c :: remove_child c cl`. Das bedeutet, dass `remove_child c cl` ein Element weniger hat, als `cl` und genau das gesuchte Element `c` aus ihr entfernt sein musste. Diese und einige ähnliche Lemmata haben wir maschinell bewiesen und damit die Korrektheit der beiden Unterprogramme gezeigt.

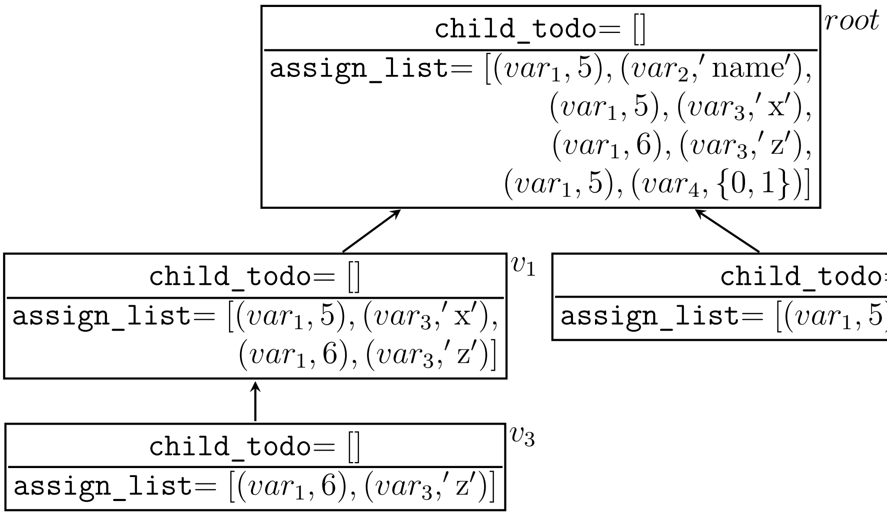
4.2.3 Der verteilte Algorithmus

Das Netzwerk startet im Initialzustand, der für jede Komponente in Code-Block 28 vorgegeben wird. Wie in Abschnitt 2.3.4 detaillierter beschrieben, kommt das Netz in den nächsten Zustand, indem eine Komponente eine interne Nachricht oder eine externe Nachricht einer anderen Komponente erhält. Der erste Fall wird in VERDI durch den sogenannten `InputHandler` gelöst, der zweite durch den `NetHandler`. Wir nehmen an, dass jede Komponente nach Terminierung des Teil-Algorithmus des ZVA eine interne Nachricht an den Teil-Checker schickt. Diese Nachricht bedeutet: „die Konsistenzprüfung darf gestartet werden.“

Sowohl der `NetHandler`, als auch der `InputHandler` einer Komponente bekommen als Eingabe die ID der Komponente und den Zustand der Komponente. Beide produzieren als Ausgabe eine Liste an Outputs (hier entweder eine leere Liste, oder das Ergebnis



(a) Die initialen Zustände der Komponenten eines Netzwerkes vor der Konsistenzprüfung. Die Kanten des Spannbaums sind eingezeichnet. Die Zuordnungen fließen von den Blättern (unten) zur Wurzel. Jedes abgearbeitete Kind wird aus der `child_todo`-Liste gestrichen.



(b) Das Netzwerk aus Abbildung 9a nach der Beendigung der Prüfung. Alle Kinder sind abgearbeitet und die `child_todo`-Listen leer. Alle Zuordnungen des Netzwerkes sammeln sich bei der Wurzel in `assign_list` an. Die Wurzel prüft die Konsistenz und gibt in diesem Fall `false` aus, denn sowohl var_1, als auch var_3 weisen widersprüchliche Zuordnungen auf.

Abb. 9: Beispielnetzwerk für unseren Konsistenzprüf-Algorithmus mit der Initialzuordnung und nach der Berechnung. (Eigene Darstellung)

der Konsistenzprüfung bei der Wurzel), einen neuen Zustand der Komponente und eine Liste an Nachrichten (hier entweder eine leere Liste oder eine Nachricht an die Eltern-Komponente mit der aktuellen Zuordnung).

Bemerkung: in unserem Spannbaum hat jede Komponente, auch die Wurzel, eine Eltern-Komponente. Die Wurzel führt sich selbst als Eltern-Komponente – diese Eigenschaft nutzen wir auch in unserem Algorithmus aus.

Codeblock 33: Die Abarbeitung einer internen Nachricht

```
Definition InputHandler (me : Component) (data: Data) :
            (list Output) * Data * list (Component * Certificate) :=
  match (children me) with
  | []  => ([], data, [( parent me, (assign_list data))])
  |  _ => ([], data, [])
  end.
```

Der `InputHandler` wird von uns dafür genutzt, die Konsistenzprüfung von den Blättern her zu starten. Bei allen Nicht-Blättern gibt es keine Auswirkung bei Erhalt einer internen Nachricht. Wenn eine Komponente vom ZVA eine interne Nachricht erhält, gibt es zwei Fälle: die Komponente hat Kinder oder hat keine Kinder:

1. Die Komponente ist kinderlos. Dies ist der Fall, wenn die Komponente ein Blatt des Spannbaums ist. Der Funktionswert vom `InputHandler` ist das Tupel `([], data, [(parent me, (assign_list data))])`. Das bedeutet: die leere Liste wird als Output ausgegeben (nichts wird ausgegeben), der Zustand der Komponente wird nicht verändert, eine Liste mit einer Nachricht wird versendet (an die Elternkomponente wird die eigene Zuordnung geschickt).
2. Die Komponente hat Kinder und ist demnach kein Blatt. In diesem Fall wird die interne Nachricht ignoriert und keine Ausgaben werden produziert.

Codeblock 34: Die Abarbeitung einer externen Nachricht

```
Definition NetHandler (me : Component) (src: Component) (msg : Certificate)
    (data: Data) : (list Output) * Data * list (Component * Certificate) :=
  match (child_todo data) with
      | []  =>
        ([ assign_list_consistent (assign_list data)], data, [])
      | [c]  =>
        ([],  (mkData ((assign_list data) ++ msg) []), [(parent me, (assign_list
            data ++ msg))])
      | _ =>
        ([],  (mkData ((assign_list data) ++ msg) (remove_child src (child_todo
            data))), [])
  end.
```

Wenn eine Komponente eine Nachricht erhält, so hat diese Nachricht einen Sender (`src`) und eine gesendete Zuordnungsliste (`msg`). Der Empfänger unterscheidet bei Erhalt einer externen Nachricht drei Fälle:

1. Die eigene `child_todo`-Liste ist leer. Dies ist ein Fall der nur bei der Wurzel auftreten kann, weil die Wurzel ihr eigener Eltern-Knoten ist. Nachdem alle Kinder abgearbeitet waren, schickte sie eine Nachricht an sich selbst. In diesem Fall kann die Prüfung der Konsistenz der Zuordnungsliste mit der Funktion

`assign_list_consistent` aus Code-Block 29 erfolgen. Das Ergebnis der Prüfung wird dann ausgegeben. Die Berechnung des Netzwerks ist damit beendet.

2. Die `child_todo`-Liste hat nur noch eine einzige Komponente. Genau diese Komponente ist der Sender. Der Empfänger nimmt die Zuordnung des Senders auf und schickt die neue Zuordnung an die Eltern-Komponente. Falls der Empfänger selbst die Wurzel ist, löst er damit den ersten Fall des `NetHandler` aus. Weiterhin wird die `child_todo`-Liste auf die leere Liste gesetzt, da nun alle Kinder abgearbeitet sind.
3. Die `child_todo`-Liste führt noch mehrere Komponenten. Der Empfänger nimmt die Zuordnung des Senders auf. Weiterhin wird der Sender aus der `child_todo`-Liste entfernt. Da noch nicht alle Kinder abgearbeitet sind, braucht keine Nachricht versendet werden.

4.3 Programmverifikation und Terminierung

4.3.1 Eigenschaften der Konsistenzprüfung

In diesem Abschnitt listen wir Eigenschaften des im vorigen Abschnitt gezeigten verteilten Algorithmus', die wir für die Korrekheitsbeweise des nächsten Abschnitts nutzen. Die gelisteten Lemmata beschreiben Zustandsinvariablen (siehe Abschnitt 2.3.4). Ein beispielhaftes Lemma beschreiben wir mit COQ-Code, den Rest listen wir nur unformal auf. Für die Zustandsinvariablen definieren wir zuerst erreichbare Zustände.

Codeblock 35: Definition erreichbarer Zustände

```
Definition Trace := list (Component * (input + list output)).

Definition net_reachable (net : network) : Prop :=
  exists (tr : Trace), refl_trans_1n_trace step_async_init net tr.
```

Wir erklären kurz die wichtigen Elemente der Definitionen aus Codeblock 35 :

- `list (Component * (input + list output))`: ein von außen beobachtbarer *Trace* von Eingaben in das Netzwerk und von Ausgaben aus dem Netzwerk.
- `network`: der Zustand des Netzwerks. Der Zustand beinhaltet die inneren Zustände der Komponenten und alle noch nicht abgearbeiteten Nachrichten.
- `refl_trans_1n_trace step_async`: die hier genutzte Zustandsübergangsrelation, sie ist asynchron und fehlerfrei, siehe 2.3.4.
- `step_async_init`: der Ausgangszustand des Netzwerks, wie in Codeblock 28 spezifiziert.

Die Definition `net_reachable` in Codeblock 35 ist zu lesen als: es existiert ein Trace, der beobachtet werden kann, wenn das Netzwerk aus dem Ausgangszustand in den Zustand `net` übergeht. Der Übergang kann dabei mehrschrittig sein. Mit diesem Baustein lässt sich formulieren: „In jedem vom Ausgangszustand `step_async_init` erreichbaren Netzwerkzustand `net` gilt folgendes ...“.

Eine Eigenschaft unseres Algorithmus’ erklären wir hier ausführlich mit Coq-Code: „Wenn die `child_todo`-Liste einer Komponente leer ist, ändert sich der interne Zustand der Komponente nicht mehr.“ Man könnte Komponenten mit leerer `todo`-Liste auch als *unveränderlich* betrachten.

Codeblock 36: Eine unveränderliche Komponente ändert ihren internen Zustand nicht mehr

```
Definition c_unchangable (net : Network) (c : Component) : Prop :=
  (nwState net c).(child_todo) = [].

Definition c_no_change (net : Network) (c : Component) : Prop :=
  forall (net2 : Network) (tr2 : Trace),
  refl_trans_1n_trace net net2 tr2 ->
  nwState net2 c = nwState net c.

Lemma terminated_no_change : forall (net : Network) (c : Component),
  net_reachable net -> c_unchangable net c ->
  c_no_change net c.
```

Die Eigenschaft `c_unchangable` gilt, wenn eine Komponente eine leere `child_todo`-Liste hat. Die Eigenschaft `c_no_change` gilt, wenn eine Komponente von einem Netzwerkzustand ausgehend nie wieder ihren internen Zustand ändert. Das Lemma `terminated_no_change` besagt, dass eine unveränderliche Komponente ihren internen Zustand nicht mehr verändert. Da wir in dem Lemma den Parameter `net` mit einem Universalquantor binden, beschreibt dieses Lemma eine Zustandsinvariable.

Wir listen im Folgenden weitere Eigenschaften der verteilten Konsistenzprüfung unformal auf. Viele der bewiesenen oder behaupteten Aussagen ähneln sich oder bauen aufeinander auf. Wir haben diejenigen mit ✓ markiert, bei denen wir einen maschinellen Beweis gefunden haben. Mit ✗ markiert sind die beiden Behauptungen, deren Beweis noch aussteht. Die Beweise dieser Lemmata können Inhalt einer zukünftigen Arbeit sein. Schließlich haben wir mit (✓) diejenigen Beweise markiert, die von den beiden noch nicht bewiesenen Lemmata abhängen. Unter der Annahme, dass die noch ausstehenden Beweise gelten, gelten die mit (✓) markierten Beweise ebenfalls.

- ✓ Eine unveränderliche Komponente ändert nie wieder ihren internen Zustand. (Codeblock 36)
- ✓ Jede Komponente empfängt nur von ihren Kindern Nachrichten und jeder Sender schickt Nachrichten nur an seine Eltern-Komponente.

✓ Nach dem Senden einer Nachricht ist die sendende Komponente unveränderlich.

✓ Die Initialzuordnung einer Komponente verbleibt in allen erreichbaren Zuständen in ihrer `assign_list`.

✓ Wenn eine Komponente keine Kinder hat (ein Blatt ist), so ist die `child_todo`-Liste leer.

✓ Nur Kinder einer Komponente können in ihrer `child_todo` List sein.

✓ Die `child_todo`-Liste enthält keine Doppelungen.

✓ Für jede Zuordnung einer Nachricht existiert eine Komponente im Unterbaum des Empfängers, bei der diese Zuordnung eine Initialzuordnung ist.

✓ Jede Zuordnung in `assign_list` einer Komponente stammt aus einer Initialzuordnung ihres Unterbaums.

✗ Für zwei Nachrichten, die noch nicht abgearbeitet wurden, ist nie der Sender der einen Nachricht der Empfänger der anderen Nachricht. (Außer bei der Wurzel, die sich zum Schluss eine eigene Nachricht sendet.)

(✓) Jede Nachricht enthält die derzeitige Zuordnung des Senders.

✗ Jeder Sender einer Nachricht ist in der `child_todo`-Liste des Empfängers. (Außer bei der Wurzel, die sich zum Schluss selbst eine Nachricht sendet.)

(✓) Alle Zuordnungen von Kindern einer Komponente, die nicht mehr in der `child_todo`-Liste sind, sind in der Zuordnung der Komponente aufgenommen.

(✓) Wenn eine Komponente unveränderlich ist, sind alle Komponenten des Unterbaums unveränderlich.

(✓) Wenn eine Komponente unveränderlich ist, sind sämtliche Zuordnungen des Unterbaums in der Zuordnung der Komponente aufgenommen.

Bei den beiden Lemmata, die mit ✗ markiert sind, haben wir bei den ersten Beweisversuchen nicht bedacht, dass die Aussagen jeweils nicht für die Wurzel gelten – der Teil, der in Klammern bei den beiden Lemmata steht. Daher ist uns ein Beweis der beiden Lemmata nicht geglückt. Aus Zeitgründen haben wir nachdem uns der Fehler aufgefallen war, nicht versucht den Beweis der Lemmata nachzuholen. Wir gehen aber davon aus, dass nach der Korrektur ein Beweis möglich ist.

4.3.2 Programmverifikation der Konsistenzprüfung

Für den Beweis der Korrektheit des gesamten Algorithmus' definieren wir zuerst in Coq die Konsistenz: für alle Komponentenpaare ist die Kombination ihrer initialen Zertifikate konsistent. Wir nutzen die Definition von `is_consistent` aus Code-Block 30.

Codeblock 37: Die Konsistenz des gesamten Netzwerks

```
Definition witness_consistent : Prop :=
  forall (c1 c2 : Component), v c1 -> v c2 ->
    is_consistent ((init_combined c1) ++ (init_combined c2))
```

Damit können wir die Haupt-Theoreme dieses Abschnitts formulieren. Wir nutzen `net_reachable'`, was ähnlich, wie `net_reachable` aus Codeblock 35 funktioniert, aber den Trace-Parameter `tr` nicht existentiell bindet.

Codeblock 38: Die Korrektheit der verteilten Konsistenzprüfung

```
Theorem output_true_witness_consistent: forall (net : Network) (tr : Trace)
  (c : Component),
    net_reachable' net tr ->
    In (c, inr [true]) tr ->
    witness_consistent.

Theorem output_false_witness_inconsistent: forall (net : Network) (tr : Trace)
  (c : Component),
    net_reachable' net tr ->
    In (c, inr [false]) tr ->
    ~witness_consistent.
```

(✓) Ist die Ausgabe in einem erreichbaren Zustand `true`, ist der Zeuge des Netzwerks konsistent.[5]

✓ Ist die Ausgabe in einem erreichbaren Zustand `false`, ist der Zeuge des Netzwerks nicht konsistent.

Nur für das erste Theorem sind die offenen Beweise aus Abschnitt 4.3.1 noch zu führen. Mit dem vollständig bewiesenen Theorem `output_false_witness_inconsistent` können wir aber falsche negative Ausgaben der Prüfung ausschließen. Mit dem lückenlosen Beweis des ersten Theorems folgt die Korrektheit der Prüfung.

4.3.3 Terminierung der Konsistenzprüfung

Damit wir wissen, dass unsere Konsistenzprüfung korrekt arbeitet, ist noch zu zeigen, dass die verteilte Prüfung immer hält. Wir argumentieren erst unformal und listen dann zwei Coq-Theoreme auf, die die Terminierung der Konsistenzprüfung beschreiben. Der Beweis der Terminierung war nicht Ziel dieser Arbeit – der Beweis ist noch offen und kann Bestandteil zukünftiger Arbeiten sein.

---

[5] Die Markierungen ✓ und (✓) folgen den Markierungen von Seite 51.

In Verdi ist die Terminierung eines Algorithmus nicht direkt beweisbar. Dies liegt daran, dass keine Annahmen über die Anzahl der internen Nachrichten getroffen werden – wenn eine nicht endende interne Nachrichtenkette versendet wird, stoppt die Berechnung des Algorithmus nicht.
So, wie wir die Konsistenzprüfung in den Teil-Checker implementieren wollen, können wir jedoch zusätzliche Annahmen treffen. Wenn der Teil-ZVA einer Komponente terminiert, so schickt er eine Nachricht an den eigenen Teil-Checker. Dies interpretiert der Teil-Checker als Startnachricht. Da jeder Teil-ZVA nur einmal, aber in jedem Fall, terminiert, erhält jeder Teil-Checker exakt eine Startnachricht.
Eine echte zusätzliche Annahme, die wir treffen ist, dass die Startnachrichten ungefähr gleichzeitig eintreffen. Zumindest sollte jede Komponente eine Startnachricht empfangen haben, bevor sie die Zuordnungen eines Kindes empfängt. Wenn der ZVA das nicht garantieren kann, müsste ein Wrapper um unseren Algorithmus geschrieben werden, der darauf wartet, dass alle Komponenten Zuordnungen haben, bevor die erste Startnachricht versendet wird.

Die Komponenten im Spannbaum, die Blätter sind, haben eine leere `children`-Liste und schicken daher direkt nach Erhalt der Startnachricht ihre Zuordnung an die Eltern-Komponente. Jedes Kind, was seine Zuordnung geschickt hat, wird aus der `child_todo` -Liste des Empfängers gestrichen. Da irgendwann alle Kinder ihre Nachricht geschickt haben, ist die `child_todo`-Liste der Komponenten, die nur Blätter als Kinder haben, irgendwann leer. Diese Eltern von Blättern schicken ihre Zuordnung nach oben, in Richtung der Wurzel und so weiter. Auch die Wurzel schickt eine Nachricht an sich selbst. Nachdem die Wurzel von sich die eigene Nachricht erhalten hat, gibt sie das Ergebnis der Prüfung aus. Damit ist der verteilte Algorithmus beendet: alle Komponenten sind unveränderlich und keine Nachrichten sind abzuarbeiten.

Codeblock 39: Axiome für: jede Komponente erhält genau eine Nachricht

```
Axiom input_to_every_comp : forall (net : network) (tr : Trace),
  net_reachable' net tr ->
  exists (net2 : network), exists (tr2 : Trace),
    refl_trans_1n_trace step_async net net2 tr2 /\
    forall (c : Component), v c-> In (c, inl start) tr.

Function filter_inputs (tr : Trace) : list Component :=
  match tr with
  | []  => []
  | (c, inl start) :: tl => c :: filter_inputs tl
  | (c, inr start) :: tl => filter_inputs tl
  end.

Axiom no_duplicate_inputs : forall (net : network) (tr : Trace),
  net_reachable' net tr -> NoDup (filter_inputs tr).
```

Wir listen in Codeblock 39 die beiden Axiome, welche zusammen annehmen, dass jede Komponente unseres Netzwerks exakt eine Nachricht erhält. Axiom `input_to_every_comp` nimmt an, dass jede Komponente eine Nachricht erhält. Axiom `no_duplicate_inputs` nimmt an, dass jede Komponente maximal eine Nachricht erhält. Die Funktion `filter_inputs` nimmt aus einem `Trace` die Input-Nachrichten heraus und speichert die Liste der Empfänger.

Wir stellen nun zwei verschiedene Ansätze vor, wie man die Terminierung in Coq beweisen könnte.

Codeblock 40: Terminierung der Prüfung, Ansatz 1

```
Theorem everything_can_end : forall (net:network) (tr:Trace) (b:bool)
  (c:Component),
    net_reachable' net tr ->
    {net2 : network & {tr2 : Trace &
      refl_trans_1n_trace step_async net net2 tr2 /\
      In (c, inr [b]) (tr ++ tr2)}}.
```

Das Theorem `everything_can_end` behauptet, dass aus jedem erreichbaren Zustand ein weiterer Zustand (in endlich vielen Schritten) erreichbar ist, in dem es eine Ausgabe im Trace gibt. Endlosschleifen werden durch dieses Theorem ausgeschlossen, da das Netzwerkmodell nicht-deterministisch arbeitet: Wenn es aus jedem Zustand einen möglichen endlichen Weg zu einem Endzustand gibt, so wird dieser auch irgendwann genommen und die Berechnung beendet. Dennoch ist das Argument des Theorems `everything_can_end` über die Terminierung nur indirekt.

Betrachten wir nun für den zweiten Ansatz die Anzahl der abgearbeiteten Nachrichten bis eine Ausgabe durch die Wurzel produziert wird: Jede Komponente bekommt exakt eine Startnachricht. Weiterhin schickt jedes Kind eine Nachricht an seine Elternkomponente. Die Wurzel schickt schließlich noch eine Nachricht an sich selbst und gibt danach das Ergebnis aus. Die Ausgabe der Wurzel führt im Netzwerk jedoch zu keiner Zustandsänderung, da die Nachricht nicht im Netzwerk empfangen wird. Aus diesem Grund wird die Ausgabe nicht mitgezählt.
Seien n Komponenten in unserem Netzwerk. Dann ist nach der Abarbeitung der n Startnachrichten und $n-1$ Nachrichten an die Eltern im Spannbaum und einer Nachricht der Wurzel an sich selbst der verteilte Algorithmus beendet. Der Algorithmus terminiert also nach $2n$ Zustandsänderungen.

Dadurch ist unser konstruktiver Ansatz aus Codeblock 41 möglich. Wir definieren dafür eine Relation `n_step`, die nicht nur die Erreichbarkeit von einem Zustand zu einem anderen definiert. Zusätzlich zur Erreichbarkeit werden die nötigen Schritte mitgezählt. Die Eigenschaft `n_step n net net2` bedeutet, dass man in `n` Schritten von Zustand `net` zu Zustand `net2` gelangen kann. Mit dem Basisfall `n_step0` verbleibt man in null Schritten in `net`. Der Fall `n_stepn` greift, wenn man in `n`−1 Schritten zu einem Zwischen-Zustand kommt und von dort aus in einem Schritt zu `net2`. Seien in unserem

Netzwerk `Component_count` Komponenten. Dann gilt, wie vorhin beschrieben, dass das Netzwerk nach `2*Component_count` Schritten eine Ausgabe produziert. Dieses Theorem ist in `terminates_after_2n` festgehalten.

Codeblock 41: Terminierung der Prüfung, Ansatz 2

```
Inductive n_step : nat -> network -> network -> Prop :=
  | n_step0 : forall net net2, net = net2 -> n_step 0 net net2
  | n_stepn : forall n net net2,
              (exists net', n_step (n-1) net net' /\
               exists tr, refl_trans_1n_trace step_async net' net2 tr) ->
                 n_step n net net2.

Variable Component_count : nat.

Theorem terminates_after_2n : forall (net : network) (b : bool) (tr : Trace),
  n_step (2*Component_count) step_async_init net ->
  net_reachable' net tr ->
  In (root, inr [b])  tr.
```

Der Beweis eines dieser beiden Ansätze würde die Terminierung unserer Prüfung zeigen.

Zusätzlich könnte noch ein Beweis geführt werden, der zeigt, dass das Netzwerk nie mehr als einen Output produziert. Wenn also ein Output im Trace existiert, ist der Algorithmus von außen betrachtet beendet. Codeblock 42 beschreibt diesen Zusammenhang: wenn in einem erreichbaren Zustand `root` einen Wert ausgegeben hat, dann sind alle Ausgaben im Trace identisch. Durch die Funktion `NoDup` werden noch zusätzlich Mehrfach-Ausgaben ausgeschlossen.

Codeblock 42: Es gibt nur einen Output

```
Theorem only_one_output: forall (net : network) (b1 b2 : bool) (c : Component)
  (tr :  Trace),
 net_reachable' net tr ->
 In (root, inr [b1])  tr ->
 In (c ,  inr [b2])  tr ->
 (c = root /\ b1 = b2 /\ NoDup tr).
```

5 Offene Probleme lösen: zertifizierender verteilter Bellman-Ford-Algorithmus

Im Zusammenhang mit den Vorarbeiten zu ZVAs sind wir auf ein offenes Problem in [Ash16] gestoßen. Asher bearbeitete dort die Verifikation des zertifizierenden verteilten Bellman-Ford-Algorithmus (siehe [BGH92], [CRKGLA89] und [VR15]) nach der Methode der formalen Instanzkorrektheit. Im Gegensatz zu den vorangegangen Kapiteln unserer Arbeit wurde in der Arbeit von Asher nicht die Coq-Bibliothek GraphBasics benutzt. Stattdessen definierte er Knoten auf der Basis von Untermengen von natürlichen Zahlen. Durch diese Definition fielen manche Beweise so komplex aus, dass Asher es auch mithilfe eines Expertenforums (siehe [Coq16]) nicht schaffte, alle Beweise zu führen. Ein Beweis insbesondere schien so schwierig, dass er vermutete, Coq alleine sei nicht für die Aufgabenstellung geeignet. Im Kontext dieser Arbeit ist es uns gelungen für die noch offenen Lemmata mit seinen Definitionen Coq-Beweise zu finden. Durch unsere Beweise schließen wir alle Lücken der Beweiskette der Arbeit von Asher.

Wir gehen davon aus, dass mit unserer Methode aus Abschnitt 5.1.2 alle noch offenen Beweisverpflichtungen (unseres Frameworks aus Abschnitt 2.2.1) aus Ashers Arbeit erfüllt werden können. Dennoch empfehlen wir eine Anpassung seiner Arbeit an eine Graphen-Bibliothek, wenn auf seiner Arbeit mit einer Implementierung und Verifikation eines Checkers aufgebaut werden soll. Unserer Meinung nach würde z.B. die Bibliothek GraphBasics eine für Beweise einfacher zu handhabende Struktur von Graphen bieten.

In Abschnitt 5.1 erläutern und beweisen wir das Lemma `arg_min_inequality`. Abschnitt 5.2 befasst sich mit dem Beweis des Lemmas `select_ok`.

5.1 Beweis des Lemmas `arg_min_inequality`

Wir führen in Abschnitt 5.1.1 das Lemma `arg_min_inequality` und die dafür nötige Notation ein und formen es in eine lesbarere Version um. In Abschnitt 5.1.2 erläutern wir die wichtigsten Beweise, die nötig waren, um das Lemma `arg_min_inequality` in Coq zu zeigen.

5.1.1 Notation und Umformungen von `arg_min_inequality`

Um das Lemma `arg_min_inequality` zu verstehen, führen wir vorher noch die Coq-Notation von Untermengen ein. Wir definieren die Menge der natürlichen Zahlen kleiner als `n` in Coq folgendermaßen:

```
Variable n : nat.
Definition mnnat := { m : nat | m < n }.
```

Es kann für `mnnat` die mathematische Notation $\{m \in \mathbb{N} \mid m < n\} = \{0, 1, 2, \ldots, n-1\}$ angenommen werden. Semantisch wird `mnnat` als Knotenmenge genutzt. Eine natürliche Zahl kleiner als `n` ist ein Knoten der Knotenmenge. Im Weiteren argumentieren wir nur syntaktisch über natürliche Zahlen.

Codeblock 43: Definition von `mnnat_y`

```
Definition ltn (m : nat) : Prop := m < n.
Variable y : nat.
Variable H : ltn y.
Definition mnnat_y : mnnat := exist ltn y H.
```

In Codeblock 43 definieren wir beispielhaft ein Element `mnnat_y` aus `mnnat`, welches der natürlichen Zahl `y` entspricht. Mit dem Schlüsselwort `exist` und den drei Parametern `ltn` (Proposition, dass `m < n` gilt), `y` (die natürliche Zahl `y`), `H` (ein Beweis, dass `y < n` gilt) wird ein Element aus `mnnat` geformt. Jedes Element aus `mnnat` trägt also zusätzlich zur natürlichen Zahl *einen Beweis* dafür, dass die natürliche Zahl kleiner als `n` ist.

Codeblock 44: Lemma `arg_min_inequality`

```
Variable f : mnnat -> nat.
Variable g : mnnat -> nat.

Lemma arg_min_inequality : forall x,
  (f x < g x) -> (exists x, f x < g x /\
                              forall y, f y < f x -> f y >= g y).
```

In Codeblock 44 stellen wir nun das Lemma `arg_min_inequality` in seiner ursprünglichen Form (siehe [Ash16], S. 53) vor. Vor einer Erklärung des Lemmas `arg_min_inequality` formen wir es für ein besseres Verständnis noch syntaktisch um:

- Die Variable `x` der Prämisse ist nicht in der Konklusion gebunden, daher
  - formen wir das `forall` der Prämisse zu `exists` um,
  - benennen wir das `x` in der Prämisse zu `z` um.
- Die Implikation in der Konklusion wird umgedreht: `f y < f x -> f y >= g y` wird zu `f y < g y -> f x >= g x`.
- Wir abstrahieren die Proposition `f x < g x` als `P`.

Nach den Umformungen erhalten wir das Lemma `exists_minimal_P_holder` aus Codeblock 45. Wir begründen kurz, warum das Lemma `exists_minimal_P_holder` gelten muss: Nehmen wir zuerst an, es existiert kein `z` der Prämisse (z.B. `n` $= 0$, oder `P` gilt nie), dann gilt das Lemma trivialerweise, da die Prämisse nie gilt. Im anderen Fall gibt es ein `z` und daher mindestens ein `x`, mit `P x`. Unter allen Elementen, wie `x`, muss eines sein, welches den minimalen `f`-Wert hat. Dieses Element ist nicht unbedingt eindeutig, falls es mehrere Elemente mit Minimalwert gibt. Nehmen wir ein `f`-kleinstes Element `x`, so gibt es kein `y` mehr, welches noch `f`-kleiner ist und bei dem `P y` gilt.

Codeblock 45: Umformung des Lemmas `arg_min_inequality`

```
Definition P x : Prop := f x < g x.

Lemma exists_minimal_P_holder :
  exists z : mnnat, P z ->
  exists x : mnnat, P x /\ (forall y : mnnat, P y -> f x <= f y).
```

Für den Beweis von `exists_minimal_P_holder` in Coq ist Induktion nicht direkt anwendbar, da `f` und `P` nur auf `mnnat` und nicht z.B. auf $\{\texttt{m} : \texttt{nat} \mid \texttt{m} < \texttt{n} + 1\}$ definiert sind. In Coq formen wir für den Beweis `P` und `f` so zu `P'` und `f'` um, dass sie `nat` statt `mnnat` als Definitionsbereich haben. Danach können wir wir einen Induktionsbeweis führen.

Konzentrieren wir uns auf die Umformung von `f` zu `f'` – die Umformung von `P'` erfolgt analog. Wenn wir `f : mnnat -> nat` durch `f' : nat -> nat` ersetzen wollen, muss `f'` die gleichen Ausgaben wie `f` erzeugen. Dafür muss `f'` zwingend `f` aufrufen, also aus einer natürlichen Zahl als Eingabe von `f'` ein Element aus `mnnat` formen, welches an `f` übergeben wird. Die Funktion `f'` muss also für jede Eingabe einen Beweis (siehe Seite 58) formen.

Nachdem wir `P'` und `f'` formen und in `exists_minimal_P_holder` einsetzen, können wir das Lemma per Induktion beweisen.

5.1.2 Die wichtigsten Beweise

Codeblock 46: Lemma `Pf'exist`

```
Lemma Pf'exist : forall A (a : A) (Pf : mnnat -> A),
  exists Pf' : nat -> A,
    (forall (x : nat) (H : x < n), Pf' x = Pf (exist ltn x H)).
```

Das Lemma `Pf'exist` zeigt, dass es für `P` und `f` jeweils ein `P'` und `f'` mit den natürlichen Zahlen als Definitionsbereich gibt, wobei der Wertebereich im `mnnat`-Zahlenbereich übereinstimmt. Es sollen also die natürlichen Zahlen auf die äquivalenten Elemente aus `mnnat` abgebildet werden. Die Menge `A` steht in dem Lemma bei `f` für `nat` und bei `P` für `Prop`.

Im Folgenden vollziehen wir den Coq-Beweis des Lemmas `Pf'exist` nach. Wir nutzen dabei die modulo-Funktion und einige ihrer Eigenschaften.

Der Beweis des Lemmas `Pf'exist` trennt sich in zwei Teile: `n = 0` und `n > 0`.
`n = 0`: `mnnat` ist leer und wir nutzen eine konstante Funktion `Pf' x = a`. Das Element `a` bekommen wir übergeben: `a` wird für `f` als 0 und für `P` als `False` übergeben. Es existiert daher die Funktion `Pf'`, die immer `a` ausgibt. Da es kein `x < 0` gibt, gilt der Rest des Lemmas trivialerweise.
`n > 0`: Wir nutzen die Annahme `n > 0` als Beweis für `n > 0` und nennen diesen Beweis `H`. Mit diesem Beweis `H` bilden wir den Beweis `mod_mnnat m H` (siehe Codeblock 47) als Beweis für die Aussage `m mod n < n`. Wir definieren damit `Pf'` als: (`Pf' = fun (m : nat) => Pf (exist ltn (m mod n) (mod_mnnat m H))`). Die Funktion `Pf'` bekommt eine natürliche Zahl als Eingabe und ruft dann die Funktion `Pf` auf. `Pf'` übergibt `Pf` ein Element aus mnnat. Nun nutzen wir die beiden Lemmata aus Codeblock 47: das erste Lemma bezeugt, dass die an `Pf` übergebene natürliche Zahl tatsächlich kleiner als `n` ist. Das zweite Lemma bezeugt, dass natürliche Zahlen kleiner als `n` auf die äquivalenten Elemente aus `mnnat` abgebildet werden.

Codeblock 47: Zwei Hilfs-Lemmata für `Pf'exist`

```
Lemma mod_mnnat : forall m,
  n > 0 -> m mod n < n.

Lemma mod_mnnat' : forall m,
  m < n -> m mod n = m.
```

In der Beweisskizze oben wird ein Detail noch nicht erwähnt: Zwischenzeitlich ist bei dem Beweis von `Pf'exist` in Coq folgendes zu zeigen:

```
x < n -> (exist ltn (x mod n)(mod_mnnat x H))= (exist ltn x H1).
```

Die Konklusion dieser Aussage bedeutet, dass zwei Elemente aus `mnnat` gleich sind, wenn die natürliche Zahl von ihnen gleich ist: `x mod n = x` und die Beweise von ihnen gleich sind: `(mod_mnnat x H) = H1`. Ersteres folgt mit Lemma `mod_mnnat'` aus Codeblock 47, zweiteres aus der Gleichheit von Beweisen desselben Typs: In `Prop` werden zwei Beweise derselben Proposition als gleich angesehen und daher ist die exakte Art des Beweises unwichtig, solange er korrekt ist (siehe [Zhu13]). Dies ist vor allem wichtig, weil wir den Beweis `H1` der rechten Seite der Aussage nicht kennen, aber dennoch den gleichen Beweis benötigen. Wir wenden hier das Coq-Lemma `proof_irrelevance` an, welches die benötigte Äquivalenz von Beweisen gleicher Propositionen zeigt.
Wir haben nun `P'` und `f'` konstruiert, welche wir zur Vereinfachung des Lemmas `exists_minimal_P_holder` nutzen.

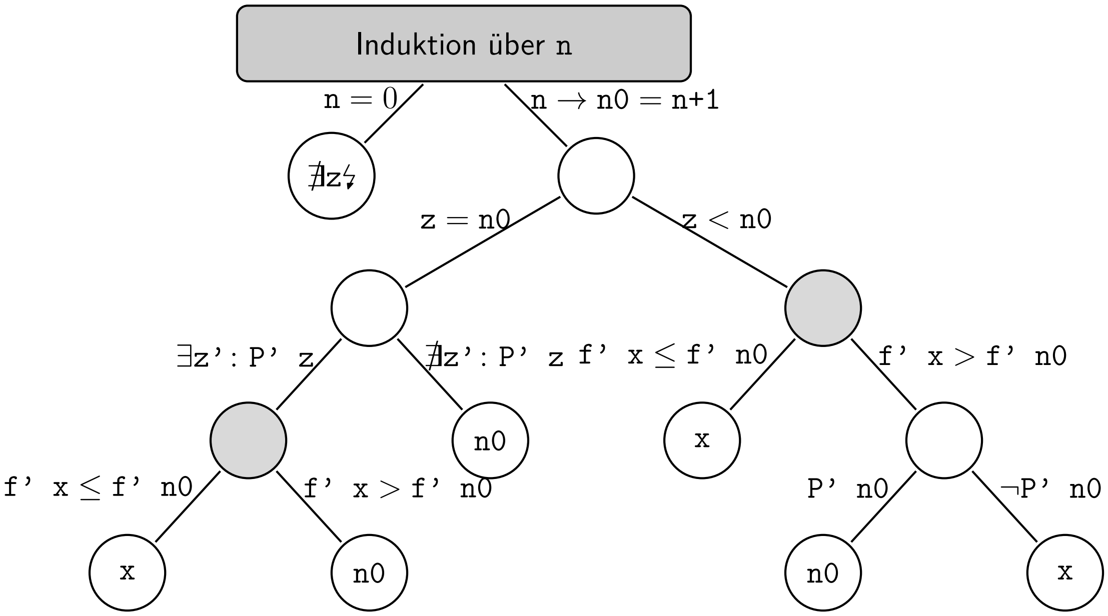


Abb. 10: Die Fallunterscheidungen im Beweis von `exists_minimal_P_holder_nat` aus Codeblock 48. (Eigene Darstellung)

Codeblock 48: `exists_minimal_P_holder` auf `nat` definiert

```
Lemma exists_minimal_P_holder_nat : forall n (f' : nat -> nat)
  (P' : nat -> Prop),
  (exists z, z < n /\ P' z) ->
  (exists x, x < n /\ P' x /\
    (forall y : nat, y < n -> P' y -> f' x <= f' y)).
```

In Codeblock 48 wird das Lemma `exists_minimal_P_holder_nat` vorgestellt. Das Lemma entsteht nach Einsetzung von `P’` und `f’` in das Lemma `exists_minimal_P_holder`. Es beruht jetzt nur noch auf den natürlichen Zahlen und nicht mehr auf `mnnat` und wir können daher einen Induktionsbeweis über `n` führen. Alle Fallunterscheidungen des Induktionsbeweises werden in Abbildung 10 gezeigt, und einige davon werden im weiteren erläutert.

Der Induktionsstart für `n` = 0 ist einfach, da keine natürliche Zahl `z` existieren kann, welche kleiner als 0 ist.

Im Induktionsschritt nehmen wir an, das Lemma gelte für alle `n` mit `n < n0` und `x` sei das `f’`-minimale Element. Falls `z` < `n0` gilt oder falls noch ein weiteres `z’` < `n0`, `P’ z’` existiert (linker grauer Kreis), können wir direkt die Induktionsvoraussetzung anwenden (rechter grauer Kreis). Falls die Induktionsvoraussetzung anwendbar ist, so haben wir unter den Elementen {`0 , ... , n0`−1} ein `f’`-Minimum `x`. Dieses muss noch mit dem Element `n0` verglichen werden, um das `f’`-Minimum von {`0 , ... , n0`} zu finden. In den Blättern des Baumes von Abb 10 sind die jeweils zu wählenden Minima eingetragen.

5.2 Beweis des Lemmas `select_ok`

In [Ash16] nimmt Asher die Existenz einer Funktion `select: nat -> component` an. Da Asher die Funktion `select` nur annimmt und nicht implementiert hat, ist Coq über die Funktion `select` außer der Funktionssignatur nichts bekannt. Es können also außer zur Funktionssignatur keine Beweise zu `select` geführt werden. Asher nimmt ein Lemma `select_ok` (siehe weiter unten) an, welches mehr als nur die Funktionssignatur benötigt. Ohne konkretere Annahmen zur Funktion bzw. eine implementierte Funktion `select` ist das Lemma `select_ok` aber nicht beweisbar.
Die Funktion `select` soll anhand der ID einer Komponente im Netzwerk gebündelte Informationen über die Komponente zurückgeben. Der Funktionswert `select i` soll sich dabei unterteilen in:

- `is_s`, ist die Komponente `i` der Startknoten oder nicht?
- `i`, die ID der Komponente `i` und
- `E_i`, die Nachbarn der Komponente `i` und die Distanz zu ihnen.

Asher nimmt das Lemma `select_ok` an:

```
Lemma select_ok: forall (i':set n), (select i').i = i'.
```

Das Lemma besagt, dass die Komponente, die durch `select` ausgewählt wird, die Richtige ist – also die ID hat, mit der wir die Funktion aufrufen. Wir implementieren `select` in Codeblock 49.

Codeblock 49: Die Funktion `select`

```
Definition select (S : set n) : component :=
    {| is_s := (proj1_sig S =? proj1_sig start_i) ; i := S ; E_i := E_dists S |}.
```

Die Funktion formt ein Drei-Tupel. Für `is_s` prüft sie, ob sie der Startknoten ist. Sie übernimmt für `i` den Übergabeparameter. Für `E_i` übernimmt sie die in `E_dist` eingetragenen Distanzen. Mit dieser vorhandenen Implementierung von `select` gelingt uns der Beweis des Lemmas `select_ok`.

6 Diskussion

Ziel dieser Arbeit war es, das Framework aus [Aki18] anzuwenden und zu vereinfachen. Bei der Vereinfachung des Frameworks haben wir uns darauf konzentriert, möglichst Beweisverpflichtungen zu entfernen, die für jeden Algorithmus erfolgen müssen. Es gelang uns, vier Beweisverpflichtungen zu einer neuen zu verschmelzen, die nur einmalig für alle zertifizierenden verteilten Algorithmen gezeigt werden muss. Für diese neue Beweisverpflichtung haben wir eine Implementierung und einen großen Anteil der Verifikation geliefert. Die Anwendung des Frameworks erfolgte an einem Fallbeispiel mit einem nicht einfach verteilbaren Zeugenprädikat, was die Praktikabilität des Frameworks aufzeigt. Weiter konnten wir alle offenen Lemmata der Arbeit Ashers [Ash16] beweisen.

In Abschnitt 6.1 fassen wir die Ergebnisse dieser Arbeit zusammen und ordnen sie ein. Darauf folgen in Abschnitt 6.2 einige unserer Erkenntnisse, die wir im Umgang mit Coq in dieser Arbeit erhalten haben. Im letzten Abschnitt 6.3 beschreiben wir Möglichkeiten, wie auf diese Arbeit aufgebaut werden kann.

6.1 Ergebnisse der Arbeit

Wir greifen hier wieder die Zielstellungen der Einleitung auf:

Framework vereinfachen. In Kapitel 2 erläuterten wir das Konzept des zertifizierenden verteilten Algorithmus und vereinfachten das in [Aki18] vorgestellte Framework. Die folgenden Beweisverpflichtungen aus [Aki18] passten wir folgendermaßen an:

- BV III: Theorem zur verteilten Prüfbarkeit der Konsistenz → entfällt bei uns
- BV IV: zusammenhängender Zeuge wird vom Algorithmus berechnet → entfällt bei uns
- BV V: vollständiger Zeuge wird vom Algorithmus berechnet → entfällt bei uns
- BV VI (i): Teil-Checker prüfen ihre Konsistenz in der Nachbarschaft → BV VI (i): Teil-Checker prüfen verteilt die Konsistenz im gesamten Netzwerk (siehe Abschnitt 2.1.3 und Kapitel 4)

Bei unserer alternativen Konsistenzprüfung entfallen die Prüfungen auf Vollständigkeit und Zusammenhang. Daher können unsere Zeugen allgemeiner sein, da sie weder vollständig noch zusammenhängend sein müssen.

Fallbeispiel Zweifärbbarkeit. In Kapitel 3 wendeten wir das Framework zur formalen Instanzverifikation auf das Fallbeispiel der Zweifärbbarkeit an. Das Fallbeispiel hatte ein komplexeres Zeugenprädikat, als die bisher in [VA17] und [Ash16] verwendeten: wir nutzten im „Graph ist nicht zweifärbbar“-Fall eine Konjunktion eines global verteilten und eines existentiell verteilten Zeugenprädikats. Für das globale Zeugenprädikat konnten wir Teile der Arbeit [VA17] verwenden – in diesem Zuge passten wir Teile dieser Vorarbeit an und bewiesen alle offenen Lemmata der Zeugeneigenschaft. Mit dem Fallbeispiel Zweifärbbarkeit erreichten wir erstmals lückenlos alle einmaligen Beweisverpflichtungen unseres Frameworks an einem komplexen Beispiel. Damit demonstrierten wir, dass der modulare Aufbau des Frameworks hilft, Definitionen oder Beweise aus Modulen anderer Arbeiten wiederzuverwenden und dass auch komplexere Beispiele für das Framework nutzbar sind. Wir sehen daher die bisherigen Fallbeispiele als Grundlage für eine mögliche Bibliothek von zertifizierenden verteilten Basis-Algorithmen, deren Bausteine spätere Arbeiten wieder verwenden können.

Konsistenzprüfung implementieren. Unsere verteilte Konsistenzprüfung wurde in Kapitel 4 implementiert und die Korrektheit bis auf zwei Lemmata bewiesen. Insbesondere falsche negative Ergebnisse der Konsistenzprüfung konnten wir ausschließen. Die offenen Beweise waren für uns im zeitlichen Rahmen der Diplomarbeit nicht mehr zu führen.
Die vereinfachte Prüfung ermöglicht es Entwickelnden von ZVAs, sich nur auf das Interface des Algorithmus’ zu konzentrieren. Beweise der Vollständigkeit des Zeugen oder des Zusammenhangs des Zeugen, die der ZVA ausgibt (siehe [Aki18]), sind nicht mehr notwendig. Dadurch kann der ZVA als Black Box gesehen werden, was dem Prinzip der verifizierten zertifizierenden Algorithmen von [ABMR14] und [Riz15] entspricht.

Beweislücken schließen. Zwei offene Lemmata aus der Arbeit von [Ash16] bewiesen wir in Kapitel 5. Dadurch sind alle Lücken in der Beweisführung in Ashers Arbeit geschlossen. Asher spekulierte, dass Coq alleine eventuell nicht für die Beweisführung der Checker-Implementierung ausreichen könnte. Durch unseren Ansatz der Abbildung von Untermengen von natürlichen Zahlen auf natürliche Zahlen selbst konnten wir zeigen, dass Coq grundsätzlich auch für diese Beweisführungen genutzt werden kann.

Von den Zielen der Arbeit haben wir damit alle, bis auf die komplette Verifikation der Konsistenzprüfung, vollständig erreicht.

6.2 Übergreifende Erkenntnisse beim Beweisen mit Coq

Beim Arbeiten mit dem Beweisassistenten Coq stellten wir bei den Beweisführungen der Kapitel 3, 4 und 5 unserer Arbeit ähnliche Schwierigkeiten fest. Hier listen wir einige davon auf und geben an, welchen Umgang wir mit ihnen empfehlen würden. Die Schwierigkeiten und Lösungen sind zwar intuitiv, aber beim ersten Arbeiten mit Coq bedenkenswert.

Beweise in Coq tauchen häufig mehrstufig in komplexen Fallunterscheidungen ab.

Hier können Beweise für die Beweisenden schnell zu einem mechanischen Manipulieren von Symbolen werden. Für solche Stellen können abstrahierende Definitionen helfen, den Überblick zu bewahren. Ein Beispiel: statt durch eine Faktensammlung über einen Knoten im Baum v: „v ist ein Knoten“, „v hat Abstand 0 zur Wurzel“ und „v ist der eigene Elternknoten“, kann die abstrahierende Definition „v ist *Wurzel* des Baums“ eingeführt werden. Diese Definition kann alle der eben genannten Fakten bündeln und diese häufig ablenkenden Details „verstecken“. Nur an den Stellen, wo die konkreten Eigenschaften der Wurzel gefragt sind, werden sie entpackt.
Es gibt noch andere Vorteile von abstrahierenden Definitionen:

- Theoreme können lesbarer formuliert werden, indem von zu feinen Details abstrahiert wird.

- Flüchtigkeitsfehler bei der Formulierung von Lemmata werden reduziert, da weniger Details zur Formulierung nötig sind.

- Ändern wir die abstrahierende Definition, so ändert sich jede Nutzung der Definition in Beweisen mit. Dieser Punkt folgt dem bekannten Prinzip “don’t repeat yourself!” (vgl. [WAB$^+$14])

Wir haben festgestellt, dass sich Definitionen mit weniger Fallunterscheidungen oder wenigstens ähnlichen Fällen einfacher in Beweisen verhalten. In Beweisen zu der Definition müssen später exakt die gleichen Fälle unterschieden werden.
Insbesondere beim Arbeiten mit VERDI stachen die Funktionen `NetHandler` und `InputHandler` aus Abschnitt 4.2.3 heraus. Diese mussten vielfach in Beweisen aus Kapitel 4 entpackt werden. Wenn man vorweg wenige Fallunterscheidungen in diesen Funktionen definiert, spart man sich hinterher viel Arbeit.

Wenn eine Beweiskette noch sehr viele Lücken hatte, waren wir verleitet, offene Lemmata oder Eigenschaften von Funktionen oder Definitionen durch Axiome zu ersetzen, um sie später zu beweisen. Manche von uns formulierten Axiome erwiesen sich auch nach vielfachem Überdenken als falsch oder ungenau. Dadurch war entweder das Axiom unbeweisbar oder alle auf das Axiom aufbauenden Beweise mussten angepasst werden. Der Arbeitsaufwand nach einer Korrektur fehlerhafter Axiome war meist enorm. Wir empfehlen daher, jedes Axiom sehr genau zu prüfen oder von den Funktionen und Definitionen her Lemmata zu beweisen, statt vom Schluss-Theorem aus.
Ähnliche Auswirkungen kann eine falsche Spezifikation der Schnittstellen des ZVA haben. Bei unseren Beweisketten konnten wir Veränderungen an den Spezifikationen ausgleichen, indem wir die erste Funktion anpassten, die den Input vom ZVA bekommt. Die Spezifikation ist das erste Glied in unserer Beweiskette. Falls sich keine anpassende Funktion für den Input finden lässt, würden sich Veränderungen an der Spezifikation, wie bei Axiomen, durch die gesamte Beweiskette ziehen.

6.3 Ausblick

Zuerst beschreiben wir, wie auf die unterschiedlichen Themen der Arbeit aufgebaut werden könnte, darauf folgen übergreifende Ausblicke.

Framework vereinfachen. In unserem Framework sind nur noch wenige Beweisverpflichtungen für jeden einzelnen Algorithmus zu führen. Eine mögliche Verbesserung des Frameworks liegt darin, auch noch „Prüft der Teil-Checker sein Teil-Prädikat korrekt?“ (BV VI (ii)) zu automatisieren. Dies wäre möglich, wenn wir für die Sprache der Teil-Prädikate eine entscheidbare Logik wählen, für die es einen Entscheidungsalgorithmus gibt. Der Entscheidungsalgorithmus würde in die Teil-Checker eingebaut. Die neue Beweisverpflichtung ist dann die Verifikation des Entscheidungsalgorithmus für die Sprache der Teil-Prädikate. Jedoch nur noch einmalig für alle ZVA. Falls ein ZVA doch eine weniger eingeschränkte Logik für die Teil-Prädikate bräuchte, könnte stattdessen das bisherige Framework dafür angewandt werden.

Konsistenzprüfung implementieren. Im Bereich der verteilten Konsistenzprüfung ist die Terminierung der Prüfung noch ohne Beweis. Wir skizzierten in Abschnitt 4.3.3 einen Ansatz die Terminierung zu beweisen.

Weiterhin könnte die verteilte Prüfung noch optimiert werden: doppelte Belegungen werden aus der Belegungsliste entfernt, Hashwerte werden statt der Belegungen verglichen und jede Komponente prüft die Konsistenz ihres Teilbaums und bricht die globale Prüfung ab, wenn Inkonsistenzen auftreten. Sofern diese Optimierungen durchgeführt werden, sind die Beweise für die Verifikation und Terminierung jedoch komplexer.

Fallbeispiel Zweifärbbarkeit. Um unser Fallbeispiel der Zweifärbbarkeits-Prüfung auf reale Instanzen anzuwenden, könnte ein ZVA implementiert werden, der Zweifärbbarkeit entscheidet. Der Algorithmus könnte vorgehen, wie in Abschnitt 3.1 beschrieben. Bei der Implementierung muss lediglich darauf geachtet werden, dass der ZVA sich in unser Interface einfügt.

Das Fallbeispiel könnte auch generalisiert werden, indem statt Zweifärbbarkeit die k-Färbbarkeit entschieden wird, oder die chromatische Zahl (siehe [EH66]) eines Graphen gefunden wird. Die Prüfung des „Graph ist k-färbbar“-Falles ist ähnlich einfach, wie bei der Zweifärbung: jeder Teil-Checker prüft, dass alle Nachbarn eine andere Farbe haben. Da allerdings schon Dreifärbung zu entscheiden NP-vollständig ist (vgl. [Ste93]), gehen wir davon aus, dass ein Checker für den „Graph ist nicht k-färbbar“-Fall sehr komplex ist.

Beweislücken schließen. Die Implementierung und Verifikation von Teil-Checkern des zertifizierenden Bellman-Ford-Algorithmus aus [Ash16] steht noch aus. Wir empfehlen zuvor die bisherigen Definitionen und Beweise an eine Graphenbibliothek, wie z.B. GRAPHBASICS anzupassen, damit der Beweisaufwand verringert wird.

Bisher extrahiert Coq nur nach OCaml, Scheme und Haskell. Ein interessantes Projekt, welches einen verifizierten Coq-Compiler namens Œuf erstellt, wurde in [MPW^{+}18] gestartet. Da das gleiche Team auch an Verdi arbeitet, würde vielleicht auch der nötige Verdi Shim portiert. In diesem Fall könnte unser Code auch direkt mit Œuf zu Assembler-Code übersetzt werden.

In unserer Arbeit beschränken wir uns auf terminierende Algorithmen, die auf statischen und fehlerfreien Netzwerken laufen. Für Ansätze, wie wir mit nicht-terminierenden, dynamischen oder fehlerhaften Netzwerken umgehen könnten, erweitern wir die Vorschläge aus [Aki18] wie folgt:

Nicht-terminierende Algorithmen Viele verteilte Algorithmen sind nicht terminierend, z.B. das Alternating-Bit-Protokoll (vgl. [BSW69]). Für nicht terminierende Algorithmen könnten wir prüfen, dass das Netzwerk nicht in einem bestimmten Zustand ist – sich also unkorrekt verhält. Das könnte zum Beispiel für selbststabilisierende Algorithmen nützlich sein, die dann eine Fehlerbehebung aktivieren. Dafür stellen wir uns drei mögliche Lösungen vor:

- Nach jedem Verarbeitungsschritt einer Komponente des Netzwerks und vor dem nächsten Schritt einer anderen Komponente wird ein globaler Checking-Prozess gestartet. Diese Lösung wäre jedoch je nach Größe des Netzwerks und Algorithmus sehr langsam.

- Jeder Verarbeitungsschritt einer Komponente startet nur den Teil-Checker der Komponente. In diesem Fall könnten universal-verteilte Teil-Prädikate vom Teil-Checker geprüft werden. Bei Nicht-Erfüllung eines Teil-Prädikates schlägt der Teil-Checker Alarm.

- Ein Hybrid der beiden Varianten wäre, dass in jedem Verarbeitungsschritt des ZVA der entsprechende Teil-Checker universal-verteilte Teil-Prädikate prüft. Nach bestimmten Zeiteinheiten oder einer Anzahl von Verarbeitungsschritten wird eine komplette Prüfung durchgeführt.

Dynamische Netzwerke Nehmen wir ein Netzwerk an, in dem, wenn eine Komponente entfernt oder hinzugefügt wird, die betroffenen Nachbarkomponenten eine Nachricht erhalten. Diese Komponenten aktualisieren so automatisch ihre veränderte Nachbarschaft.
Formale Instanzkorrektheit ist für solche Netzwerke am einfachsten beweisbar, wenn nach jeder Änderung im Netzwerk der gesamte Checking-Prozess neu gestartet wird. Wenn der Checker dann terminiert, hatte er bei dem gesamten Durchlauf ein statisches Netzwerk. Dieses „statische Netzwerk“ entspricht dann den Voraussetzungen, um unser Framework darauf anzuwenden.
Effizienter dagegen wäre es, wenn das Netzwerk berechnet, welche bisher abgelaufenen Prozesse von der Topologie-Änderung betroffen sind und es nur diese wiederholt.

Fehlerhafte Kommunikation In Netzwerken können Nachrichten verloren gehen, verdoppelt werden oder in der Reihenfolge vertauscht werden. Die von uns genutzte Bibliothek VERDI bietet Möglichkeiten an, um mit solchen Fehlerquellen umzugehen. Man beweist zuerst alle Eigenschaften des verteilten Algorithmus' mit der vereinfachenden Annahme der fehlerlosen Kommunikation. VERDI transformiert dann den verteilten Algorithmus, der mit den Fehlern in der Kommunikation, soweit möglich, umgehen kann.

Literatur

[ABMR11] Alkassar, Eyad ; Böhme, Sascha ; Mehlhorn, Kurt ; Rizkallah, Christine: Verification of certifying computations. In: *International Conference on Computer Aided Verification* Springer, 2011, S. 67–82

[ABMR14] Alkassar, Eyad ; Böhme, Sascha ; Mehlhorn, Kurt ; Rizkallah, Christine: A framework for the verification of certifying computations. In: *Journal of Automated Reasoning* 52 (2014), Nr. 3, S. 241–273

[AC11] Armoogum, Sandhya ; Caully, Asvin: Obfuscation techniques for mobile agent code confidentiality. In: *Journal of Information & Systems Management* 1 (2011), Nr. 1, S. 83–94

[Aki18] Akili, Samira: *Verteilte Prüfung der Konsistenz im Rahmen eines Verifikations-Frameworks für Zertifizierende Verteilte Algorithmen*, Humboldt-Universität zu Berlin, Diplomarbeit, 2018

[Ash16] Asher, David: *Verifikation eines zertifizierenden verteilten Algorithmus*, Humboldt-Universität zu Berlin, Diplomarbeit, 2016

[BGH92] Bertsekas, Dimitri P. ; Gallager, Robert G. ; Humblet, Pierre: *Data networks*. Bd. 2. Prentice-Hall International New Jersey, 1992

[Bro41] Brooks, R. L.: On coloring the nodes of a network. In: *Proceedings of the Cambridge Philosophical Society* 37 (1941), S. 194–197

[BSW69] Bartlett, Keith A. ; Scantlebury, Roger A. ; Wilkinson, Peter T.: A note on reliable full-duplex transmission over half-duplex links. In: *Communications of the ACM* 12 (1969), Nr. 5, S. 260–261

[BSZL$^+$18] Breitner, Joachim ; Spector-Zabusky, Antal ; Li, Yao ; Rizkallah, Christine ; Wiegley, John ; Weirich, Stephanie: Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report). In: *Proc. ACM Program. Lang.* 2 (2018), Juli, Nr. ICFP, 89:1–89:16. `http://dx.doi.org/10.1145/3236784`. – DOI 10.1145/3236784. – ISSN 2475–1421. – Eingesehen am 6. Oktober 2018

[CLRS09] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to algorithms*. MIT press, 2009

[Coq] *GitHub-Seite unseres* Coq*-Codes*. `github.com/voellinger/verified-certifying-distributed-algorithms`. – Eingesehen am 6. Oktober 2018

[Coq16] Coq*-Mailinglisten-Thread von Asher*. `sympa.inria.fr/sympa/arc/`

`coq-club/2016-03/msg00171.html`. Version: 2016. – Eingesehen am 6. Oktober 2018

[Coq18] *Dokumentation von* Coq. `coq.inria.fr/documentation`. Version: 2018. – Eingesehen am 6. Oktober 2018

[CRKGLA89] Cheng, Chunhsiang ; Riley, Ralph ; Kumar, Srikanta P. ; Garcia-Luna-Aceves, Jose J.: A loop-free extended Bellman-Ford routing protocol without bouncing effect. In: *ACM SIGCOMM Computer Communication Review* Bd. 19 ACM, 1989, S. 224–236

[Dup01] Duprat, Jean ; Ecole Normale Superieur de Lyon (Hrsg.): *A Coq toolkit for graph theory.* : Ecole Normale Superieur de Lyon, 2001

[EH66] Erdős, Paul ; Hajnal, András: On chromatic number of graphs and set-systems. In: *Acta Mathematica Academiae Scientiarum Hungarica* 17 (1966), Nr. 1-2, S. 61–99

[FZWK17] Fonseca, Pedro ; Zhang, Kaiyuan ; Wang, Xi ; Krishnamurthy, Arvind: An empirical study on the correctness of formally verified distributed systems. In: *Proceedings of the Twelfth European Conference on Computer Systems* ACM, 2017, S. 328–343

[GHS83] Gallager, Robert G. ; Humblet, Pierre A. ; Spira, Philip M.: A distributed algorithm for minimum-weight spanning trees. In: *ACM Transactions on Programming Languages and systems (TOPLAS)* 5 (1983), Nr. 1, S. 66–77

[Gon08] Gonthier, Georges: Formal proof–the four-color theorem. In: *Notices of the AMS* 55 (2008), Nr. 11, S. 1382–1393

[HK07] Heggernes, Pinar ; Kratsch, Dieter: Linear-time certifying recognition algorithms and forbidden induced subgraphs. In: *Nord. J. Comput.* 14 (2007), Nr. 1-2, S. 87–108

[HKPM16] Huet, Gérard ; Kahn, Gilles ; Paulin-Mohring, Christine: *The Coq Proof Assistant A Tutorial*, 2016. – Eingesehen am 6. Oktober 2018

[Jou16] Jourdan, Jacques-Henri: *Verasco: a Formally Verified C Static Analyzer*, Université Paris 7 Diderot, PhD thesis, Mai 2016

[JTL12] Jang, Dongseok ; Tatlock, Zachary ; Lerner, Sorin: Establishing Browser Security Guarantees through Formal Shim Verification. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12).* Bellevue, WA : USENIX, 2012. – ISBN 978–931971–95–9, S. 113–128

[LBC16] Lesani, Mohsen ; Bell, Christian J. ; Chlipala, Adam: Chapar:

certified causally consistent distributed key-value stores. In: *ACM SIGPLAN Notices* Bd. 51 ACM, 2016, S. 357–370

[LD03] Linn, Cullen ; Debray, Saumya: Obfuscation of executable code to improve resistance to static disassembly. In: *Proceedings of the 10th ACM conference on Computer and communications security* ACM, 2003, S. 290–299

[Ler18] Leroy, Xavier ; INRIA Paris (Hrsg.): *The CompCert verified compiler.* : INRIA Paris, 2018

[Lyn96] Lynch, Nancy A.: *Distributed Algorithms.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996. – ISBN 1558603484

[MMNS11] McConnell, Ross M. ; Mehlhorn, Kurt ; Näher, Stefan ; Schweitzer, Pascal: Certifying algorithms. In: *Computer Science Review* 5 (2011), Nr. 2, S. 119–161

[MN98] Mehlhorn, Kurt ; Näher, Stefan: From algorithms to working programs: On the use of program checking in LEDA. In: *International Symposium on Mathematical Foundations of Computer Science* Springer, 1998, S. 84–93

[MPW$^+$18] Mullen, Eric ; Pernsteiner, Stuart ; Wilcox, James R. ; Tatlock, Zachary ; Grossman, Dan: Œuf: minimizing the Coq extraction TCB. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* ACM, 2018, S. 172–185

[Mul18] Mullen, Eric: *Pushing the Limits of Compiler Verification*, University of Washington, PhD thesis, 2018

[NMS$^+$08] Nanevski, Aleksandar ; Morrisett, Greg ; Shinnar, Avraham ; Govereau, Paul ; Birkedal, Lars: Ynot: dependent types for imperative programs. In: *ACM Sigplan Notices* 43 (2008), Nr. 9, S. 229–240

[Ong14] Ongaro, Diego: *Consensus: Bridging theory and practice*, Stanford University, PhD thesis, 2014

[PAC$^+$18] Pierce, Benjamin C. ; Amorim, Arthur A. ; Casinghino, Chris ; Gaboardi, Marco ; Greenberg, Michael ; Hriţcu, Cătălin ; Sjöberg, Vilhelm ; Yorgey, Brent: *Software Foundations. Electronic textbook.* `softwarefoundations.cis.upenn.edu`. Version: 2018. – Eingesehen am 6. Oktober 2018

[Riz15] Rizkallah, Christine: *Verification of Program Computations*, Universität Saarbrücken, Diss., 2015

[RS04] Rogaway, Phillip ; Shrimpton, Thomas: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: *International workshop on fast software encryption* Springer, 2004, S. 371–388

[San06] Santoro, Nicola: *Design and analysis of distributed algorithms.* Bd. 56. John Wiley & Sons, 2006

[Ste93] Steinberg, Richard: The state of the three color problem. In: *Annals of discrete mathematics* Bd. 55. Elsevier, 1993, S. 211–248

[Tea18] Team, Coq D.: *The Coq Proof Assistant Reference Manual*, 2018. `coq.inria.fr/distrib/current/refman`. – Eingesehen am 6. Oktober 2018

[VA17] Völlinger, Kim ; Akili, Samira: Verifying a class of certifying distributed programs. In: *NASA Formal Methods Symposium* Springer, 2017, S. 373–388

[VA18] Völlinger, Kim ; Akili, Samira: On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems* Springer, 2018, S. 161–180

[Vera] *GitHub-Seite des Frameworks* Verdi. `github.com/uwplse/verdi`. `github.com/uwplse/verdi`. – Eingesehen am 6. Oktober 2018

[Verb] *Homepage des* Coq-*Frameworks* Verdi. `verdi.uwplse.org/`. – Eingesehen am 6. Oktober 2018

[Völ17] Völlinger, Kim: Verifying the Output of a Distributed Algorithm Using Certification. In: *International Conference on Runtime Verification* Springer, 2017, S. 424–430

[VR15] Völlinger, Kim ; Reisig, Wolfgang: Certification of Distributed Algorithms Solving Problems with Optimal Substructure. In: *Software Engineering and Formal Methods.* Springer, 2015, S. 190–195

[WAB+14] Wilson, Greg ; Aruliah, Dhavide A. ; Brown, C T. ; Hong, Neil P C. ; Davis, Matt ; Guy, Richard T. ; Haddock, Steven H. ; Huff, Kathryn D. ; Mitchell, Ian M. ; Plumbley, Mark D. et al.: Best practices for scientific computing. In: *PLoS biology* 12 (2014)

[Wie03] Wiedijk, Freek: Comparing mathematical provers. In: *International Conference on Mathematical Knowledge Management* Springer, 2003, S. 188–202

[WWA+16] Woos, Doug ; Wilcox, James R. ; Anton, Steve ; Tatlock, Zachary ; Ernst, Michael D. ; Anderson, Thomas: Planning for change in a formal verification of the Raft consensus protocol. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* ACM, 2016, S. 154–165

[WWP+15] Wilcox, James R. ; Woos, Doug ; Panchekha, Pavel ; Tatlock, Zachary ; Wang, Xi ; Ernst, Michael D. ; Anderson, Thomas: Verdi: a framework for implementing and formally verifying distributed systems. In: *ACM SIGPLAN Notices* Bd. 50 ACM, 2015, S. 357–368

[Zhu13] Zhu, Ming-Yuan: *When is a Proof Irrelevant? The Proof-Irrelevance in PowerEpsilon*. `researchgate.net/publication/308966512`. Version: November 2013. – Eingesehen am 6. Oktober 2018

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 6. Oktober 2018 ……………………………………………………