Abstrakte Übergangsrelationen als Mittel zur Verifikation von Programmeigenschaften

Universität des Saarlandes

Masterarbeit

von
Martin Schäf

Betreuer
Prof. Dr. Andreas Podelski
FB 14 Informatik
Universität des Saarlandes

Saarbrücken, 3. April 2006

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit zum Thema “Abstrakte Übergangsrelationen als Mittel zur Verifikation von Programmeigenschaften” vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Desweiteren habe ich sämtliche Zitate kenntlich gemacht.

Saarbrücken, 3. April 2006

Martin Schäf

Danksagungen

Ich möchte vorab Professor, Dr. Andreas Podelski danken, der es mir ermöglicht hat diese Arbeit in einer sehr angenehmen Atmosphäre zu erstellen. Ich möchte mich bei Andrey Rybalchenko für die sehr gute Betreuung meiner Arbeit bedanken. Thomas Wies und Alexander Malkis möchte ich danken für ihre wertvollen Hinweise bei der Erstellung dieser Arbeit. Ich bedanke mich bei Prof. Bernd Finkbeiner für die Zweitkorrektur dieser Arbeit. Mein größter Dank gebührt meinen Eltern Gerd und Doris Schäf, die mich während meiner gesamten Studienzeit moralisch und logistisch unterstützt haben.

Zusammenfassung

Die Entwicklung von Software ist ein fehleranfälliger Prozess. Allein die Tatsache, dass ein Programm ausführbar ist und dass für eine begrenzte Menge von Testfällen keine Fehler in der Ausführung des Programms auftritt, sagt nicht aus, dass ein Programm fehlerfrei ist. Vor allem in Bereichen, in denen Software nicht einfach im Fall eines Versagens abgeschaltet und repariert werden kann ist es wichtig beweisen zu können, dass Software fehlerfrei ist. Es existieren verschiedene Methoden diesen Beweis zu erbringen. Die gängigste Methode ist deduktive Verifikation, in der die Software von Hand verifiziert wird. Dieser Vorgang ist allerdings teuer und fehleranfällig.
Formale Verifikation ist eine jüngere Methode die vor allem für die Verifikation von Hardware verwendet wird. Mit formaler Verifikation können Systeme mit endlicher Zustandsmenge automatisch bewiesen werden. Um die Methode auf Software anzuwenden, muss diese abstrahiert werden, da die Zustandsmenge eines Programms unendlich gross sein kann. Das Überprüfen von Programmeigenschaften auf einem abstrakten Modell wird auch Model Checking genannt.
Es existieren verschiedene Ansätze um durch Model Checking Programmeigenschaften zu beweisen. In [esp] wird ein Ansatz vorgestellt temporäre Sicherheitseigenschaften eines Programms automatisch zu Beweisen. [tin] stellt eine Möglichkeit vor durch Model Checking Terminierung und Fairness zu beweisen.
Diese Ansätze arbeiten immer auf dem gesamten Programm. In dieser Arbeit wird eine Methode hergeleitet, ein Programm oder einen Teil eines Programms in eine abstrakte Übergangsrelation zu übersetzen und durch diese, Annahmen über das Verhalten der Programmvariablen zu beweisen. Der Vorteil dieser Methode ist die Wiederverwendbarkeit von Übergangsrelationen. Da diese unabhängig vom Kontrollfluss sind, genügt es, für einen Programmteil einmalig die Übergangsrelation zu berechnen, egal wie oft dieser Teil verwendet wird. Diese Eigenschaft bietet es an, dieses Verfahren in den Prozess der Softwareentwicklung einzubeziehen um die Fehlersuche zu erleichtern, oder Fehler frühzeitig zu erkennen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Laut einer Statistik des National Institute of Standards and Technology [nis] werden in der Softwareentwicklung 80% des Budgets eines Softwareprojekts für die Suche nach Programmfehlern und deren Behebung aufgewendet. Diese Kosten entstehen vor allem durch Fehler (engl. defect), die nur in bestimmten Programmläufen zu einem sichtbaren Fehlschlagen des Programms führen, so genannte Laufzeitfehler (engl.: run time error). Diese Fehler können mit den gängigen Hilfsmitteln nicht automatisch erkannt werden, da der Quelltext, der diesen Fehler verursacht, durchaus syntaktisch korrekt sein kann. Das Versagen des Programms entsteht meist aus einer unerlaubten arithmetischen Operation, wie eine Division durch Null, oder bei dem Versuch auf geschützten Speicher zuzugreifen. Die Ursache dieser Fehler, die so genannte Infektion, muss nicht die gleiche Stelle im Quelltext sein, die auch zum Versagen des Programms führt. Eine falsche Wertzuweisung innerhalb des Quelltexts, beispielsweise, kann zum Zeitpunkt der Zuweisung keine Auswirkung haben, aber im weitern Verlauf des Programms zu einer Speicherzugriffsverletzung führen. Der Zeitpunkt des Fehlschlagens gibt, für solche Fehler, keinen Hinweis auf den Zeitpunkt der Infektion, also auf die eigentlich Fehlerursache. Es gibt verschiedene Ansätze solche Fehler zu finden, oder frühzeitig zu vermeiden. Die in der Praxis am häufigsten verwendete Methode ist eine Kombination aus systematischem Testen und Debugging. Für einzelne Teile des Programms werden Testdaten erzeugt, bestehend aus Eingabedaten und den erwarteten Ausgabedaten. Treten Fehler bei der Ausführung der Testfälle auf, kann mit einem Debugger (z.B. [gdb]) das Programm während der Ausführen an bestimmten Stellen, den so genannten Breakpoints, unterbrochen werden und der Programmierer kann, anhand des aktuellen Programmzustands, versuchen Rückschlüsse auf die Fehlerursache zu finden. Der grundlegende Nachteil des Debuggings ist, dass nur für bestimmte Testfälle die Korrektheit des Programms garantiert ist.
Mit der zunehmend Verbreitung von Softwaresystem in Anwendungsgebieten, die die Abwesenheit von Fehlern verlangen, wie Flugzeugsteuerung oder Medizintechnik, steigt der Bedarf an Methoden zur Verifikation von Programmeigenschaften. Für kritische Programme wird oft deduktive Verifikation verwendet. Das Programm wird stückweise von Hand verifiziert. Dieser Ansatz ist allerdings sehr zeitaufwendig und da die Verifikation von Menschen ausgeführt wird, kann die Korrektheit nur bedingt garantiert werden.
Die Möglichkeiten eine automatisierte Verifikation von Software zu entwickeln sind gemäß der Berechenbarkeitstheorie durch das Halteproblem beschränkt. Es ist bewiesen, dass man in einer Programmiersprache kein Programm schreiben kann, dass für alle in dieser Programmiersprache geschriebenen Programme entscheiden kann, ob sie terminieren.
Eine Methode die Verifikation von Software zu automatisieren ist Model Checking [MoCh], ein Verfahren, um nebenläufige Systeme mit endlicher Zustandsmenge automatisch zu verifizieren. Da die Zustandsmenge von Programmen unendlich groß sein kann, besteht die Hauptaufgabe des Model Checkings darin ein abstrak-

tes Modell eines Programms zu erzeugen, dessen Zustandsmenge endlich ist, das aber genügend Informationen bietet um die gewünschte Eigenschaft beweisen zu können. Es existieren bereits einige Anwendungen, die diesen Ansatz verwenden, wie BLAST [BLA] von Microsoft.
Die größte Schwierigkeit bei der Konstruktion eines Modells für ein bestimmtes Programm liegt darin, dass die Zustandsmenge für umfangreiche Programme extrem groß wird. Dieses Problem tritt vor allem auf, wenn ein Programm aus vielen, miteinander interagierenden, Komponenten besteht, wie es bei nebenläufigen Programmen der Fall sein kann, oder wenn ein Programm Datenstrukturen verwaltet, die viele verschiedene Werte annehmen können, z.B. durch Veränderungen eines Wertes innerhalb einer Schleife. Aus diesem Grund wird Model Checking momentan vornehmlich für die Verifikation von kleinen Anwendungen, wie Treibern und Protokollen verwendet.
Im Rahmen dieser Arbeit wird ein Programm zur automatischen Überprüfung von Programmeigenschaften entwickelt. Das Programm überprüft die Erfüllbarkeit von Hilfsaussagen für ein Programm oder einen Programmteil.

2 Motivation und Vorgehensweise

Um die Sicherheit von Software zu erhöhen und die Kosten bei der Suche nach Programmfehlern zu reduzieren, wird im Folgenden eine Methode entwickelt, die automatisch bestimmte Eigenschaften der Programmvariablen in einem Programm oder Programmteil überprüft. Die Automatisierung soll so gestaltet sein, dass die zu überprüfende Eigenschaft als eine Formel der Prädikatenlogik erster Ordnung formuliert wird, deren Erfüllbarkeit sich automatisch testen lässt. So erhält man für die Gültigkeit der Eigenschaft eine klare Ja- oder Nein-Antwort. Solche Eigenschaften können entweder automatisch generiert werden, z.B. die Eigenschaft, dass bei einer Division von zwei Variablen der Nenner nie Null ist, oder vom Benutzer manuell festgelegt werden, um bestimmte Eigenschaften eines Programms zu garantieren.
Dieses Ziel wird realisiert, indem zuerst zu jeder Art von Programmaussage eine abstrakte Übergangsrelation [1] definiert wird. Eine Übergangsrelation ist eine Relation auf der Menge der möglichen Belegungen der Programmvariablen vor und nach der Ausführung einer Programmaussage, die beschreibt, wie sich die Belegungen der Programmvariablen durch eine Programmaussage ändern können. Weiterhin wird eine Verknüpfungsoperation für Übergangsrelationen definiert, die es erlaubt eine Übergangsrelation für einen Programmblock aus der Verknüpfung der Übergangsrelationen der einzelnen Programmaussagen dieses Programmblocks zu erzeugen. Dieser Ansatz wird zunächst formal beschrieben. Anschließend wird eine Implementierung dieser Übergangsrelationen durch Formeln der linearen Temporallogik (**LTL**) angegeben, deren Erfüllbarkeit durch einen Theorembeweiser getestet werden kann.
Da dieser Ansatz unabhängig von der analysierten Programmiersprache ist, werden in dieser Arbeit nur die gängigsten Arten von Programmaussagen betrachtet. Die Methode ist für die Programmiersprache **C** implementiert, beschränkt sich allerdings auf nicht rekursive Funktionen, Zuweisungen, `if`- und `while`-Aussagen. Weiterhin werden keine Zeiger oder Felder betrachtet. Eine Erweiterung dieser Methode auf den vollen Umfang von **C** ist möglich, übersteigt jedoch den Umfang dieser Arbeit.

[1] Im Folgenden wird aus Gründen der besseren Lesbarkeit von Übergangsrelationen, statt von abstrakten Übergangsrelationen gesprochen.

3 Verwandte Arbeiten

Der hier verwendete Ansatz ist eng verwandt mit den Themen Prädikat Abstraktion (engl. predicate abstraction) und Übergangs-Prädikat Abstraktion (engl. transition predicate abstraction). Als ein Prädikat wird in diesem Zusammenhang eine bool'sche Aussage über Programmvariablen bezeichnet. Prädikat Abstraktion ist eine Methode, automatisch ein abstraktes Modell eines Programms zu konstruieren. Erstmals wird diese Technik in [gra] beschrieben. In [bal] wird sie als Abstraktion für ein **C**-Programm verwendet. Prädikat Abstraktion bildet die Zustände eines Programms auf abstrakte Zustände ab, die durch Auswertung einer Menge von Prädikaten in diesem Zustand bestimmt werden. Diese Methode ist bereits in einigen bestehenden Model Checking Anwendungen, z.B. BLAST ([BLA]), implementiert.
Übergangs-Prädikat Abstraktion [pod] geht einen Schritt weiter und betrachtet nicht die Menge der abstrakten Zustände eines Programms, sondern die Menge der abstrakten Übergangsrelationen zwischen den Zuständen eines Programms. Statt Prädikaten werden Übergangs-Prädikate [2] betrachtet. Ein Übergangs-Prädikat ist eine binäre Relation zwischen Zuständen. Eine abstrakte Übergangsrelation wird hier durch eine Konjunktion von Übergangs-Prädikaten dargestellt. In [pod] wird beschrieben, wie mittels Übergangs-Prädikat Abstraktion Terminierung und Fairness eines Programms bewiesen werden können.
In dieser Arbeit wird der Ansatz der Übergangs-Prädikat Abstraktion erweitert um, unabhängig vom Kontrollfluss eines Programms, zu jeder Programmaussage eine abstrakte Übergangsrelation zu berechnen. Vorteil dieses Ansatzes ist, dass die Berechnung der Übergangsrelationen unabhängig vom Kontrollfluss ist, was zu einer höheren Wiederverwendbarkeit der einzelnen Relationen führt. In dieser Arbeit werden die Relationen anhand des Syntaxbaums und nicht wie in anderen Ansätzen, anhand des Kontrollflussgraphen erzeugt. Offensichtlich liegt der Vorteil dieser Methode darin, dass bei Änderungen in einem Programm die Übergangsrelationen nur für den entsprechenden Knoten im Syntaxbaum und dessen Vorgängern neu berechnet werden müssen.
Um dieses Ziel zu erreichen wird zuerst die theoretische Berechnung der Relationen betrachtet und anschließend eine Implementierung dieser Relationen durch Übergangs-Prädikate beschrieben.

[2] Im Folgenden ist mit dem Wort Prädikat immer ein Übergangs-Prädikat gemeint.

4 Grundlagen

In [pod] wird eine Methode beschrieben um ein Programm automatisch in ein endliches, abstraktes Übergangsprogramm, durch Berechnungen auf dem Kontrollflussgraphen, zu übersetzen. In dieser Arbeit wird ein Ansatz gezeigt, wie dieses Ziel durch Berechnungen auf dem Syntaxbaum erreicht werden kann.
In diesem Kapitel wird die Berechnung der abstrakten Übergangsrelationen für die in Abbildung 4 gezeigte Programmiersprache gezeigt.

```
Ausdrücke:
E ::=  n | x
     | E+E | E-E | E/E | E*E
     | E=E | E>E | E<E | NOT(E)
     | E AND E | E OR E

Programmaussagen:
S ::=  x = E
     | E
     | S;S
     | if (E) S else S
     | while(E) S
     | return E | break | continue
     | f(x) | x = f(y)

globale Programmaussagen:
P ::=  x = E
     | f(x) = S
     | P;P
```

Abbildung 1: Einfache Syntax einer Programmiersprache

Im Folgenden wird mit V und V' die Menge der ungestrichenen und gestrichenen Programmvariablen bezeichnet. Die Variablen in V' entsprechen den Variablen in V nach Ausführung einer Programmaussage. Weiterhin sei D_i die Menge aller gültigen Werte die ein Variable $v_i \in V$ annehmen kann. Es gelte $D := D_1 \times \ldots \times D_n$. Eine Funktion $s : V \rightarrow D$, die jeder Programmvariablen einen Wert zuweist wird Zustand (engl.: state) genannt. Für eine Variable $v_i \in V$ beschreibt $s(v_i)$ den Wert, der der Variablen v_i durch die Funktion s zugewiesen wird [3]. Als Übergangsprädikat wird eine binäre Relation über Zuständen bezeichnet. Diese wird üblicherweise durch eine Zusicherung über gestrichenen und ungestrichenen Programmvariablen dargestellt. Eine Übergangsrelation ρ_τ einer Programmaussage τ ist die Menge aller Paare von Zuständen (s, s') für die gilt, dass s' durch τ von s aus erreicht werden kann. Diese Menge lässt sich auch als Formel über

[3] Aus Gründen der Vereinfachung gilt für einen Zustand $s' : V' \rightarrow D$, dass $s'(v'_i) = s'(v_i)$.

den gestrichenen und ungestrichenen Programmvariablen formulieren. Eine solche Formel ist genau für alle Zustandspaare $(s, s') \in \rho_\tau$ erfüllt. Da beide Schreibweisen Vorteile haben wird im Umfang dieser Arbeit immer die Mengendarstellung und die Darstellung durch eine Formel betrachtet.
Es gilt nun im Folgenden Regeln zur Berechnung der Übergangsrelationen für die verschiedenen Typen von Programmaussagen aus Abbildung 4 aufzustellen.
Es wird weiterhin ein abstrakter Syntaxbaum konstruiert. In diesem abstrakten Syntaxbaum wird zu jedem Knoten des Syntaxbaums eines Programms ein Knoten erzeugt, der mit der Übergangsrelation der durch den ursprünglichen Knoten beschriebenen Programmaussage beschriftet ist.

4.1 Ausdrücke

Für die Behandlung von Ausdrücken wird die Eigenschaft vorausgesetzt, dass Ausdrücke immer als bool'sche Aussage ausgewertet werden. Für Ausdrücke, die eine vergleichende Funktion haben ist dies intuitiv klar. Arithmetische Ausdrücke werden immer als wahre Aussage angesehen. Die Menge $\phi(E)$ bezeichnet die Menge aller Zustände, für die der Ausdruck E als wahr ausgewertet wird.
Gilt für einen Zustand s, dass $s \in \phi(E)$, so ist der Ausdruck E für die, den Programmvariablen durch s zugewiesenen Werte erfüllt. Weiterhin sei $(\neg\phi(E))$ die Menge aller Zustände s für die der Ausdruck E nicht erfüllt ist. Die Vereinigung der disjunkten Mengen $\phi(E)$ und $(\neg\phi(E))$ ergibt die Menge aller Zustände.
Ein Ausdruck $E \equiv x = 3$ lässt sich also durch die Menge von Zuständen

$$\phi(x = 3) \equiv \{s | s(x) = 3\}$$

beschreiben.

4.2 Übergangsrelationen von Zuweisungen

Zuweisungen stellen den einfachsten Typ von Programmaussagen dar, da sie nicht aus anderen Programmaussagen zusammengesetzt sind. Folglich haben sie innerhalb des Syntaxbaums keine Nachfolger. Eine Zuweisung $\tau \equiv x = E$, in einem Programm, dass nur eine Programmvariable x verwendet, wird durch die Übergangsrelation $\rho_\tau \equiv x' = E$ beschrieben. Alternativ lässt sich die Übergangsrelation als die Menge $\{(s, s') | s'(x) = E\}$ darstellen.
Innerhalb der Übergangsrelation müssen auch alle Programmvariablen beachtet werden, die nicht durch die zugehörige Programmaussage verändert werden. Seien beispielsweise $V = \{x, y\}$ und $V' = \{x', y'\}$ die Mengen der gestrichenen und ungestrichenen Programmvariablen, dann ergibt sich für eine Programmaussage $\tau \equiv x = E$ die Übergangsrelation $\rho_\tau \equiv x' = E \wedge y' = y$, bzw $\rho_\tau \equiv \{(s, s') | s'(x) = E \wedge s(y) = s'(y)\}$. Diese Übergangsrelation beschreibt die Tatsache, dass die Variable x nach Ausführung der Programmaussage τ den Wert E annimmt und allen anderen Programmvariablen unverändert bleiben.

4.3 Übergangsrelationen von Programmblöcken

Ein Programmblock ist die sequenzielle Komposition einzelner Programmaussagen. Die Übergangsrelation für einen solchen Programmblock errechnet sich aus der relationalen Komposition der Übergangsrelationen der einzelnen Programmaussagen dieses Blocks. Für eine Programmaussage $\tau \equiv \tau_1; \tau_2$ ergibt sich die Übergangsrelation ρ_τ aus der relationalen Komposition $\rho_{\tau_1} \circ \rho_{\tau_2}$. Diese wird wie folgt definiert:

$$\rho_{\tau_1} \circ \rho_{\tau_2} := \left\{ (s, s') \,\middle|\, \exists s'' : (s, s'') \in \rho_{\tau_1} \wedge (s'', s') \in \rho_{\tau_2} \right\}$$

Für zwei Übergangsrelationen $\rho_{\tau_1} \equiv x' = x + 1$ und $\rho_{\tau_2} \equiv x' = x + 2$ ist die Komposition folglich $\rho_{\tau_1} \circ \rho_{\tau_2} \equiv x' = (x + 1) + 2$.
Im weiteren Verlauf dieses Kapitels wird diese Definition erweitert um den Effekt von Sprunganweisungen korrekt darstellen zu können.

4.4 Komposition von Übergangsrelationen und Ausdrücken

In manchen Fällen ist es nötig zu einer Übergangsrelation bestimmte Bedingungen hinzuzufügen. Zu diesem Zweck wird eine Verknüpfung $\rho_\tau \circ \phi(E)$ definiert, die eine Menge beschreibt, die alle Zustandspaare (s, s') enthält, für die gilt, dass die Bedingung E nach Ausführung der Programmaussage τ erfüllt ist. Diese Verknüpfung wird wie folgt definiert:

$$\rho_\tau \circ \phi(E) := \left\{ (s, s') \,\middle|\, (s, s') \in \rho_\tau \wedge s' \in \phi(E) \right\}$$

Analog gilt:

$$\phi(E) \circ \rho_\tau := \left\{ (s, s') \,\middle|\, (s, s') \in \rho_\tau \wedge s \in \phi(E) \right\}$$

Um eine Verknüpfung zwischen Mengen von Zuständen und Mengen von Tupeln von Zuständen verwenden zu können muss die Assoziativität erhalten bleiben. Zu diesem Zweck wird die Verknüpfung zweier Mengen von Zuständen wie folgt definiert:

$$\phi(E) \circ \phi(E') := \left\{ s \,\middle|\, s \in \phi(E) \wedge s \in \phi(E') \right\}$$

4.5 Übergangsrelationen von bedingten Programmaussagen

Zu einer bedingten Programmaussage $\tau \equiv \texttt{if E } \tau_1 \texttt{ else } \tau_2$ lässt sich die Übergangsrelation $\rho_\tau \equiv \phi(E) \wedge \rho_{\tau_1} \vee \neg\phi(E) \wedge \rho_{\tau_2}$ einfach herleiten. Die Relation ρ_τ lässt sich auch als Menge

$$\rho_\tau \equiv \left\{ (s, s') \,\middle|\, \begin{array}{ll} s \in \phi(E) \wedge & (s, s') \in \rho_{\tau_1} \\ \vee & \\ s \notin \phi(E) \wedge & (s, s') \in \rho_{\tau_2} \end{array} \right\}$$

schreiben. Für die Konstruktion des abstrakten Syntaxbaums gilt, tritt eine bedingte Programmaussage $\tau_i \equiv \texttt{if E}\ \tau_\texttt{j}\ \texttt{else}\ \tau_\texttt{k}$ innerhalb eines Programmblocks $\tau_0 \dots \tau_i \dots \tau_n$ auf, so werden alle Programmaussagen $\tau_i + 1 \dots \tau_n$, die τ_i innerhalb dieses Programmblocks folgen entfernt und sowohl mit τ_j, als auch τ_k sequenziell komponiert.

4.6 Übergangsrelationen von Schleifen

Für eine Schleife $\tau \equiv \texttt{while}(E)\ \tau_B$ ist die Menge

$$\rho_\tau \equiv \bigcup_{i=0}^{\infty} (\phi(E) \circ \rho_{\tau_B})^i \circ \neg\phi(E)$$

der Paare von Zuständen (s, s') für die gilt, s' ist durch τ von s aus erreichbar gemäß dem Halteproblem für den allgemeinen Fall nicht berechenbar. Aus diesem Grund soll anstatt der nicht berechenbaren Relation ρ_τ eine abstrakte Übergangsrelation $\rho_\tau^\#$ berechnet für die gilt $\rho_\tau \subseteq \rho_\tau^\#$. Die Menge $\rho_\tau^\#$ enthält also alle Paare von Zuständen, die auch die gesuchte Relation ρ_τ enthält. Es existieren allerdings auch Paare von Zuständen, die zwar in $\rho_\tau^\#$ jedoch nicht in ρ_τ enthalten sind.
Um eine möglichst kleine Übermenge von ρ_τ zu erhalten wird in dieser Arbeit der Ansatz der Übergangsprädikatenabstraktion aus [pod] verfolgt. Übergangsprädikatenabstraktion abstrahiert eine Relation mittels einer Menge von Übergangsprädikaten. Für eine Programmaussage τ über den Programmvariablen V mit Übergangsrelation ρ_τ und eine Menge von Übergangsprädikaten

$$\mathcal{P} := \bigcup_{v_i \in V} \{(v_i = v_i'), (v_i \geq, v_i'), (v_i \leq v_i')\}$$

wird die Übergangsprädikatenabstraktion $\alpha(\rho_\tau)$ von ρ_τ durch die Formel

$$\alpha(\rho_\tau) \equiv \bigwedge \{p \in \mathcal{P} \,|\, \rho_\tau \subseteq p\}$$

beschrieben. Abbildung 2 zeigt die Mengenschreibweise von $\alpha(\rho_\tau)$ für die Übergangsprädikatmenge $\mathcal{P}$.

$$\alpha(\rho_\tau) = \left\{ (s, s') \;\middle|\; \forall v_i \in V : \begin{cases} s(v_i) = s'(v_i) & \text{falls} \forall (s_1, s_2) \in \rho_\tau : s_1(v_i) = s_2(v_i) \\ s(v_i) \geq s'(v_i) & \text{falls} \forall (s_1, s_2) \in \rho_\tau : s_1(v_i) \geq s_2(v_i) \\ s(v_i) \leq s'(v_i) & \text{falls} \forall (s_1, s_2) \in \rho_\tau : s_1(v_i) \leq s_2(v_i) \end{cases} \right\}$$

Abbildung 2: Abstraktion einer Relation ρ_τ

Die, abstrakte Relation $\alpha(\rho_\tau)$ reduziert die Relation ρ_τ auf die Information, ob der Wert einer Variablen v_i durch die Programmaussage τ unverändert bleibt, monoton

wächst oder sinkt, oder ob die Veränderung nicht vorhersehbar ist. Für jede Programmvariable v_i wird getestet, ob für alle Zustandspaare $(s, s') \in \rho_\tau$ gilt, dass eine der Bedingungen $s(v_i) = s'(v_i)$, $s(v_i) \leq s'(v_i)$, $s(v_i) \geq s'(v_i)$ erfüllt ist. Die Reihenfolge, in der diese Bedingungen getestet werden ist nicht beliebig, da $s(v_i) = s'(v_i)$ auch die beiden anderen Bedingung erfüllt. Es wird zuerst getestet, ob $s(v_i) = s'(v_i)$ erfüllt ist. Falls nicht, werden die beiden anderen Bedingungen überprüft. Wird der Wert von v_i durch die Relation ρ_τ nicht verändert, so gilt auch für alle Zustandspaare in $\alpha(\rho_\tau)$, dass $s'(v_i) = s(v_i)$. Ist eine der Bedingungen $s(v_i) \leq s'(v_i)$ oder $s(v_i) \geq s'(v_i)$ erfüllt, so enthält $\alpha(\rho_\tau)$ alle Paar von Zuständen (s, s'), für die diese Bedingung erfüllt ist. Es gilt also $\rho_\tau \subseteq \alpha(\rho_\tau)$.

Die abstrakte Relation $\rho_\tau^\#$ einer Schleife $\tau \equiv \texttt{while}(E)\ \tau_B$ wird im Folgenden aus mehreren, zueinander disjunkten, Fällen zusammengesetzt. Es werden die Fälle unterschieden, dass die Schleife entweder gar nicht betreten wird oder, genau in einer der ersten n Iterationen verlassen wird, sowie der Fall, dass die Schleife in den ersten n Iterationen nicht verlassen wird, es sich jedoch nicht um eine Endlosschleife handelt.
Mit $(\rho_\tau)^n := \rho_\tau \circ \ldots \circ \rho_\tau$ wird im Folgenden die n-fache Komposition einer Relation ρ_τ mit sich selbst bezeichnet.
Zu einer Schleife $\tau \equiv \texttt{while}(E)\ \tau_B$ sei

$$\beta_n(\rho_{\tau_B}, \phi(E)) \equiv (\phi(E) \circ \rho_{\tau_B})^n \circ \neg\phi(E)$$

die Relation, die die Menge aller Zustandspaare (s, s') enthält, für die gilt, dass der Zustand s' von s aus nach genau n Iterationen der Schleife erreicht wird.
Die Relation $\sigma_n(\rho_{\tau_B}, \phi(E))$ zu einer Schleife $\tau \equiv \texttt{while}(E)\ \tau_B$ enthält alle Zustandspaare (s, s') für die gilt, dass die Schleife betreten wird, innerhalb der ersten n Iterationen nicht verlassen wird und die Abbruchbedingung für die Abstraktion des Schleifenkörpers erfüllbar ist.

$$\sigma_n(\rho_{\tau_B}, \phi(E)) = \left\{ (s, s') \;\middle|\; \begin{array}{l} \bigwedge_{i=1}^{n} \left((s, s') \notin \beta_n(\rho_{\tau_B}, \phi(E)) \right) \\ \wedge\ (s, s') \in (\phi(E) \circ \alpha(\rho_{\tau_B})) \circ (\neg\phi(E) \end{array} \right\}$$

Mit den beiden gezeigten Relationen $\sigma_n(\rho_{\tau_B}, \phi(E))$ und $\beta_n(\rho_{\tau_B}, \phi(E))$ lässt sich jetzt die gesuchte Relation $\rho_\tau^\#$, wie in Abbildung 3 gezeigt, konstruieren. Die Anzahl der Schleifeniterationen für die eine konkrete Relation angegeben wird bestimmt der Parameter n. Die abstrakte Relation einer Schleife $\tau \equiv \texttt{while}(E)\ \tau_B$ wird in Abhängigkeit von n geschrieben als $\rho_\tau^\#(n)$. Offensichtlich gilt $\rho_\tau \subseteq \rho_\tau^\#(n)$.

4.7 Übergangsrelationen von Funktionsaufrufen und globalen Programmaussagen

Globale Programmaussagen müssen gesondert betrachtet werden, da sich die Übergangsrelation eines Programms nicht aus der relationalen Komposition der Über-

$$\rho_\tau^\#(n) = \left\{ (s,s') \;\middle|\; \begin{array}{l} (\bigvee_{i=1}^{n}(s,s') \in \beta_i(\rho_{\tau_B}, \phi(E)) \\ \vee\ s \notin \phi(E) \wedge s = s' \\ \vee\ (s,s') \in \sigma_n(\rho_{\tau_B}, \phi(E)) \end{array} \right\}$$

Abbildung 3: Abstrakte Relation einer Schleife mit Genauigkeit n

gangsrelationen der einzelnen globalen Programmaussagen zusammensetzt, sondern aus der Komposition der Initialwerte der Programmvariablen mit der Relation der Hauptfunktion des Programms.
Es werden zwei Typen von globalen Programmaussagen unterschieden. Globale Zuweisungen und Funktionsdefinitionen. Diese dürfen nicht direkt mit einander verknüpft werden, da Funktionen in verschiedenen Zusammenhängen aufgerufen werden können. Übergangsrelationen von Funktionen dürfen also nicht mit anderen Relationen verknüpft werden.
Um die Semantik und die Übergangsrelation eines Funktionsaufrufs zu beschreiben wird nun die Menge der Programmvariablen V um die Menge aller Funktionssymbole und die Menge der formalen Variablen einer Funktion erweitert.
Die Übergangsrelation einer Funktionsdefinition $\tau \equiv f(v_i, \ldots, v_j) = \tau_B$ mit den formalen Variablen $v_i, \ldots, v_j$ und dem Funktionskörper τ_B wird durch die Menge $\{(s,s')|(s,s') \in \tau_B\}$ beschrieben. Der Wert $s'(f)$ beschreibt den Rückgabewert der Funktion.
Um diesen Rückgabewert zu erhalten wird im Folgenden die Übergangsrelation für Programmaussagen vom Type `return E` definiert und die Definition der relationalen Komposition für die Programmaussage `return` erweitert.
In einer Funktion $f(v_i, \ldots, v_j)$ wird eine Programmaussage `return E` als ein Zuweisung $f = E$ interpretiert, die der Variable f den Ausdruck E zuweist. Die Übergangsrelation ρ_τ einer Programmaussage $\tau \equiv$ `returnE` in einer Funktion f wird folglich geschrieben als

$$\rho_\tau \equiv \left\{ (s,s') \,\middle|\, s'(f) = E \right\}$$

Weiterhin muss für die Berechnung von Übergangsrelationen gelten, dass innerhalb eines Funktionskörpers nach einer `return`-Aussage keine weiteren Programmaussagen ausgeführt werden können. Um dies zu garantieren wird die Definition der relationalen Komposition erweitert. Seien ρ_{τ_1} und ρ_{τ_2} Übergangsrelationen in einem Funktionskörper einer Funktion f. Für die Komposition $\rho_{\tau_1} \circ \rho_{\tau_2}$ gilt:

$$(\rho_{\tau_1} \circ \rho_{\tau_2}) = \begin{cases} \rho_{\tau_1} \circ \rho_{\tau_2} & \text{wenn } \forall (s,s') \in \rho_{\tau_1} :\ s(f) = s'(f) \\ \rho_{\tau_1} & \text{sonst} \end{cases}$$

Die Übergangsrelation für den Aufruf einer Funktion $\tau \equiv f(v_i, \ldots, v_j)\tau_B$ mit

Parametern $w_i, \ldots, w_j \in D$ wird beschrieben durch

$$\rho_\tau \equiv \left\{ (s, s') \,\middle|\, \bigwedge_{k=i}^{j} (s(v_k) = w_k) \wedge (s, s') \in \rho_{\tau_B} \right\}$$

Für einen Funktionsaufruf $\tau \equiv x = f(v_i, \ldots, v_j)$, der einer Programmvariablen den Rückgabewert der Funktion zuweist íst die Übergangsrelation folglich:

$$\rho_\tau \equiv \left\{ (s, s') \,\middle|\, \bigwedge_{k=i}^{j} (s(v_k) = w_k) \wedge (s, s') \in \rho_{\tau_B} \wedge s'(x) = s'(f) \right\}$$

5 Implementierung

Für die Implementierung der Arbeit wurde die funktionale Programmiersprache **OCAML** [OCAML] verwendet. Die Analyse wurde für die Programmiersprache **C** [C99] entwickelt. Als Parser wird **CIL**[CIL] verwendet.

5.1 Datentypen

Das entwickelte Programm berechnet für jede Programmaussage in einem gegebenen **C** Programm die zugehörige Übergangsrelation. Eine Übergangsrelation ρ_τ einer Programmaussage τ wird als Hashtabelle implementiert. Als Suchschlüssel wird die Menge der Variablennamen verwendet. In der Zieldatenstruktur wird für jede Variable eine Liste aus Tupeln erstellt. Ein solches Tupel besteht aus einer Bedingung (engl. guard) und dem Wert der gestrichenen Variablen. Dies hat den Vorteil, dass schnell auf die Relationen einzelner Programmvariablen zugegriffen werden kann. Für das Code Fragment `if (x>0) x++; else x--` sieht der entsprechende Hashtabelleneintrag in einer Hashtabelle `H` für die Variable x wie folgt aus:

$$\begin{aligned}\texttt{H}(x) :=&(((x > 0), \quad x + 1);\\ &((x \leq 0), \quad x - 1))\end{aligned}$$

Dieser Hashtabelleneintrag lässt sich als die Formel

$$Eval_{\texttt{H}}(x) := (x > 0) \wedge (x' = x + 1) \vee (x \leq 0) \wedge (x' = x - 1)$$

darstellen. Die Formel $Eval_{\texttt{H}}(v_i)$ lässt sich für eine beliebige Variable v_i konstruieren, indem man für den Hashtabelleneintrag $\texttt{H}(v_i)$ die Disjunktion aller Elemente der Tupelliste bildet und für jedes Tupel $(cond, val)$ die Formel $(cond) \wedge (v_i' = val)$ einsetzt. Der Hashtabelleneintrag einer Programmvariablen v_i wird mit $(\texttt{true}, \texttt{v}_\texttt{i})$ initialisiert.
Ein Zustandspaar (s, s') ist also genau dann Teil einer Übergangsrelation mit zugehöriger Hashtabelle `H`, wenn $Eval_{\texttt{H}}(v_i)$ für alle (v_i, v_i') für die, durch s und s' zugewiesenen Werte, wahr ist.

Die Übergangsrelationen von Funktionen werden in einer eigenen Hashtabelle gespeichert. In der Zieldatenstruktur dieser Hashtabelle wird neben der oben beschriebenen Tupelliste auch die Menge der Namen der formalen Variablen und eine Tupelliste für den Rückgabewert der Funktion gespeichert, um ein einfaches Auswerten eines Funktionsaufrufs zu ermöglichen. Für die Funktion `int f(int x)` in Abbildung 5.1 besteht die Menge der Formalen Variablen nur aus `int x`. Die Tupelliste für den Rückgabewert der Funktion ist $((\texttt{x} > \texttt{0}), \texttt{x} + \texttt{1}), ((\texttt{x} \leq \texttt{0}), \texttt{x} - \texttt{1})$. Da in dieser Funktion nur formale Variablen verändert werden ist die Tupelliste, die die Veränderungen der Variablenbelegungen durch einen Aufruf dieser Funktion beschreibt, nicht relevant.
Ein Funktionsaufruf `int i = f(3)` führt dann zu dem Hashtabelleneintrag $((\texttt{3} > \texttt{0}), \texttt{3} + \texttt{1}), ((\texttt{3} \leq \texttt{0}), \texttt{3} - \texttt{1})$ oder vereinfacht $(\texttt{true}, \texttt{4})$ für die Variable i.

```
int f ( int x )
{
     if (x>0) x++;
     else x--;
     return x;
}
```

Abbildung 4: Einfache **C**-Funktion

Elemente der Tupelliste, für die die Bedingung nicht erfüllbar ist, können ignoriert werden, da dieser Fall nicht eintreten kann.

5.2 Relationale Komposition

Für zwei Hashtabellen $\mathtt{H}_1$ und $\mathtt{H}_2$ wird die relationale Komposition $\mathtt{H}_1 \circ \mathtt{H}_2$ berechnet, indem die Hashtabelle $\mathtt{H}_1$ als Initialbelegung für die Hashtabelle $\mathtt{H}_2$ verwendet wird. Dies kann entweder während der Berechnung von $\mathtt{H}_2$ geschehen oder durch nachträgliches Einsetzen.

```
if (x>0) x++;
else x--;

if (x==3) x = x * 2;
else x = 0 ;
```

Sei nun $\mathtt{H}_1$ die Hashtabelle zu der ersten `if`-Aussage in Abbildung 5.2 und $\mathtt{H}_2$ die Hashtabelle der zweiten `if`-Aussage. Folglich ergeben sich für die Variable x die Hashtabelleneinträge:

$$\begin{aligned}\mathtt{H}_1(x) =&(((x > 0), \quad x + 1),\\ &((x \leq 0), \quad x - 1))\end{aligned}$$

$$\begin{aligned}\mathtt{H}_2(x) =&(((x = 3), \quad x * 2),\\ &((x \neq 3), \quad 0))\end{aligned}$$

Algorithmisch bildet man die Komposition $\mathtt{H}_1 \circ \mathtt{H}_2$ für eine Variable x wie folgt: Für jedes Tupel $(cond, val)$ aus der Liste $\mathtt{H}_1(x)$ wird eine neue Liste konstruiert, indem man in der Tupelliste $\mathtt{H}_2(x)$ jedes Auftreten der Variablen x durch val ersetzt und anschließend die Bedingung jedes Tupels in $\mathtt{H}_2(x)$ mit $cond$ verundet. Die Konkartination dieser Listen entspricht $(\mathtt{H}_1 \circ \mathtt{H}_2)(x)$. Treten in $\mathtt{H}_2(x)$ mehrere Variablen auf müssen die Elemente der Tupellisten aller Variablen in $\mathtt{H}_1(x)$ permutiert werden. Als Beispiel dient Abbildung 5.

$$\begin{aligned}(\mathtt{H}_1 \circ \mathtt{H}_2)(x) =&(((x > 0 \wedge x + 1 = 3), \quad 2 * (x + 1)),\\ &((x \leq 0 \wedge x - 1 = 3), \quad 2 * (x - 1))\\ &((x > 0 \wedge x + 1 \neq 3), \quad 0)\\ &((x \leq 0 \wedge x - 1 \neq 3), \quad 0))\end{aligned}$$

```
S1: if (x>0) x++;
    else x--;

S2: if (x==3) y = y * 2;
    else y = 3 ;
```

$$\begin{aligned}\mathtt{H}_{\mathtt{S1}}(x) =&(((x > 0), \quad x + 1),\\ &((x \leq 0), x - 1))\\ \mathtt{H}_{\mathtt{S1}}(y) =&((true, \quad y))\end{aligned}$$

$$\mathtt{H}_{\mathtt{S2}}(x) =((true, \quad x))$$

$$\begin{aligned}\mathtt{H}_{\mathtt{S2}}(y) =&(((x = 3), \quad 2 * y),\\ &(x \neq 3), \quad 3))\end{aligned}$$

$$\begin{aligned}(\mathtt{H}_{\mathtt{S1}} \circ \mathtt{S2})(x) =&(((x > 0), \quad x + 1),\\ &((x \leq 0), \quad x - 1))\end{aligned}$$

$$\begin{aligned}(\mathtt{H}_{\mathtt{S1}} \circ \mathtt{S2})(y) =&(((x > 0 \wedge x + 1 = 3), \quad 2 * y),\\ &((x \leq 0 \wedge x - 1 = 3), \quad 2 * y),\\ &((x > 0 \wedge x + 1 \neq 3), \quad 3),\\ &((x \leq 0 \wedge x - 1 \neq 3), \quad 3),\end{aligned}$$

Abbildung 5: Relationale Komposition

Die Hashtabelle die die Übergangsrelation eines Programmblocks beschreibt ist folglich die Hashtabelle, die durch relationale Komposition der Hashtabellen der einzelnen Programmaussagen dieses Blocks entsteht.

5.3 Komposition von Bedingungen und Hashtabellen

In manchen Fällen ist notwendig zu einer Hashtabelle nachträglich eine Bedingung hinzuzufügen. Diese Bedingung ergibt sich aus der Komposition $\mathtt{H} \circ c$ einer Bedingung c mit einer Hashtabelle $\mathtt{H}$. Hierzu ersetzt man zuerst alle Variablen der Bedingung durch die entsprechenden Einträge in der Hashtabelle. Man permutiert alle

möglichen Variablenbelegungen, die in der Hashtabelle eingetragen sind und verundet die Bedingung mit der Bedingung des entsprechenden Hashtabelleneintrags. Man erhält so eine Liste von Bedingungen. Durch Veroderung der Listeneinträge erhält man die neue Bedingung.

```
S0: int k, x, n;

S1: if (k<3) x++;
    else x--;

S2: while (x<n)
    ...
```

$$\mathtt{H}_{\mathtt{S1}}(x) = (((k < 3), x + 1), \\ ((k \geq 3), x - 1))$$
$$\mathtt{H}_{\mathtt{S1}}(n) = ((true, n))$$

Die Komposition von $\mathtt{H}_{\mathtt{S1}}$ und der Bedingung $(x < n)$ lautet dann:

$$(\mathtt{H}_{\mathtt{S1}} \circ (x < n)) = ((k < 3) \wedge \quad (x + 1 < n)) \\ \vee \\ ((k \geq 3) \wedge \quad (x - 1 < n))$$

Abbildung 6: Komposition von Bedingungen und Hashtabelle

Abbildung 6 zeigt ein Beispiel für die Komposition von einer Hashtabelle und einer Bedingung.

5.4 Schleifen und Abstraktion

Die Übergangsrelation für Schleifen besteht aus zwei Teilen. Der erste ist Teil ist die Relation für die n-fache Ausführung einer Schleife und lässt sich mit den bis jetzt gezeigten Implementierungsschritten einfach realisieren. Um diese Hashtabelle zu erzeugen wird die `while`-Aussage in eine `if`-Aussage umgeformt, deren `then`-Block aus dem Schleifenkörper besteht und deren `else`-Block leer ist. Es wird ein neuer Block $\mathtt{B}_n$ erzeugt, indem die `if`-Aussage n-mal hintereinander ausgeführt wird. Dieser Block wird erweiterter Schleifenkörper genannt. Die Bedingung für der Verlassen der Schleife nach n Iterationen wird erzeugt, indem man die Hashtabelle von $\mathtt{B}_n$ bildet und sie mit der negierten Abbruchbedingung verknüpft. Diese Bedingung wird mit der Bedingung jedes Tupels der Hashtabelle von $\mathtt{B}_n$ verundet und daraus ergibt sich die Hashtabelle für die n-fache Iteration der Schleife.
Der zweite Teil beinhaltet die in Abbildung 2 gezeigte Prädikatenabstraktion für eine Menge von Übergangsprädikaten $\mathcal{P}$. Für die Implementierung der Abstraktion

wird der Theorembeweiser **CLP-Prover** [CLP] von Andrey Rybalchenko verwendet. Der Theorembeweiser testet die Erfüllbarkeit einer Formel über gestrichenen und ungestrichenen Programmvariablen.
Sei nun $\tau \equiv \texttt{while(C)}\ \tau_B$ eine Schleife. Mit $\texttt{H}_1$ wird die Hashtabelle für die Relation $(\phi(C) \circ \rho_{\tau_B}) \circ \neg\phi(C)$ also für das einmalige Ausführen der Schleife bezeichnet. $\texttt{H}_2$ ist die Hashtabelle für die Relation $(\phi(C) \circ \rho_{\tau_B})^2 \circ \neg\phi(C)$, die das zweimalige Ausführen der Schleife beschreibt. Für eine Variable v_i lassen sich aus den Hashtabellen $\texttt{H}_1$ und $\texttt{H}_2$, wie bereits gezeigt, zwei Formeln $B_1(v_i) := Eval_{\texttt{H}_1}(v_i)$ und $B_2(v_i) := Eval_{\texttt{H}_2}(v_i)$ konstruieren. Die Frage $\rho_\tau \subseteq p$, ob ein Prädikats $p \in \mathcal{P}$ eine gültige Übermenge der Übergangsrelation einer Schleife $\tau \equiv \texttt{while(C)}\ \tau_B$ ist, wird durch den Theorembeweiser beantwortet. Diese Frage wird für jede Programmvariable als Formel formuliert, die besagt, dass die Änderung des Wertes der Programmvariable durch Ausführung des Schleifenkörpers das Prädikat p impliziert. Für die Implementierung dieser Frage wird eine neue Menge von Prädikaten $\mathcal{P} := \{(a' = a), (a' \geq a), (a' \leq a)\}$ konstruiert mit $a \notin V$. Es gilt $Q_p(B_1(v_i), B_2(v_i)) := (a = B_1(v_i) \wedge a' = B_2(v_i)) \rightarrow p$. Abbildung 7 zeigt die Frage nach der Implikation $a' = a$ für die Programmvariable v_i.

$$\begin{aligned} Q_{(a'=a)}(B_1(v_i), B_2(v_i) &:= \big(a = B_1(v_i) \wedge a' = B_2(v_i)\big) \rightarrow \big(a' = a\big) \\ &= \neg\big(a = B_1(v_i) \wedge a' = B_2(v_i)\big) \vee \big(a' = a\big) \end{aligned}$$

Abbildung 7: Beispiel für eine Frage an den Theorembeweiser

Durch den Theorembeweiser kann jetzt zwischen vier möglichen Fällen unterschieden werden um eine Hashtablle H für die abstrakte Relation für eine Variable v_i zu bilden. Zu diesem Zweck werden die im letzten Abschnitt eingeführten Hashtabellen $\texttt{H}_1$ und $\texttt{H}_2$ konstruiert, sowie die zugehörigen Formeln $B_1(v_i)$ und $B_2(v_i)$. Es werden folgende Fälle unterschieden:

- $Q_{(a'=a)}(B_1(v_i), B_2(v_i)$ ist erfüllbar
  $\texttt{H}(v_i) := (true, v_i)$
- $Q_{(a'\geq a)}(B_1(v_i), B_2(v_i)$ ist erfüllbar
  $\texttt{H}(v_i) := (\epsilon_i \geq 0, v_i + \epsilon_i)$
- $Q_{(a'\leq a)}(B_1(v_i), B_2(v_i)$ ist erfüllbar
  $\texttt{H}(v_i) := (\epsilon_i \geq 0, v_i - \epsilon_i)$
- sonst
  $\texttt{H}(v_i) := (\epsilon_i = \bot, v_i + \epsilon_i)$

Die Füllvariable ϵ_i wird bei Bedarf erfüllt und kann nicht durch andere Programmaussagen verändert werden. Sie wird verwendet um die Verwaltung von Ungleichungen zu umgehen.

Um aus dieser Hashtabelle die Hashtabelle für die abstrakte Relation einer Schleife zu bilden müssen die Bedingungen hinzugefügt werden, dass die Schleife in den ersten n Iterationen nicht verlassen wird, was sich mit den bisher gezeigten Mitteln einfach realisieren lässt, und die Bedingung, dass die Schleife nach einer beliebigen Anzahl an Iterationen verlassen wird. Die Bedingung erhält man durch Komposition der Hashtabelle für die abstrakte Relation und der negierten Abbruchbedingung der Schleife.
Aus diesen Teilschritten lässt sich die Hashtabelle für Übergangsrelationen von Schleifen algorithmisch berechnen, indem man für alle Variablen die Tupellisten der einzelnen Teilschritte konkartiniert. Für eine Schleife `while(x<10) x++;` ist der Hashtabelleneintrag für x mit einer Genauigkeit von 2 und Füllvariable ϵ_x wie folgt:

$$\begin{aligned}\texttt{H}(x) =(&((x \geq 10), && x),\\ &((x < 10 \wedge x + 1 \geq 10), && x + 1),\\ &((x < 10 \wedge x + 1 < 10 \wedge x + 2 \geq 10), && x + 2),\\ &((x < 10 \wedge x + 1 < 10 \wedge x + 2 < 10 \wedge x + \epsilon_x \geq 10 \wedge \epsilon_x \geq 0), && x + \epsilon_x))\end{aligned}$$

Abbildung 8: Hashtabelle für `while(x<10) x++;`

5.5 Sprunganweisungen

Um ein Programm analysieren zu können, müssen die Programmaussagen `return`, `break` und `continue` für die Übergangsrelationen von Programmblöcken berücksichtigt werden. Diese Anweisungen werden auch Sprunganweisungen genannt. Andere Sprunganweisungen wie `goto` werden in dieser Arbeit nicht berücksichtigt, lassen sich aber analog behandeln.

- `return`:
  Durch eine `return`-Anweisung wird der aktuelle Funktionskörper sofort verlassen, unabhängig davon, ob sich diese Anweisung innerhalb ein Blocks befindet, der dem Funktionskörper untergeordnet ist (z.B. ein Schleifenkörper). Weiterhin gibt `return` einen Rückgabewert für die aktuelle Funktion zurück.

- `break`:
  Eine `break`-Anweisung hat nur einen Effekt, wenn sie innerhalb eines Schleifenkörpers oder einer `switch`-Anweisung steht. `Switch`-Anweisungen werden nicht näher betrachtet, da sie sich einfach in `if`-Anweisung ohne `break` umbauen lassen. Eine `break`-Anweisung in einem Schleifenkörper, oder in einem Programmblock, der innerhalb eines Schleifenkörpers steht, hat den Effekt, dass die Schleife an dieser Stelle verlassen wird.

- `continue`: Die `continue`-Anweisung hat nur innerhalb einer Schleife einen Effekt. Durch diese Anweisung wird die Ausführung des Schleifenkörpers an dieser Stelle abgebrochen, und falls die Abbruchbedingung der Schleife weiterhin erfüllt ist, wird der Schleifenkörper von Beginn an ausgeführt.

Um die Berechnung der Hashtabelle für die Übergangsrelation eines Programmblocks unter Berücksichtigung von `return, break` und `continue` zu realisieren, wird ein Kellerspeicher (engl. Stack) angelegt. Jedes mal, wenn die Übergangsrelation einer Programmaussage berechnet wird, die untergeordnete Programmblöcke enthält (z.B. `if` oder `while`), wird diese Programmaussage auf den Kellerspeicher gelegt. Nach der Berechnung dieser Übergangsrelation wird die entsprechende Programmaussage vom Kellerspeicher abgehoben.

Das unterste Element des Kellerspeichers ist das Programm selbst. `return`-, `break`- und `continue`-Aussagen können den Kellerspeicher verändern. Die Auswirkung dieser Programmaussagen auf den Kellerspeicher lassen sich wie folgt beschreiben:

- Durch eine `return`-Anweisung werden solange Elemente von dem Kellerspeicher abgehoben, bis das zuletzt abgehobene Element eine Funktionsdefinition ist. Es wird angenommen, dass jede Ausführung einer Funktion mit einer `return`-Anweisung endet.
- Durch eine `break`-Anweisung werden solange Elemente von dem Kellerspeicher abgehoben, bis das zuletzt abgehobene Element eine Schleife ist.
- Durch eine `continue`-Anweisung werden solange Elemente von dem Kellerspeicher abgehoben, bis das zuletzt abgehobene Element eine Schleife ist. Anschließend wird dieses Element wieder auf den Kellerspeicher gelegt.

Abbildung 10 zeigt verschiedene Zustände des Kellerspeichers nach Ausführung der entsprechenden Programmaussage des in Abbildung 9 beschriebene Programm.

Zur Berechnung der Tupelliste des Rückgabewerts einer Funktion, wird bei der Analyse eines Programmblocks nicht nur die zugehörige Hashtabelle berechnet, sondern auch die Tupelliste für den Rückgabewert des Programmblocks. Enthält der Programmblock, oder ein ihm untergeordneter Programmblock, keine `return`-Anweisung, so wird eine leere Liste zurückgegeben. Weiterhin muss für jede Art von Programmaussage ein Rückgabewert definiert werden.

- Zuweisung, `break`- und `continue`-Anweisungen:
  Diese Anweisungen geben eine leere Liste zurück.
- `return`-Aussagen: Falls durch eine `return`-Aussage ein Wert zurückgegeben wird, also die Funktion in der die `return`-Aussage steht, nicht `void` zurückgibt, so wird der Rückgabewert mit der Hashtabelle der vorangehenden Programmaussage ausgewertet, und die so entstandene Liste wird als Rückgabewert zurück gegeben.

```
Program P:
S0: int foo() {
S1:    while(x<0) {
S2:       if (x<u) {
S3:          x++;
S4:          continue;
          }
S5:       if (x==u) {
S6:          break;
           } else {
             x--;
S7:          return x;
S8:        }
S9:    }
S10:   return x;
S11: }
```

Abbildung 9: Beispielprogramm für das Verhalten des Kellerspeichers

- `if`-Aussagen: Eine `if`-Aussage besteht aus einer Bedingung und zwei Programmblöcken, dem `then`-Block, der ausgeführt wird, wenn die Bedingung erfüllt ist, und dem `else`-Block, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Sollten einer `if`-Aussage in dem gleichen Programmblock andere Programmaussagen folgen, kann man diese wegfallen lassen, wenn man sie, wie in Abbildung 11 gezeigt, sowohl an das Ende des `then`-Blocks, als auch an das Ende des `else`-Blocks anhängt. Diese Blöcke werden erweitere Blöcke genannt. Diese Änderung ist notwendig, da z.B. in dem Fall, dass der `then`-Block eine `return`-Aussage enthält, der `else`-Block jedoch nicht, es nicht möglich ist die zu der `if`-Aussage gehörende Hashtabelle mit der einer nachfolgenden Programmaussage zu verknüpfen, da sonst die `return`-Aussage ignoriert werden würde.

  Der Rückgabewert für die `if`-Aussage ergibt sich also aus der Veroderung der Rückgabewerte des erweiterten `then`-Blocks und des erweiterten `else`-Blocks.

- `while`-Aussagen: Für `while`-Aussagen ist es nicht möglich, wie bei `if`-Aussagen, den Schleifenkörper um die, der `while`-Aussage folgenden Programmaussagen zu erweitern, da der Schleifenkörper mehrmals in Folge ausgeführt wird. In diesem Fall müssen die Programmaussagen, die der Schleifenaussage folgen, als eigener Block $\texttt{B}_{rest}$ gespeichert werden. Der erweiterte Schleifenkörper $\texttt{B}_n$, für die n-fache Ausführung des Schleifenkörpers wird unter Berücksichtigung von `return, break` und `continue` wie folgt erzeugt:

  Die Schleife wird zu einer `if`-Aussage umgewandelt, deren Bedingung der

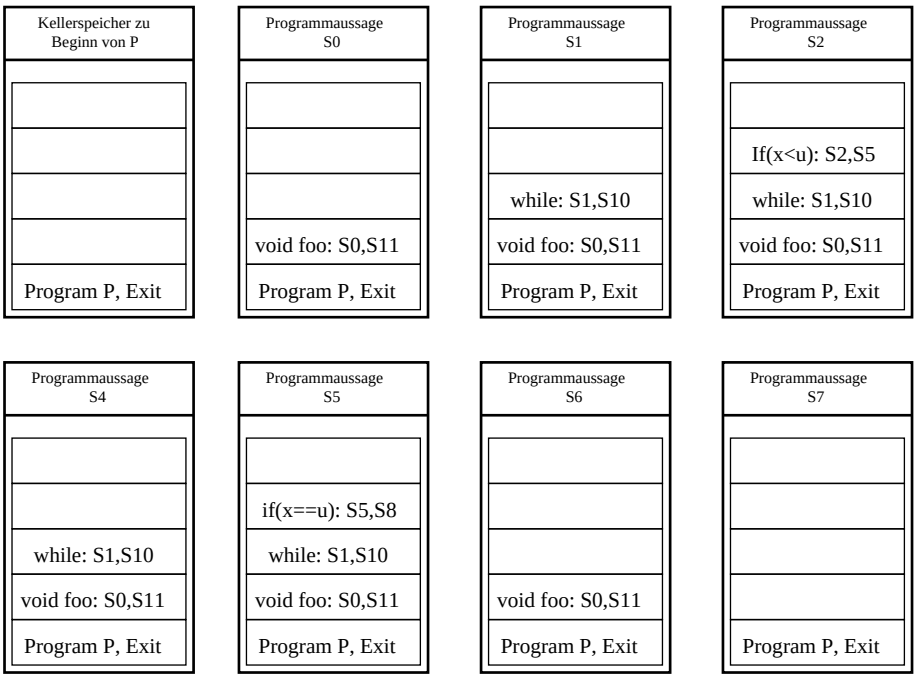

Abbildung 10: Zustand des Kellerspeichers für das Programm P in Abbildung 9

Abbruchbedingung der Schleife entspricht mit dem Schleifenkörper als `then`-Block und einem leeren `else`-Block.

Diese Aussage wird durch die im letzten Punkt beschriebenen Änderungen für `if`-Aussagen umgeformt und alle Programmaussagen, die einer `continue`-Aussage folgen werden entfernt. Diese `if`-Aussage wird n-mal hintereinander gehängt und ergibt so einen neuen Block. An diesen Block wird noch eine weitere `if`-Aussage angehängt, deren Bedingung die Abbruchbedingung der Schleife ist. Der `then`-Block dieser Aussage ist leer und der `else`-Block ist der oben beschriebene Block $\mathtt{B}_rest$. Der neue Block wird durch die im letzten Punkt beschriebenen Änderungen für `if`-Aussagen umgeformt. Jetzt werden alle Programmaussagen, die einer `return`-Aussage folgen, entfernt. Weiterhin werden alle Programmaussagen, die einer `break`-Aussage folgen, entfernt und die `break`-Aussage wird durch den Block $\mathtt{B}_rest$ ersetzt.

Der Rückgabewert einer für eine Schleifenaussage ergibt sich unter Berücksichtigung dieser Änderungen aus der Disjunktion der Rückgabewerte der einzelnen Teilschritte.

Mit dieser erweiterten Methode, die Hashtabelle für einen Programmblock zu berechnen lässt sich gleichzeitig auch der abstrakte Rückgabewert für diesen Pro-

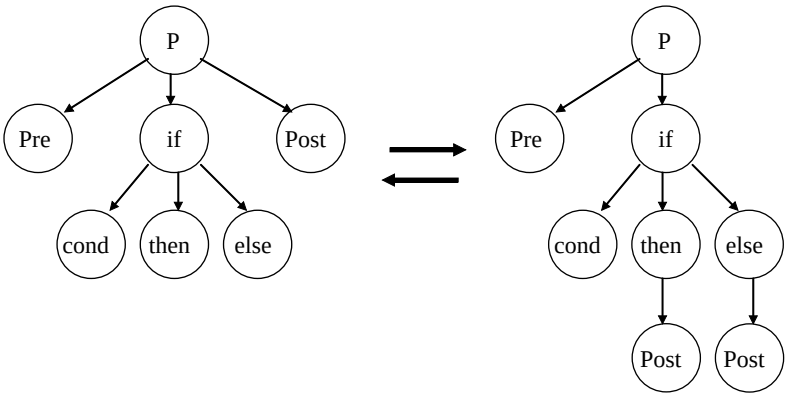

Abbildung 11: Transformation des Syntaxbaums für `if`-Aussagen

grammblock bestimmen.

5.6 Globale Programmaussagen

Globale Programmaussagen können Wertzuweisungen, Funktions- oder Variablendefinitionen sein. Die Handhabung von Variablendefinition und Wertzuweisungen ist aus der vorherigen Kapiteln bekannt. Für die Analyse von Funktionsdefinitionen hält **CIL** eine Liste der formalen- und lokalen Variablen der Funktion vor, sowie den Funktionskörper. Um aus dem Funktionskörper einen Rückgabewert, bzw. eine Übergangsrelation für die Funktion zu berechnen, muss eine Hashtabelle für die Menge aller, in dieser Funktion erscheinenden Variablen erstellt werden. Also eine Hashtabelle die alle bis zu diesem Punkt definierten globalen Variablen, sowie die lokalen- und formalen Variablen dieser Funktion enthält. Allen Variablen v_i dieser Hashtabelle wird die Initalbelegung $(true, v_i)$ zugewiesen. Auch globalen Variablen, denen schon ein Wert zugewiesen wurde, haben diese Initialbelegung, da nicht bekannt ist, an welcher Stelle im Programmcode diese Funktion aufgerufen wird. Es wird jetzt eine künstliche Programmaussage erzeugt, deren Übergangsrelation durch diese Initalhashtabelle gegeben ist. Dieses Programmaussage wird vor die erste Programmaussage des Funktionskörper gehängt. Übergangsrelation und Rückgabewert der Funktion lassen sich mit der im letzten Kapitel gezeigten Methode berechnen. Da jetzt Rückgabewert, Menge der formalen Variablen und Hashtabelle einer Funktion bekannt sind, kann diese als neuer Eintrag zur Funktionshashtabelle hinzugefügt werden.

5.7 Funktionsaufrufe

Durch einen Funktionsaufruf werden, abhängig von den Eingabeparametern der Funktion, möglicherweise globale Variablen verändert und ein Rückgabewert erzeugt. Da eine Funktion vor dem ersten Aufruf definiert sein muss, ist sichergestellt, dass zu jeder aufgerufenen Funktion ein Eintrag in der Hashtabelle al-

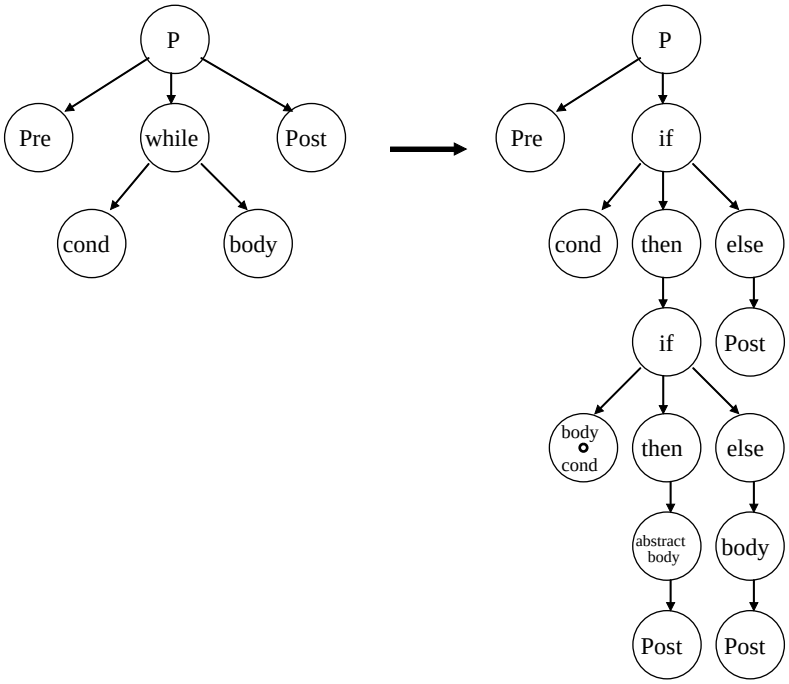

Abbildung 12: Transformation des Syntaxbaums für `while`-Aussagen mit Genauigkeit 1

ler Funktionen existiert und folglich die Menge der formalen Variablen, sowie der Rückgabewert und die Hashtabelle der globalen Veränderungen bekannt sind. Durch einen Funktionsaufruf wird jeder formalen Variablen eine Belegung zugewiesen. So kann eine Hashtabelle $\mathtt{H}_{fun}$ erzeugt werden. Alle Variablen v_i, die nicht in der Menge der formalen Variablen enthalten sind erhalten den Initialwert $(true, v_i)$. Durch die Parameter des Funktionsaufruf kann jeder formalen Variablen in $\mathtt{H}_{fun}$ ein eindeutiger Wert zugewiesen werden. Die Änderungen der globalen Variablen durch den Funktionsaufruf erhält man aus der Komposition von $\mathtt{H}_{fun}$ mit der Hashtabelle der globalen Änderungen dieser Funktion. Den Rückgabewert erhält man durch Komposition von $\mathtt{H}_{fun}$ mit dem allgemeinen Rückgabewert der Funktion. Die Komposition lässt sich analog zu der Komposition von Hashtabellen realisieren.

Für die in Abbildung 13 gezeigte Funktion `foo` ergibt sich der Hashtabellen-

```
int x=0;

F0: int foo(val val)
    {
       if (abs<0)
       {
         x++;
         return -1*val;
       }
       return val;
    }

F1: void main()
    {
S0:    int k = -3;
S1:    k = x+foo(k);
S2:    k = foo(k)+x;
    }
```

Abbildung 13: Funktionsaufruf

eintrag:

`Formale Variablen :` $\{val\}$

`globale Änderungen :`

$$\texttt{H}(x) = ((val < 0, x + 1), (val \geq 0, x))$$

`Rückgabewert :`

$$((val < 0, -val), (val \geq 0, val))$$

Für die in den Zeilen `S1` und `S2` gezeigten Programmaussagen ist zu beachten, dass der Funktionsaufruf in beiden Programmaussagen zu unterschiedlichen Zeitpunkten ausgewertet werden muss, da der Wert der globalen Variablen x innerhalb von `foo` verändert wird. Das verwendete Forntend **CIL** spaltet arithmetische Ausdrücke, die einen Funktionsaufruf enthalten, automatisch in die in Abbildung 14 gezeigten Teilaussagen auf. Es ist folglich sichergestellt, dass ein Funktionsaufruf nie Teil eines arithmetischen Ausdrucks ist.

Für die Abbildung 13 gezeigte Funktion `main` berechnet sich der entsprechende

```
S1: tmp_1 = x;
    tmp_2 = foo(k);
    k = tmp_1 + tmp_2;
S2: tmp_3 = foo(k);
    tmp_4 = x;
    k = tmp_3 + tmp_4;
```

Abbildung 14: Darstellung der Programmaussagen `S1` und `S2` durch **CIL**

Hashtabelleneintrag für die Änderungen der globalen Variablen wie folgt:

$$
\begin{aligned}
\mathtt{H_{S0}}(k) &= (true, -3) \\
\mathtt{H_{S0}}(x) &= (true, x) \\
(\mathtt{H_{S0}} \circ \mathtt{H_{S1}})(k) &= ((-3 < 0, x + (-1) * (-3)), \\
&\quad\ (-3 \geq 0, x + (-3))) \\
&= (true, x + 3) \\
(\mathtt{H_{S0}} \circ \mathtt{H_{S1}})(x) &= ((-3 < 0, x + 1), \\
&\quad\ (-3 \geq 0, x)) \\
&= (true, x + 1) \\
(\mathtt{H_{S0}} \circ \mathtt{H_{S1}} \circ \mathtt{H_{S2}})(k) &= ((x + 3 < 0, (-1) * (x + 3)) + (x + 1 + 1)), \\
&\quad\ (x + 3 \geq 0, (x + 3) + (x + 1)))) \\
&= ((x + 3 < 0, -1), \\
&\quad\ (x + 3 \geq 0, 2 * x + 4)) \\
(\mathtt{H_{S0}} \circ \mathtt{H_{S1}} \circ \mathtt{H_{S2}})(x) &= ((x + 3 < 0, (x + 1 + 1)), \\
&\quad\ (x + 3 \geq 0, x + 1)) \\
&= ((x + 3 < 0, x + 2), \\
&\quad\ (x + 3 \geq 0, x + 1))
\end{aligned}
$$

Da `main` vom Typ `void` ist, und keine formalen Variablen besitzt, lässt sich das Verhalten von `main` allein durch die Hashtabelle $(\mathtt{H_{S0}} \circ \mathtt{H_{S1}} \circ \mathtt{H_{S2}})$ beschreiben.

6 Auswertung

Die beschriebene Methode wird exemplarisch auf den Bubblesort Algorithmus angewendet. Gleichzeitig werden die Vorteile dieser Methode gegenüber vergleichbaren Methoden aufgezeigt. Da im Umfang dieser Arbeit die Behandlung von Zeigern nicht enthalten ist, wird, für den in Abbildung 15 beschriebenen Algorithmus, nur untersucht, ob die verwendeten Schleifen terminieren.

```
void bubblesort(int *ptr, int N)
{
   int temp;
   int i= 0;
   while(i<N-1)
   {
    int j=0;
      while( j < N-1-i )
      {
          if ( *(ptr+j) > *(ptr+j+1) )
         {
             temp = *(ptr+j);
             *(ptr+j) = *(ptr+j+1);
             *(ptr+j+1) = temp;
         }
         j++;
      }
      i++;
   }
   return;
}
```

Abbildung 15: Bubblesort in **C**

Der Vorteil dieser Methode, der sich aus der Verwendung des abstrakten Syntaxbaums ergibt, wird an den Programmvariablen `i` und `j` aufgezeigt. Abbildung 16 zeigt eine vereinfachte Darstellung des Kontrollflussgraphen der Bubblesort Funktion. Abbildung 17 zeigt eine vereinfachte Darstellung des Syntaxbaums.
Die beschriebene Methode berechnet die Übergangsrelationen der einzelnen Konten des Syntaxbaums, beginnend bei den Blättern, rekursiv bis zur Wurzel. Der Vorteil dieser Vorgehensweise lässt sich an der Abstraktion der Schleifen aufzeigen. Bei der beschriebenen Methode wird zuerst die abstrakte Übergangsrelation für die Schleife `while( j < N-1-i )` konstruiert. Diese wird einmalig für den zugehörigen Knoten berechnet. Während der Berechnung der abstrakten Übergangsrelation der Schleife `while(i<N-1)` ist folglich die Übergangsrelation des Schleifenkörpers bereits bekannt. Daraus folgt, dass der, für die Berechnung der Übergangsrelation eine Knotens, entstehende Zeit- und Platzaufwand unabhängig

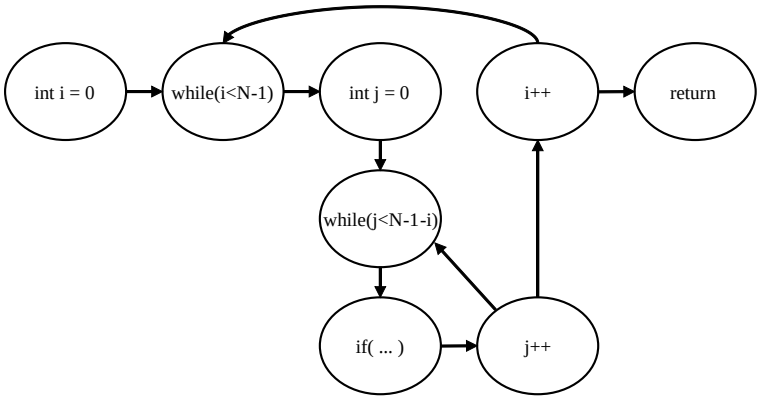

Abbildung 16: vereinfachter CFG für Bubblesort

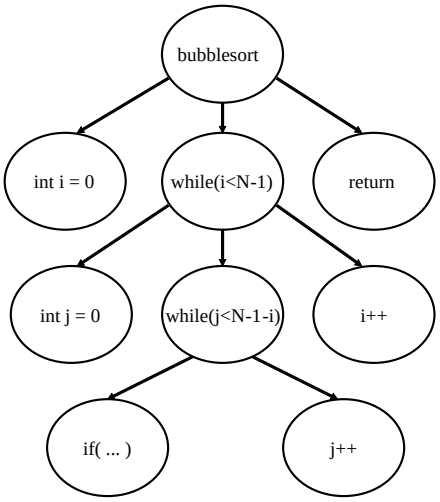

Abbildung 17: vereinfachter Syntaxbaum für Bubblesort

von der Beschaffenheit seiner Kinder ist.

Bei der Berechnung der Übergangsrelationen basierend auf dem Kontrollflussgraphen, beschreibt der erste Knoten den Startpunkt und der letzte Knoten den Austrittspunkt des Programms. Ein offensichtlicher Nachteil des Kontrollflussgraphen ist, dass die Abstraktion einer Schleife erst berechnet werden kann, wenn der Schleifenkörper bekannt ist. Zu diesem Zweck muss der Kontrollflussgraph durchlaufen werden, bis das Ende des Schleifenkörpers erreicht ist. Die so entstehenden Kosten treten bei der hier beschriebenen Methode nicht auf. Ein weiterer Vorteil, der sich aus der Verwendung des Syntaxbaums in diesem Beispiel, ergibt ist, dass automatisch zuerst die Abstraktion der inneren Schleife `while( j < N-1-i )` berechnet wird und das Ergebnis der Abstraktion direkt für die Berechnung der Schleife `while(i<N-1)` verwendet werden kann. Bei der Verwendung des Kon-

trollflussgraphen muss diese Optimierung von Hand implementiert werden.

Ein weiterer Punkt der für die Verwendung des Syntaxbaums spricht ist die Wiederverwendbarkeit der einzelnen Übergangsrelationen. Für die beschriebene Methode gilt, wird ein Knoten innerhalb des Syntaxbaums geändert, so müssen alle Übergangsrelationen neu berechnet werden, die auf dem Pfad von der Wurzel des Syntaxbaums zu dem betroffenen Knoten liegen. Alle anderen Relationen bleiben unverändert. Dies ist ein großer Vorteil gegenüber der Verwendung des Kontrollflussgraphen, da sich der Kontrollfluss durch eine Änderung im Programmcode stark verändern kann und daher in vielen Fällen alle Übergangsrelationen neu berechnet werden müssen.

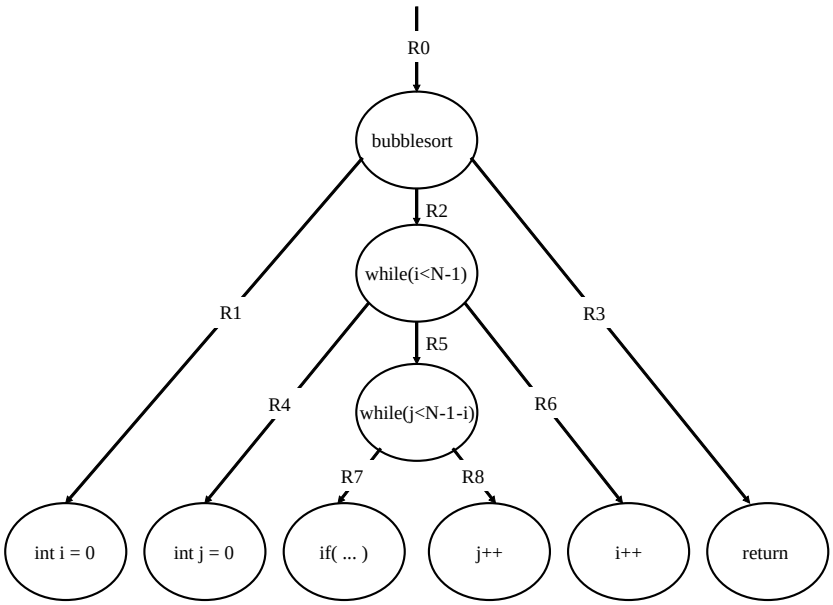

Abbildung 18: AST für Bubblesort mit Übergangsrelationen

Im Folgenden wird gezeigt, wie durch die beschriebene Methode die Übergangsrelationen für die Programmvariablen `i` und `j` für den in Abbildung 15 gezeigten *Bubblesort*-Algorithmus hergeleitet werden und die Aussagekraft dieser Relationen untersucht.

Der in Abbildung 18 gezeigte abstrakte Syntaxbaum ist mit den zu berechnenden Übergangsrelationen `R0` bis `R8` annotiert. Zuerst werden die Übergangsrelationen der Blätter berechnet.

$$
\begin{aligned}
R1(i) =& (i' = 0) \\
R1(j) =& (j' = j)
\end{aligned}
$$

$$
\begin{aligned}
R4(i) =& (i' = i) \\
R4(j) =& (j' = 0)
\end{aligned}
$$

$$
\begin{aligned}
R7(i) =& (i' = i) \\
R7(j) =& (j' = j)
\end{aligned}
$$

$$
\begin{aligned}
R8(i) =& (i' = i) \\
R8(j) =& (j' = j + 1)
\end{aligned}
$$

$$
\begin{aligned}
R6(i) =& (i' = i + 1) \\
R6(j) =& (j' = j)
\end{aligned}
$$

$$
\begin{aligned}
R3(i) =& (i' = i) \\
R3(j) =& (j' = j)
\end{aligned}
$$

Mit den Relationen R7 und R8 wird jetzt die Übergangsrelation R5 berechnet. Die Übergangsrelation für den Schleifenkörper ergibt sich aus $\texttt{R7} \circ \texttt{R8}$. Für die Berechnung der Schleifenrelation wird die beschriebene Abstraktion mit einer Genauigkeit von zwei verwendet.

$$R5(i) = (i' = i)$$

$$
\begin{aligned}
R5(j) =& (j \geq N - 1 - i) \wedge (j' = j) \\
& \vee (j < N - 1 - i) \wedge (j + 1 \geq N - 1 - i) \wedge (j' = j + 1) \\
& \vee (j < N - 1 - i) \wedge (j + 1 < N - 1 - i) \wedge (j + 2 \geq N - 1 - i) \wedge (j' = j + 2) \\
& \vee (j < N - 1 - i) \wedge (j + 1 < N - 1 - i) \wedge (j + 2 < N - 1 - i) \\
& \wedge (j + \epsilon_j \geq N - 1 - i) \wedge (\epsilon_j \geq 0) \wedge (j' = j + \epsilon_j)
\end{aligned}
$$

Der für die Berechnung der Relation R2 benötigte Schleifenkörper K ergibt sich aus der Komposition $\texttt{R4} \circ \texttt{R5} \circ \texttt{R6}$.

$$K(i) = (i' = i + 1)$$

$$\begin{aligned} K(j) =& (0 \geq N - 1 - i) \wedge (j' = 0) \\ & \vee (0 < N - 1 - i) \wedge (1 \geq N - 1 - i) \wedge (j' = 1) \\ & \vee (0 < N - 1 - i) \wedge (1 < N - 1 - i) \wedge (2 \geq N - 1 - i) \wedge (j' = 2) \\ & \vee (0 < N - 1 - i) \wedge (1 < N - 1 - i) \wedge (2 < N - 1 - i) \\ & \wedge (\epsilon_j \geq N - 1 - i) \wedge (\epsilon_j \geq 0) \wedge (j' = \epsilon_j) \end{aligned}$$

Daraus ergibt sich für R2 die Übergangsrelation:

$$\begin{aligned} R2(i) =& (i \geq N - 1) \wedge (i' = i) \\ & \vee (i < N - 1) \wedge (i + 1 \geq N - 1) \wedge (i' = i + 1) \\ & \vee (i < N - 1) \wedge (i + 1 < N - 1) \wedge (i + 2 \geq N - 1) \wedge (i' = i + 2) \\ & \vee (i < N - 1) \wedge (i + 1 < N - 1) \wedge (i + 2 < N - 1) \\ & \wedge (i + \epsilon_i \geq N - 1) \wedge (\epsilon_i \geq 0) \wedge (i' = i + \epsilon_i) \end{aligned}$$

$$\begin{aligned} R2(j) =& (0 \geq N - 1 - i) \wedge (j' = 0) \\ & \vee (0 < N - 1 - i) \wedge (1 \geq N - 1 - i) \wedge (j' = 1) \\ & \vee (0 < N - 1 - i) \wedge (1 < N - 1 - i) \wedge (2 \geq N - 1 - i) \wedge (j' = 2) \\ & \vee (0 < N - 1 - i) \wedge (1 < N - 1 - i) \wedge (2 < N - 1 - i) \\ & \wedge (\epsilon_j \geq N - 1 - i) \wedge (\epsilon_j \geq 0) \wedge (j' = \epsilon_j) \end{aligned}$$

An $R2(j)$ lässt sich der Vorteil dieser Methode aufzeigen. Da die Implikation $(x = K(j) \wedge x' = (K \circ K)(j)) \rightarrow (x' = x)$ erfüllt ist muss für $R2(j)$ keine neue Relation für berechnet werden, sondern es gilt $R2(j) = K(j)$. Zuletzt wird die Übergangsrelation $R0$ aus der Komposition $R1 \circ R2 \circ R3$ berechnet. Es gilt:

$$\begin{aligned} R0(i) =& (0 \geq N - 1) \wedge (i' = 0) \\ & \vee (0 < N - 1) \wedge (1 \geq N - 1) \wedge (i' = 1) \\ & \vee (0 < N - 1) \wedge (1 < N - 1) \wedge (2 \geq N - 1) \wedge (i' = 2) \\ & \vee (0 < N - 1) \wedge (1 < N - 1) \wedge (2 < N - 1) \\ & \wedge (\epsilon_i \geq N - 1) \wedge (\epsilon_i \geq 0) \wedge (i' = \epsilon_i) \end{aligned}$$

$$\begin{aligned} R0(j) =& (0 \geq N - 1) \wedge (j' = 0) \\ & \vee (0 < N - 1) \wedge (1 \geq N - 1) \wedge (j' = 1) \\ & \vee (0 < N - 1) \wedge (1 < N - 1) \wedge (2 \geq N - 1) \wedge (j' = 2) \\ & \vee (0 < N - 1) \wedge (1 < N - 1) \wedge (2 < N - 1) \\ & \wedge (\epsilon_j \geq N - 1) \wedge (\epsilon_j \geq 0) \wedge (j' = \epsilon_j) \end{aligned}$$

Durch die gezeigten Schritte lässt sich jetzt zu der Funktion *Bubblesort* eine Formel in Prädikatenlogik erster Ordnung konstruieren, deren Erfüllbarkeit der Terminierung des *Bubblesort*-Algorithmus entspricht.

$$\begin{aligned} R0 =& (0 \geq N-1) \wedge (i' = 0) \wedge (j' = 0) \\ & \vee (0 < N-1) \wedge (1 \geq N-1) \wedge (i' = 1) \wedge (j' = 1) \\ & \vee (0 < N-1) \wedge (1 < N-1) \wedge (2 \geq N-1) \wedge (i' = 2) \wedge (j' = 2) \\ & \vee (0 < N-1) \wedge (1 < N-1) \wedge (2 < N-1) \\ & \wedge (\epsilon_i \geq N-1) \wedge (\epsilon_i \geq 0) \wedge (i' = \epsilon_i) \\ & \wedge (\epsilon_j \geq N-1) \wedge (\epsilon_j \geq 0) \wedge (j' = \epsilon_j) \end{aligned}$$

Diese Formel ist erfüllbar, also kann davon ausgegangen werden, dass der hier gezeigte *Bubblesort*-Algorithmus terminiert.
Für die Berechnung dieser Formel werden acht Übergangsrelationen berechnet und insgesamt sechs Fragen an den Theorembeweiser gestellt. Der Theorembeweiser wird nur für Schleifen verwendet. Pro Programmvariable müssen höchstens drei Fragen an den Theorembeweiser gestellt werden. Dies ist der Fall, wenn die ersten beiden getesteten Prädikate, in dieser Implementierung sind dies die Prädikate $x' = x$ und $x' \geq x$, nicht erfüllbar sind. Für ein Programm mit n Variablen werden also pro Schleife höchstens $3 * n$ Fragen an den Theorembeweiser gestellt. Die Anzahl der Programmaussagen innerhalb des Schleifenkörpers ist nicht relevant, da vor der Berechnung der abstrakten Übergangsrelation der Schleife eine Übergangsrelation für den gesamten Schleifenkörper berechnet wird.
Der wichtigste Punkt für die Berechnung der Kosten dieser Methode ist die Vereinfachung der Prädikatenlogikformeln. Durch die naive Komposition zweier Übergangsrelationen können sehr große Formeln entstehen. Diese müssen vereinfacht werden, da sonst, sogar für kleine Programme, ein Speicherüberlauf unvermeidbar ist. Um dem vorzubeugen, wird in dieser Arbeit jedes Disjunkt einer durch Komposition entstandenen Relation durch den Theorembeweiser auf Erfüllbarkeit getestet. Nicht erfüllbare Disjunkte werden aus der Relation entfernt, da sie für die Erfüllbarkeit der Disjunktion keine Auswirkung haben. Eine obere Abschätzung für die so entstehenden Kosten für die Berechnung der Komposition zweier Programmaussagen ergibt sich aus dem Produkt der Anzahl der Kinder beider Programmaussagen. Diese Vorgehensweise ist sehr simpel und kostenintensiv, für eine erste Implementierung dieser Methode jedoch ausreichend.

7 Fazit

Mit der gezeigten Methode lässt sich zu einem Programm, oder einem Programmteil eine abstrakte Übergangsrelation erzeugen. Diese Übergangsrelation kann als Prädikatenlogikformel über der Menge der gestrichenen Programmvariablen und ungestrichenen Programmvariablen interpretiert werden. Aufgrund der für Schleifen verwendeten Abstraktion gilt, wenn ein Belegungspaar die Formel nicht erfüllt, so kann es auch nicht von dem Programm berechnet werden, es existieren allerdings Belegungspaare, die zwar die Formel erfüllen, jedoch nicht von dem Programm berechnet werden können.
Mit Hilfe dieser Formel können jetzt Annahmen über das Verhalten von Programmvariablen verifiziert werden. Möchte man beispielsweise für einen Programmteil garantieren, dass eine Variable x innerhalb dieses Programmteils nicht verkleinert werden kann, also, ob $x' \geq x$ gilt, so bildet man die Konjunktion der für den Programmblock berechneten Formel und $x' < x$, der Negation der Annahmen. Jetzt testet man die Erfüllbarkeit der Konjunktion mit einem Theorembeweiser. Stellt der Theorembeweiser fest, dass die Formel nicht erfüllbar ist, so kann garantiert werden, dass x' immer größer, oder gleich x ist. Der Umkehrschluss gilt nicht.
Diese Methode kann auch verwendet werden, um die Abwesenheit von Laufzeitfehlern zu garantieren. Um beispielsweise die Abwesenheit von Divisionen durch Null zu garantieren, wird in der betrachteten Formel jede Division $\frac{a}{b}$ mit einem Prädikat $b = 0$ konjugiert. Dieser Schritt kann automatisiert werden und es gilt, ist die so konstruierte Formel nicht erfüllbar, so kann auch keine Division durch Null auftreten. Das Auftreten anderer Laufzeitfehler, wie Speicherzugriffsverletzungen kann analog überprüft werden.
Da diese Arbeit nur einen Teil der Programmiersprache **C** behandelt, kann noch nicht abgeschätzt werden, wie gut sich diese Methode zur Analyse größerer Programme eignet. Gute Ergebnisse wurden für kleine Programmteile, vor allem aber für Programmteile, die wenig Abstraktion benötigen, erzielt. So ist es denkbar, diese Methode in eine Entwicklungsumgebung zu integrieren und als Hilfsmittel für die Suche nach Fehlern zu verwenden.

8 Zukünftige Arbeiten

Um die Möglichkeiten dieses Ansatzes zu testen, soll diese Methode auf den vollen Umfang der Programmiersprache **C** erweitert werden. Vor allem die Behandlung von Feldern, Zeigern und Rekursion soll genauer betrachtet werden.
Die Laufzeit dieser Methode soll durch eine effizientere Vereinfachung der Prädikatenlogikformeln erheblich verbessert werden. Das häufige Auftreten von Tautologien und Widersprüchen innerhalb der Formeln soll genutzt werden um die Formeln ohne Hilfe des Theorembeweisers stark zu vereinfachen. Gelingt es, die Größe der Formeln ohne Verwendung des Theorembeweisers ausreichend zu reduzieren, so muss der Theorembeweiser nur für die Prädikatenabstraktion von Schleifen verwendet werden.
Um die bestehende Methode zu optimieren, soll die Verwaltung von Funktionsaufrufen dahingehend optimiert werden, dass innerhalb der Übergangsrelationen auch abstrakte Funktionssymbole verwendet werden können, um die Veränderung der Programmvariablen durch einen Funktionsaufruf zu beschreiben. Durch diesen Ansatz soll die Anzahl der Übergangsrelationen, die durch eine Änderung im Programmcode neu berechnet werden müssen, stark reduziert werden. Als Grundlage dieser Optimierung dient der Ansatz, dass eine Änderung in einem Programm als die Änderung eines Knotens im Syntaxbaum dieses Programms zu sehen ist. Es wurde gezeigt, dass die beschriebene Methode die abstrakten Übergangsrelationen für jeden Knoten des Syntaxbaums bestimmt. Durch die Optimierung soll erreicht werden, dass, wenn sich ein Knoten im abstrakten Syntaxbaum ändert, nur die Übergangsrelationen für Knoten, die auf dem Pfad zwischen Wurzel und dem geänderten Knoten liegen, neu berechnet werden müssen.
Weiterhin soll überprüft werden, inwiefern sich die Eigenschaft von **C**, dass Speicher explizit angefordert werden muss, dazu verwenden lässt, bei der Berechnung der Übergangsrelationen von Schleifen auf eine Abstraktion zu verzichten. Dieser Ansatz entsteht aus der Überlegung, dass in Programmen oft über die Länge von Feldern iteriert wird und diese in **C** eine konstante Größe haben.
Mit diesen Erweiterungen soll die in dieser Arbeit vorgestellte Methode auf größeren Programmen getestet werden. Basierend auf diesen Tests soll die Methode so erweitert werden, dass automatisch Übergangs-Prädikate konstruiert werden, die verschiedene Programmeigenschaften garantieren.
Langfristig soll diese Methode auf Laufzeit optimiert werden und in eine Entwicklungsumgebung integriert werden, um die, durch den Compiler gegebene, Syntaxkorrektur um eine Art Semantikkorrektur zu erweitern. Die Methode soll den Programmierer auf mögliche Fehler im Programm hinweisen, die während der Programmausführung zu einem Versagen führen können. Es bleibt abzuwarten, ob diese Erweiterung dem Programmierer als echte Hilfe dient, oder ob durch die notwendige Abstraktion zu viele Informationen verloren gehen um dem Programmierer sachliche Hinweise auf Fehlerquellen geben zu können.

Literatur

[esp] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. 29 PLDI (2002), 58–70.

[tin] Andreas Podelski, Andrey Rybalchenko: Transition Invariants. Proc. 19th IEEE Symposium on Logic in Computer Science (LICS 2004), Turku, Finland. July 13-17, 2004.

[ART] B. Cook, A. Podelski, A. Rybalchenko: Abstraction-refinement for Termination. In Static analysis : 12th International Symposium, SAS 2005

[SPL] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Safety. Springer, 1995.

[nis] National Institute of Standards and Technology (USA). – http://www.nist.gov/

[gdb] GDB: The GNU Project Debugger. – http://www.gnu.org/software/gdb/gdb.html

[del] Deltadebugging - From automated testing to automated debugging. – http://www.st.cs.uni-sb.de/dd/

[SLM] Microsoft SLAM and Static Driver Verifier http://research.microsoft.com/slam/

[BLA] BLAST - Berkeley Lazy Abstraction Software Verification Tool. - http://embedded.eecs.berkeley.edu/blast/

[MoCh] Model Checking, Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled

[OCAML] The CAML Language - http://caml.inria.fr

[C99] ISO/IEC 9899:1999 Standard für C

[CIL] CIL - Infrastructure for C Program Analysis and Transformation, http://manju.cs.berkeley.edu/cil

[CLP] Holzbaur C., OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

[bal] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In SIGPLAN Conference on Programming Language Design and Implementation, pages 203–213, 2001.

[col] M. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Computer Aided Verification, pages 293–304, 1998.

[gra] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, Proc. 9th INternational Conference on Computer Aided Verification (CAV'97), volume 1254, pages 72–83. Springer Verlag, 1997.

[pod] A. Podelski, A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005