Verification Condition Generation

Viktor Barie

Technische Universität Kaiserslautern
vbarie@rhrk.uni-kl.de

Abstract

Verification Condition Generation ist eine State-of-the-Art Vorgehensweise um Korrektheit von Programmen zu zeigen. In diesem Paper wird darauf eingegangen worauf es basiert und wie es weiterentwickelt wurde. Des Weiteren werden die Probleme und deren Lösungsversuche von Verification Condition Generation angesprochen, wie zum Beispiel die Größe der Verificaton Conditions, die in Bezug auf die Größe des betrachteten Programms exponentiell wachsen kann. Der Fokus des Papers liegt dabei auf der partiellen Korrektheit von Programmen. Im Anschluss werden spezielle Tools und deren Vorgehensweisen zum Erstellen von Verification Conditions vorgestellt.

Keywords and phrases Hoare-Logik, schwächste Vorbedingung, Predicate Transformer, Basic Paths, P-Induktiv

1 Einführung

Verification Condition Generation (VCG) ist eine State-of-the-Art Vorgehensweise, mit Hilfe derer Programme auf Korrektheit geprüft werden können. Allgemein geht es bei VCG darum, sogenannte Verification Conditions (VCs) aufzustellen. Diese sind prädikatenlogische Formeln erster Stufe, welche anstelle des Programms überprüft werden. In diesem Paper möchte ich darauf eingehen, wie VCs erzeugt werden und welche Probleme dabei entstehen können. Der Fokus liegt dabei auf der partiellen Korrektheit und ich gehe wie folgt vor:
Zunächst erläutere ich die Hoare-Logik, welche grundlegend für die Entwicklung der VCG ist. Dann stelle ich die grundlegende Vorgehensweise vor, mit der VCs gebildet werden. Des Weiteren werde ich deren Probleme und Verbesserungsvorschläge ansprechen. Im letzten Kapitel folgt eine Vorstellung bekannter Tools von VCG. Im folgenden Paper benutze ich IMP [6] als Sprache auf der die Berechnungen durchgeführt werden.

▶ **Definition 1** (IMP: An Imperative Language).

- Int Integer Literale n
- $Bool$ Boolean $\{true, false\}$
- Loc Variablen $\{x, y, z, \dots\}$
- $Aexp$ Arithmetische Ausdrücke e
- $Bexp$ Bool'sche Ausdrücke b
- $Comm$ Befehle c

$$
\begin{array}{lll}
b ::= & true & \\
 & \mid false & \\
 & \mid e_1 = e_2 & \text{für } e_1, e_2 \in Aexp \\
 & \mid e_1 < e_2 & \text{für } e_1, e_2 \in Aexp \\
 & \mid \neg b & \text{für } b \in Bexp \\
 & \mid b_1 \vee b_2 & \text{für } b_1, b_2 \in Bexp \\
 & \mid b_1 \wedge b_2 & \text{für } b_1, b_2 \in Bexp
\end{array}
$$

$$
\begin{array}{lll}
e ::= & n & \text{für } n \in Int \\
 & \mid x & \text{für } x \in Loc \\
 & \mid e_1 + e_2 & \text{für } e_1, e_2 \in Aexp \\
 & \mid e_1 - e_2 & \text{für } e_1, e_2 \in Aexp \\
 & \mid e_1 * e_2 & \text{für } e_1, e_2 \in Aexp
\end{array}
\qquad
\begin{array}{lll}
c ::= & skip & \\
 & \mid x := e & \text{für } x \in L \wedge e \in Aexp \\
 & \mid c_1; c_2 & \text{für } c_1, c_2 \in Comm \\
 & \mid if\ b\ then\ c_1\ else\ c_2 & \text{für } b \in Bexp \wedge c_1, c_2 \in Comm \\
 & \mid while\ b\ do\ c & \text{für } c \in Comm \wedge b \in Bexp
\end{array}
$$

2 Hoare-Logik

Die Hoare-Logik wurde 1969 in "An axiomatic Basis for computer programing" von C. A. R. Hoare vorgestellt [5]. Sie befasst sich mit der formalen Korrektheit von Programmen in einer imperativen Sprache. Das zentrale Werkzeug der Hoare-Logik ist das Hoare-Tripel $\{P\}\ c\ \{Q\}$. Dabei steht P für die Vorbedingung, Q für die Nachbedingung und c für die Ausführungen des Programms. Des Weiteren ist auf dem Hoare-Tripel die partielle Korrektheit definiert.

▶ **Definition 2** (Partielle Korrektheit)**.** Partielle Korrektheit, notiert $\{P\}\ c\ \{Q\}$, bedeutet: Wenn das Programm terminiert und der Anfangszustand σ die Formel P erfüllt, dann transformiert c den Zustand σ zu σ' und σ' erfüllt Q.

Das Hoare-Kalkül befasst sich mit der Semantik und dem Beweis von Hoare-Tripel.

▶ **Definition 3** (Hoare-Kalkül)**.** Das Hoare-Kalkül besteht aus folgenden 6 Regeln.

$$1: \frac{}{\{P\}\ SKIP\ \{P\}} \qquad 2: \frac{}{\{P[u := t]\}\ u := t\ \{P\}}$$

$$3: \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1\ ;\ c_2\ \{Q\}} \qquad 4: \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ if\ b\ then\ c_1\ else\ c_2\ \{Q\}}$$

$$5: \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ while\ b\ do\ c\ \{P \wedge \neg b\}} \qquad 6: \frac{(P' \to P)\ \{P\}\ c\ \{Q\}\ (Q \to Q')}{\{P'\}\ c\ \{Q'\}}$$

Skip(1) ist naheliegend. Im Programm wird nichts an P verändert, deswegen bleibt P auch nach dem Programm erhalten. Die Wertzuweisung(2) ist lediglich eine Rückwärtssubstitution in P. Die Sequentielle Komposition(3) ist die Abfolge sequentieller Programme c_1 und c_2. Bei der bedingten Anweisung(4) und der Schleifenregel(5) wird die Bedingung mit in die Vorbedingung geholt. Die Konsequenzregel(6) ist eine generelle Aussage über Folgerung von Aussagen. Die Beweise zu den Regeln befinden sich in Apt & Olderog [1].
Ein kleines Beispielprogramm:

Listing 1 (Beispielprogramm 1)

$\{true\}$
$if\ \ y \leq 0\ \ then\ \ x := 1\ \ else\ \ x := y$
$\{x > 0\}$

Zuerst wenden wir die bedingte Anweisung an, dann die Wertzuweisung und zum Abschluss die Konsequenzregel.

$$\frac{\dfrac{(true \wedge (y \leq 0)) \to (1 > 0) \vdash \{1 > 0\}x := 1\{x > 0\}}{\vdash \{true \wedge y \leq 0\}x := 1\{x > 0\}} \qquad \dfrac{(true \wedge (y > 0)) \to (y > 0) \vdash \{y > 0\}x := y\{x > 0\}}{\vdash \{true \wedge y > 0\}x := y\{x > 0\}}}{\vdash \{true\}if\ y \leq 0\ then\ x := 1\ else\ x := y\{x > 0\}}$$

Nun ein Beispiel mit einer Schleife.

Listing 2 (Beispielprogramm 2)

$\{x \leq 0\}$
$while\ \ x \leq 5\ \ do\ \ x := x + 1$
$\{x = 6\}$

Die Anwendung der Schleifen-, Wertzuweisungs- und Konsequenzregel ergibt:

$$\frac{\dfrac{(x \leq 6 \wedge x \leq 5) \to (x + 1 \leq 6) \vdash \{x + 1 \leq 6\}x := x + 1\{x \leq 6\}}{\vdash \{x \leq 6 \wedge x \leq 5\}x := x + 1\{x \leq 6\}}}{\vdash \{x \leq 6\}while\ x \leq 5\ do\ x := x + 1\{x \leq 6 \wedge x > 5\}}$$

Daraus folgt mit der Konsequenzregel (wobei W die Schleife darstellt):

$$\frac{(x \leq 0) \rightarrow (x \leq 6) \qquad \vdash \{x \leq 6\} W \{(x \leq 6) \wedge (x > 5)\} \qquad [(x \leq 5) \wedge (x > 5)] \rightarrow (x = 6)}{\{x \leq 0\} W \{x = 6\}}$$

Wie man sieht, haben wir uns hier eines kleinen Tricks bedient. Wir haben die Invariante $x \leq 6$ benutzt, damit nicht eine unbestimmte Anzahl an Wertzuweisungen vorgenommen werden muss. Eine Invariante muss vor dem ersten und vor jedem weiteren Erreichen des Schleifenguards und am Ende der Schleife gelten. Oben genannte Beispiele können in Jhala's Vorlesungsnotizen nachgelesen werden [6].

3 Verification Conditions

Eine Verification Condition ist eine prädikatenlogische Formel erster Stufe, deren Auswertung gleichzusetzen ist mit der Bewertung, ob ein Programm beziehungsweise ein Programmabschnitt korrekt ist. Die grundlegende Vorgehensweise beinhaltet die Methode der schwächsten Vorbedingung. Diese ist ein Predicate Transformer welche aus der Nachbedingung und dem behandelten Programm eine Formel erstellt. Die schwächste Vorbedingung sei definiert [3]:

▶ **Definition 4** (Schwächste Vorbedingung)**.** Die schwächste Vorbedingung $WP(Q, c)$ hat die Eigenschaft, dass wenn Zustand $\sigma \models WP(Q, c)$ und Ausführung c in Zustand σ ausgeführt wird und zu Zustand σ' führt, dann gilt $\sigma' \models Q$.

3.1 Berechnung der schwächsten Vorbedingung

Die schwächste Vorbedingung WP wird folgendermaßen berechnet:

▶ **Lemma 5** (Predicate Transformer)**.** *Sei $c_1 \ldots c_n$ ein Programm und Q die Nachbedingung. Dann gilt für WP(Q, $c_1 \ldots c_n$):*

$$\begin{aligned}
&WP(Q[x], x := e) && \Leftrightarrow Q[e] \\
&WP(Q, c_1 \ldots c_n) && \Leftrightarrow WP(WP(Q, c_n), c_1 \ldots c_{n-1}) \\
&WP(Q, if\ b\ then\ c_1\ else\ c_2) && \Leftrightarrow b \rightarrow WP(Q, c_1) \wedge \neg b \rightarrow WP(Q, c_2) \\
&W = WP(Q, while\ b\ do\ c) && \Leftrightarrow (b \rightarrow WP(W, c)) \wedge (\neg b \rightarrow Q))
\end{aligned}$$

Um zur Verification Condition zu kommen, wird die Vorbedingung P vor WP gesetzt: $P \rightarrow WP(Q, c)$. Nun können wir Verification Conditions bilden und auflösen.
Sehen wir uns nochmal Beispielprogramm 1 an.

Listing 3 (Auswertung der VC für Beispielprogramm 1)

```
Zur Erinnerung:
{true}if y ≤ 0 then x := 1 else x := y{x > 0}

P → WP(Q, c) = {true} → WP(x > 0, if y ≤ 0 then x := 1 else x := y)
WP(x > 0, if y ≤ 0 then x := 1 else x := y)
⇔ [(y ≤ 0) → WP(x > 0, x := 1)] ∧ [(y > 0) → WP(x > 0, x := y)]
⇔ [(y ≤ 0) → (x > 0)[x:=1]] ∧ [(y > 0) → (x > 0)[x:=y]]
⇔ [(y ≤ 0) → (1 > 0)] ∧ [(y > 0) → (y > 0)]
Daraus folgt:
true → [(y ≤ 0) → (1 > 0)] ∧ [(y > 0) → (y > 0)]
```

Die Verification Condition, die wir nun abgeleitet haben, wird mittels eines automatischen Theorembeweisers geprüft. In diesem Fall lässt es sich jedoch noch gut per Hand zu *true* auflösen. So dargestellte VCs können in der Größe explodieren. Umformung 3 führt Konjunktionen zweier sich eventuell sogar wiederholenden Teilformeln ein, Umformung 4 beinhaltet Rekursion. Allerdings kann statt der rekursiven Formel W die Invariante I eingesetzt werden [6]. Dadurch wäre die Rekursion aufgelöst, das Finden von Invarianten für Schleifen ist jedoch nicht unbedingt leichter zu gestalten. Im folgenden Abschnitt gehe ich auf die Basic Paths Erweiterung ein, welche versucht diese Schwachstellen zu beseitigen.

3.2 Basic Paths

Die Basic Paths Erweiterung, vorgestellt von Bradley/Manna [3], arbeitet mit Pfaden eines Programmes, um die Schwachstellen der ursprünglichen Berechnung der schwächsten Vorbedingung zu umgehen.

3.2.1 Grundlagen

Ein Basic Path ist eine Sequenz von Programmausführungen, welcher an der Vorbedingung oder an einer Schleifeninvariante startet und an einer Schleifeninvariante oder an der Nachbedingung endet. Wie man sehen kann bleibt hier das Problem des Findens der Schleifeninvariante bestehen, sobald eine Schleife in dem zu prüfenden Programm auftaucht. In einem Basic Path gibt es keine Guards mehr. Guards sind Eintrittsbedingungen bei Loop-Schleifen, While-Schleifen und If-Abfragen. Diese werden in zwei sich widersprechende Annahmen umgewandelt (siehe *assume* in Lemma 6). Dementsprechend gibt es pro Guard zwei Basic Paths. In einem wird dieser angenommen und dessen Folgeausführungen werden abgehandelt. Im anderen wird die Negation angenommen und der Guard und dessen Folgeausführungen werden übersprungen. Demnach liegt ein Schleifendurchlauf nur noch innerhalb eines Basic Paths und nicht mehr übergreifend. Bei einer If-Bedingung wird innerhalb eines Basic Paths entweder die then- oder die else-Ausführungen beachtet. Folgende Änderungen an der Berechnung ergibt sich aus der Erweiterung mit Basic Paths:

▶ **Lemma 6** (Predicate Transformer mit Basic Paths)**.**

$$
\begin{aligned}
&WP(Q[x], x := e) && \Leftrightarrow Q[e] \\
&WP(Q, assume\ c) && \Leftrightarrow c \rightarrow Q \\
&WP(Q, c_1 \dots c_n) && \Leftrightarrow WP(WP(Q, c_n), c_1 \dots c_{n-1})
\end{aligned}
$$

Die Basic Paths zu Beispielprogramm 1 sind in den folgenden Listings zu sehen.

Listing 4 (Basic Path und VC 1)

$\{true\}$
$assume\ \ y \leq 0$
$x := 1$
$\{x > 0\}$

$WP(x > 0, assume\ \ y \leq 0; x := 1)$
$\Leftrightarrow WP(WP(x > 0, x := 1), assume\ \ y \leq 0)$
$\Leftrightarrow WP((x > 0)_{[x:=1]}, assume\ \ y \leq 0)$
$\Leftrightarrow [(y \leq 0) \rightarrow (x > 0)_{[x:=1]}]$
$\Leftrightarrow [(y \leq 0) \rightarrow (1 > 0)]$
`Also:`
$true \rightarrow [(y \leq 0) \rightarrow (1 > 0)]$

Listing 5 (Basic Path und VC 2)

$\{true\}$
$assume\ \ y > 0$
$x = y$
$\{x > 0\}$

$WP(x > 0, assume\ \ y > 0; x = y)$
$\Leftrightarrow WP(WP(x > 0, x = y), assume\ \ y > 0)$
$\Leftrightarrow WP((x > 0)_{[x=y]}, assume\ \ y > 0)$
$\Leftrightarrow [(y > 0) \rightarrow (x > 0)_{[x=y]}]$
$\Leftrightarrow [(y > 0) \rightarrow (y > 0)]$
`Also:`
$true \rightarrow [(y > 0) \rightarrow (y > 0)]$

Betrachten wir nun das etwas komplexere Programm LinearSearch.

Listing 6 (LinearSearch)

```
@Vorbedingung P: (0 ≤ l) ∧ (u < |a|)
@Nachbedingung Q: rv ↔ ∃i((l ≤ i ≤ u) ∧ (a[i] = e))
bool LinearSearch(int[] a, int l, int u, int e) {
        int i := l;
        while
        @Invariante I: (l ≤ i) ∧ (∀j((l ≤ j < i) → (a[j] ≠ e)))
                (i ≤ u) do{
                        if (a[i] = e) return true;
                        i := i+1
                }
        return false;
}
```

Die folgenden vier Basic Paths und dazugehörige Verification Conditions ergeben sich:
Der erste Basic Path geht von der Vorbedingung P bis zur Invariante I der Schleife. Also entsteht die VC aus der Verknüpfung von P und WP, während bei WP die Invariante als Nachbedingung des Basic Paths dient und die einzige Ausführung dessen ist $i := l$.

Listing 7 (Basic Path und VC 1 zu LinearSearch)

```
@P: (0 ≤ l) ∧ (u < |a|)
i := l;
@I: (l ≤ i) ∧ (∀j((l ≤ j < i) → (a[j] ≠ e)))

P → WP(I, i := l)
⇔ P → I[i:=l]
⇔ [(0 ≤ l) ∧ (u < |a|)] → [(l ≤ l) ∧ (∀j((l ≤ j < l) → (a[j] ≠ e)))]
```

Der zweite Basic Path beginnt bei der Invariante I und endet bei der Nachbedingung Q. Er betritt die Schleife durch $assume\ i \leq u$ und die If-Bedingung durch $assume\ a[i] = e$. Durch Ausführung des Return-Befehls innerhalb der If-Bedingung endet das Programm mit der Nachbedingung.

Listing 8 (Basic Path und VC 2 zu LinearSearch)

```
@I: (l ≤ i) ∧ (∀j((l ≤ j < i) → (a[j] ≠ e)))
assume i ≤ u;
assume a[i] = e;
rv := true;
@Q: rv ↔ ∃i((l ≤ i ≤ u) ∧ (a[i] = e))

P → WP(Q, assume i ≤ u; assume a[i] = e; rv := true)
⇔ P → WP(Q[rv:=true], assume i ≤ u; assume a[i] = e)
⇔ P → WP(((a[i] = e) → Q[rv:=true]), assume i ≤ u)
⇔ P → [(i ≤ u) → ((a[i] = e) → Q[rv:=true])]
```

Der dritte Basic Path beginnt ebenfalls bei der Invariante und betritt die Schleife, aber sie erfüllt nicht die If-Bedingung. Dadurch wird $i := i + 1$ ausgeführt und der Pfad endet an der Invariante, um zu prüfen, ob die Schleife erneut betreten werden muss (allerdings nicht im selben Lauf des Basic Paths).

Listing 9 (Basic Path und VC 3 zu LinearSearch)

$\texttt{@I:}\ \ (l \leq i) \wedge (\forall j((l \leq j < i) \rightarrow (a[j] \neq e)))$
$assume\ \ i \leq u;$
$assume\ \ a[i] \neq e;$
$i := i + 1;$
$\texttt{@I:}\ \ l \leq i \wedge (\forall j\ (l \leq j < i \rightarrow a[j] \neq e))$

$I \rightarrow WP(I, assume\ \ i \leq u, assume\ \ a[i] \neq e, i := i + 1)$
$\Leftrightarrow I \rightarrow ((i \leq u) \rightarrow ((a[i] \neq e) \rightarrow I_{[i:=i+1]}))$

Der letzte Basic Path beginnt wieder bei der Invariante I, geht diesmal jedoch nicht in die Schleife hinein. Im Pfad wird der Return-Befehl ausgeführt und er endet dadurch bei der Nachbedingung Q.

Listing 10 (Basic Path und VC 4 zu LinearSearch)

$\texttt{@I:}\ \ (l \leq i) \wedge (\forall j((l \leq j < i) \rightarrow (a[j] \neq e)))$
$assume\ \ i > 0;$
$rv := false;$
$\texttt{@Q:}\ \ rv \leftrightarrow \exists i((l \leq i \leq u) \wedge (a[i] = e))$

$I \rightarrow WP(Q, assume\ \ i > u, rv := false)$
$\Leftrightarrow I \rightarrow ((i > u) \rightarrow Q_{[rv:=false]})$

Das Beispiel kann in Bradley/Manna [3] nachgelesen werden.

3.2.2 P-Invariant und P-Induktiv

In diesem Kapitel möchte ich erläutern, dass es ausreicht nur eine endliche Anzahl an Basic Paths zu betrachten, um die Korrektheit eines Programmes nachzuweisen (siehe auch Kapitel 5.2.5 Bradley/Manna [3]).

▶ **Definition 7.** Eine Annotation eines Programmes P ist P-Invariant, wenn diese bei jedem Erreichen valide ist. Eine Annotation ist jedwege Zusicherung über ein Verhalten von Parametern innerhalb des Programmes wie zum Beispiel eine Invariante.

Das ist jedoch nicht automatisch zu prüfen, da eine unendliche Anzahl an Berechnungen überprüft werden müsste sobald eine Schleife ins Spiel kommt. Daher werden Basic Paths benutzt. Daraus folgt:

▶ **Theorem 8.** *Wenn für jeden Basic Path $\{F\}c_1 \ldots c_n\{G\}$ von Programm P die Verification Condition gültig ist, dann sind die Annotationen im Programm P P-Induktiv.*

Proof. Zu zeigen ist: Die Annotationen von einem Lauf des Programmes P sind gültig.
Voraussetzung: Alle Basic Paths erzeugen eine korrekte Verification Condition.
Ein Lauf eines Programmes P besteht aus einer Anzahl von Basic Paths. Durch Induktion über die Anzahl an Basic Paths in einem Lauf lässt sich zeigen, dass die Annotationen dieses Laufes gültig sind, wenn die Verification Conditions aller Basic Paths gültig sind.
Induktionsanfang:
Der Lauf von P besteht aus einem Basic Path. Da für jeden Basic Path die VC korrekt ist, ist damit für alle Basic Paths des Laufes die Korrektheit gezeigt.
Induktionsvoraussetzung:
Gegeben ist ein Lauf der Länge n (n Basic Paths). Wenn alle n Basic Paths gültig sind, ist auch der Lauf inklusive seiner Annotationen gültig.

Induktionsschritt:
Habe der Lauf des Programmes P $n+1$ Basic Paths bp_i, mit den Annotationen F und G_i: $\{F\}bp_1\{G_1\}bp_2, \ldots, bp_n\{G_n\}bp_{n+1}\{G_{n+1}\}$
So gilt nach der Induktionsvoraussetzung die Annotation G_n. Mit der sequentiellen Komposition von Hoare gilt für jeden weiteren beliebigen Basic Path bp_{n+1} auch G_{n+1}. Daher sind alle Annotationen G_i des Laufes von P gültig.
Da wir nun gezeigt haben, dass die Annotation G_{n+1}, und mit ihr auch alle anderen Annotationen G_i, gültig ist, sind alle Annotationen P-Induktiv. ◀

3.2.3 Ausblick auf Funktionsaufrufe in Basic Paths

Funktionsaufrufe innerhalb eines Programmes, also Rekursion, führen genau wie Schleifen zu einer endlichen jedoch unbeschränkten Anzahl an Aufrufen. Bei Schleifen wurden daher die Invarianten eingeführt, bei Funktionsaufrufen bedienen wir uns der Summaries, damit Basic Paths nicht mehrere Funktionsaufrufe innehalten. Eine Summary ist die Nachbedingung mit eingesetzten Werten des Rekursionsaufrufs im Basic Path. Dies ist möglich, da die Nachbedingung die Zusammenfassung des erwarteten Ergebnisses der Funktion beinhaltet, daher auch Summary. In einem Basic Path der einen Funktionsaufruf enthält ist demnach an der Position des Aufrufes ein *assume* Q'. Q' ist dabei die Nachbedingung, in welcher die Parameter an den veränderten Funktionsaufruf angepasst sind. In diesem Fall gibt es keinen Basic Path mit *assume* $\neg Q'$.
Nachfolgend ein Beispiel zur Funktion BinarySearch(int[a], int l, int u, int e).

Listing 11 (Beispiel eines Rekursiven Aufrufs in einem Basic Path)

```
BinarySearch(int[] a, int l, int u, int e)
@Nachbedingung: rv ↔ ∃i((l ≤ i ≤ u) ∧ (a[i] = e))

Der rekursive Aufruf return BinarySearch(a, m + 1, u) wird im Basic Path zu:
assume rv' ↔ ∃i(((m + 1) ≤ i ≤ u) ∧ (a[i] = e))
```

Das komplette Beispiel dazu befindet sich in Bradley/Manna [3].

3.2.4 Ausblick auf Totale Korrektheit in Basic Paths

▶ **Definition 9** (Totale Korrektheit)**.** Totale Korrektheit, notiert $[P]\ c\ [Q]$, bedeutet: $\{P\}\ c\ \{Q\}$ und das Programm terminiert sicher.

Die totale Korrektheit unterscheidet sich nur bei der Frage der Terminierung von der partiellen Korrektheit. Es ist demnach zusätzlich zu zeigen, dass das betrachtete Programm, bei einem Input welcher die Vorbedingung erfüllt, immer terminiert. Terminierungsbeweise basieren auf Well-Founded Relations und Ranking Functions. Man zeigt, dass jeder Aufruf der Funktion auf der betrachteten Relation immer kleiner wird. Auf nähere Ausführungen verzichte ich im Rahmen dieses Papers.

3.3 Ausblick auf Pointer im Hoare-Kalkül

Es gibt wesentliche Probleme bei der Behandlung von Pointern im Hoare-Kalkül. Eines davon bezieht sich auf Pointer Aliasing. Unter Pointer Aliasing versteht man die Situation, dass mehrere unterschiedliche Pointer auf die gleiche Speicheradresse zeigen. Dadurch entstehen Abhängigkeiten, die im Hoare-Kalkül nicht richtig behandelt werden können. Ein weiteres Problem ist die erhöhte Komplexität der Beweise, da diese zusätzlich über die Lokalität von

Zuweisungsbefehlen geführt werden müssen. Bezüglich Pointern wird McCarthy's Regel zur Speicheradressierung zum Hoare-Kalkül hinzugefügt (sie auch [6]), um genannte Probleme zu umgehen.

▶ **Definition 10** (McCarthy's Regel)**.** Sei E_i eine Speicheradresse und M eine Speichervariable, die den Speicher als ein Objekt darstellt, dann gilt:

- $upd(M, E_1, E_2)$: Aktualisiere M an Adresse E_1 mit dem Wert E_2.
- $sel(M, E_1)$: Lese M an Adresse E_1.
- $sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2, & \text{falls } E_1 = E_3 \\ sel(M, E_3), & \text{falls } E_1 \neq E_3 \end{cases}$
- Die angepasste Zuweisungsregel ändert nun den Wert im Speicher: $\{P[M := upd(M, E_1, E_2)]\}E_1^* := E_2\{P\}$

Weitere Ausführungen können in Bornat [2] und Jhala [6] gefunden werden.

4 Beispiel-Tools für Verification Condition Generation

4.1 Extended Static Checker

Extended Static Checker (ESC) ist ein Tool zum Finden von Fehlern in Programmen, welche sonst nur während der Laufzeit oder gar nicht gefunden werden würden. Der Input in ESC\Java ist ein Java-Programm mit oder ohne eventuellen Anmerkungen wie Vorbedingungen, Nachbedingungen oder Invarianten. Der Output ist eine Liste möglicher Defekte des Programms. Des Weiteren wird jede Methode mit ihren Korrektheitsbedingungen in eine Verification Condition umgewandelt und deren Validität getestet.

4.1.1 Guarded Command Sprache

Zuerst verwandelt ESC das Programm in eine äquivalente Guarded Command Repräsentation. In dieser sind komplexe Eigenschaften von Java-Programmen vereinfacht dargestellt. Diese Guarded Command Sprache bringt das Programm in eine Form, in der viele komplexe Eigenschaften bezüglich Soundness und Completeness eliminiert werden [4].

▶ **Definition 11** (Guarded Command Sprache)**.**

A, B, S	∈ Stmt
e	∈ Expr (expressions)
x	∈ Var (variables)

Stmt ::= assert e | assume e | x := e | A ; B | A [] B

Die Berechnung der schwächsten Vorbedingung sieht in ESC wie folgt aus.

▶ **Lemma 12** (Predicate Transformer von ESC)**.**

$$
\begin{aligned}
&WP(Q, assert\ e) && \Leftrightarrow e \wedge Q \\
&WP(Q, assume\ e) && \Leftrightarrow e \rightarrow Q \\
&WP(Q, x := e) && \Leftrightarrow Q(x := e) \\
&WP(Q, A; B) && \Leftrightarrow WP(WP(Q, B), A) \\
&WP(Q, A \,[]\, B) && \Leftrightarrow WP(Q, A) \wedge WP(Q, B)
\end{aligned}
$$

Bisher ist alles recht nahe an der Methode wie auch in Bradley/Manna vorgestellt. $assert\ e$ bedeutet, dass e behauptet wird und demnach zusätzlich gilt. $A \,[]\, B$ bezieht sich auf die Aussage, dass entweder A oder B ausgeführt wird und zwar willkürlich. Es ist vergleichbar mit einer bedingten Anweisung. Ein Beispiel mit dem Programm Betrag [4]:

Listing 12 (Betrag)

```
static int abs(int x)
@ensures result ≤ 0
{
        if (x < 0){
                x = −x;
                @assert x > 0;
        }
        if (c > 0){
                c − −;
        }
        return x;
}
```

Diese Methode wird in die Guarded Command Sprache überführt:

Listing 13 (Betrag in Guarded Command)

$$((assume\ x < 0; x = -x; assert\ x > 0)$$
$$[](assume\ \neg(x < 0)));$$
$$((assume\ c > 0; c = c - 1)$$
$$[](assume\ \neg(c > 0)));$$
$$assert\ x \geq 0$$

Und daraus wird die VC gebildet:

Listing 14 (Betrag VC)

$$((x < 0) \rightarrow ((-x > 0) \wedge (((c > 0) \rightarrow (-x \geq 0)) \wedge (\neg(c > 0) \rightarrow (-x \geq 0))))) \wedge$$
$$(\neg(x < 0) \rightarrow (((c > 0) -> (x \geq 0)) \wedge (\neg(c > 0) \rightarrow (x \geq 0))))$$

4.1.2 Schleifen in ESC

Schleifen in ESC werden wie folgt behandelt:

Listing 15 (While-Schleife in ESC)

```
While {Invariant I} b do c end;
wird überführt in
```
$assert\ I;$
$x_1 := y_1; \ldots; x_n := y_n$
$assume\ I;$
$((assume\ b; c; assert\ I; assume\ false)\ []\ assume\ \neg b)$

Zuerst wird behauptet, dass die Invariante I gilt. Dann werden die Variablen, die in Programm c gegeben sind, $x_1 \ldots x_n$, frischen Variablen mit einem willkürlichen Wert $y_1 \ldots y_n$, zugeordnet. Die willkürlichen Werte stammen aus dem initialen Zustand des Programmes. Anschließend wird die Invariante I angenommen und durch [] wird gesagt, dass entweder $\neg b$ oder b gilt. Letztes führt zur Ausführung von c und der erneuten Behauptung, dass die Invariante auch nach der Ausführung noch gilt. $assume\ false$ ist jetzt jedoch der Trick, der das Ganze erst möglich macht. Dadurch wird sichergestellt, dass in der Verification Condition geprüft wird, ob die Invariante valide ist. In der Berechnung führt $assume\ false$ dazu, dass $false \rightarrow (vorherige\ WP - Berechnung)$ eine Tautologie ist. Die nächste Durchführung der Berechnung ergibt dann, dass $I \wedge true$ geprüft wird. Dadurch entfällt jegliches Anhängsel und die Invariante wird auf Korrektheit geprüft.

4.1.3 Passify

Um mehrfach vorkommende Ausdrücke zu verhindern, führt ESC nun noch Passify ein. Passify macht aus dem Guarded Command Programm eine passive Form ohne Seiteneffekte, damit die dadurch verbundenen Fehlerquellen eliminiert werden. Kurz gefasst verwandelt Passify jede Wertzuweisung in eine Assume-Ausführung um und verwendet dafür neue Instanzen für Variablen. $x := e$ wird also zu $assume\ x_1 := e$. Mehr dazu in [4].
Das Programm Betrag in passiver Form sieht nun wie folgt aus:

Listing 16 (Betrag in passiver Form in ESC)

$$((assume\ x < 0; assume\ x_1 = -x; assert\ x_1 > 0; assume\ x_2 = x_1)$$
$$[](assume\ \neg(x < 0); assume\ x_2 = x));$$
$$((assume\ c > 0; assume\ c_1 = c - 1; assume\ c_2 = c_1)$$
$$[](assume\ \neg(c > 0); assume\ c_2 = c));$$
$$assert\ x \geq 0$$

Durch die Passifikation kann die Semantik auf 2 Ausgaben heruntergebrochen werden:

▶ **Definition 13.** Sei S ein Programm, N und W Zustände eines Programms. Dann beschreibt N die Zustände, bei denen die Ausführungen von S normal terminieren können. W beschreibt die Zustände, bei denen die Ausführungen von S schief gehen können. Dann gilt:

S	$N.S$	$W.S$
$assert\ e$	e	$\neg e$
$assume\ e$	e	$false$
$A; B$	$N.A \wedge N.B$	$W.A \vee (N.A \wedge W.B)$
$A\ []\ B$	$N.A \vee N.B$	$W.A \vee W.B$

Des Weiteren gilt:

▶ **Theorem 14.** *Wenn S in passiver Form ist und $WP.S.Q$ die schwächste Vorbedingung von Programm S mit der Nachbedingung Q ist, dann: $WP.S.Q = \neg(W.S) \wedge (N.S \rightarrow Q)$*

Proof. Durch strukturelle Induktion. ◀

Dadurch gilt für alle passiven Ausführungen S $WP.S.true \leftrightarrow \neg(W.S)$. Die VC von dem Programm Betrag ist dadurch die Folgende:

Listing 17 (VC von Betrag in passiver Form)

$$\neg[((x < 0) \wedge (x_1 = -x) \wedge \neg(x_1 > 0)) \vee (Q_1 \wedge Q_2 \wedge \neg(x_2 \geq 0))]$$
`mit`
$$Q_1 = [((x < 0) \wedge (x_1 = -x) \wedge (x_1 > 0) \wedge (x_2 = x_1)) \vee (\neg(x < 0) \wedge (x_2 = x)]$$
$$Q_2 = [((c > 0) \wedge (c_1 = c - 1) \wedge (c_2 = c_1)) \vee (\neg(c > 0) \wedge (c_2 = c))]$$

Es sind noch Duplikate von Formelabschnitten vorhanden. Doch durch die Disjunktion muss der automatische Theorembeweiser nicht alle Duplikate abarbeiten. Die resultierenden VCs werden maximal quadratisch in Bezug auf die Größe des betrachteten Programms S.

▶ **Theorem 15.** *Wenn S in passiver Form ist, dann ist*
- *$|N.S|$ in $\mathcal{O}(|S|)$, und*
- *$|W.S|$ in $\mathcal{O}(|S^2|)$.*

Proof. Durch strukturelle Induktion über S. ◀

Durch die Betrachtung von Ausnahmenbehandlung würde die VC wieder exponentielle Größe annehmen und die Ausgabewerte des passiven Programms müssten angepasst werden. Ich verzichte an dieser Stelle darauf weiter einzugehen.

K. R. M. Leino hat in [8] gezeigt, dass die Vorgehensweise von ESC nur eine Abwandlung der Berechnung der schwächsten Vorbedingung ist. Bei dieser wurde eine Eigenschaft genutzt, die nur in bestimmten Programmklassen gilt. Diese Eigenschaft ist die schwächste liberale Vorbedingung(WLP). Während WP bei erfüllter Vorbedingung nicht "schief gehen kann" und in einem Zustand terminiert der die Nachbedingung Q erfüllt, muss WLP nicht unbedingt terminieren. Der Zusammenhang zur schwächsten Vorbedingung ist wie folgt:

▶ **Lemma 16.** $\forall Q(WP(S,Q)) \equiv\ WP(S,true) \wedge\ WLP(S,Q)$

Daraus ergeben sich die folgenden angepassten Ausgaben, wobei für die Ausnahmenbehandlung wiederum eine Erweiterung eingeführt werden müsste [8].

▶ **Lemma 17.** *Wenn ¬W(S) alle Zustände beschreibt, deren Ausführung von S garantiert nicht schief gehen können und ¬N(S) alle Zustände beschreibt, deren Ausführung von S garantiert nicht normal terminieren können, dann gilt:*
¬W(S) = WP(S, true)
¬N(S) = WLP(S, false)

Damit ist gezeigt, dass die Vorgehensweise in ESC\Java tatsächlich nur eine Abwandlung der Berechnung schwächsten Vorbedingung ist.

4.2 Boogie

Ein wesentlicher Vorteil von Boogie ist die Architektur, welche die Komplexität von Verification Condition Generation verringert. Zu sehen ist dies vor allem bei den Umformungen von While-Schleifen und Rekursionsaufrufen. Des Weiteren findet Booogie in Verbindung mit dem Spec#-Compiler nicht nur semantische Fehler, wie Verletzungen von Annotationen, sondern auch syntaktische Fehler innerhalb des Programms. In Boogie wird die Berechnung der schwächsten Vorbedingung erweitert um folgende Umformungen. Siehe auch [7].

▶ **Lemma 18** (Zusätzliche Berechnungsregeln der schwächsten Vorbedingung)**.**

$WP(Q, assert\ x) \quad \Leftrightarrow (x \wedge Q)$
$WP(Q, havoc\ x) \quad \Leftrightarrow (\forall x : Q)$
$while\ b(Invariante\ I)\ do\ c;\ wird\ \text{überführt in}$
$\quad assert\ I;$
$\quad havoc\ xs; assume\ I;$
$\quad if\ (b)\{S; assert\ I; assume\ false;\}\ else\ \{\};$

Havoc beschreibt die Funktion, dass x ein willkürlicher Wert zugeschrieben wird und demnach Q für alle x valide sein muss. Diese Funktion wird bei der Umformung von While-Schleifen benötigt.

Die While-Schleife hat dadurch folgende Aussage:

Listing 18 (While-Schleife in Boogie nach Umformung durch Predicate Transformer)

$$
\begin{aligned}
& WP(Q, while\ b(Invariante\ I)\ do\ c) \\
\Leftrightarrow\ & WP(Q, assert\ I; havoc\ xs; assume\ I; if\ b\ do\ (S; assert\ I; assume\ false)\ else\ ()) \\
\Leftrightarrow\ & WP((b \to WP(Q, S; assert\ I; assume\ false)) \wedge (\neg b \to Q), assert\ I; havoc\ xs; assume\ I) \\
\Leftrightarrow\ & WP([(b \to WP(false \to Q, S; assert\ I)) \wedge (\neg b \to Q)], assert\ I; havoc\ xs; assume\ I) \\
\Leftrightarrow\ & WP([(b \to WP(I \wedge true, S)) \wedge (\neg b \to Q)], assert\ I; havoc\ xs; assume\ I) \\
\Leftrightarrow\ & WP([(I \to (b \to WP(I, S)) \wedge (\neg b \to Q))], assert\ I; havoc\ xs) \\
\Leftrightarrow\ & WP([\forall xs((I \to (b \to WP(I, S)) \wedge (\neg b \to Q)))], assert\ I) \\
\Leftrightarrow\ & I \wedge \forall xs((I \to (b \to WP(I, S)) \wedge (\neg b \to Q)))
\end{aligned}
$$

Demnach muss die Invariante gelten. Außerdem muss für alle xs gelten: Wenn die Schleifenbedingung erfüllt ist, dann folgt daraus, dass die Ausführungen der Schleife die Gültigkeit der Invariante nicht verändern darf. Zusätzlich muss gelten, dass, wenn die Schleifenbedingung nicht erfüllt ist die Bedingung gilt, die nach der Schleife zu gelten hat.

Die Behandlung von einem Rekursionsaufruf erfolgt ebenfalls durch Umformulierung dessen und ähnlich zur Rekursionsbehandlung bei Basic Paths wird die Nachbedingung der Funktion durch die veränderten Parameter im Aufruf angepasst.

▶ **Lemma 19.** *Sei P die Vorbedingung, gs_i ersetzte Variablen von gs, $Stmts_i$ sind Ausführungen, die durch die Berechnung der schwächsten Vorbedingung umgewandelt wurden, $Post'$ ist die Nachbedingung mit eingesetzten gs_i. So gilt:*

$$Impl = assume\ P; gs_i := gs; Stmts_i; assert\ Post'$$

Pro Methode des Boogie-Programms wird eine Verification Condition erstellt und zum Schluss werden alle VC's von einem automatischen Theorembeweiser auf Validität geprüft.

References

1 K. R. Apt and E.-R. Olderog. Programmverifikation. 1994. `http://csd.Informatik.Uni-Oldenburg.DE/~skript/pub/Papers/Errata.ps` Errata-Liste bzw. `http://csd.Informatik.Uni-Oldenburg.DE/~skript/pub/Papers/Errata_long.ps` Errata-Liste mit Tippfehlern.

2 Richard Bornat. Proving pointer programs in hoare logic. In Roland Backhouse and JoséNuno Oliveira, editors, Mathematics of Program Construction, volume 1837 of Lecture Notes in Computer Science, pages 102–126. Springer Berlin Heidelberg, 2000.

3 Aaron R. Bradley and Zohar Manna. The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

4 Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM.

5 C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, October 1969.

6 Ranjit Jhala. Floyd-hoare logic. University Lecture, 2013.

7 K. Rustan M. Leino. Specification and verification of object-oriented software.

8 K. Rustan M. Leino. Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research, April 2004.