Implementing a Termination Analysis using Configurable Program Analysis

Masterarbeit

an der Fakultät für Informatik und Mathematik
der Universität Passau

Prüfer

Prof. Dr. Dirk Beyer
Prof. Dr. Christian Lengauer

Sebastian Ott
19. Oktober 2016

Zusammenfassung

Termination ist eine fundamentale und wichtige Eigenschaft jedes Programmes, die daher automatisch geprüft werden sollte. Diese Arbeit stellt hierfür einen modularen Ansatz vor, der auf einer beliebigen Erreichbarkeitsanalyse und einer beliebigen Komponente zur Synthese von Terminations- und Nicht-Terminationsargumenten aus möglicherweise nicht terminierenden Pfaden basiert. Es wird ein Algorithmus zur Kombination dieser beiden Komponenten erläutert sowie dessen Umsetzung und die Integration der Bibliothek LassoRanker zur Analyse der möglicherweise nicht terminierenden Pfade in CPAchecker vorgestellt. Diese Implementierung wird auf Basis von Verifikationsproblemen der SV-COMP 2016 evaluiert. CPAchecker erzielt dabei sehr respektable Resultate, ist jedoch für Programme mit Zeigern noch nicht so effizient wie andere Verifikationswerkzeuge.

Inhaltsverzeichnis

Verzeichnisse **5**
Algorithmusverzeichnis . . . 5
Programmeverzeichnis . . . 5
Abbildungsverzeichnis . . . 5
Tabellenverzeichnis . . . 6

1 Einleitung **7**

2 Verwandte Arbeiten **8**

3 Grundlagen **10**
3.1 Programmtermination . . . 10
3.2 CPAchecker . . . 11
3.2.1 Programmrepräsentation . . . 11
3.2.2 Configurable Program Analyis . . . 13
3.2.3 CPA-Algorithmus . . . 15
3.2.4 Prädikaten-Analyse . . . 17
3.3 LassoRanker . . . 19
3.3.1 Geometrisches Nicht-Terminationsargument . . . 20
3.3.2 Terminationsargumente . . . 22
3.4 Komposition von Terminationsargumenten . . . 28

4 Implementierung **30**
4.1 Algorithmus . . . 30
4.2 Einweben von Terminationsargumenten in Programme . . . 32
4.3 Synthese von Terminations- und Nicht-Terminationsargumenten . . . 34
4.3.1 Konstruktion von Lassos aus Fehlerpfaden . . . 34
4.3.2 Transformation von Terminationsargumenten . . . 37
4.4 Export von Terminations- und Nicht-Terminationsargumenten . . . 38
4.5 Konfiguration . . . 38
4.6 Integration der Prädikaten-Analyse . . . 40
4.7 Einschränkungen . . . 41

5 Evaluation **43**
5.1 Vergleich linearer und nicht linearer Analyse von Lassos . . . 44

5.2 Vergleich mit anderen Verifikationswerkzeugen 50
5.2.1 Schleifen-Programme 50
5.2.2 Schleifen-Programme ohne Zeiger 55

6 Zusammenfassung **58**

Literatur **59**

Danksagung **65**

Erklärung zur Masterarbeit **66**

Verzeichnisse

Algorithmusverzeichnis

1 CPA-Algorithmus 16

2 Terminationsalgorithmus 31

Programmeverzeichnis

3.1 C-Programm mit Schleife und Verzweigung 12
3.2 C-Programm mit Geschachtelter Abstiegsrelation 27

Abbildungsverzeichnis

3.1 CFA von Abbildung 3.1 12
3.2 Lasso-Programm 20
3.3 Werte der Variablen von Program 3.2 über die Schleifeniterationen 27

5.1 Nach CPU-Zeit sortierte korrekte Ergebnisse der Konfigurationen *termination-linear* und *termination-non-linear* 46
5.2 Streudiagramm der CPU-Zeiten der Konfigurationen *termination-linear* und *termination-non-linear* 48
5.3 Streudiagramm des Speicherverbrauchs der Konfigurationen *termination-linear* und *termination-non-linear* 49
5.4 Nach CPU-Zeit sortierte korrekte Ergebnisse der Verifikationswerkzeuge . . 53
5.5 Nach CPU-Zeit sortierte akkumulierte Punkte der Verifikationswerkzeuge gemäß des Schemas der SV-COMP 2016 54
5.6 Nach CPU-Zeit sortierte korrekte Ergebnisse der Verifikationswerkzeuge für Programmen ohne Zeiger 56

Tabellenverzeichnis

5.1 Resultate der Konfigurationen *termination-linear* und *termination-non-linear* 45
5.2 Resultate der Verifikationswerkzeuge 51
5.3 Ressourcenverbrauch der Verifikationswerkzeuge 52
5.4 Resultate der Verifikationswerkzeuge für Programme ohne Zeiger 56

1 Einleitung

Das uniforme Halteproblem ist in der Informatik sehr berühmt und stammt noch aus einer Zeit vor der Erfindung des modernen Computers. Trotzdem ist es heute noch sehr aktuell, da das Terminationsverhalten jedes Programmes von großer Bedeutung in praktischen Anwendungen ist. Wenn beispielsweise eine Routine eines Gerätetreibers nicht terminiert, kann es zum Einfrieren der Rechners kommen oder eine Interaktion mit dem Benutzer wird nicht vollständig verarbeitet. Auch im Kontext von Anwendungsprogrammen ist Nicht-Termination in den meisten Fällen unerwünscht und die daraus resultierenden Konsequenzen werden wohl meistens negativ wahrgenommen. Daher entstand der Wunsch automatische Beweise im Bezug auf das Terminationsverhalten von Computerprogrammen führen zu können. Diese Arbeit beschäftigt sich mit einer Terminationsanalyse basierend auf einer konfigurierbaren Programmanalyse sowie einer Bibliothek zur Synthese von Terminations- und Nicht-Terminationsargumenten, die beide flexibel ausgetauscht werden können. Ein zu diesem Zweck entwickelter Algorithmus wird vorgestellt und durch dessen Umsetzung in CPAchecker die praktische Umsetzbarkeit des Konzeptes gezeigt.

Diese Arbeit untergliedert sich in sechs Kapiteln, von denen dieses das Erste ist. Zunächst werden in Kapitel 2 einige Veröffentlichungen kurz vorgestellt, die in Zusammenhang mit dieser Arbeit stehen. In Kapitel 3 werden Grundlagen zu Programmtermination, der Funktionsweise von CPAchecker und der Theorie hinter der Synthese von Terminations und Nicht-Terminationsargumenten mittels der Bibliothek LassoRanker erläutert. Das Kapitel 4 widmet sich der Implementierung in CPAchecker, die in Kapitel 5 evaluiert wird, indem zwei Konfigurationen gegenübergestellt werden und anschließend eine der beiden mit anderen Verifikationswerkzeugen verglichen wird. Die Arbeit schließt mit einer Zusammenfassung der Ergebnisse und einem Ausblick in Kapitel 6.

2 Verwandte Arbeiten

TERMINATOR war das erste vollautomatische Softwareverifikationswerkzeug, das in der Praxis in der Lage war Termination oder Nicht-Termination von großen[1] C-Programmen mit komplexen Kontrollstrukturen und Zeigern zu beweisen. Es wurde erfolgreich eingesetzt, um Nicht-Termination in Windows-Geräte-Treibern aufzuspüren, wodurch die Stabilität des Betriebssystemes verbessert werden konnte [20, 21]. Ein Verfahren zur Synthese von Terminations-Argumenten für Programme, deren Termination von der Veränderung des dynamischen Speichers abhängt, wird in [5] vorstellt. Dabei kann es sich zum Beispiel um verkettete Listen handeln, die durch Einfügen einer Längen- und Indexmetavariable leichter analysierbar werden. Der Nachfolger T2 von TERMINATOR ist unter anderem auch in der Lage Programme mit nicht linearen Zuweisungen durch Divergenzbeweise verarbeiten [2]. Die Verringerung der notwendigen Laufzeit kann durch die gleichzeitige Synthese von Terminationsargumenten und diese unterstützende Invarianten erreicht werden, indem zwei das Programm repräsentierende und miteinander verbundene Graphen verwendet werden, so dass Programmzustände entweder auf Grund der gefundenen Invariante nicht erreichbar sind oder wegen eines Terminationsargumentes nicht weiter betrachtet werden müssen [15].

[41] beschäftigt sich mit der Adaption der Konzepte der Prädikatenabstraktion für Erreichbarkeitsprobleme, um sie für Terminationsanalysen zu verwenden. Ziel ist ein endliches abstraktes Programm in Form einer Menge von abstrakten Transitionen. Eine weitere Möglichkeit ist Abstraktion der Abstiegsfunktion eines Programmes in Kombination mit gewöhnlicher Prädikatabstraktion. Wenn die Abstraktion der Abstiegsfunktion fehlschlägt, ist zu prüfen, ob die Prädikaten- oder Abstiegsabstraktion zu verfeinern ist [4]. [24] beschreibt ein Verfahren, welches die von SAT- und SMT-Solvern bekannten konflikt-getriebenen Lernverfahren auf Beweisverfahren für Nicht-Termination überträgt, so dass die Präzision der Analyse verbessert werden kann. Ferner basieren einige neuere Ansätze auf Lösungsverfahren für linearen Rekursionsgleichungen [39] und dem Berechnen Rekurrenter Mengen durch die kontinuierliche Verfeinerung mittels einer Rückwärts- und einer Vorwärtsanalyse [3].

Weitere Arbeiten befassen sich mit Programmen, die neben linearen auch polynomiale Ausdrücke enthalten dürfen. [47] beschreibt eine Möglichkeit Terminationsbeweise von Programmen mit poylnominalen Schleifenbedingungen und ansonsten linearen Termen durch

[1] Programme mit mehr als 20000 Zeilen Quellcode.

Reduktion auf semialgebraische Systeme zu führen. Auf Bäumen basierende Terminationsargumente erlauben auch polynomiale Terme in Zuweisungen [13]. Solche Ansätze lassen sich nicht nur für imperative sondern auch für logische Programmiersprachen benutzen [38].

SeaHorn extrapoliert Pfade, deren Termination es bewiesen hat, zu Kandidaten für Terminationsargumente, welche für eine möglichst große Menge von Pfaden gültig sind [46]. Die gefunden Abstiegsfunktionen werden als Zähler implementiert, die in jeder Schleifeniteration dekrementiert werden und nicht kleiner als Null werden dürfen. Außerdem ist es in der Lage inkonsistenten Quellcode zu finden, dessen Erreichbarkeit Nicht-Termination zur Folge hätte und damit ein starkes Indiz für einen Fehler im Programm oder überflüssigen Quellcode darstellt. Der Ansatz ist deutlich feingranularer als bisherige Verfahren, da er den Kontrollfluss der Schleife nicht überapproximiert, sondern auf Basis von Hornklauseln die Pfadüberdeckung berechnet [31].

AProVE unterstützt durch die Verwendung von LLVM zahlreiche Programmiersprachen und prüft im Falle von C vor der eigentlichen Terminationsanalyse, ob das Programm auf Grund von fehlerhaftem Zugriff auf dynamischen Speicher undefiniertes Verhalten zeigen kann und somit nicht als terminierend betrachtet werden sollte [26]. Die Terminationsanalyse beherrscht außerdem Zeiger [43] und kann Datentypen als Bitvektoren statt mathematischen Ganzen Zahlen repräsentieren, so dass beispielsweise Überläufe von nicht Vorzeichen behafteten Datentypen korrekt modelliert werden [29]. Am Beispiel Java wird gezeigt, wie Programme mit zyklischen Datenstrukturen in Termersetzungssysteme, welche die interne Repräsentation der Programme in AProVE sind, übersetzt werden können und anschließend deren Termination gezeigt werden kann [16].

Ultimate Automizer verwendet einen auf Automaten basierenden Ansatz zur Extraktion von möglicherweise nicht-terminierenden Schleifen, für die mittels der integrierten Bibliothek LassoRanker Terminations- und Nicht-Terminationsargumente synthetisiert werden [11, 27]. Bei Terminationsargumenten handelt es sich um auf affinen Funktionen basierten Abstiegsrelationen, die auf Basis von Schablonen und mittels eines SMT-Solvers berechnet werden. Um bereits als terminierend klassifizierte Pfade von der weiteren Analyse ausschließen zu können, ist die effiziente Berechnung des Kompliments eines Automaten wichtig [12].

3 Grundlagen

Dieses Kapitel erläutert die wichtigsten Konzepte, die Bibliotheken LassoRanker und zentrale Komponenten von CPAchecker, welche durch die Terminierungsanalyse verwendet werden.

3.1 Programmtermination

Ein Programm terminiert, wenn jede mögliche Ausführung aus endlichen vielen Schritten besteht, anderenfalls terminiert es nicht [18]. Damit handelt es sich um eine Lebendigkeitseigenschaft eines Programmes [1]. Da auf realen Computern der verfügbare Speicher endlich ist, kann ein Programm nicht unendlich viele verschiedene Zustände annehmen. Daher wird ein nicht terminierendes Programm nach endlich vielen Schritten einen Programmzustand erreichen, der vollkommen identisch mit einem früheren ist.

Die Frage, ob eine gegebenes Programm nicht terminiert, ist im allgemeinen nicht entscheidbar [45][1]. Es gibt demnach beweisbar mindestens ein Programm, dessen Terminationsverhalten nicht berechnet werden kann. In praktischen Anwendungen sind die Ressourcen nicht nur endlich, sondern tatsächlich beschränkt, so dass es Programme gibt, die zwar theoretisch analysierbar sind, aber in der Praxis dafür nicht die notwendigen Ressourcen zur Verfügung stehen. Für bestimmte Klassen von Programmen ist das Halteproblem jedoch durchaus berechenbar. Es kann gezeigt werden, dass das Halteproblem für einfach lineare Schleifen-Programme[2] entscheidbar ist [14, 44]. Daher konzentriert sich diese und zahlreiche andere Arbeiten auf Programme, deren Arithmetik linear ist.

In strukturierten imperativen Programmiersprachen kann Nicht-Termination von endlichen Programmen nur durch Schleifen, Rekursion und Sprunganweisungen[3] verursacht werden, sofern angenommen wird, dass es immer einen Berechnungsfortschritt gibt. Das heißt, dass alle Programme ohne solche Kontrollstrukturen trivialerweise terminieren. In

[1] Streng genommen beweist Turing in [45] nicht die Unentscheidbarkeit des Halteproblems, dies ist jedoch eine unmittelbare Folge.

[2] Programme der Form $while(Bx > b \wedge Cx \geq c)\{x := Ax + a\}$, mit $A, B, C \in \mathbb{R} \times \mathbb{R}$, $a, b, c \in \mathbb{R}$ und x Vektor der Programmvariablen. Gleiches gilt für die rationalen Zahlen und, falls $C = 0 \wedge c = 0$ gilt, ebenso für die ganzen Zahlen.

[3] Typischerweise darf das Ziel einer Sprunganweisung nicht außerhalb der aktuellen Methode liegen. Ausnahmen wie die Anweisung `longjmp` aus C werden in dieser Arbeit nicht betrachtet.

manchen Sprachen wie zum Beispiel C kann außerdem undefiniertes Verhalten auftreten, das unter anderem Nicht-Termination verursachen kann. Im folgenden wird dies jedoch explizit ausgeschlossen, obwohl das Nichtvorhandensein von undefiniertem Verhalten eine notwendige Bedingung für die Termination eines C-Programmes ist, die gegebenenfalls vor oder nach einer Terminationsanalyse separat geprüft werden kann.

3.2 CPAchecker

CPAchecker[4] [8] ist ein in Java[5] geschriebenes Framework für konfigurierbare Softwareverifikation, das unter der Apache-Lizenz 2.0 verfügbar ist. Es bietet zahlreiche Analysen und unterstützt die Programmiersprachen C und Java. Die im folgenden beschriebenen grundlegenden Konzepte von CPAchecker wurden zum Teil zuerst in seinem Vorgänger Blast[6] implementiert.

3.2.1 Programmrepräsentation

Das zu analysierende Programm wird intern als Kontroll-Fluss-Automat (CFA) dargestellt, der aus einer endlichen Menge L von Programmstellen zur Modellierung des Befehlszähler pc, einem initialen Wert des Befehlszähler pc_0 sowie einer endlichen Menge $G \subseteq L \times Ops \times L$ von Kontrollfluss-Kanten zur Modellierung der Programmanweisungen. Dabei ist Ops die Menge der Möglichen Programmanweisungen einer Programmiersprache, wobei komplexe Anweisungen geteilt werden, sodass wir uns im wesentlichen auf Annahmen und Zuweisungen beschränken können[7]. Sei X die Menge der Programmvariablen und C die Menge der konkreten Programmzustände. Ein konkreter Programmzustand $c \in C$ weist jeder Variable aus der Menge $X \cup \{ pc \}$ einen Wert zu. Durch $g \in G$ wird eine Transitionsrelation $\xrightarrow{g} \subseteq C \times g \times C$ definiert. Die vollständige Transitionsrelation $\rightarrow$ ist durch die Vereinigung über alle Kanten geben:

$$\rightarrow \; = \bigcup_{g \in G} \xrightarrow{g} \tag{3.1}$$

$c \xrightarrow{g} c'$ und $c \rightarrow c'$ sind eine abkürzende Schreibweise für $(c, g, c') \in \rightarrow$ sowie $\exists g \in G : (c, g, c') \in \rightarrow$. Damit lässt sich die Erreichbarkeit eines konkreten Zustandes c_n von einer sogenannten Region $r \subset C$ definieren:

$$c_n \in Reach(r) \Leftrightarrow \exists c_0 \in r, \langle c_0, c_1, ..., c_n \rangle : \forall_{i=1}^{n} : c_{i-1} \rightarrow c_i \tag{3.2}$$

[4] `https://cpachecker.sosy-lab.org/`
[5] Version 1.8
[6] `http://www.sosy-lab.org/~dbeyer/Blast/index-epfl.php`
[7] Die Analyse von interprozeduralen Programmen wird ebenfalls unterstützt.

```
1  int main() {
2    int a = 9;
3    int b = -3;
4
5    while (a > 0) {
6      if (b > 0) {
7        a = a - b;
8      } else {
9        b = b + 1;
10     }
11   }
12
13   return 0;
14 }
```

Program 3.1: C-Programm mit Schleife und Verzweigung

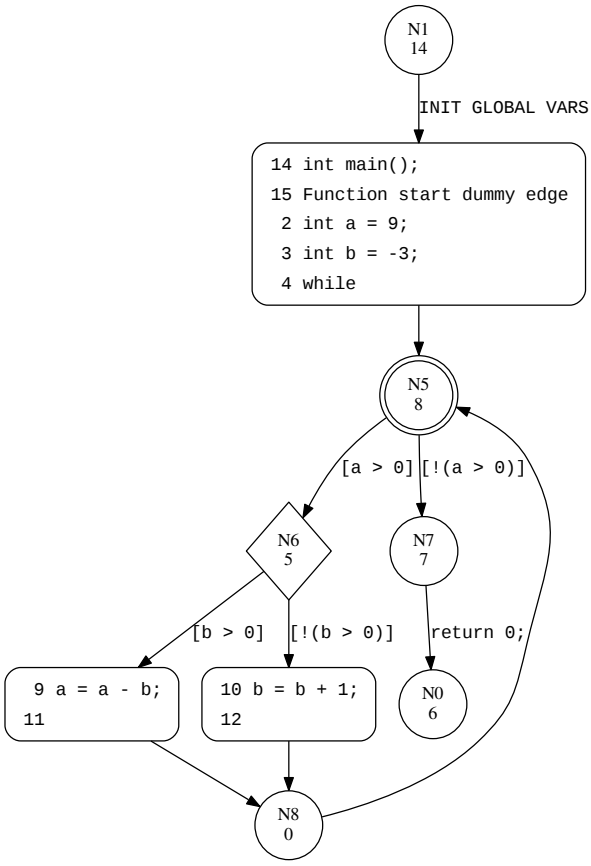

Abbildung 3.1: CFA von Abbildung 3.1

Dabei bezeichnet $\langle c_0, c_1, ..., c_n \rangle$ eine Folge von konkreten Zuständen aus C. [6, 7]

CPAchecker verwendet den C-Parser des Eclipse-Projektes CDT und transformiert den erzeugten Syntax-Baum in einen CFA. Abbildung 3.1 zeigt den von CPAchecker erzeugten CFA des Programms 3.1 auf der linken Seite neben der Abbildung. Die lineare Kette von Anweisungen zwischen *N1* und *N5* ist zusammengefasst. *N5* ist der Kopf der Schleife und als solcher mit einem doppelten Rand gezeichnet. Mit der Kante zu *N7* verlässt der Kontrollfluss die Schleife. Nach Knoten *N6* verzweigt sich der Kontrollfluss, wodurch die `if-else`-Struktur aus dem Programm abbildet wird. Die Bedingungen stehen an den ausgehenden Kanten und sind immer von eckigen Klammern umgeben. Außerdem ist zu erkennen, dass es auch Kanten gibt, die keine Anweisungen tragen und nur der Verbindung von Knoten dienen.

3.2.2 Configurable Program Analyis

Eine *Configurable Program Analysis* (CPA) ist ein Tupel $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$ bestehend aus einer abstrakten Domäne D, einer Transferrelation $\rightsquigarrow$, einem Operator $merge$ zum Kombinieren zweier Elemente und dem Abbruch-Operator $stop$:

1. Die *abstrakte Domäne* $D = (C, \mathcal{E}, [\![\cdot]\!])$ besteht aus der Menge C der konkreten Zustände, einem Halbverband $\mathcal{E}$ sowie einer Konkretisierungsfunktion $[\![\cdot]\!]$. Der Halbverband $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ ist durch die Menge E der Elemente, das größte Element $\top \in E$, das kleinste Element $\bot \in E$, eine partielle Ordnung $\sqsubseteq \subseteq E \times E$ und eine totale Funktion $\sqcup : E \times E \to E$ als Vereinigung-Operator gegeben. Die abstrakten Zustände der Analyse sind die Element des Halbverbandes E, welche durch die Konkretisierungsfunktion $[\![\cdot]\!] : E \to 2^C$ auf eine Menge von konkreten Zuständen abgebildet werden, wodurch die Bedeutung jener definiert wird. Damit die Analyse korrekt ist, müssen folgende Bedingungen erfüllt sein:

$$[\![\top]\!] = C \wedge [\![\bot]\!] = \emptyset \tag{3.3}$$

$$\forall e, e' \in E : e \sqsubseteq e' \implies [\![e]\!] \subseteq [\![e']\!] \tag{3.4}$$

$$\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!] \tag{3.5}$$

   Die Konkretisierung des größten und kleinsten Elementes werden durch (3.3) sinnvoll definiert. Die partielle Ordnung $\sqsubseteq$ muss wegen (3.4) mit der Konkretisierungsfunktion $[\![\cdot]\!]$ verträglich sein. Auf Grund von (3.5) ist der Vereinigungs-Operator $\sqcup$ präzise oder überapproximiert. Insgesamt legt die abstrakte Domäne das Analyseziel einer CPA fest.

2. Durch die *Transferrelation* $\rightsquigarrow \subseteq E \times G \times E$ wird die Menge der abstrakten Nachfolgezustände eines abstrakten Zustandes definiert. Jede Transition ist mit einer CFA-Kante g gekennzeichnet. $e \overset{g}{\rightsquigarrow} e'$ und $e \rightsquigarrow e'$ sind abkürzende Schreibweisen für $(e, g, e') \in \rightsquigarrow$ sowie $\exists g \in G : (e, g, e') \in \rightsquigarrow$. Es werden die folgenden Eigenschaften gefordert:

$$\forall e \in E : \exists e' \in E : e \rightsquigarrow e' \tag{3.6}$$

$$\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{\, c' \mid c \xrightarrow{g} c' \,\} \tag{3.7}$$

   Durch (3.6) wird die Totalität der Transferrelation gewährleistet, so dass die Analyse Fortschritt erzielen kann. Mit (3.7) ergibt sich eine präzise oder überapproximierende Transferrelation.

3. Der Operator $merge : E \times E \to E$ fasst die Information zweier abstrakter Zustände zusammen. Dabei wird $e' \sqsubseteq merge(e, e')$ gefordert um die Korrektheit der Analyse zu garantieren, sodass das Ergebnis abhängig von e zwischen e' und $\top$ liegt. *merge* ist weder kommutativ noch ist es dasselbe wie der Operator $\sqcup$. *merge* kann jedoch auf $\sqcup$ aufbauen. Zwei häufig verwendete Varianten sind $merge^{sep}(e, e') = e'$ sowie $merge^{join}(e, e') = e \sqcup e'$.

4. Der Abbruch-Operator $stop : E \times 2^E \to \mathbb{B}$ überprüft, ob der als erstes Argument gegebene abstrakte Zustand vollständig von der als zweites Argument gegebenen Menge von abstrakten Zuständen abgedeckt ist. Es gibt zahlreiche Möglichkeiten *stop* zu implementieren, in CPAchecker finden jedoch die folgenden beiden Versionen häufig Verwendung:

$$stop^{sep}(e, R) = (\exists e' \in R : e \sqsupseteq e') \tag{3.8}$$

$$stop^{join}(e, R) = (e \sqsupseteq \bigsqcup_{e' \in R} e') \tag{3.9}$$

   Dabei ist zu beachten, dass $stop^{join}$ eine Potenzmengen-Domäne[8] erfordert.

Der Berechnungsaufwand und die Präzision einer CPA werden durch diese vier Komponenten beeinflusst. Dadurch ergibt sich eine große Flexibilität bei der Implementierung und Zusammenstellung der Komponenten.

Ein Beispiel ist die *LocationCPA* $\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, merge_{\mathbb{L}}, stop_{\mathbb{L}})$ mit der Aufgabe die Erreichbarkeit von CFA-Knoten zu verfolgen. Die Abstrakte Domäne basiert auf dem flachen Halbverband der Menge L von CFA-Knoten. Für die Transferrelation gilt folgendes:

$$l \overset{g}{\rightsquigarrow} l' \iff \exists g \in G : g = (l, op, l') \tag{3.10}$$

$$l \overset{g}{\rightsquigarrow} \bot \iff \nexists g \in G : g = (l, op, l') \tag{3.11}$$

Dabei wird die Operation einer CFA-Kante ignoriert. Weiter sind $merge_{\mathbb{L}} = merge^{sep}$ und $stop_{\mathbb{L}} = stop^{sep}$ definiert.

Außerdem gibt es die *CompositeCPA* zum kombinieren von mehreren CPAs, deren abstrakte Domäne das kartesische Produkt der Domänen der in ihr enthaltenen CPAs ist. Die anderen Komponenten sind entsprechend definiert. Eine präzisere Transferrelation der CompositeCPA wird erreicht, indem der Austausch von Informationen zwischen zwei CPAs durch einen Operator $\downarrow: E_1 \times E_2 \to E_1$ ermöglicht wird, wobei $\downarrow (e, e') \sqsupseteq e$ gelten muss.

[8] Für eine Potenzmengen-Domäne gilt $[\![e \sqcup e']\!] = [\![e]\!] \cup [\![e']\!]$.

Zur Kombination einer CPA mit einer anderen ist es auch möglich, dass eine CPA als Hülle um eine zweite fungiert. Sowohl die Operatoren als auch die Transferrelation benutzen die entsprechenden Komponenten der inneren CPA und erledigen davor und danach eigene Aufgaben. Der Abstrakte Zustand der äußeren CPA besteht aus dem abstrakten Zustand der inneren CPA und der für die äußere CPA spezifischen Informationen. Ein Beispiel ist die *ARGCPA*, die während der Analyse einen abstrakten Erreichbarkeitsgraphen (ARG) aufbaut, in dem jeder abstrakte ARG-Zustand neben dem gekapselten Zustand der inneren CPA unter anderem Referenzen auf Eltern- und Kinderzustände speichert. Typischerweise verwenden Analysen die *ARGCPA* als äußerste Analyse.

Die ursprüngliche Idee einer CPA wurde in [6] veröffentlicht, wo der interessierte Leser weitere Details finden kann.

Eine Erweiterung ist das Konzept einer CPA mit dynamischer Anpassung der Präzision (CPA+[9]) $\mathbb{D} = (D, \Pi \rightsquigarrow, merge, stop, prec)$, die zusätzlich die Menge Π der Präzisionen und den Operator $prec : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ umfasst, der für einen abstrakten Zustand, seine Präzision und alle bisher erreichten abstrakten Zustände und deren Präzisionen einen neuen abstrakten Zustand mit einer neuen Präzision berechnet. Dabei muss folgende Bedingung gewährleistet sein:

$$\forall e, \hat{e} \in E, \pi, \hat{\pi} \in \Pi, R \subseteq E \times \Pi : (\hat{e}, \hat{p}) = prec(e, \pi, R) \implies [\![e]\!] \subseteq [\![\hat{e}]\!] \tag{3.12}$$

Ein Tupel $(e, \pi) \in E \times \Pi$ bezeichnet einen abstrakten Zustand e zusammen mit seiner Präzision p, die zur Berechnung des Zustandes verwendet wurde. Die Operatoren *merge* sowie *stop* erhalten entsprechend ein zusätzliches Argument $\pi \in \Pi$ und die Transferrelation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ ist nun ein Viertupel. Die Erweiterung zur CPA+ mit dynamischer Anpassung der Präzision ist erstmals in [7] beschrieben worden.

3.2.3 CPA-Algorithmus

Zur Berechnung der Erreichbarkeit von abstrakten Programmzuständen verwendet CPAchecker den *CPA-Algorithmus*, der vereinfacht als Pseudocode in Algorithmus 1 dargestellt ist. Das Ergebnis des Verfahrens ist eine Überapproximation der vom initialen Zustand e_0 erreichbaren abstrakten Zustände. Es werden zwei Mengen von abstrakten Zuständen mit deren Präzisionen verwendet: *reached* beinhaltet alle bisher gefundenen erreichbaren Zustände und *waitlist* enthält alle noch zu bearbeitenden Zustände. Beide enthalten zu Beginn den initialen Zustand e_0 sowie die dazugehörige Präzision π_0. Solange *waitlist* nicht leer ist, wird ein abstrakter Zustand mit Präzision (e, π) entnommen und in Zeile 6 dessen

[9] In CPAchecker und dieser Arbeit findet CPA+ Verwendung und mit CPA ist im folgenden CPA+ gemeint.

Nachfolger Zustände unter Verwendung der Transferrelation $\rightsquigarrow$ berechnet. Anschließend wird in Zeile 7 für jeden Nachfolger e' dessen Präzision angepasst, bevor der resultierende Zustand $\hat{e}$ in Zeile 10 mit jedem abstrakten Zustand aus *reached* kombiniert und gegebenenfalls das Ergebnis e_{new} zusammen mit der Präzision des Nachfolgers $\hat{\pi}$ in *waitlist* und *reached* gegen den verwendeten Zustand e'' und seine Präzision π'' ausgetauscht wird. Der letzte Schritt in der äußeren Schleife ist in Zeile 17 die Überprüfung, ob der neue Zustand e_{new} in *waitlist* und *reached* eingefügt werden muss, weil er nicht vollständig von den bereits erreichten abstrakten Zuständen in *reached* abgedeckt wird.

Algorithm 1 CPA-Algorithmus

Input: CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$, initialer abstrakter Zustand $e_0 \in E$ mit Präzision $\pi_0 \in \Pi$
Output: eine Menge von erreichbaren abstrakten Zuständen

```
1:  waitlist := { (e_0, π_0) }
2:  reached := { (e_0, π_0) }
3:  while waitlist ≠ ∅ do
4:    (e, π) := waitlist.pop()
5:
6:    for each e' : e ⇝ (e', π) do
7:      (ê, π̂) := prec(e', π)
8:
9:      for each (e'', π'') ∈ reached do
10:       e_new := merge(ê, e'', π̂)
11:       if e_new ≠ e'' then
12:         waitlist := (waitlist ∪ { (e_new, π̂) }) \ { (e'', π'') }
13:         reached := (reached ∪ { (e_new, π̂') }) \ { (e'', π'') }
14:       end if
15:     end for
16:
17:     if ¬stop(e' { e | (e, ·) ∈ reached }, π̂) then
18:       waitlist := waitlist ∪ { (ê, π̂) }
19:       reached := reached ∪ { (ê, π̂) }
20:     end if
21:   end for
22: end while
23: return { e | (e, ·) ∈ reached }
```

Es kann gezeigt werden, dass die Menge von abstrakten Zuständen als Resultat des Aufrufs $CPA(\mathbb{D}, e_0, \pi_0)$ des CPA-Algorithmus mit der CPA $\mathbb{D}$ und dem initialen Zustand e_0 mit Präzision π_0 eine Überapproximation der konkreten Zustände ist, die von einem durch den initialen abstrakten Zustand repräsentierten konkreten Zustand aus erreichbar sind:

$$\bigcup_{e \in CPA(\mathbb{D}, e_0, \pi_0)} [\![e]\!] \supseteq Reach([\![e_0]\!]) \tag{3.13}$$

Der CPA-Algorithmus wurde ursprünglich in [6] beschrieben. Die in Zusammenhang mit CPA+ stehenden Erweiterungen können in [7] nachgelesen werden.

3.2.4 Prädikaten-Analyse

Die Prädikaten-Analyse nutzt Prädikatenabstraktion und repräsentiert Prädikate als SMT-Formeln, zu deren Verarbeitung die Bibliothek JavaSMT[10] verwendet wird. Mit $\mathcal{P}$ wird im Folgenden die Menge der Prädikate über die Variablen des analysierten Programmes in der verwendeten Logik bezeichnet. Die Prädikaten-Analyse basiert auf der *PredicateCPA* $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, merge_{\mathbb{P}}, stop_{\mathbb{P}}, prec_{\mathbb{P}})$, deren Bestandteile im folgenden erklärt werden:

1. Die abstrakte Domäne $D_{\mathbb{P}} = (C_{\mathbb{P}}, \Pi, [\![\cdot]\!]_{\mathbb{P}})$ definiert den Verband der abstrakten Zustände $\mathcal{E}_{\mathbb{P}} = (E_{\mathbb{P}}, \top, \bot, \sqcup)$, dessen Elemente $(\psi, l^{\psi}, \varphi) \in \mathcal{P} \times (L \cup \{ l_T \}) \times \mathcal{P}$ aus einer Abstraktionsformel ψ, einem CFA-Knoten l^{ψ} und einer Pfadformel φ bestehen. Dabei ist ψ eine boolesche Kombination von Prädikaten aus der Menge $\pi(l^{\psi})$, die durch die Präzision des abstrakten Zustandes, von dem die Abstraktion berechnet wurde, und den CFA-Knoten l^{ψ}, an dem diese Abstraktion berechnet wurde, gegeben ist. φ ist eine Formel die einige oder alle Pfade von l^{ψ} zum aktuellen Knoten repräsentiert.

2. Die Menge der Präzisionen $\Pi_{\mathbb{P}} = \{ \pi : L \to 2^{\mathcal{P}} \}$ enthält Funktionen, die CFA-Knoten auf Teilmengen der Prädikate $\mathcal{P}$, welche Aussagen über die Menge X der Programmvariablen treffen, abbilden.

3. Die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ operiert nur auf der Pfadformel, so dass für eine Kante $g = (l, op, l')$ und zwei abstrakte Zustände $e = (\psi, l^{\psi}, \varphi)$ und $e' = (\psi', l^{\psi\prime}, \varphi')$

   $$(e, g, e') \in \rightsquigarrow_{\mathbb{P}} \Leftrightarrow \psi' = \psi \wedge l^{\psi\prime} = l^{\psi} \wedge \varphi' = SP_{op}(\varphi)) \tag{3.14}$$

   gilt. $SP_{OP}(\varphi)$ berechnet aus φ und der Operation *op* die stärkste Nachbedingung φ'.

4. Der Operator $merge_{\mathbb{P}}$ kombiniert die Pfadformeln zweier Zustände $e = (\psi, l^{\psi}, \varphi)$ und $e' = (\psi', l^{\psi\prime}, \varphi')$ disjunktiv, wenn Abstraktionsformel und Abstraktionsknoten gleich sind:

   $$merge_{\mathbb{P}}(e, e', \pi) = \begin{cases} (\psi', l^{\psi\prime}, \varphi \vee \varphi') & \text{falls } \psi' = \psi \wedge l^{\psi\prime} = l^{\psi} \\ e' & \text{sonst} \end{cases} \tag{3.15}$$

[10] `https://github.com/sosy-lab/java-smt`

5. $stop_{\mathbb{P}} = stop^{sep}$.

6. Der Operator $prec_{\mathbb{P}}$ führt die Berechnung der Abstraktion am Ende eines Blocks durch. Das Blockende legt der Operator $blk : E \times G \to \mathbb{B}$ fest. Wenn l der aktuelle CFA-Knoten ist, gilt folgendes:

$$prec_{\mathbb{P}}(e, \pi) = \begin{cases} ((\psi \wedge \varphi)^{(\pi(l))}, l, true) & \text{falls } blk(e, g) \\ e & \text{sonst} \end{cases} \tag{3.16}$$

   Dabei ist $(\psi \wedge \varphi)^{(\pi(l))}$ die boolesche oder kartesische Prädikatenabstraktion für die Konjunktion der Pfad- und Abstraktionsformel sowie die Prädikatenmenge $\pi(l)$ der Präzision π am aktuellen CFA-Knoten l. Die Pfadformel wird auf $true$ zurückgesetzt und der aktuelle CFA-Knoten gespeichert.

Ein wichtiger Aspekt ist die durch den Operator *blk* anpassbare Blocklänge. Zum Beispiel kann *blk* so gewählt werden, dass immer an einem Schleifenkopf eine Abstraktion durchgeführt wird. [9]

Um eine vollständige Analyse zu erhalten, werden noch weitere CPAs benötigt, wie zum Beispiel die *LocationCPA* zum Verfolgen der Erreichbarkeit von CFA-Knoten und die *ARGCPA* zum Aufbau des abstrakten Erreichbarkeitsgraphen, die durch Schachtelung und Verwendung der *CompositeCPA* zu einer CPA verknüpft werden.

Es bleibt die Frage, auf welche Weise die Präzision konstruiert wird, das heißt wie passende Prädikate gefunden werden, die eine möglichst abstrakte Repräsentation des Programmes zulassen, allerdings die Erreichbarkeit aller Programmzustände ausschließt, welche durch die gegebene Spezifikation als Fehlerzustände definiert sind. Eine Möglichkeit dieses Problem zu lösen ist *counterexample-guided abstraction refinement* (CEGAR)[17]. Die Idee ist, mit einem sehr abstrakten Modell zu beginnen und es iterativ zu verfeinern, indem neue Information in die Präzision eingefügt wird. Dafür wird die Analyse unterbrochen, sobald ein die Spezifikation verletzender Zustand erreicht ist, und ein Pfad zu diesem Zustand analysiert. Zuerst wird geprüft, ob der abstrakte Fehlerpfad einen konkreten Pfad repräsentiert. In diesem Fall verletzt das Programm die Spezifikation und der konkrete Fehlerpfad kann als Gegenbeispiel verwendet werden. Ansonsten ist der abstrakte Fehlerpfad auf Grund des zu ungenauen Modells nicht konkret genug, damit die Analyse ihn ausschließen kann. Deshalb ist es nun notwendig, geeignete Information in die Präzision einzufügen, so dass dieser Pfad von der weiteren Exploration ausgeschlossen wird. Es gibt verschiedene Ansätze um diese Information aus dem Fehlerpfad zu extrahieren, beispielsweise Craig-Interpolation [23, 30] oder die Synthese von Invarianten [10].

3.3 LassoRanker

LassoRanker[11] ist eine in Java geschrieben Bibliothek zum automatischen Synthetisieren von Terminiations- und Nicht-Terminationsargumenten. Sie ist Teil des Softwareverifikationsframeworks Ultimate[12] und findet im Softwareverifikationswerkzeug Ultimate Automizer[13] Anwendung.

Im folgenden sei $n \in \mathbb{N}_0$[14] die Zahl der Programmvariablen, $x' \in \mathbb{R}^n$ der Vektor der Werte der Programmvariablen vor und $x \in \mathbb{R}^n$ nach einer Transition. Als Eingabe erwartet LassoRanker ein sogenanntes *Lasso-Programm* $P = (S, L)$, das aus den beiden binären Relationen Stamm $S \subseteq \mathbb{R} \times \mathbb{R}$ und Schleife $L \subseteq \mathbb{R} \times \mathbb{R}$ besteht. Der Stamm repräsentiert einen Pfad vom Beginn des Programms bis zum Anfang der Schleife, welche einen Pfad repräsentiert, dessen erster und letzter CFA-Knoten, der Honda[15] genannt wird, identisch sind. Wie in Abbildung 3.2 dargestellt bilden Stamm und Schleife zusammen ein Lasso, das eine endliche Darstellung für eine möglicherweise unendliche Ausführung ist. S und L sind durch affine Ungleichungssysteme der Form

$$A \begin{pmatrix} x' \\ x \end{pmatrix} \leq b \tag{3.17}$$

mit einer Matrix $A \in \mathbb{R}^{m \times 2n}$ [16] und einem Spaltenvektor $b \in \mathbb{R}^m$ definiert, die den Zusammenhang zwischen dem Vektor der Variablenwerte $x' \in \mathbb{R}^n$ vor der Transition s beziehungsweise l und dem Vektor der Variablenwerte $x \in \mathbb{R}^n$ danach darstellen. Diese Ungleichungssysteme werden als SMT-Formeln repräsentiert, die Konjunktionen von affinen Ungleichungen sind. Eine unendliche Folge $(x_t)_{t \in \mathbb{N}_0}$ von Programmzuständen ist eine unendliche Ausführung eines Lassos $P = (S, L)$ genau dann, wenn $(x_0, x_1) \in S$ und $\forall i \in \mathbb{N} : (x_i, x_{i+1}) \in L$. Im folgenden werden nur noch Lasso-Programme betrachtet, für deren Schleife $L \neq \emptyset$ gilt, da Programme mit leerer Schleifenrelation offensichtlich terminieren und so technische Schwierigkeiten vermieden werden können.

LassoRanker nutzt zum Lösen von SMT-Formeln SMTInterpol[17] oder Z3[18], wobei letzteres Programm nicht direkt aus Java angesprochen werden kann, sondern als separater

[11] `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=lasso_ranker`
[12] `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/`
[13] `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer`
[14] In dieser Arbeit gilt $0 \notin \mathbb{N}$.
[15] Der Hondaknoten wird zum Knüpfen des Auges bei der Schlinge eines zum Einfangen von Tieren vorgesehen Lassos verwendet.
[16] Falls $n = 0$ lässt sich (3.17) zu $\mathbf{0} \leq b$ vereinfachen und es gilt $S, L \subseteq \mathbb{R}^0 \times \mathbb{R}^0 = \{ (\mathbf{0}, \mathbf{0}) \}$
[17] `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[18] `https://github.com/Z3Prover/z3`

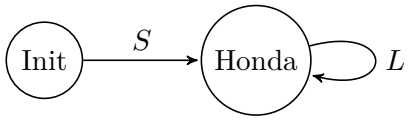

Abbildung 3.2: Lasso-Programm mit Stamm S und Schleife L

Prozess ausgeführt wird, mit dem LassoRanker über die Standard-Eingabe und -Ausgabe kommuniziert.

3.3.1 Geometrisches Nicht-Terminationsargument

Für $k \in \mathbb{N}$ und ein wie in Abschnitt 3.3 definiertes Lasso-Programm $P = (S, L)$ heißt das Tupel

$$N = (x_0, x_1, y_1, ..., y_k, \lambda_1, ..., \lambda_k, \mu_1, ..., \mu_{k-1})$$

Geometrisches Nicht-Terminationsargument der Größe k genau dann, wenn die folgenden Bedingungen gelten:

$$x_0, x_1, y_1, ..., y_k \in \mathbb{R}^n \tag{3.18}$$

$$\lambda_1, ..., \lambda_k, \mu_1, ..., \mu_{k-1} \in \mathbb{R}_0^+ \tag{3.19}$$

$$(x_0, x_1) \in S \tag{3.20}$$

$$A \begin{pmatrix} x_1 \\ x_1 + \sum_{i=0}^{k} y_i \end{pmatrix} \leq b \tag{3.21}$$

$$A \begin{pmatrix} y_1 \\ \lambda_1 y_1 \end{pmatrix} \leq \mathbf{0} \tag{3.22}$$

$$A \begin{pmatrix} y_i \\ \lambda_i y_i + \mu_{i-1} y_{y-1} \end{pmatrix} \leq \mathbf{0}, i \in 2, ..., k \tag{3.23}$$

Dabei sind A und b die Matrix und der Vektor der Ungleichung (3.17), durch welche die Schleifenrelation L definiert ist. Die Erfüllbarkeit dieser nicht linearen Bedingungen ist entscheidbar und eine Lösung kann mit Hilfe eines SMT-Solvers gesucht werden. Ein gefundenes Nicht-Terminationsargument für P ist nicht nur ein Beweis der Nicht-Termination, sondern gibt auch explizit durch

$$x_0,\ x_1,\ x_1 + YU^0\mathbf{1},\ x_1 + YU^0\mathbf{1} + YU^1\mathbf{1},\ x_1 + YU^0\mathbf{1} + YU^1\mathbf{1} + YU^2\mathbf{1}, ...$$

eine nicht terminierende Ausführung von P an. Dabei repräsentiert x_0 den initialen Programmzustand, x_1 den Programmzustand vor der ersten Schleifeniteration und jedes weitere

Folgenglied den Zustand zwischen zwei Schleifeniterationen. Die Matrix Y besteht aus dem Spaltenvektoren $y_1, ..., y_k$ und U ist folgendermaßen definiert:

$$U = \begin{pmatrix} \lambda_1 & \mu_1 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_2 & \mu_2 & \cdots & 0 & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_1 & \mu_1 \\ 0 & 0 & 0 & \cdots & 0 & \lambda_1 \end{pmatrix}$$

(3.18) sowie (3.19) geben den Definitionsbereich der Variablen an. Die Erreichbarkeit des durch x_1 gegeben Programmzustandes vor der ersten Schleifeniteration vom durch x_0 gegebenen initialen Programmzustand wird durch (3.20) sichergestellt. Ungleichung (3.21) garantiert das Vorhandensein einer Variablenbelegung $x_2 := x_1 + YU^0\mathbf{1}$ nach der ersten Schleifeniteration, sodass $(x_1, x_2) \in l$ gilt. Schließlich sorgen die Bedingungen (3.22) und (3.23) für die Existenz der unendlich vielen weiteren Programmzustände, durch welche sich die Nicht-Termination des Lasso-Programms ergibt.

Falls ein Lasso-Programm $P = (S, L)$ einen Stamm $s = \mathbb{R} \times \mathbb{R}$ besitzt, gibt es eine unendliche beschränkte Ausführung $(x_t)_{t \in \mathbb{N}_0}$ genau dann, wenn ein Fixpunkt x^* der Schleife l existiert, also $(x^*, x^*) \in L$ gilt. Außerdem existiert ein Geometrisches Nicht-Terminationsargument der Größe 0, wenn es einen solchen Fixpunkt gibt. Falls $s \neq \mathbb{R} \times \mathbb{R}$, kann es sein, dass der Fixpunkt der Schleife nicht vom initialen Programmzustand aus erreichbar ist und damit auch die Nicht-Termination von P nicht sicher gegeben ist.

Gibt es für die Schleife L eines nicht terminierenden Lasso-Programms P Matrizen $G \in \mathbb{R}^{m \times n}$, $M \in \mathbb{R}^{n \times n}$ sowie Vektoren $g \in \mathbb{R}^m$, $m \in \mathbb{R}^n$, so dass L

$$(x, x') \in L \iff Gx \leq g \wedge x' = Mx + m \tag{3.24}$$

erfüllt, und alle Eigenvektoren von M reell und nicht negativ sind, dann gibt es ein Geometrisches Nicht-Terminationsargument der Größe n für P. Außerdem vereinfacht sich die Suche nach einem Geometrischen Nicht-Terminationsargument, falls die Eigenwerte von M bekannt sind, da diese dann $\lambda_1, ..., \lambda_k$ in den Bedingungen (3.22) sowie (3.23) entsprechen, sodass durch Einsetzen die Ungleichungen linear werden. In (3.24) repräsentieren G und g die Schleifenbedingung. Durch M und m ist der Schleifenrumpf definiert.

Falls die Programmvariablen ganzzahlig sind, müssen für die Existenz eines Geometrischen Nicht-Terminationsarguments statt (3.18) und (3.19)

$$x_0, x_1, y_1, ..., y_k \in \mathbb{Z}^n \tag{3.25}$$

$$\lambda_1, ..., \lambda_k, \mu_1, ..., \mu_{k-1} \in \mathbb{N}_0 \tag{3.26}$$

gelten, da Programme, die über den reellen Zahlen nicht terminieren, möglicherweise über den Ganzen Zahlen terminieren. Die Erfüllbarkeit von nichtlinearen Ungleichungssystemen über den Ganzen Zahlen ist im allgemeinen nicht entscheidbar, aber es ist möglich die Werte für $\lambda_1, ..., \lambda_k$ auf eine endliche Menge zu beschränken, wodurch die Ungleichungen linear werden und damit effizient lösbar sind. Die Vollständigkeit ist dann jedoch nicht mehr gegeben.

Eine detailliertere Beschreibung und die Beweise zu den angegeben Resultaten finden sich für den Spezialfall $k = 0$ in [32] und für den allgemeinen Fall in [35].

3.3.2 Terminationsargumente

LassoRanker ist in der Lage für Lasso-Programme Terminationsargumente, die auf Abstiegsfunktionen basieren, zu synthetisieren. Eine reellwertige Funktion $f : \mathbb{R}^n \to \mathbb{R}$ ist genau dann eine *Abstiegsfunktion* einer Schleife L, wenn folgende Bedingungen erfüllt sind:

$$\exists \delta \in \mathbb{R}^+ : \forall x', x : (x', x) \in L \implies f(x') - f(x) \geq \delta \tag{3.27}$$

$$\forall x', x : (x', x) \in L \implies f(x') > 0 \tag{3.28}$$

Durch (3.27) wird sichergestellt, dass die Abstiegsfunktion in jeder Schleifeniteration mindestens um einen festen Wert δ kleiner wird. Durch die Beschränkung nach unten in (3.28) wird die Termination der Schleife garantiert, falls es eine solche Abstiegsfunktion f existiert. Dabei ist die Zahl der Schleifeniterationen einer konkreten Ausführung $x_1, x_2, ...$ der Schleife durch $\frac{f(x_1)}{\delta}$ nach oben beschränkt. Falls die Schleife L eines Lasso-Programms (3.27) und (3.28) genügt, ist die Termination jedes Lasso-Programms $P = (S, L)$ gegeben. Dabei handelt es sich jedoch um keine notwendigen Bedingungen, da die tatsächlich erreichbaren Transitionen von L durch S eingeschränkt sein können.

Ein *induktive Invariante* eines Lasso-Programms $P = (S, L)$ ist ein Prädikat $I(x)$, welches

$$\forall x', x : (x', x) \in S \implies I(x) \tag{3.29}$$

$$\forall x', x : I(x') \wedge (x', x) \in L \implies I(x) \tag{3.30}$$

erfüllt. Eine *Abstiegsfunktion mit induktiver Invariante* eines Lasso-Programms $P = (S, L)$ besteht neben einer Abstiegsfunktion f auch aus einer induktiven Invarianten I für die folgende weitere Eigenschaften

$$\exists \delta \in \mathbb{R}^+ : \forall x', x : I(x') \wedge (x', x) \in L \implies f(x') - f(x) \geq \delta \tag{3.31}$$

$$\forall x', x : I(x') \wedge (x', x) \in L \implies f(x') > 0 \tag{3.32}$$

erfüllt sein müssen, wodurch unter anderem Informationen aus dem Stamm S Verwendung im Terminationsargument finden. Durch $I(x)$ werden die möglichen Ausführungen der Schleife L unter Berücksichtigung von S überapproximiert, weshalb (3.31) (3.27) und (3.32) (3.28) entspricht. Durch eine reellwertige Funktion f und eine Konstante $\delta \in \mathbb{R}^+$ kann analog zu (3.27) und (3.28) eine wohlfundierte[19] Relation

$$R = \{ (x', x) \mid f(x') - f(x) \geq \delta \wedge f(x') > 0 \} \tag{3.33}$$

definiert werden, die einen Zustand x' vor einer Schleifeniteration mit einem Zustand x danach verknüpft. LassoRanker synthetisiert Abstiegsrelationen mit induktiven Invarianten als Terminationsargumenten und unterstützt die folgenden Arten von Abstiegsrelationen:

- Affine Abstiegsrelation
- Lexikographische Abstiegsrelation
- Zusammensetzte Lexikographische Abstiegsrelation
- Mehrphasige Abstiegsrelation
- Geschachtelte Abstiegsrelation
- Parallele Abstiegsrelation
- Abschnittsweise definierte Abstiegsrelation

Dazu gibt es jeweils eine Schablone, mit deren Hilfe aus einem Lasso die SMT-Formel generiert wird, deren Erfüllbarkeit die Existenz einer entsprechenden Abstiegsrelation beweist. In diesem Fall ergibt sich aus einer erfüllenden Belegung der freien Variablen die konkrete Abstiegsrelation. In den folgenden Kapiteln dieser Arbeit werden nur Affine, Lexikographische und Geschachtelte Abstiegsrelation verwendet. Die übrigen Abstiegsrelationen werden daher hier nicht weiter erläutert.

[19] Eine Relation R ist wohlfundiert gdw. $\nexists (x_i)_{i \in \mathbb{N}} : \forall j \in \mathbb{N} : (x_j, x_{j+1}) \in R$. Die Literatur zu Termination von Programmen weicht hier von der in der Mathematik üblichen Definition ab. In der Mathematik wird diese Eigenschaft üblicherweise terminierend genannt und wohlfundiert ist eine Relation R gdw. R^{-1} terminierend ist.

Die *Affine Abstiegsrelation* besteht aus einer affine Funktion

$$f(x) = a^T x + a_0 \tag{3.34}$$

mit dem Koeffizientenvektor $a \in \mathbb{R}^n$ und der Konstante $a_0 \in \mathbb{R}$ sowie einer[20] affinen induktiven Invariante

$$I(x) = s^T x + s_0 \lhd 0 \tag{3.35}$$

mit $\lhd \in \{ \leq, < \}$, dem Koeffizientenvektor $s \in \mathbb{R}^n$ und der Konstante $s_0 \in \mathbb{R}$. Die Schablone der Affinen Abstiegsrelation besteht aus einer Schablone für eine wohldefinierte Relation der Form (3.33), wobei f die affine Funktion aus (3.34) und einer Schablone für die affine Invariante aus (3.35) besteht. Das Einsetzen von f aus (3.34) sowie I aus (3.35) in die Konjunktion der Ungleichungen (3.29), (3.30), (3.31) und (3.31) ergibt eine Formel, für die mittels eines SMT-Solvers eine erfüllende Belegung ν der freien Variablen gesucht werden kann. Falls es eine solche Belegung gibt, kann damit aus der Schablone T eine konkrete Instanz $\nu(T) = (R, I)$ erzeugt werden, für die

$$\forall x', x : (I(x') \wedge (x', x) \in L) \implies (x', x) \in R \tag{3.36}$$

gilt und damit die Termination des Lasso-Programms $P = (S, L)$ bewiesen ist.

Da die zu lösenden Formeln jedoch universell quantifiziert sind, muss noch eine Transformation durchgeführt werden, um ausschließlich existenzquantifizierte nur durch Konjunktionen verknüpfte lineare Formeln zu erhalten, für die effiziente Verfahren zur Lösungssuche bekannt sind. Zuerst werden die $I(x)$-Terme durch Anwendung von

$$\Phi_1 \wedge \Phi_2 \implies \Psi \equiv \Phi_2 \implies \Psi \vee \neg\Phi_2 \tag{3.37}$$

auf die rechte Seite der Implikationen gebracht. Die hierdurch verursachten Disjunktionen werden nach folgendem Schema

$$m \geq 0 \vee n < 0 \quad \rightsquigarrow \quad m + n \leq 0 \tag{3.38}$$

beziehungsweise durch Entfernen des Terms $I(x)$ in (3.32) eliminiert. Auf die resultierenden Formeln wird Farkas Lemma angewandt. Die ursprüngliche Variante wurde von Farkas in [25] veröffentlicht. Die hier verwendete Version besagt, dass für ein erfüllbares lineares Ungleichungssystemen $Ax \leq b$ und ein lineares Gleichungssystem $c^T x \leq \delta$ folgendes

$$\forall x : (Ax \leq b \implies c^T x \leq \delta) \quad \iff \quad \exists \lambda (\lambda \geq 0 \wedge \lambda^T A = c^T \wedge \lambda^T b \leq \delta) \tag{3.39}$$

[20] Die Implementierung in LassoRanker unterstützt mehre induktive affine Invarianten.

gilt. Mit Hilfe dieser Äquivalenztransformation wird schließlich auch noch die Beseitigung aller Universalquantoren in den vier Formeln erreicht.

Für die weiteren Typen von Terminationsargumenten muss die Definition einer Abstiegsrelation erweitert werden. Eine binäre Relation R heißt Abstiegsrelation, wenn es eine Funktion $\rho : \mathbb{R}^n \to \alpha$ mit einer Wohlordnung $(\alpha, \succ)$ und Ordinalzahl[21] α gibt, so dass

$$\forall x', x : (x', x) \in R \implies \rho(x') \succ \rho(x) \tag{3.40}$$

erfüllt ist. R ist genau dann wohldefiniert, wenn eine solche Funktion ρ existiert. Damit kann die Termination der Schleife L eines Lasso-Programms $P = (S, L)$ bewiesen werden, indem $L \subset R$ gezeigt wird, da in diesem Fall L auch eine terminierende Relation sein muss. Für die Termination von P ist es jedoch ausreichend Ausführungen von L zu betrachten, die tatsächlich unter Berücksichtigung von S erreichbar sind. Daher genügt es

$$\forall (x_1, x_2, ...) : (x_0, x_1) \in S \implies (\forall j \in \mathbb{N} : (x_j, x_{j+1}) \in L \implies (x_j, x_{j+1}) \in R) \tag{3.41}$$

zu zeigen.

Die *Lexikographische* Schablone verwendet einen Parameter $k \in \mathbb{N}$, eine Funktion $F : \mathbb{R}^n \to \mathbb{R}^k$ und die Lexikographische Ordnung auf $\mathbb{R}^k$, so dass sich folgende konjunktiv zu verknüpfende Bedingungen ergeben:

$$\bigwedge_{i=1}^{k} \delta_i > 0 \tag{3.42}$$

$$\bigwedge_{i=1}^{k} f_k(x) > 0 \tag{3.43}$$

$$\bigwedge_{i=1}^{k-1} (f_i(x') \leq f_i(x) \vee \bigvee_{j=1}^{i-1} f_j(x') < f_j(x)\delta_j \tag{3.44}$$

$$\bigvee_{i=1}^{k} f_i(x') < f_i(x) - \delta_i \tag{3.45}$$

Dabei ist f_i die i. Komponente von F, die jeweils eine affine Funktion der Form (3.34) ist. Durch den Parameter k wird die Zahl der verwendeten Funktionen und damit auch der notwendige Rechenaufwand, um nach einer Lösung zu suchen, gesteuert. Die Lexikographische Schablone kann sich zum Beispiel eigenen, wenn in der Schleife eines Programms eine Verzweigung vorhanden ist. Jedoch enthalten die Bedingungen, aus deren Lösung sich die

[21] Eine Menge α ist genau dann eine Ordinalzahl, wenn $\forall a \in \alpha : a \subset \alpha$ gilt und $(\alpha, \ni)$ eine wohlgeordnete Menge ist. Da die Definition von wohlfundiert nicht der in der Mathematik üblichen folgt, muss bei der Definition einer Ordinalzahl die Relation $\ni$ statt $\in$ verwendet werden.

Werte der in die Schablone einzusetzenden Parameter ergeben, nicht lineare Anteile, weshalb es potentiell erheblich aufwendiger ist eine Lösung zu finden. Außerdem unterstützen auch nicht alle SMT-Solver das Lösen von nicht linearen Formeln.

Die *Geschachtelte* Schablone erzeugt Abstiegsrelationen, die aus mehreren Phasen bestehen, denen jeweils eine affine Funktion zugeordnet ist, deren Wert nicht um mehr als den Wert der zur vorhergehenden Phase zugehörigen Funktion zunehmen darf. Eine Phase i endet sobald der Wert der dazugehörigen Funktion f_i negativ wird und alle vorigen Phasen beendet sind. Sobald eine Phase i beendet ist, nimmt demgemäß der Wert der Funktion f_{i+1}, welche der nächsten Phase $i+1$ zugeordnet ist, ab. Für einen Parameter $k \in \mathbb{N}$, eine Konstante $\delta \in \mathbb{R}$ und eine Funktion $F : \mathbb{R}^n \to \mathbb{R}^k$ ist ergibt sich die Schablone aus folgenden konjunktiv zu verknüpfenden Formeln:

$$\delta > 0 \tag{3.46}$$

$$f_1(x) < f_1(x') - \delta \tag{3.47}$$

$$f_k(x') > 0 \tag{3.48}$$

$$\bigwedge_{i=2}^{k} f_i(x) < f_i(x') + f_{i-1}(x') \tag{3.49}$$

Der Parameter k legt die Zahl der Phasen fest. (3.46) stellt in Verbindung mit (3.47) sicher, dass die Funktion f_1 der ersten Phase in jeder Schleifeniteration um mindestens δ abnimmt. (3.48) beschränkt den Wert der Funktion f_k der letzten Phase nach unten. Zusammen mit (3.49), welche die Zunahme des Funktionswertes einer Phase auf den Wert der Funktion der vorhergehenden Phase in der letzten Iteration beschränkt, ergibt sich eine Schablone für eine wohlfundierte Relation über x' und x. Die Geschachtelte Schablone ist eine Spezialfall der Mehrphasen-Schablone, die im Gegensatz zu erstgenannten das Lösen von nicht linearen Formeln erfordert.

Program 3.2 zeigt eine typische Schleife, für die mit Hilfe der Geschachtelten Schablone eine Abstiegsrelation gefunden werden kann. Der Wert der Variable `x` ist initial positiv und in jeder Schleifeniteration wird der Wert von `y` abgezogen. Da `y` zu Beginn negativ ist, steigt der Wert von `x` zunächst, nimmt jedoch ab der fünften Schleifeniteration ab, weil `y` jeweils dekrementiert wird. Für $k = 2$ lässt sich Beispielsweise eine aus

$$\delta = 0.5 \tag{3.50}$$

$$f_1(x, y) = -y + 1 \tag{3.51}$$

$$f_2(x, y) = x \tag{3.52}$$

```c
int main() {
  int x = 9;
  int y = -3;

  while (x > 0) {
    x = x - y;
    y = y + 1;
  }

  return 0;
}
```

Program 3.2: C-Programm mit Geschachtelter Abstiegsrelation

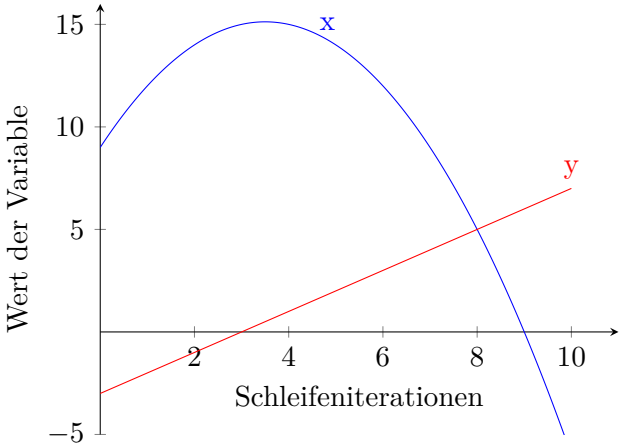

Abbildung 3.3: Werte der Variablen von Program 3.2 über die Schleifeniterationen

bestehende Geschachtelte Abstiegsrelation finden. Der Verlauf der Werte der Variablen nach jeder Schleifeniteration ist in Abbildung 3.3 als kontinuierliche Kurve dargestellt. Der Wert von `x` beschreibt eine Parabel, der von `y` eine affine Kurve. Im C-Programm nehmen die Variablen nur diskrete Werte an, die genau den Werten der beiden Kurven entsprechen.

Die Synthese der Abstiegsrelationen ist vollständig in dem Sinn, dass eine Abstiegsrelation für ein terminierendes Lasso-Programm synthetisiert wird, falls es eine Relation gibt, die den Bedingungen der entsprechenden Schablone entspricht. An Stelle der Rellen Zahlen können auch die Rationalen oder Ganzen Zahlen verwendet werden. In letzterem Fall ist die Synthese jedoch nicht mehr vollständig, solange nicht die ganzzahlige Hülle berechnet wird. Dies ist jedoch bisher nicht in LassoRanker implementiert.

Für die Korrektheit und Vollständigkeit der erläuterten Abstiegsrelationen ist es zulässig, dass die Relationen S und L durch eine Konjunktion von strikten und nicht strikten affinen Ungleichungen darstellbar sind. Es muss also für S eine Darstellung

$$(x', x) \in S \Longleftrightarrow (A \begin{pmatrix} x' \\ x \end{pmatrix} + b \leq 0 \wedge C \begin{pmatrix} x' \\ x \end{pmatrix} + d < 0) \tag{3.53}$$

mit $A \in \mathbb{R}^{m \times 2n}$, $C \in \mathbb{R}^{l \times 2n}$, $b \in \mathbb{R}^m$ sowie $d \in \mathbb{R}^l$ geben und entsprechend auch für L existieren. Dann muss jedoch Farkas Lemma (3.39) zu *Motzkins Transpositions-Theorem* [36] erweitert werden, welches besagt, dass die folgenden Formeln (3.54) und (3.55) äquivalent sind:

$$\forall x \in \mathbb{R}^{2n} : \neg(Ax \leq b \wedge Cx < d) \tag{3.54}$$

$$\begin{aligned} \exists \lambda \in \mathbb{R}^m, \mu \in \mathbb{R}^l : \; & \lambda \geq 0 \wedge \mu \geq 0 \\ \wedge \; & \lambda^T A + \mu^T C = 0 \\ \wedge \; & \lambda^T b + \mu^T d \leq 0 \wedge (\lambda^T < 0 \vee \mu \neq 0) \end{aligned} \tag{3.55}$$

Eine ausführliche Darstellung der in LassoRanker verwendeten Terminationsargumente sowie die Beweise zu den in diesem Abschnitt erläuterten Abstiegsrelationen finden sich in [28, 33, 34].

3.4 Komposition von Terminationsargumenten

LassoRanker ist nur in der Lage Terminationsargumente für Lasso-Programme zu synthetisieren. Um auch für komplexere Programme, die zum Beispiel Verzweigungen und mehrere möglicherweise geschachtelte Schleifen enthalten können, Terminationsbeweise führen zu können, ist es in aller Regel notwendig mehre Terminationsargumente, die für einzelne Teile eins Programms gefunden wurden, zu verknüpfen.

R^+ bezeichnet die transitive Hülle der binären Relation R. Eine Relation R ist disjunktiv wohlfundiert, wenn $R \subseteq T_1 \cup T_2 \cup ... \cup T_n$ gilt und $T_1, T_2, ..., T_n$ wohlfundiert sind. T ist jedoch im allgemeinen nicht wohlfundiert. Es kann allerdings gezeigt werden, dass eine Relation R wohlfundiert ist, wenn ihre transitive Hülle R^+ disjunktiv wohlfundiert ist. Weiter sei $R_I^+ = R^+ \cap (Reach(r_0) \times Reach(r_0))$ die Einschränkung der transitiven Hülle auf die von einer Startregion r_0 tatsächlich erreichbaren Transitionen. Damit kann die Termination eines Programmes, das durch seine Transitions-Relation $R = \{ (c, c') \mid c \to c' \}$ gegeben ist,

durch die Komposition mehrerer Terminationsargumente, die als wohlfundierte Relationen $T_1, T_2, ..., T_n$ gegeben sind, durch

$$R_I^+ \subseteq T_1 \cup T_2 \cup ... \cup T_n \tag{3.56}$$

gezeigt werden. Für nicht zyklische Teile eines CFA können triviale wohlfundierte Relationen gefunden werden, die zum Beispiel auf den Knotennummern basieren, so dass es ausreichend ist, diesen Test für starke Zusammenhangskomponenten durchzuführen, weshalb in aller Regel auch nur hierfür explizit Terminationsargumente berechnet werden. [22, 40]

Die Terminationsargumente $T_1, T_2, ..., T_n$ können durch kontinuierliche Verfeinerung des Gesamt-Terminationsargumentes gewonnen werden. Dabei wird mit der leeren Menge von Terminationsargumenten T begonnen und ein Pfad gesucht, der dieser nicht genügt. Solange ein solcher Pfad gefunden wird, kann daraus ein Lasso-Programm konstruiert sowie versucht werden für dieses ein Terminationsargument zu synthetisieren. Falls dies erfolgreich ist, wird es zu T hinzugefügt. Dies wird solange wiederholt bis entweder ein Pfad gefunden wird, der tatsächlich ein Beispiel für Nicht-Termination des Programmes ist oder kein Pfad mehr extrahiert werden kann, der nicht schon von den bisher berechneten Terminationsargumenten abgedeckt ist. In [19] wird dies mit der Abstraktion des Pfades zu CEGAR für Termination kombiniert.

4 Implementierung

Dieses Kapitel beschreibt die Umsetzung einer Terminierungsanalyse in CPAchecker unter zu Hilfenahme der Bibliothek LassoRanker und einer bereits in CPAchecker existierenden Erreichbarkeitsanalyse. Dabei werden Ideen von Terminator [21] aufgegriffen.

4.1 Algorithmus

Für die Terminationsanalyse ist ein Algorithmus erforderlich, der das Zusammenspiel der verwendeten Erreichbarkeitsanalyse und der Synthese von Terminations- und Nicht-Terminationsargumenten steuert. Das dafür benutze Verfahren ist in Algorithmus 2 in vereinfachter Form zu lesen und in `TerminationAlgorithm` in CPAchecker implementiert.

Jede Schleife des Programmes wird einzeln analysiert. Dabei wird *reached* und *waitlist* zunächst mit dem initialen Zustand e_0 und der dazugehörigen Präzision p_0 vorbereitet sowie der Analyse Informationen über die gerade analysierte Schleife l bereitgestellt. Solange es noch zu bearbeitende Zustände gibt, wird die Erreichbarkeitsanalse S aufgerufen und das Ergebnis analysiert. In Zeile 9 wird geprüft, ob ein Fehlerzustand erreicht wurde. Falls dem so ist, wird versucht ein Terminations- oder Nicht-Terminationsargument aus dem dazugehörige Fehlerpfad zu synthetisieren. Wenn es ein Terminationsargument für diesen Pfad gibt, wird die Abstiegsrelation für diese Schleife entsprechend der Ausführungen in Abschnitt 3.4 aktualisiert und die möglicherweise ebenfalls synthetisierten Invarianten der Erreichbarkeitsanalyse zur Verfügung gestellt. Vor der nächsten Iteration ist noch der Fehlerpfad aus *waitlist* und *reached* zu entfernen. Falls der Fehlerpfad ein tatsächliches Beispiel für Nicht-Termination des Programmes ist, ist der Algorithmus fertig und gibt dieses Resultat zurück. Wenn mittels LassoRanker weder Termination noch Nicht-Termination des durch den Fehlerpfad repräsentierten Lassos gezeigt werden kann, kann zunächst Termination nicht mehr bewiesen werden[1]. Es ist jedoch weiterhin möglich Nicht-Termination zu beweisen, indem die Analyse S verwendet wird um weitere möglicherweise nicht terminierende Lassos zu suchen oder andere Schleifen zu analysieren. Falls die Erreichbarkeitsanalyse

[1] Die Implementierung des Algorithmus in CPAchecker kann eventuell auch noch Termination beweisen, falls im weiteren Verlauf für einen anderen Fehlerpfad zu dieser Schleife ein Terminationsargument gefunden wird.

Algorithm 2 Terminationsalgorithmus

Input: Analyse S, initialer abstrakter Zustand $e_0 \in E$ mit Präzision $\pi_0 \in \Pi$, Menge der Schleifen *loops*

Output: $TRUE \mid FALSE \mid UNKNOWN$

```
1:  result = TRUE
2:  for each l : loops do
3:    waitlist := { (e_0, π_0) }
4:    reached := { (e_0, π_0) }
5:    updateLoop(l)
6:
7:    while waitlist ≠ ∅ do
8:      S.run(reached, waitlist)
9:      if targetState ∈ reached then
10:       argument := analyseLasso(targetState)
11:
12:       if argument.isTerminationArgument() then
13:         updateRankingRelations(result)
14:         updateInvariants(result)
15:         removeTargetPath(waitlist, reached, targetState)
16:
17:       else if argument.isNonTerminationArgument() then
18:         return FALSE
19:
20:       else
21:         result := UNKNOWN
22:       end if
23:     end if
24:   end while
25: end for
26:
27: if isRecursive(S, e_0, π_0) then
28:   result := UNKNOWN
29: end if
30: return result
```

feststellt, dass kein Lasso mehr existiert, das nicht die bisher konstruierte Abstiegsrelation erfüllt, terminierend diese Schleife immer und die nächste Schleife wird untersucht. Nach der Analyse aller Schleifen wird in Zeile 27 noch überprüft, ob das Programm Rekursion enthält, da durch Rekursion ebenfalls Nicht-Termination verursacht werden kann.

Es ist erforderlich, dass die äußerste CPA die `ARGCPA` ist, wobei dies in der Praxis keine echte Einschränkung darstellt, da dies in aller Regel ohnehin der Fall ist oder sogar auf Grund der verwendeten Erreichbarkeitsanalyse zwingend der Fall sein muss. Damit der beschriebene Fehlerzustand als solcher richtig erkannt wird, muss eine spezielle Spezifikation verwendet werden, die auch sicherstellt, dass Aufrufe von Methoden, welche die sofortige Termination des Programmes zu Folge haben[2], nicht zurückkehren.

4.2 Einweben von Terminationsargumenten in Programme

Das Instrumentieren des zu analysierenden Programmes mit den gefundenen Terminationsargumenten besteht im wesentlichen aus zwei Teilen, die beide von der `TerminationCPA` $\mathbb{T} = (D_{\mathbb{T}}, \Pi_{\mathbb{T}}, \rightsquigarrow_{\mathbb{T}}, merge_{\mathbb{T}}, stop_{\mathbb{T}}, prec_{\mathbb{T}})$ wahrgenommen werden, welche die für die Erreichbarkeitsanalyse verwendete CPA $\mathbb{S} = (D_{\mathbb{S}}, \Pi_{\mathbb{S}} \rightsquigarrow_{\mathbb{S}}, merge_{\mathbb{S}}, stop_{\mathbb{S}}, prec_{\mathbb{S}})$ kapselt. Der `TerminationAlgorithm` stellt die für das Einweben der Terminationsargumente notwendigen Daten durch ein `TerminationLoopInformation`-Objekt bereit, in das nur der `TerminationAlgorithm` schreiben kann. $\mathbb{T}$ besteht aus folgenden Komponenten:

- Die abstrakte Domäne $D_{\mathbb{T}}$ ergibt sich durch das Kreuzprodukt aus dem flachen Halbverband $\mathcal{E}_{\mathbb{T}} = \{ stem, loop \}$ und $\mathcal{E}_{\mathbb{S}}$ der gekapselten abstrakten Domäne D_S. Ein `TerminationState` $(t, e) \in \mathcal{E}_{\mathbb{T}} \times \mathcal{E}_{\mathbb{S}}$ als Element der abstrakten Domäne enthält einen abstrakten Zustand der Analyse und die Information, ob der abstrakte Zustand zum Stamm oder der Schleife gehört, in dem für den zweiten Fall auch der Honda-Knoten gespeichert wird. Auf diese Weise wird ein abstrakter Befehlszählerzustand pc', der den Wert des Befehlszählers zu Beginn der Stamm- oder Schleifentransition repräsentiert, umgesetzt. Der künstlich nach dem Schleifenkopf eingefügte Fehlerzustand enthält weitere Informationen, wie die nicht erfüllte Abstiegsrelation als SMT-Formel und die im ursprünglichen Programm nicht enthaltene Kante von Schleifenkopf zum Fehlerzustand. Es wird nur der Operator $\sqsubseteq_{\mathbb{T}}$ und nicht $\sqcup_{\mathbb{T}}$ unterstützt. $\sqsubseteq_{\mathbb{T}}$ verwendet $\sqsubseteq_{\mathbb{S}}$:

$$(t, e) \sqsubseteq_{\mathbb{T}} (t', e') \iff e \sqsubseteq_{\mathbb{S}} e' \wedge \ t = t'. \tag{4.1}$$

[2] In C sind das zum Beispiel `abort` und `exit`. Wenn ein `ERROR`-Label erreicht wird gilt gleiches.

- Die Transferrelation $\rightsquigarrow_{\mathbb{T}}$ erledigt das eigentliche Einweben der Terminationsargumente. Falls lh ein Schleifenkopf der aktuell analysierten Schleife ist, gilt folgendes:

$$\begin{aligned}((t,e),&(l,op,l'),(t',e'),\pi) \in \rightsquigarrow_{\mathbb{T}} \iff \\ &(((e,(l,op,l'),e',\pi) \in \rightsquigarrow_{\mathbb{S}} \wedge t = t') \\ &\vee\ (l = lh \wedge t = stem \wedge t' = loop \wedge e' = startLoop(e,(l,op,l'),\pi)) \\ &\vee\ (l = lh \wedge t = loop \wedge t' = loop \wedge e' = checkRankingRelation(e,\pi)))\end{aligned} \tag{4.2}$$

  Die Transferrelation verwendet die Transferrelation der gekapselten CPA und greift selbst nur an den Schleifenköpfen der Schleife, deren Terminationsverhalten gerade untersucht wird, ein[3].

  startLoop sorgt während des Übergangs von Stamm zur Schleife dafür, dass die Werte aller lokalen Programmvariablen der aktuellen Funktion und aller globalen Variablen gesichert werden: `__PRIMED__a = a; __PRIMED__b = b; ...`. Zusätzlich wird für alle diese Variablen der Wert der Dereferenzierung gespeichert, falls es sich um einen Zeiger-Typ handelt. Anschließend werden noch die abstrakten Nachfolger bezüglich der *op* und $\rightsquigarrow_{\mathbb{S}}$ berechnet.

  checkRankingRelation erzeugt temporär einen Knoten l' im CFA, der über eine Kante (l, op, l') erreichbar ist, falls die bisher für die Schleife synthetisierte Abstiegsrelation nicht erfüllt ist. *op* ist dabei die Negation der Vereinigung der Abstiegsrelationen. Alle abstrakten Zustände, die den Knoten l' repräsentieren, werden von der verwendeten Spezifikation als Fehlerzustand betrachtet.

- $merge_{\mathbb{T}}$ kombiniert zwei abstrakte Zustände nur, wenn beide Teil der Schleife oder beide Teil des Stammes sind:

$$merge_{\mathbb{T}}((t,e),(t',e'),\pi) = \begin{cases}(t', merge_{\mathbb{S}}(e,e',\pi)) & \text{falls } t = t' \\ (t',e') & \text{sonst}\end{cases} \tag{4.3}$$

- $stop_{\mathbb{T}}$ berücksichtigt die Trennung von Stamm und Schleife:

$$stop_{\mathbb{T}}((t,e),R,\pi) = stop_{\mathbb{S}}(e,\{\, e' \mid (t,e') \in R \,\},\pi) \tag{4.4}$$

- $prec_{\mathbb{T}}$ ruft $prec_{\mathbb{S}}$ auf und sorgt dabei für das Entkapseln der Eingabe und Kapseln der resultierenden Zustände.

[3] Damit das instrumentierte Programm syntaktisch korrekt ist, werden die zum Speichern der Werte der Variablen am Übergang von Stamm zu Schleife verwendeten Variablen außerhalb der Schleife definiert. Dies wird jedoch von der gekapselten Analyse in aller Regel ignoriert.

Für alle initialen Zustände (t, e) gilt $t = stem$. Die `TerminationCPA` darf nicht innerhalb einer `CompositeCPA` verwendet werden, da die `TerminationTransferrelation` die Methode `getAbstractSuccessorsForEdge` nicht unterstützt.

Damit auch Analysen verwendet werden können, die den *CEGAR*-Algorithmus verwenden, ist es notwendig alle Fehlerpfade um die im vorherigen Absatz beschrieben CFA-Kanten zu erweitern. Dafür wird die Klasse `TerminationARGBasedRefiner` verwendet, die den `AbstractARGBasedRefiner` der Erreichbarkeitsanalyse kapselt und den instrumentierten Fehlerpfad als `TerminationARGPath` bereitstellt. Dafür sind die von `TerminationAlgorithm` bereitgestellten Informationen und die im Fehlerzustand gespeicherte CFA-Kante zu selbigem notwendig.

4.3 Synthese von Terminations- und Nicht-Terminationsargumenten

Für die Synthese von Terminations- und Nicht-Terminationsargumenten werden verschiedene Komponenten verwendet. Zuerst müssen aus dem Gegenbeispiel ein oder mehrere Lassos konstruiert werden. Diese werden anschließend an LassoRanker übergeben. Dabei wird zuerst nach einem Nicht-Terminationsargument gesucht, da dies im Mittel erheblich schneller ist als die Synthese von Terminationsargumenten. Die möglicherweise resultierenden Terminationsargumente müssen in eine für die weitere Verwendung geeignete Darstellung transformiert werden.

4.3.1 Konstruktion von Lassos aus Fehlerpfaden

Die Klasse `LassoBuilder` erzeugt aus einem `CounterexampleInfo`-Objekt und den für das Terminationsverhalten dieses Lassos relevanten Variablen, die durch ihre Deklarationen gegeben sind, eine Menge von Lassos. Zunächst werden die CFA-Kanten des Fehlerpfades in Stamm und Schleife geteilt, indem diese Information aus den jeweiligen `TerminationState` extrahiert wird. Als nächster Schritt wird der Stamm in eine SMT-Formel umgewandelt, wobei auf eine bereits existierende `PathFormulaManager`-Implementierung zurückgegriffen werden kann, so dass hier keine explizite Behandlung von Quellcode notwendig ist und andere Programmiersprachen leicht unterstützt werden könnten, sofern es einen Konverter zum Erstellen der Formel gibt. Bei der Erzeugung der Pfadformel der Schleife muss darauf geachtet werden, dass das `PathFormula`-Objekt mit den Metainforamtionen des Stammes initialisiert wird, da sonst die Variablen in den Formeln von Stamm und Schleife nicht richtig indiziert werden. Außerdem ist es notwendig, den ersten Index einer Variable in der Schleife während der Erzeugung der Pfadformel zu speichern, sofern diese Variable nicht

schon im Stamm vorgekommen ist. Zeiger werden mittels Uminterpretierter Funktionen abgebildet.

Die Formeln müssen in Disjunktive Normalform gebracht werden, wobei jede Klausel ein wie in Abschnitt 3.3 beschriebenes lineares Ungleichungssystem darstellen muss. Um dies zu erreichen werden die folgenden Transformationen durchgeführt:

1. Ganzzahlige *Division* und *Modulo* werden durch eine Äquivalenzumformung entfernt. Dazu wird der entsprechende Term durch eine Hilfsvariable ersetzt, die das Ergebnis der Operation repräsentiert, und ein zusätzlicher Term, der nur das richtige Ergebnis der Operation als Wert der Hilfsvariable zulässt, konjunktiv mit der Formel verknüpft. Eine Division a/b wird durch die Variable d ersetzt und die folgenden Bedingungen ergänzt[4]:

$$b > 0 \wedge a \geq 0 \implies d * b \leq a < (d+1) * b \tag{4.5}$$
$$b < 0 \wedge a \geq 0 \implies d * b \leq a < (d-1) * b \tag{4.6}$$
$$b > 0 \wedge a < 0 \implies d * b \geq a > (d-1) * b \tag{4.7}$$
$$b < 0 \wedge a < 0 \implies d * b \geq a > (d+1) * b \tag{4.8}$$

   Falls die Division nicht ganzzahlig ist, sollte bereits der Kehrwert in der Formel als Faktor einer Multiplikation anstatt einer Division verwendet werden. Einer Modulo-Operation $a\%b$, die durch die Variable d ersetzt wird, werden die folgenden Formeln hinzugefügt:

$$a = d * b + r \tag{4.9}$$
$$b > 0 \implies 0 \leq r < b \tag{4.10}$$
$$b < 0 \implies 0 \leq r < -b \tag{4.11}$$

2. *Nicht lineare Multiplikation* $a * b$ wird durch eine neue Variable c, die das Ergebnis überapproximiert, ersetzt. Für jeden Faktor $f \ in \{ a, b \}$ werden einige Fälle unterschieden:
   - $f = 0$
   - $f = 1$
   - $f = -1$
   - $0 < f < 1$

[4]Division durch 0 bewirkt in C undefiniertes Verhalten und wird daher an dieser Stelle nicht betrachtet.

- $1 < f$
- $-1 < f < 0$
- $f < -1$

Für jede Kombination der Fälle für die beiden Faktoren wird eine Bedingung an das Resultat c und die Faktoren in Form mehrerer linearer Ungleichungen aufgestellt und diese konjunktiv verknüpft. Durch diese Umformung erhöht sich jedoch die Anzahl an Disjunktionen in erheblichem Umfang, da alle Fälle disjunktiv zusammengefügt werden.

3. *Uninterpretierte Funktionen* dienen unter anderem der Modellierung von Zeigern. Jedes Vorkommen einer Funktionsanwendung wird durch eine neue Variable ersetzt. Damit die Semantik erhalten bleibt, muss für je zwei Applikationen einer Funktion sichergestellt werden, dass die Ergebnisse gleich sind, falls alle Argumente übereinstimmen. Für zwei durch die Variablen f_j und f_K ersetzte Anwendungen einer uninterpretierten Funktion f und Argumenten $x_1, x_2, ..., x_n$ sowie $y_1, y_2, ..., y_n$ muss folgende zusätzliche Bedingung hinzugefügt werden.

$$(\bigwedge_{i=1}^{n} x_i = y_i) \implies f_j = f_k \tag{4.12}$$

Auf Grund der Implikation und der Notwendigkeit dies für jede Kombination aus zwei verschiedenen Anwendungen der selben Funktionen durchzuführen, ergibt sich eine in der Zahl an Funktionsanwendungen einer Funktion exponentielle Anzahl an Disjunktionen.

Auf eine konsistente Ersetzung in Stamm und Schleife ist zu achten. Dies wird erreicht, indem die für den Stamm durchgeführte Ersetzung auch auf die Schleife angewandt wird und auch die Funktionsapplikationen im Stamm bei der Erzeugung der Bedingungen (4.12) für die Schleife berücksichtigt werden.

4. Eine Verwendung des *if-then-else*-Operators $if\ a\ then\ b\ else\ c$ wird durch eine neue Variable d ersetzt, deren korrekter Wert durch die zusätzliche Bedingung

$$(d = b \wedge a) \vee (d = c \wedge \neg a) \tag{4.13}$$

festgelegt wird.

5. Die Formeln werden in *Negationsnormalform* durch Anwendung der in JavaSMT bereits vorhandenen Implementierung überführt.

6. Der *Ungleichheits*-Operator wird durch Verwendung des strikten kleiner- und größer-Operators eliminiert. Dabei wird $a \neq b$ durch $a < b \vee a > b$ substituiert.

7. *Gleichungen* der Form $a = b$ werden durch zwei nicht strikte Ungleichungen $a \leq b \vee a \geq b$ ersetzt.

8. Während der Berechnung der *Disjunktiven Normalform* werden bereits nicht erfüllbare Klauseln ausgefiltert, damit die Zahl der resultierenden Klauseln möglichst gering gehalten werden kann.

Durch Durchführung dieser Schritte können die Formeln in die gewünschte Form gebracht werden, wobei jedoch die Zahl der Disjunktionen möglicherweise sehr hoch sein kann, sodass die abschließende Berechnung der DNF nicht immer möglich sein wird.

Jede Klausel der in DNF vorliegenden Formel ist ein Stamm beziehungsweise eine Schleife. Für jede erfüllbare Kombination aus einer Stamm- und Schleifenklausel wird ein Lasso erzeugt. Dafür muss zunächst noch die Menge der Ein- und Ausgabevariablen der Stamm- und Schleifentransition bestimmt werden. Dabei werden nur die für den aktuell betrachteten Schleifenkopf relevanten Programmvariablen betrachtet und jeweils die entsprechende Variable in der Formel mit dem höchsten Index als Ausgabevariable betrachtet. Der Stamm besitzt keine Eingabevariablen. Die Eingabevariablen der Schleife werden wie zu Beginn dieses Abschnittes beschrieben während der Konstruktion der Schleifenformel aus dem Fehlerpfad berechnet. Die bei der Elimination der uninterpretierten Funktionen verwendete Substitution muss ebenfalls beachtet werden, damit die Werte von Zeigern korrekt als Eingabe- bzw. Ausgabevariable erfasst werden. Als Identifikator, durch den der Zusammenhang zwischen einer Eingabe- und Ausgabevariable hergestellt wird, dient der nicht instantiierte Term der ursprünglichen Pfadformel.

4.3.2 Transformation von Terminationsargumenten

Alle resultierenden Terminationsargumente bestehen aus einer Abstiegsrelation und einer Menge von Schleifeninvarianten. Die Abstiegsrelation wird unter Verwendung der gestrichenen Variablen, welche die Werte der Programmvariablen vor der Schleifentransition darstellen, und Originalvariablen in einen C-Term und eine SMT-Formel umgewandelt. Es werden Affine, Geschachtelte sowie Lexikographische Abstiegsrelationen unterstützt und in wie in Unterabschnitt 3.3.2 beschriebene Formeln beziehungsweise C-Ausdrücke umgewandelt. Die Invarianten werden nur als SMT-Formel dargestellt.

4.4 Export von Terminations- und Nicht-Terminationsargumenten

Die berechneten Terminations- und Nicht-Terminationsargumente werden nach Schleifen sortiert und während der Ausgabe der Statistiken in eine Datei exportiert, falls die Ausgabe von Dateien nicht grundsätzlich deaktiviert ist. Das gewählte Format ist für die Verwendung durch Menschen gedacht und nicht als maschinenlesbares Format konzipiert.

Falls die Nicht-Termination des Programmes bewiesen werden kann, entfernt der Terminations-Algorithmus den Fehlerzustand, der keinen Zustand des ursprünglichen Programmes repräsentiert, markiert den vorhergehenden abstrakten Zustand am Schleifenkopf als Fehlerzustand und bereitet den Fehlerpfad des Gegenbeispiels entsprechend auf. Auf diese Weise kann das Gegenbeispiel ohne weitere Anpassungen an CPAchecker im einem maschinenlesbaren Format als *graphml*-Datei[5] und in Form eines interaktiven Berichts [37] exportiert werden. In beiden Fällen wird der unendliche Fehlerpfad durch den Stamm und eine Iteration der Schleifentransition des nicht terminierenden Lassos repräsentiert. An diesem Beispiel ist deutlich zu erkennen, wie umfangreich die Integration der Terminationsanalyse in CPAchecker ist, sowie die daraus resultierenden Möglichkeiten im Bezug auf die Wiederverwendung vorhandener und zukünftige Komponenten.

4.5 Konfiguration

Eine auf den in diesem Kapitel beschriebenen Komponenten basierende Analyse lässt sich flexibel mit einer vorhanden Erreichbarkeitsanalyse kombinieren. Im folgenden werden die dafür notwendigen Optionen und weitere Konfigurationsmöglichkeiten einer Terminationsanalyse beschrieben:

- *analysis.algorithm.termination* aktiviert die Verwendung des Terminations-Algorithmus.
- *cpa.termination.refiner* setzt den von `TerminationARGBasedRefiner` verwendeten `Refiner`. Bei der Verwendung des CEGAR-Algorithmus muss diese Option genutzt und *cegar.refiner = cpa.termination.TerminationARGBasedRefiner* gesetzt werden.
- *TerminationCPA.cpa* setzt die Kind-CPA der `TerminationCPA`. In aller Regel wird dies die `CompositeCPA` sein. Die Verwendung der `TerminationCPA` kann zum Beispiel durch *ARGCPA.cpa = cpa.termination.TerminationCPA* erreicht werden.

[5] Bisher wurde kein genaues Format für Nicht-Termination definiert. Näheres findet sich auf der Webseite der SV-COMP 2016: `https://sv-comp.sosy-lab.org/2016/witnesses/`

- *termination.check* wird intern verwendet, um Konfigurationen zu ermöglichen, die für verschiedene Programmeigenschaften geeignet sind und je nach zu prüfender Eigenschaft eine passende Analyse laden.
- *termination.config* legt fest, welche Konfiguration für eine Terminationsanalyse geladen werden soll.
- *termination.lassoAnalysis.eigenvectors* bestimmt die Anzahl der in der Nicht-Terminationsanalyse verwendeteten generalisierten Eigenvektoren. Eine größere Zahl ermöglicht die Nicht-Termination von mehr Schleifen zu beweisen, verursacht jedoch gleichzeitig einen deutlich erhöhten Rechenaufwand. Standardmäßig werden drei Eigenvektoren verwendet.
- *termination.lassoAnalysis.externalSolverCommand* definiert die Kommandozeile, mit der ein externer SMT-Solver von LassoRanker gestartet wird. Die Standardeinstellung ist passend für Z3 gewählt.
- *termination.lassoAnalysis.linear.analysisType* legt die Analyseart für lineare Terminationsargumente fest:
  - *DISABLED* deaktiviert die Analyse.
  - *LINEAR* verwendet lineare Formeln.
  - *LINEAR_WITH_GUESSES* verwendetet lineare Formeln und versucht Eigenwerte der Schleife zu erraten. Dies ist die Standardeinstellung.
  - *NONLINEAR* ermöglicht eine nicht lineare Analyse, benötigt jedoch einen hierfür geeigneten SMT-Solver.
- *termination.lassoAnalysis.linear.externalSolver* aktiviert die Verwendung des externen SMT-Solvers an Stelle von SMTInterpol und ist standardmäßig abgeschaltet. Diese Option hat nur Einfluss auf die Synthese linearer Terminationsargumente.
- *termination.lassoAnalysis.maxTemplateFunctions* legt die maximale Zahl an Funktionen fest, die in der Abstiegsrelation eines Terminationsargumentes verwendet werden. Eine höhere Zahl ermöglicht die Synthese von Abstiegsrelationen für mehr Schleifen, steigert jedoch auch die Komplexität der Synthese. Standardmäßig werden maximal drei Funktionen verwendet.
- *termination.lassoAnalysis.nonStrictInvariants* bestimmt die maximale Zahl der nicht strikten Invarianten, die während der Synthese eines Terminationsargumentes generiert werden. Der Standardwert ist 3.

- *termination.lassoAnalysis.nonlinear.analysisType* legt die Analyseart für nicht lineare Terminationsargumente und Nicht-Terminationsargumente fest. Es können die selben Werte wie für *termination.lassoAnalysis.linear.analysisType* genutzt werden.
- *termination.lassoAnalysis.nonlinear.externalSolver* aktiviert die Verwendung des externen SMT-Solvers an Stelle von SMTInterpol und ist standardmäßig abgeschaltet. Diese Option hat nur Einfluss auf nicht lineare Terminationsargumente und Nicht-Terminationsargumente.
- *termination.lassoAnalysis.strictInvariants* bestimmt die maximale Anzahl der strikten Invarianten, die während der Synthese von Terminationsargumenten generiert werden. Der Standardwert ist 2.
- *termination.lassoBuilder.simplify* aktiviert die Vereinfachung der Stamm- und Schleifenformel bevor daraus Lassos erzeugt werden. Dies ist potentiell sehr teuer und daher standardmäßig deaktiviert.
- *termination.maxRepeatedRankingFunctionsPerLoop* legt fest, nach welcher Zahl an synthetisierten Terminationsargumenten, die bereits für die aktuell bearbeitet Schleife bekannt waren, die Analyse der aktuellen Schleife beendet wird, weil voraussichtlich kein Fortschritt mehr erzielt werden wird. Der Standardwert ist 10.
- *termination.resetReachedSetStrategy* definiert die Strategie zum Entfernen des Fehlerzustandes, nachdem ein Terminationsargument synthetisiert werden konnte. Dabei stehen die folgenden Optionen zu Auswahl:
  - *REMOVE_TARGET_STATE* entfernt nur den Fehlerzustand.
  - *REMOVE_LOOP* entfernt alle abstrakten Zustände die zur Schleife gehören. Dies ist die Standardeinstellung.
  - *RESET* entfernt alle Zustände und beginnt die nächste Iteration beim initialen Zustand.
- *termination.resultFile* beschreibt den Pfad der Datei, in die alle gefundenen Terminations- und Nicht-Terminationsargumente in einem für den Menschen lesbaren Format exportiert werden.

4.6 Integration der Prädikaten-Analyse

Die Verwendung der Prädikaten Analyse als Erreichbarkeitsanalyse erscheint sinnvoll, da sie in der Lage ist Beziehungen zwischen Variablen herzustellen und auf Grund der Bedingungen

der Abstiegsrelationen diese Anforderung zu erfüllen ist. Es sind jedoch einige Anpassungen an der Konfiguration notwendig beziehungsweise hilfreich um eine vernünftige Analyse durchführen zu können.

Zunächst ist es wichtig, die mit der Option *cpa.predicate.ignoreIrrelevantVariables* die Optimierung abzuschalten, welche dafür sorgt, dass nur Prädikate über für ein Erreichbarkeitsproblem relevante Variablen in die Pfadformel aufgenommen werden. Diese Optimierung würde zur Folge haben, dass die zur Sicherung aller Variablen am Schleifenanfang verwendeten Variablen nicht in der Pfadformel vorkommen würden und möglicherweise fälschlich angenommen würde, dass manche Variablen nicht relevant seien, obwohl dies nicht des Fall ist.

Weiterhin beherrscht die lineare Prädikaten-Analyse keine Bit-Operationen, die jedoch verwendet werden, um die Abstiegsrelation als C-Anweisung zu kodieren. Daher muss die durch den `TerminationState` ebenfalls als SMT-Formel bereitgestellte negierte Abstiegsrelation im $\downarrow_{\mathbb{P}}$-Operator der Prädikatenanalyse bereit gestellt werden. Dafür ist die Option *cpa.predicate.strengthenWithFormulaReportingStates* zu aktivieren.

Damit die von LassoRanker zur Synthese von Terminationsargumeten berechneten Schleifeninvarianten genutzt werden können, müssen einige zusätzliche Einstellungen aktiviert werden. Die Option *cpa.predicate.invariants.addToPrecision* bewirkt das Einfügen der Invarianten in die Präzision der Prädikaten-Analyse. Da es sich nicht sicher um Invariaten der ganzen Schleife, sondern nur des einen Lassos handelt, ist es nicht möglich die Invarianten durch Konjunktion mit Pfad- oder Abstraktionsformel zu verbinden. Weitere Details zur Verarbeitung der Invarianten durch die Prädikaten-Analyse und deren Nutzen sind in [42] beschrieben.

Außerdem muss die Überprüfung der von der Prädikatenanalyse generierten Gegenbeispiele durch eine weitere Analyse abgeschaltet werden, da die notwendige Instrumentierung des Gegenbeispiels bisher nicht implementiert ist.

Die Konfiguration *terminationAnalysis*[6] kombiniert die lineare Prädikatenanalyse mit dem Terminationsalgorithmus und der `TerminationCPA` zu einer vollständigen Terminationsanalyse. Dabei werden die in diesem Abschnitt genannten Anpassungen der Konfiguration vorgenommen.

4.7 Einschränkungen

Die in diesem Kapitel beschriebene Implementierung unterliegt einigen Restriktionen. Es kann nur das Terminationsverhalten von Schleifen und nicht von Rekursion untersucht wer-

[6] `https://svn.sosy-lab.org/software/cpachecker/trunk/config/terminationAnalysis.properties?p=23000`

den. Dies ist nicht darin begründet, dass das theoretische Konzept nicht auch auf rekursive Aufrufe anwendbar ist, sondern es ist bisher nicht implementiert worden, da die Unterstützung für Rekursion in CPAchecker nicht den für eine Terminationsanalyse notwendigen Umfang aufweist und daher im Rahmen dieser Arbeit nicht die Möglichkeit bestand die Terminationsanalyse auf Rekursion auszudehnen. Beispielsweise wäre es notwendig, nicht nur alle im CFA enthaltenen Schleifen zu bestimmen, sondern auch alle Rekursionen, die damit eine starke Zusammenhangskomponente über mehrere Funktionen hinweg bilden. Außerdem müsste zwingend eine Erreichbarkeitsanalyse benutzt werden, die Rekursion unterstützt.

Weiterhin ist es nicht möglich alle Abstiegsrelationen als Bedingung in C darzustellen, da unter anderem die Multiplikation eines Zeigers mit einer Konstanten keine zulässige Operation ist. In solchen Fällen wird versucht die Abstiegsrelation ausschließlich als SMT-Formel darzustellen. Falls der $\downarrow$-Operator der Erreichbarkeitsanalyse nicht in der Lage ist, diese Formel aus dem `TerminationState` zu extrahieren und entsprechend zu verwenden, wird das synthetisierte Terminationsargument keine Auswirkung haben, da der dazugehörige Fehlerpfad nicht ausgeschlossen werden kann.

Elemente eines Arrays dürfen nicht Teil einer Abstiegsrelation sein, da der entsprechende Wert vor der Schleifeniteration nicht zur Verfügung steht. Um diesem Umstand abzuhelfen, wäre es notwendig, entweder das ganze Array oder gezielt die relevanten Zellen zu Beginn der Schleife zu kopieren. Der erste Ansatz ließe sich in C mit der Funktion `memcpy` umsetzten, wobei jedoch die Länge des zu kopierenden Speicherbereiches bekannt sein müsste. Diese richtig zu bestimmen, würde zusätzlichen Aufwand erfordern. Die alternative Möglichkeit ist ebenfalls aufwendig, da die am Schleifenanfang zu sichernde Information nicht nur vom Programm, sondern auch von den bisher gefunden Terminationsargumenten abhinge und daher die Wiederverwendung des `ReachedSet`s nach einer Iteration des Algorithmus nicht immer möglich wäre.

Die verwendete Erreichbarkeitsanalyse muss entweder Bitoperationen präzise verarbeiten können oder der $\downarrow$-Operator muss die Extraktion der Abstiegsrelation als SMT-Formel aus einem `FormulaReportingState` unterstützen, da anderenfalls alle Abstiegsrelationen ignoriert würden.

Falls *CEGAR* in der Erreichbarkeitsanalyse zum Einsatz kommt, setzt die Terminationsanalyse zur Zeit die Verwendung eines `ARGBasedRefiner` voraus, damit auch der Fehlerpfad entsprechend der Beschreibung in Abschnitt 4.2 ergänzt werden kann.

5 Evaluation

In diesem Kapitel wird nun gezeigt, dass das Konzept und die zugehörige Implementierung in CPAchecker aus den vorangegangenen Kapiteln nicht nur theoretisch verwertbar sind, sondern auch in der praktischen Verwendung gute Ergebnisse erzielen. Dafür werden zunächst zwei unterschiedliche Konfigurationen der in CPAchecker implementierten Terminationsanalyse gegenübergestellt. Anschließend wird CPAchecker mit drei anderen Verifikationswerkzeugen in Bezug auf die darin implementierten Terminationsanalysen verglichen.

Diese Evaluation bezieht sich auf Revision *23354* des *trunk* von CPAchecker[1]. Es kommt jeweils der in Kapitel 4 beschriebene Algorithmus in Verbindung mit der Prädikaten-Analyse zum Einsatz. Die Grundkonfiguration ist in den Abschnitten 4.5 und 4.6 beschrieben.

Alle Experimente wurden auf baugleichen Rechnern durchgeführt, welche mit einer CPU vom Typ Intel Xeon E5-2650 v2 ausgerüstet sind, die eine Taktfrequenz von 2,60 GHz aufweist, 135 GB Speicher besitzen und unter Ubuntu 16.04 mit einem Linux Kernel 4.4.0 liefen. Als Java-Laufzeitumgebung kam Java HotSpot(TM) 64-Bit Server VM 1.8.0_101 zum Einsatz. Jede Ausführung eines Verfikationswerkzeuges wurde durch BenchExec[2] auf 2 CPU-Kerne, 15 GB Speicher und 15 min CPU-Zeit eingeschränkt. Damit entspricht die Ausführungsumgebung weitgehend der der *Competition on Software Verification* (SV-COMP) 2016[3].

Als Problemstellung dienten alle Programme der SV-COMP 2016[4] ohne Rekursion aus der Kategorie *Termination*, die um die Programme aus der Kategorie *Loops* erweitert wurden. Das Terminationsverhalten der letztgenannten war in den meisten Fällen noch nicht bekannt und wurde durch übereinstimmende Resultate mehrerer Verifikationswerkzeuge oder, falls diese widersprüchlich waren, durch manuelle Überprüfung festgestellt und die Programme entsprechend gekennzeichnet. Da nicht alle Programme durch mindestens ein Verfikationswerzeug lösbar sind, gibt es jedoch nach wie vor Programme, deren Terminationsverhalten unbekannt ist. Außerdem mussten einige fehlerhafte Programme korrigiert werden, für die

[1] `https://svn.sosy-lab.org/software/cpachecker/trunk/?p=23354`
[2] `https://github.com/sosy-lab/benchexec`
[3] Die SV-COMP 2016 erlaubte die Verwendung von acht CPU-Kernen und als Betriebssystem kam noch Ubuntu 14.04 zum Einsatz.
[4] `https://github.com/sosy-lab/sv-benchmarks/`

CPAchecker teilweise erstmals zeigen konnte, dass die bisherige Annahme nicht zutreffend ist. Für die im folgenden beschriebenen Experimente wurden insgesamt 733 Programme verwendet, von denen 570 terminieren und 136 nicht terminieren. Das Terminationsverhalten von 27 Programmen ist unbekannt.

Die Regeln[5] der SV-COMP 2016 legen das Verhalten einiger verwendeter Funktionen fest. Sie besagen, dass kein Aufruf der Funktion `void __VERIFIER_error()` jemals zurückkehrt und `void __VERIFIER_error() { abort(); }` als Implementierung angenommen werden kann, weswegen durch diese Funktion keine Nicht-Termination verursacht werden kann. Für die Funktion `void __VERIFIER_assume(int)` gilt laut den Regeln hingegen, dass sie eine Endlosschleife ausführt, falls der übergebene Parameter 0 ist und anderenfalls ohne Seiteneffekt sofort zurückkehrt[6]. Da keiner der Teilnehmer an der SV-COMP 2016 dies als Nicht-Termination identifiziert hat und es fraglich erscheint, ob diese Art von Nicht-Termination tatsächlich beabsichtigt ist, soll ein Aufruf der Funktion `void __VERIFIER_assume(int)` mit 0 als Parameter im Folgenden äquivalentes Verhalten wie ein Aufruf von `void __VERIFIER_error()` aufweisen.

5.1 Vergleich linearer und nicht linearer Analyse von Lassos

In diesem Abschnitt werden zwei unterschiedliche Konfiguration verglichen:

- *termination-linear* ist die zu Beginn dieses Kapitels beschriebene Grundkonfiguration, die SMTInterpol und nur lineare Formeln während der Synthese von Terminations- und Nicht-Terminationsargumenten verwendet.

- *termination-non-linear* basiert auf der Grundkonfiguration, benutzt jedoch lineare Formeln und SMTInterpol nur während der Synthese von linearen Terminationsargumenten. Für nicht lineare Terminationsargumente sowie für Nicht-Terminationsargumente kommen nicht lineare Formeln und Z3 zum Einsatz, da SMTInterpol keine nicht linearen Formeln verarbeiten kann.

In allen anderen Optionen stimmen die beiden Konfigurationen überein.

Tabelle 5.1 zeigt die Resultate der beiden Analysen im Vergleich. Dabei sind die richtigen Ergebnisse grün und die falschen rot dargestellt. Ein Verifikationswerkzeug gibt *true* aus, wenn es beweisen konnte, dass ein Programm terminiert, *false*, wenn sie zeigen konnte, dass ein Programm eine nicht terminierende Ausführung besitzt, und *unknown*, wenn es weder das eine noch das andere beweisen konnte. Die in grüner Schrift gehaltenen Stati sind korrekte

[5] https://sv-comp.sosy-lab.org/2016/rules.php

[6] `void __VERIFIER_assume(int expression) { if (!expression) { LOOP: goto LOOP; }; return; }` ist als Referenzimplementierung angegeben.

Tabelle 5.1: Resultate der Konfigurationen *termination-linear* und *termination-non-linear*

Status	*termination-linear*	*termination-non-linear*
true	269	270
false	59	59
false	1	1
unknown	136	136
timeout	255	254
parsing failed	13	13

Ergebnisse, die roten falsche. Wenn die Analyse sich mit einem Fehler beendet oder eine Ressourcenbeschränkung überschritten wurde, wird das Ergebnis in Magenta dergestalt. Die Ergebnisse der beiden Konfigurationen weisen fast keine Unterscheide auf. *termination-linear* kann 269 terminierende Programme lösen, *termination-non-linear* eins mehr. Die Zahl der richtig als nicht terminierend erkannten Programme liegt bei beiden Konfigurationen bei 59. Beide Analysen geben fälschlicherweise für ein terminierendes Programm Nicht-Termination als Resultat an. Dies ist in der zu unpräzisen Modellierung von unbeschränkten Arrays begründet. In 136 Fällen geben die Analysen auf und 255 beziehungsweise 254 Mal wird das Zeitlimit überschritten. 13 Programme können von CPAchecker nicht verarbeitet werden, da sie noch `#include`-Anweisungen enthalten. Insgesamt gibt es nur drei Programme, deren Ergebnisse unterschiedlich sind. *termination-linear* kann ein terminerendes richtig lösen, das bei *termination-non-linear* zu einer Überschreitung des Zeitlimits führt und für zwei Programme trifft genau das Gegenteil zu. Damit wird noch deutlicher, dass es keine signifikanten Unterschiede im Bezug auf die Resultate zwischen den beiden Konfigurationen gibt.

In Abbildung 5.1 ist die CPU-Zeit der korrekten Ergebnisse der beiden Konfigurationen zu erkennen. Auf der horizontalen Achse sind die jeweils bezüglich der CPU-Zeit aufsteigend sortierten Ergebnisse aufgetragen. Die vertikale Achse stellt die benötigte CPU-Zeit jedes Ergebnisses auf einer logarithmischen Skala zur Basis zehn dar. Die Kurve einer schnelleren Analyse verläuft tiefer und je weiter rechts die Kurve endet, desto mehr Programme konnten erfolgreich analysiert werden. Es ist zu erkennen, dass die orangefarbene Kurve, die *termination-linear* repräsentiert, nur sehr geringe Unterschiede zu der violetten Kurve, die *termination-non-linear* repräsentiert, aufweist. Beide beginnen bei ungefähr 6 s und enden beim Zeitlimit von 900 s und 328 beziehungsweise 329 richtigen Resultaten. Im Bereich zwischen 10 s und 50 s ist *termination-linear* minimal schneller. Der größte Unterschied von circa 10 s liegt beim 300. schnellsten korrekten Resultat.

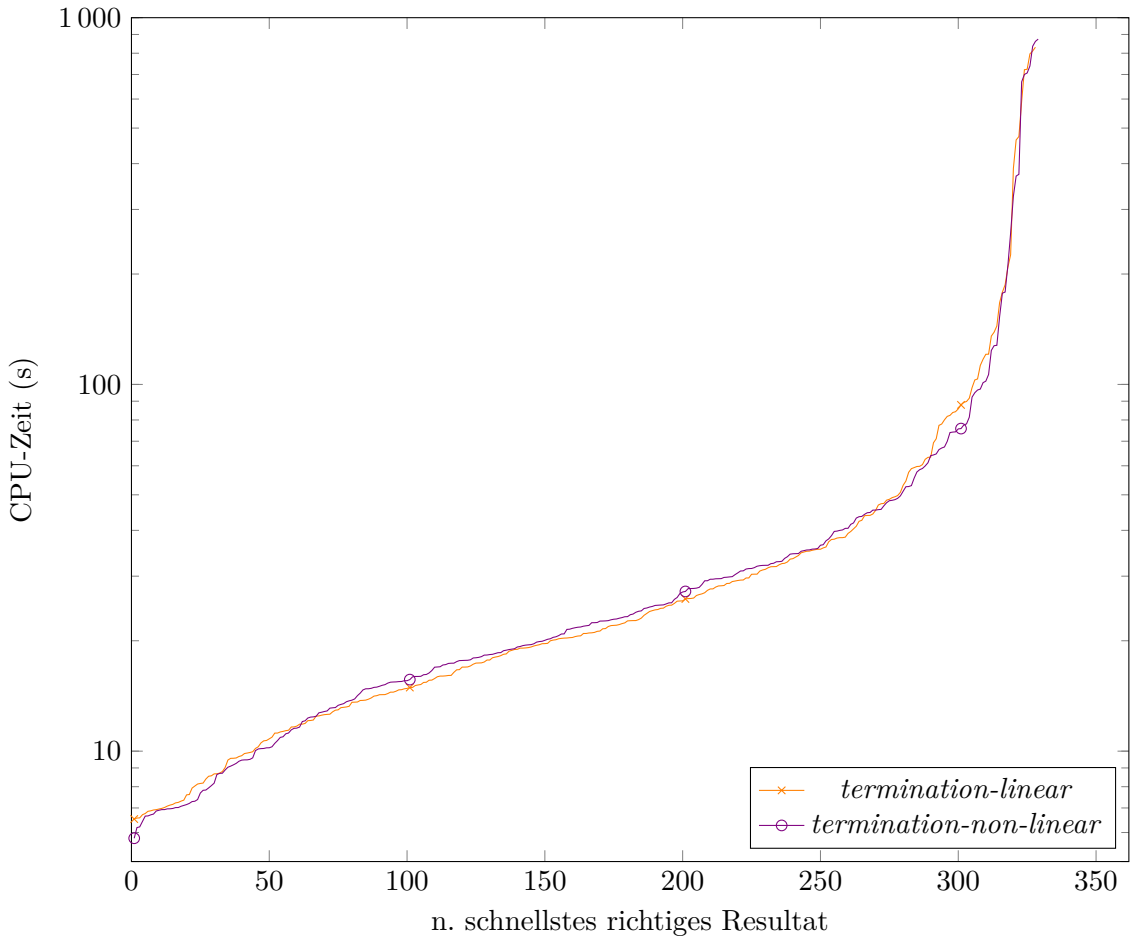

Abbildung 5.1: Nach CPU-Zeit sortierte korrekte Ergebnisse der Konfigurationen *termination-linear* und *termination-non-linear*

Die Abbildungen 5.2 und 5.3 zeigen zwei Streudiagramme, von denen das erste die CPU-Zeit und das zweite den Speicherverbrauch darstellt. Die Werte von *termination-linear* verwenden die horizontale Ache und die von *termination-non-linear* die vertikale Ache. Beide Achse sind logarithmisch zur Basis zehn skaliert. Jedes blaues Kreuz repräsentiert ein von beiden Konfigurationen analysiertes Programm.

Da fast alle Datenpunkte in Abbildung 5.1 nahe bei der Winkelhalbierenden der beiden Achsen liegen, gibt es kaum Programme mit besonders bedeutenden Unterschieden im Verbrauch von Rechenzeit zwischen den beiden Konfigurationen. Nur für einzelne Programme ist das Verhältnis der CPU-Zeiten größer als 2 beziehungsweise kleiner als 0,5. Auch lässt sich gut erkennen, dass das erfolgreiche Analysieren der meisten Problemstellungen eine CPU-Zeit zwischen 10 s und 100 s benötigt. Bei der Gruppe von Punkten links unterhalb davon, handelt es sich um die Programme, welche schon beim Einlesen zu einem Fehler führen.

Aus den von CPAchecker für jedes Programm ausgegebenen Statistiken ergibt sich eine Gesamtzeit[7] von 2 310s zum Synthetisieren von Terminationsargumenten mit *termination-linear* und 4 200s mit *termination-non-linear*. Wenn nur die richtig gelösten Programme betrachtet werden, betragen die Werte 182s und 228s. Da es sich dabei jedoch nur um circa ein Achtel der für die Konstruktion der Lassos und weniger als ein Zehntel der für die Synthese von Terminationsargumenten benötigten Zeit handelt, ist kein Unterschied bei der Betrachtung der insgesamt von den Analysen verbrauchten Zeit erkennbar.

Die Datenpunkte des Speicherverbrauchs liegen wie in Abbildung 5.3 zu erkennen im Vergleich zum Diagramm der CPU-Zeit in Abbildung 5.2 noch deutlich näher an der Winkelhalbieren und damit ist der Speicherverbrauch der beiden Konfigurationen für die meisten analysierten Programme nahezu gleich. Die Streuung der Punkte nimmt mit höherem Speicherverbrauch zu und erreicht ihr Maximum bei circa 4GB, ist jedoch nur für vier Programme besonders signifikant ausgeprägt. Daraus lässt sich schließen, dass die von der nicht linearen Analyse verwendete Instanz von Z3 im Vergleich zum Rest der Analyse keine relevante Menge an Speicher benötigt, obwohl Z3 als externer Prozess gestartet wird. Auffällig ist die Gruppe von Datenpunkten links unten, welche wie in Abbildung 5.2 Programme repräsentieren, die schon durch Frontend nicht verarbeitet werden können.

Dieser Abschnitt zeigte die geringen Unterschiede der beiden Konfigurationen sowohl im Hinblick auf die Analyseergebnisse als auch bezüglich der zur Berechnung dieser notwendigen Ressourcen auf. Dies ist damit zu erklären, dass der Anteil an nicht linearen Berechnungen verhältnismäßig gering ist.

[7] Hier wird die tatsächlich vergangene Zeit und nicht die CPU-Zeit gemessen.

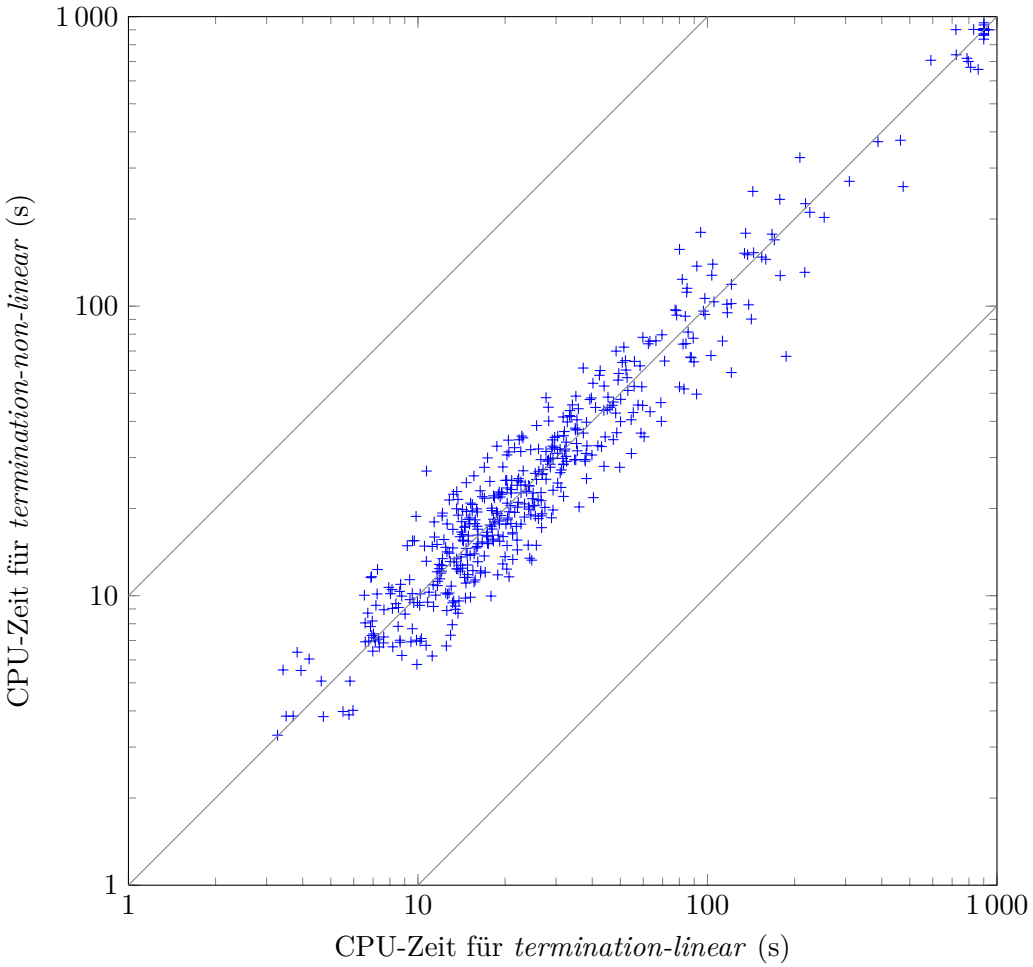

Abbildung 5.2: Streudiagramm der CPU-Zeiten der Konfigurationen *termination-linear* und *termination-non-linear*

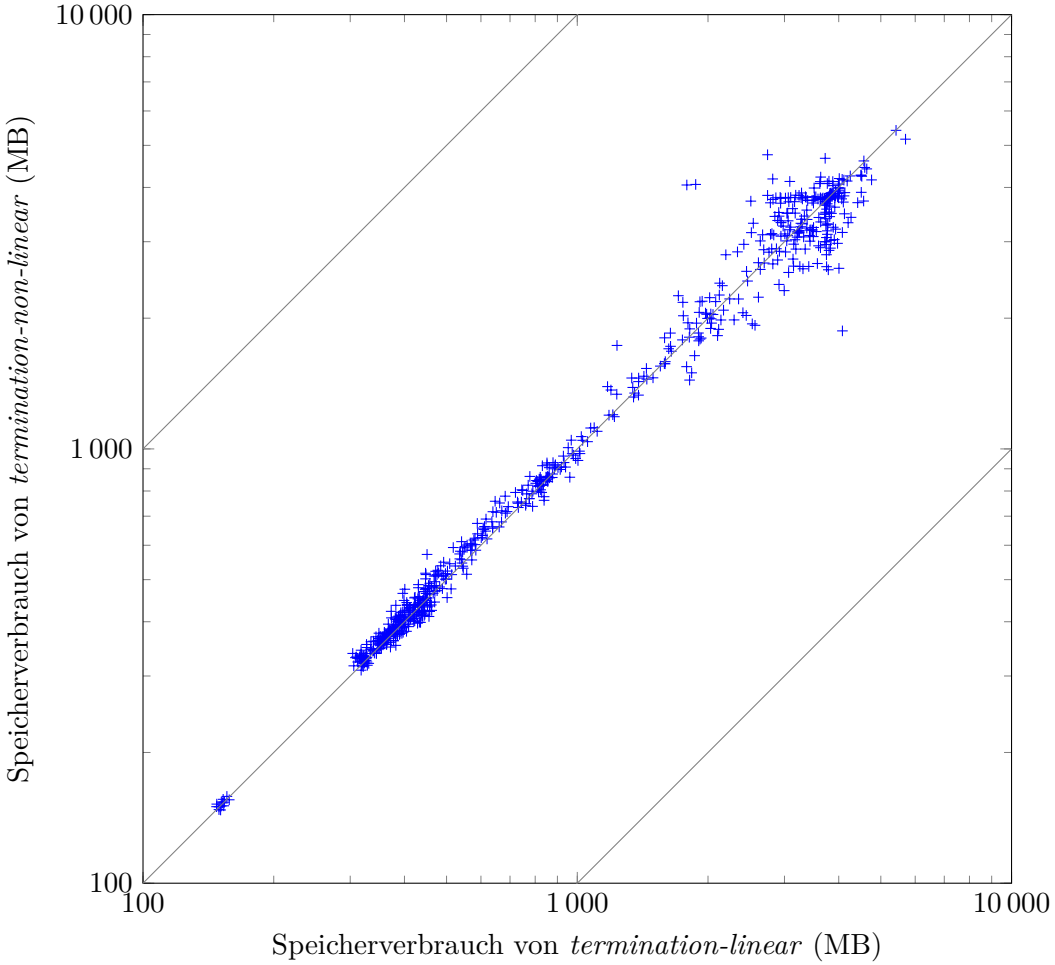

Abbildung 5.3: Streudiagramm des Speicherverbrauchs der Konfigurationen *termination-linear* und *termination-non-linear*

5.2 Vergleich mit anderen Verifikationswerkzeugen

Um zu demonstrieren, dass die Implementierung in CPAchecker durchaus konkurrenzfähig ist, wird die Konfiguration *termination-linear* mit der aktuellen stabilen Version aller anderen Verifikationswerkzeuge verglichen, die im Rahmen der SV-COMP 2016 ernsthaft[8] an der Kategorie *Termination* teilgenommen haben:

- AProVE [9] in Revision 05e59fa[10]
- SeaHorn[11] in Revision 0.1.0[12]
- Ultimate Automizer[13] in Revision 9517084e[14]

5.2.1 Schleifen-Programme

Tabelle 5.2 zeigt die Resultate der vier Verifikationswerkzeuge im Vergleich. Die Anzahl der richtigen Antworten für terminierende Programme weist für AProVE (278), CPAchecker (269) und SeaHorn (261) nur geringe relative Unterschiede auf. AProVE kann die Nicht-Termination von 71, CPAchecker von 59 und SeaHorn von 83 nicht terminierenden Programmen beweisen. Ultimate Automizer hingegen ist in der Lage Termination von 431 und Nicht-Termination von 111 Programmen korrekt zu berechnen und kann somit circa 60 % mehr terminierende Programme erfolgreich analysieren. Ultimate Automizer ist auch im Stande das Terminationsverhalten von erheblich mehr nicht terminierenden Programme korrekt zu bestimmen und keines der Ergebnisse ist falsch. AProVE behauptet fälschlich für drei terminierende Programme sie würde nicht terminieren und CPAchecker für eines. SeaHorn hingegen tut dies für 44 Programme und erkennt bei zwei Programmen fälschlich Termination. Beides ist sehr negativ zu bewerten, da mehr als ein Drittel der *false*-Antworten inkorrekt sind und es für den Benutzer von SeaHorn keine Möglichkeit gibt zuerkennen, wenn Programme fälschlich als terminierend klassifiziert werden. Somit sind die Ergebnisse von SeaHorn praktisch wertlos. Da CPAchecker für als nicht terminierend klassifizierte Programme in der Lage ist ein Gegenbeispiel in einem maschinenlesbaren Format und als interaktiven Bericht in Form einer HTML-Seite auszugeben, fällt das eine

[8] Es werden hier nur Verifikationswerkzeuge mit positiver Punktezahl berücksichtigt.
[9] `http://aprove.informatik.rwth-aachen.de/index.asp?subform=home.html`
[10] Die aktuelle Version ist jeweils unter `http://aprove.informatik.rwth-aachen.de/downloads/cli/aprove.jar` verfügbar.
[11] `http://seahorn.github.io/`
[12] `https://github.com/seahorn/seahorn/releases/download/v0.1.0/SeaHorn-0.1.0-Linux-x86_64.tar.gz`
[13] `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer`
[14] `http://www.informatik.uni-freiburg.de/~heizmann/UltimateAutomizer-20160701-TerminationWithoutBlockEncoding.zip`

Tabelle 5.2: Resultate der Verifikationswerkzeuge

Status	APROVE	CPACHECKER	SEAHORN	ULTIMATE AUTOMIZER
true	278	269	261	431
false	71	59	83	111
true	0	0	2	0
false	3	1	44	0
unknown	0	136	0	98
timeout	281	255	115	66
out of memory	1	0	0	0
error	99	13	228	27

falsche Ergebnis nicht so stark ins Gewicht, weil das Gegenbeispiel leicht überprüft werden kann. Nur CPACHECKER und ULTIMATE AUTOMIZER sind in der Lage Programme zu erkennen, die sie nicht lösen können, und darauf hin die Analyse ohne Ergebnis abzubrechen. Einzig APROVE genügt das Speicherlimit von 15 GB in einem Fall nicht. APROVE und SEAHORN beenden sich mit 99 und 228 Mal häufig mit einem Fehler. Insgesamt ist bei Betrachtung der einzelnen Verifikationsprobleme zu erkennen, dass die drei Teilnehmer der SV-COMP 2016 bei der Bearbeitung der Programme, welche nicht Teil der Termination-Kategorie in der SV-COMP 2016 waren, merklich schlechtere Ergebnisse erzielen. Dies ist bei SEAHORN besonders deutlich, da alle 44 nicht korrekten *false*-Antworten die zusätzlichen Programme betreffen und für diese außerdem nur 22 korrekte Resultate vorliegen.

In Tabelle 5.3 sind die mittlere CPU-Zeit und der mittlere Speicherverbrauch für die vier Verifikationswerkzeuge aufgelistet. Dabei enthalten die beiden oberen Zeilen die Werte gemittelt über alle Programme und die beiden unteren gemittelt über die korrekten Ergebnisse eines Verifikationswerkzeuges. APROVE und CPACHECKER sind mit durchschnittlich 409 s beziehungsweise 346 s erheblich langsamer als SEAHORN und ULTIMATE AUTOMIZER mit 168 s beziehungsweise 132 s. APROVE weist deutlich den höchsten Speicherverbrauch von 2 880 MB auf. SEAHORN benötigt jedoch im Durchschnitt nur 67,6 MB Speicher und damit weniger als ein Fünfzehntel der drei anderen Verifikationswerkzeuge. Falls nur die richtig gelösten Ergebnisse betrachtet werden, reduziert sich jener Wert sogar auf 41,8 MB. Das ist weniger als 10 % der Werte von ULTIMATE AUTOMIZER (532 MB) und CPACHECKER (598 MB). Die zur Berechnung der korrekten Ergebnisse von APROVE benötigte durchschnittlich Speichermenge ist mit 1290 MB noch einmal mehr als doppelt hoch. Die mittlere CPU-Zeit liegt mit 44,8 s knapp unter der von CPACHECKER mit 47,5 s. SEAHORN ist bei dieser Betrachtung mit durchschnittlich 16,9 s eindeutig das schnellste Verifikationswerkzeug. Der sehr geringe Ressourcenverbrauch von SEAHORN ist mit hoher Sicherheit auch auf die

Tabelle 5.3: Ressourcenverbrauch der Verifikationswerkzeuge

Status	APROVE	CPACHECKER	SEAHORN	ULTIMATE AUTOMIZER
CPU-Zeit (s)	409	346	168	132
Speicher (MB)	2 880	1 560	67,6	1 170
CPU-Zeit (s) (korrekte Ergebnisse)	44,8	47,5	16,9	31,1
Speicher (MB) (korrekte Ergebnisse)	1 290	598	41,8	532

Implementierung in C und C++ zurückzuführen. Die anderen drei Verifikationswerkzeuge sind in Java verfasst.

Die hohe Geschwindigkeit von SEAHORN ist auch in Abbildung 5.4 zu erkennen, die in gleicher Art wie Abbildung 5.1 die CPU-Zeit der richtigen Ergebnisse zeigt. Die rote Kurve von SEAHORN beginnt bei weniger als einer Sekunde und es ist zu erkennen, dass SEAHORN in der Lage ist über 300 Programme erfolgreich in unter zehn Sekunden zu analysiert. In der weiteren Zeit bis zum Zeitlimit von 900 s kann jedoch SEAHORN weniger als 30 zusätzliche korrekte Resultate berechnen. Die grüne Kurve von APROVE beginnt bei 4 s und verläuft bis circa zum 200. schnellsten Erdgebiss relativ flach und steigt im weiteren Verlauf in der logarithmischen Skalierung annähernd konstant, so dass sie knapp rechts der roten Kurve endet. CPACHECKER benötigt mindestens sieben Sekunden um ein richtiges Ergebnis zu produzieren. Die gelbe Kurve liegt immer oberhalb der roten und nur im Bereich um 100 s minimal unterhalb der grünen. Damit ist CPACHECKER etwas langsamer als APROVE und liegt bei der Zahl der richtigen Ergebnisse knapp hinter APROVE und SEAHORN. Die blaue Kurve von ULTIMATE AUTOMIZER beginnt bei über 10 s, liegt jedoch für mehr als 400 Ergebnisse unterhalb von 30 s und endet bei knapp 550 richtigen Resultaten weit rechts der anderen drei Kurven. Damit zahlt sich der offenbar hohe initiale Aufwand von ULTIMATE AUTOMIZER insgesamt aus.

Um auch die falschen Ergebnisse mit in die Bewertung der einzelnen Verifikationswerkzeuge einfließen zu lassen, soll wird nun das Bepunktungsschema der SV-COMP 2016[15] herangezogen. Dabei werden richtige Ergebnisse für Programme ohne Verletzung der Spezifikation mit zwei und mit Verletzung der Spezifikation mit einem Punkt bewertet. Falsche Ergebnisse erhalten eine negative Punktzahl in Höhe des 16-fachen Wertes des entsprechenden richtigen Resultates.

[15] `https://sv-comp.sosy-lab.org/2016/rules.php#scores`

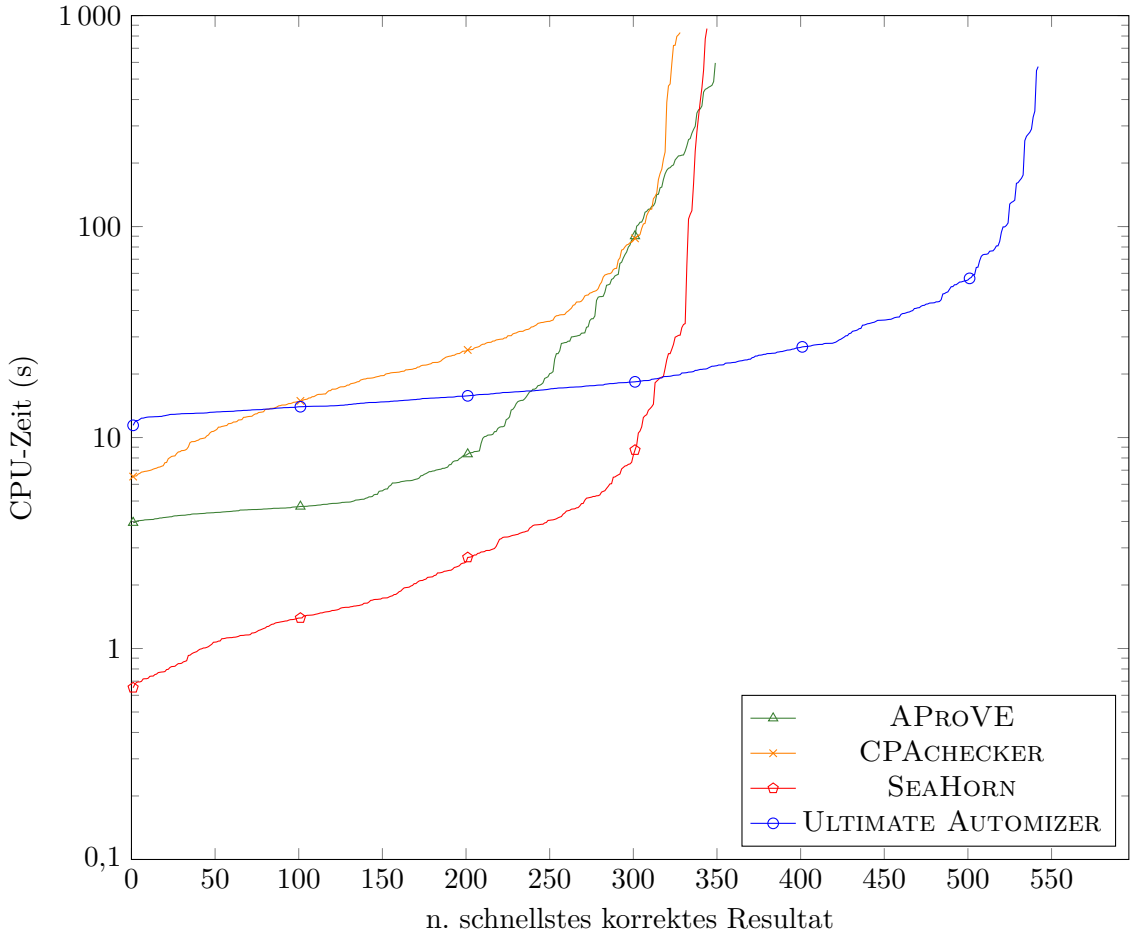

Abbildung 5.4: Nach CPU-Zeit sortierte korrekte Ergebnisse der Verifikationswerkzeuge

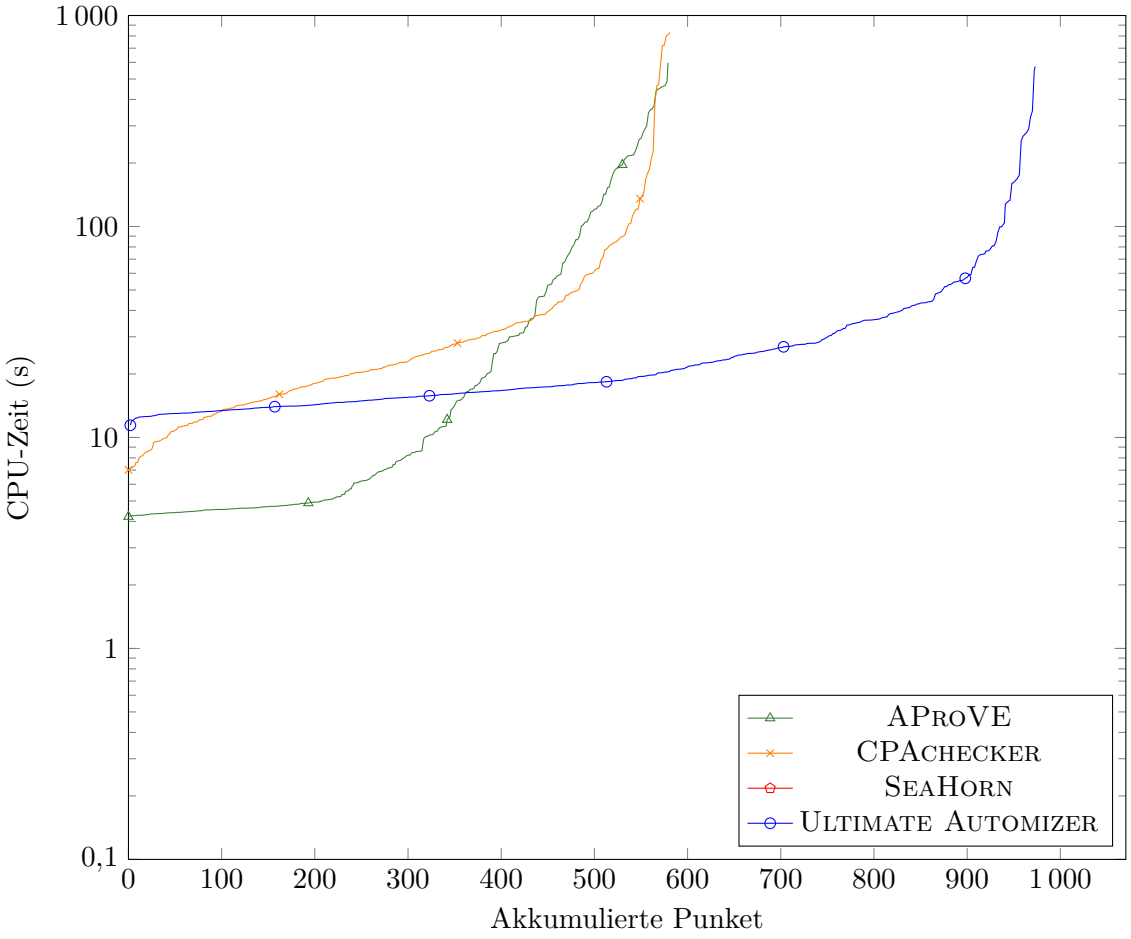

Abbildung 5.5: Nach CPU-Zeit sortierte akkumulierte Punkte der Verifikationswerkzeuge gemäß des Schemas der SV-COMP 2016

Abbildung 5.5 zeigt wie Abbildung 5.4 die von den Verifikationswerkzeugen jeweils benötigte CPU-Zeit. Diesmal sind jedoch auf der horizontalen Achse die akkumulieren Punkte dargestellt. Die Kurve von SeaHorn liegt vollständig links außerhalb des dargestellten Bereichs, da sie auf der Grund sehr hohen Zahl an falschen Resultaten bereits bei -163 endet. Die anderen drei Kurven weisen einen sehr ähnlichen Verlauf wie in Abbildung 5.4 auf, wobei die gelbe Kurve von CPAchecker in einem erheblich größeren Intervall von 30 s bis 300 s unterhalb der grünen von AProVE verläuft, als dies in Abbildung 5.4 der Fall ist, und minimal rechts der grünen Kurve endet. Unter Berücksichtigung der falschen Ergebnisse ist somit CPAchecker nicht mehr schlechter zu bewerten als AProVE.

Die in diesem Abschnitt erläuterten Daten lassen klar erkennen, dass die Terminationsanalyse in CPAchecker je nach Berwertungsmaßstab minimal besser oder schlechter als AProVE bei der Analyse von C Programmen mit Schleifen ist. Der Ressourcenverbrauch von SeaHorn ist in den meisten Fällen zwar bedeutend geringer der von CPAchecker und AProVE, aber die Ergebnisse von SeaHorn sind auf Grund der außergewöhnlich hohen Zahl nicht korrekter Ergebnisse nicht verwertbar. Ultimate Automizer ist mit einem moderaten Ressourcenverbrauch und der mit großem Abstand maximalen Zahl richtiger Ergebnisse als das beste der vier Verifikationswerkzeuge zur Bestimmung des Terminationsverhaltens von C-Programmen mit Schleifen anzusehen.

5.2.2 Schleifen-Programme ohne Zeiger

Nun soll die Menge der Verifikationsprobleme auf die 496 Programme eingeschränkt werden, die keine Zeiger enthalten. Die in Tabelle 5.4 aufgelisteten Resultate ähneln denen aus dem vorangegangen Abschnitt. Größter Unterschied ist die mit 289 um 142 geringere Zahl an durch Ultimate Automizer richtig analysierten terminierenden Programmen. An der Differenz zwischen dieser Tabelle und Tabelle 5.2 lässt sich erkennen, das Ultimate Automizer für kein nicht terminierendes Programm mit Zeigern die Nicht-Termination beweisen kann. AProVE gelingt dies in einem Fall, CPAchecker in vier und SeaHorn in drei Fällen. CPAchecker kann mit 249 terminierenden Programmen ohne Zeiger circa 25 % mehr richtig analysieren als AProVE und SeaHorn.

Abbildung 5.6 zeigt die CPU-Zeit der richtigen Resultate aufsteigend geordnet nach der CPU-Zeit für die vier Verifikationswerkzeuge. Das Diagramm unterscheidet sich von der entsprechenden Abbildung 5.4 aus dem vorhergehenden Abschnitt in der relativ zu den anderen Kurven weniger weit nach rechts verlaufenden blauen Kurve von Ultimate Automizer und in der Reihenfolge der rechten Enden der drei anderen Kurven. Die gelbe Kurve von CPAchecker endet am weitesten rechts und die grüne von AProVE liegt vollständig oberhalb der roten von SeaHorn.

Tabelle 5.4: Resultate der Verifikationswerkzeuge für Programme ohne Zeiger

Status	APROVE	CPACHECKER	SEAHORN	ULTIMATE AUTOMIZER
true	202	249	205	289
false	70	55	80	111
true	0	0	2	0
false	3	0	43	0
unknown	0	115	0	70
timeout	104	70	100	14
out of memory	1	0	0	0
error	47	4	66	12

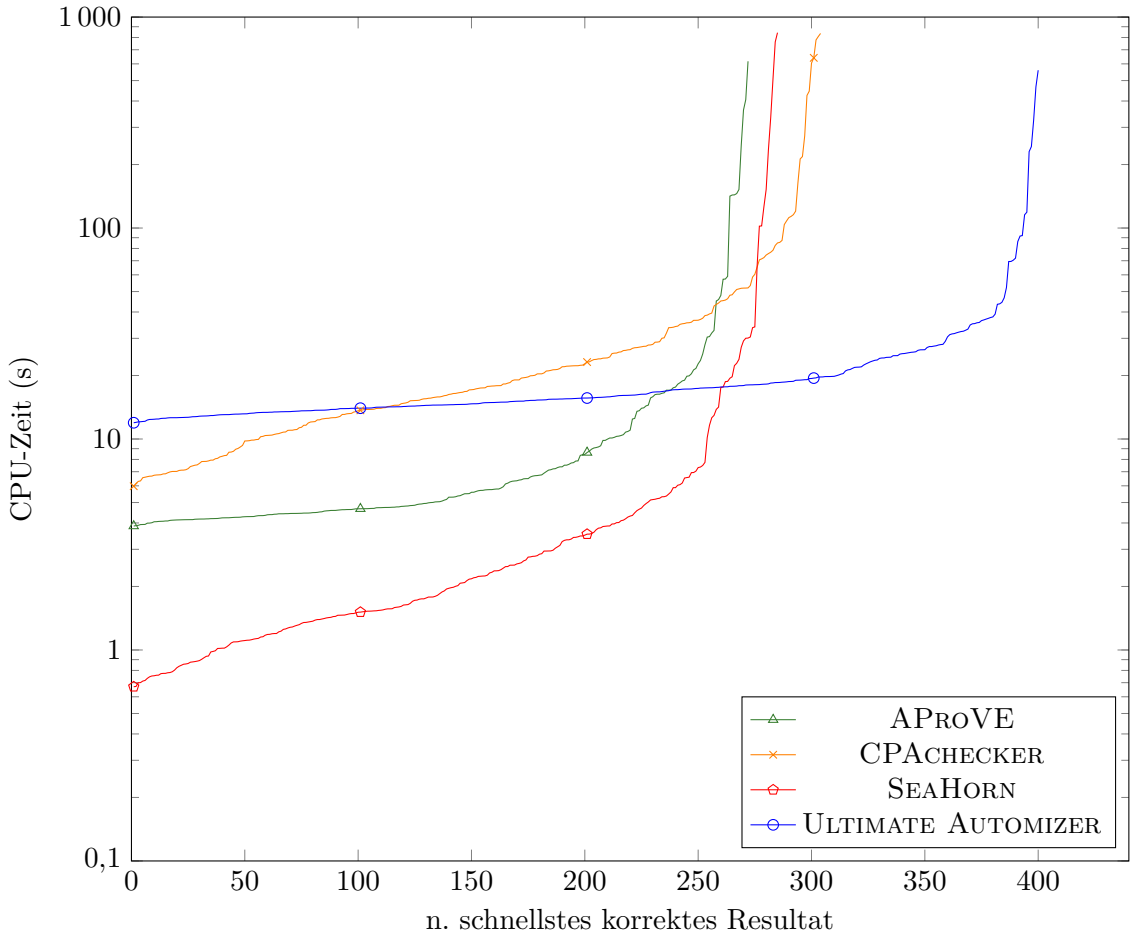

Abbildung 5.6: Nach CPU-Zeit sortierte korrekte Ergebnisse der Verifikationswerkzeuge für Programmen ohne Zeiger

Insgesamt lässt sich deutlich erkennen, dass CPAchecker etwas bessere Ergebnisse als AProVE erzielt, falls nur Programme ohne Zeiger betrachtet werden. Auch ohne Berücksichtigung der hohen Zahl falscher Ergebnisse schneidet SeaHorn etwas schlechter als CPAchecker ab. Ultimate Automizer erreicht auch auf den hier ausgewählten Programmen eindeutig die besten Resultate, auch wenn der Unterschied erheblich geringer ausfällt, da Ultimate Automizer bei der Analyse terminierender Programme mit Zeigern offenbar auffallend gut ist.

6 Zusammenfassung

Der in dieser Arbeit beschriebene Ansatz zur Umsetzung einer Terminationsanalyse mit Hilfe des Konzepts einer CPA ermöglichte die erfolgreiche Implementierung eines neuen Algorithmus und Analysetyps in CPAchecker, die den Funktionsumfang um die bisher nicht unterstützte Programmeigenschaft Termination erweitert. Am Beispiel der Prädikaten-Analyse konnte demonstriert werden, dass nicht nur die Wiederverwendung einzelner Komponenten durchführbar ist, sondern sogar die direkte Nutzung ganzer vorhandener Analysen möglich ist, wodurch die bisherigen und zukünftigen Möglichkeiten von CPAchecker sehr gut genutzt werden können. Der Vergleich mit anderen Verifikationswerkzeugen zeigte, dass die Implementierung in CPAchecker nur von einem von drei Verifikationswerkzeugen übertroffen wird, wobei Programme mit Zeigern auf Grund der hohen Zahl an Disjunktionen Schwierigkeiten bereiten, die jedoch durch Verbesserungen bei der Integration von LassoRanker zu überwinden sein müssten, indem weitere dort vorhandene Komponenten genutzt werden.

Die Integration weiterer Verfahren zur Synthese von Terminations- oder Nicht-Terminationsargumenten könnte für Pfade hilfreich sein, bei denen bisher kein geeignetes Argument berechnet werden kann. Eine Parallelisierung dieser Berechnungen und die gleichzeitige Analyse aller Schleifen könnte zu besseren Ergebnissen führen. Die Unterstützung von Rekursion ist vornehmlich ein Implementierungsproblem, da die vorgestellten theoretischen Konzepte direkt drauf angewandt werden können. Weitere Experimente mit anderen Erreichbarkeitsanalysen und verschiedenen Konfigurationen könnten weitere Hinweise für mögliche Optimierungen geben. Die Möglichkeit zur Validierung von Gegenbeispielen und Korrektheitsbeweisen anderer Verifikationswerkzeuge würde die Vertrauenswürdigkeit der Ergebnisse durch die automatische Überprüfung der Resultate erhöhen.

Insgesamt wurden die Fähigkeiten von CPAchecker in erheblichem Umfang erweitert, da nun erstmals eine Lebendigkeitseigenschaft eines Programmes bewiesen werden kann. Darauf aufbauend sind in Zukunft auch Analysen beliebiger Kombinationen von Lebendigkeits- und Sicherheitseigenschaften denkbar.

Literatur

[1] B. Alpern und F. B. Schneider. »Defining Liveness«. In: *Inf. Process. Lett.* 21.4 (1985), S. 181–185. DOI: 10.1016/0020-0190(85)90056-0.

[2] D. Babic u. a. »Proving termination of nonlinear command sequences«. In: *Formal Asp. Comput.* 25.3 (2013), S. 389–403. DOI: 10.1007/s00165-012-0252-5.

[3] A. Bakhirkin und N. Piterman. »Finding Recurrent Sets with Backward Analysis and Trace Partitioning«. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Hrsg. von M. Chechik und J. Raskin. Bd. 9636. Lecture Notes in Computer Science. Springer, 2016, S. 17–35. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_2.

[4] I. Balaban, A. Pnueli und L. D. Zuck. »Modular Ranking Abstraction«. In: *Int. J. Found. Comput. Sci.* 18.1 (2007), S. 5–44. DOI: 10.1142/S0129054107004553.

[5] J. Berdine u. a. »Automatic Termination Proofs for Programs with Shape-Shifting Heaps«. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Hrsg. von T. Ball und R. B. Jones. Bd. 4144. Lecture Notes in Computer Science. Springer, 2006, S. 386–400. ISBN: 3-540-37406-X. DOI: 10.1007/11817963_35.

[6] D. Beyer, T. A. Henzinger und G. Théoduloz. »Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis«. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Hrsg. von W. Damm und H. Hermanns. Bd. 4590. Lecture Notes in Computer Science. Springer, 2007, S. 504–518. ISBN: 978-3-540-73367-6. DOI: 10.1007/978-3-540-73368-3_51.

[7] D. Beyer, T. A. Henzinger und G. Théoduloz. »Program Analysis with Dynamic Precision Adjustment«. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 2008, S. 29–38. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.13.

[8] D. Beyer und M. E. Keremoglu. »CPAchecker: A Tool for Configurable Software Verification«. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Hrsg. von G. Gopalakrishnan und S. Qadeer. Bd. 6806. Lecture Notes in Computer Science. Springer, 2011, S. 184–190. ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_16.

[9] D. Beyer, M. E. Keremoglu und P. Wendler. »Predicate abstraction with adjustable-block encoding«. In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. Hrsg. von R. Bloem und N. Sharygina. IEEE, 2010, S. 189–197. ISBN: 978-1-4577-0734-6.

[10] D. Beyer u. a. »Path Invariants«. In: *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation (PLDI 2007, San Diego, CA, June 10-13)*. ACM Press, New York (NY), 2007, S. 300–309. ISBN: 978-1-59593-633-2.

[11] A. Biere und R. Bloem, Hrsg. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Bd. 8559. Lecture Notes in Computer Science. Springer, 2014. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9.

[12] F. Blahoudek u. a. »Complementing Semi-deterministic Büchi Automata«. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Hrsg. von M. Chechik und J. Raskin. Bd. 9636. Lecture Notes in Computer Science. Springer, 2016, S. 770–787. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_49.

[13] A. R. Bradley, Z. Manna und H. B. Sipma. »Termination of Polynomial Programs«. In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. Hrsg. von R. Cousot. Bd. 3385. Lecture Notes in Computer Science. Springer, 2005, S. 113–129. ISBN: 3-540-24297-X. DOI: 10.1007/978-3-540-30579-8_8.

[14] M. Braverman. »Termination of Integer Linear Programs«. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Hrsg. von T. Ball und R. B. Jones. Bd. 4144. Lecture Notes in Computer Science. Springer, 2006, S. 372–385. ISBN: 3-540-37406-X. DOI: 10.1007/11817963_34.

[15] M. Brockschmidt, B. Cook und C. Fuhs. »Better Termination Proving through Cooperation«. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Hrsg. von N. Sharygina und H. Veith. Bd. 8044. Lecture Notes in Computer Science. Springer, 2013, S. 413–429. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_28.

[16] M. Brockschmidt u. a. »Automated Termination Proofs for Java Programs with Cyclic Data«. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Hrsg. von P. Madhusudan und S. A. Seshia. Bd. 7358. Lecture Notes in Computer Science. Springer, 2012, S. 105–122. ISBN: 978-3-642-31423-0. DOI: 10.1007/978-3-642-31424-7_13.

[17] E. M. Clarke u. a. »Counterexample-guided abstraction refinement for symbolic model checking«. In: *J. ACM* 50.5 (2003), S. 752–794. DOI: 10.1145/876638.876643.

[18] B. Cook. »Principles of program termination«. In: *Engineering Methods and Tools for Software Safety and Security* 22 (2009), S. 161.

[19] B. Cook, A. Podelski und A. Rybalchenko. »Abstraction Refinement for Termination«. In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Hrsg. von C. Hankin und I. Siveroni. Bd. 3672. Lecture Notes in Computer Science. Springer, 2005, S. 87–101. ISBN: 3-540-28584-9. DOI: 10.1007/11547662_8.

[20] B. Cook, A. Podelski und A. Rybalchenko. »Termination proofs for systems code«. In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. Hrsg. von M. I. Schwartzbach und T. Ball. ACM, 2006, S. 415–426. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1134029.

[21] B. Cook, A. Podelski und A. Rybalchenko. »Terminator: Beyond Safety«. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Hrsg. von T. Ball und R. B. Jones. Bd. 4144. Lecture Notes in Computer Science. Springer, 2006, S. 415–418. ISBN: 3-540-37406-X. DOI: 10.1007/11817963_37.

[22] B. Cook, A. Podelski und A. Rybalchenko. »Proving program termination«. In: *Commun. ACM* 54.5 (2011), S. 88–98. DOI: 10.1145/1941487.1941509.

[23] W. Craig. »Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem«. In: *J. Symb. Log.* 22.3 (1957), S. 250–268. DOI: 10.2307/2963593.

[24] V. D'Silva und C. Urban. »Conflict-Driven Conditional Termination«. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Hrsg. von D. Kroening und C. S. Pasareanu. Bd. 9207. Lecture Notes in Computer Science. Springer, 2015, S. 271–286. ISBN: 978-3-319-21667-6. DOI: 10.1007/978-3-319-21668-3_16.

[25] J. Farkas. »Theorie der einfachen Ungleichungen.« In: *Journal für die reine und angewandte Mathematik* 124 (1902), S. 1–27.

[26] J. Giesl u. a. »Proving Termination of Programs Automatically with AProVE«. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*. Hrsg. von S. Demri, D. Kapur und C. Weidenbach. Bd. 8562. Lecture Notes in Computer Science. Springer, 2014, S. 184–191. ISBN: 978-3-319-08586-9. DOI: 10.1007/978-3-319-08587-6_13.

[27] M. Heizmann, J. Hoenicke und A. Podelski. »Termination Analysis by Learning Terminating Programs«. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Hrsg. von A. Biere und R. Bloem. Bd. 8559. Lecture Notes in Computer Science. Springer, 2014, S. 797–813. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_53.

[28] M. Heizmann u. a. »Linear Ranking for Linear Lasso Programs«. In: *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*. Hrsg. von D. V. Hung und M. Ogawa. Bd. 8172. Lecture Notes in Computer Science. Springer, 2013, S. 365–380. ISBN: 978-3-319-02443-1. DOI: 10.1007/978-3-319-02444-8_26.

[29] J. Hensel u. a. »Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution«. In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. Hrsg. von R. D. Nicola und eva Kühn. Bd. 9763. Lecture Notes in Computer Science. Springer, 2016, S. 234–252. ISBN: 978-3-319-41590-1. DOI: 10.1007/978-3-319-41591-8_16.

[30] T. A. Henzinger u. a. »Abstractions from proofs«. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Hrsg. von N. D. Jones und X. Leroy. ACM, 2004, S. 232–244. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964021.

[31] T. Kahsai u. a. »Finding Inconsistencies in Programs with Loops«. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Hrsg. von M. Davis u. a. Bd. 9450. Lecture Notes in Computer Science. Springer, 2015, S. 499–514. ISBN: 978-3-662-48898-0. DOI: 10.1007/978-3-662-48899-7_35.

[32] J. Leike und M. Heizmann. »Geometric Series as Nontermination Arguments for Linear Lasso Programs«. In: *CoRR* abs/1405.4413 (2014).

[33] J. Leike und M. Heizmann. »Ranking Templates for Linear Loops«. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Hrsg. von E. Ábrahám und K. Havelund. Bd. 8413. Lecture Notes in Computer Science. Springer, 2014, S. 172–186. ISBN: 978-3-642-54861-1. DOI: 10.1007/978-3-642-54862-8_12.

[34] J. Leike und M. Heizmann. »Ranking Templates for Linear Loops«. In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015.

[35] J. Leike und M. Heizmann. »Geometric Nontermination Arguments«. unveröffentlicht. 2016.

[36] T. S. Motzkin. »Beiträge zur Theorie der linearen Ungleichungen«. Diss. 1936.

[37] M. Murr. »Towards Understandable CPAchecker Counterexamples«. Bachelorarbeit. Universität Passau, 2016.

[38] M. T. Nguyen u. a. »Polytool: Polynomial interpretations as a basis for termination analysis of logic programs«. In: *TPLP* 11.1 (2011), S. 33–63. DOI: 10.1017/S1471068410000025.

[39] J. Ouaknine und J. Worrell. »On linear recurrence sequences and loop termination«. In: *SIGLOG News* 2.2 (2015), S. 4–13. DOI: 10.1145/2766189.2766191.

[40] A. Podelski und A. Rybalchenko. »Transition Invariants«. In: *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 2004, S. 32–41. ISBN: 0-7695-2192-4. DOI: 10.1109/LICS.2004.1319598.

[41] A. Podelski und A. Rybalchenko. »Transition predicate abstraction and fair termination«. In: *ACM Trans. Program. Lang. Syst.* 29.3 (2007). DOI: 10.1145/1232420.1232422.

[42] T. Stieglmaier. »Augmenting Predicate Analysis with Auxiliary Invariants«. Masterarbeit. Universität Passau, 2016.

[43] T. Ströder u. a. »Proving Termination and Memory Safety for Programs with Pointer Arithmetic«. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*. Hrsg. von S. Demri, D. Kapur und C. Weidenbach. Bd. 8562. Lecture Notes in Computer Science. Springer, 2014, S. 208–223. ISBN: 978-3-319-08586-9. DOI: 10.1007/978-3-319-08587-6_15.

[44] A. Tiwari. »Termination of Linear Programs«. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Hrsg. von R. Alur und D. A. Peled. Bd. 3114. Lecture Notes in Computer Science. Springer, 2004, S. 70–82. ISBN: 3-540-22342-8. DOI: 10.1007/978-3-540-27813-9_6.

[45] A. M. Turing. »On computable numbers, with an application to the Entscheidungsproblem«. In: *Proceedings of the London Mathematical Society* 42.2 (1936), S. 230–265.

[46] C. Urban, A. Gurfinkel und T. Kahsai. »Synthesizing Ranking Functions from Bits and Pieces«. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Hrsg. von M. Chechik und J. Raskin. Bd. 9636. Lecture Notes in Computer Science. Springer, 2016, S. 54–70. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_4.

[47] B. Wu u. a. »Termination of Loop Programs with Polynomial Guards«. In: *Computational Science and Its Applications - ICCSA 2010, International Conference, Fukuoka, Japan, March 23-26, 2010, Proceedings, Part IV*. Hrsg. von D. Taniar u. a. Bd. 6019. Lecture Notes in Computer Science. Springer, 2010, S. 482–496. ISBN: 978-3-642-12188-3. DOI: 10.1007/978-3-642-12189-0_42.

Danksagung

An dieser Stelle möchte ich noch einigen Personen danken, die entscheidend zum Gelingen dieser Arbeit beigetragen haben. Professor Dirk Beyer ermöglichte mir die Wahl und Bearbeitung dieses Themas und stellte dafür auch Ressourcen seines Lehrstuhls bereit, die insbesondere zur Durchführung der Experimente verwendet wurden. Außerdem ermöglichte er eine Reise zur Universität Freiburg. Mein Betreuer Matthias Dangl stellte den Kontakt zu Matthias Heinzmann her und stand mir jederzeit mit hilfreichen Ratschlägen zur Seite. Philipp Wendler gab mir sehr nützliche Hinweise zu CPAchecker und der Prädikaten-Analyse. Matthias Heinzmann beantwortete mir zahlreiche Fragen zu LassoRanker und (Nicht-)Terminationsanalysen im Allgemeinen per E-Mail und persönlich während meines Besuchs des Lehrstuhls von Professor Andreas Podelski in Freiburg.

Erklärung zur Masterarbeit

Hiermit erkläre ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe und dass die Masterarbeit in gleicher oder anderer Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 19. Oktober 2016

Sebastian Ott