Wird erscheinen in: Ada-Deutschland-Tagung 2002 6.- 8. März 2002, Jena

EIN VERGLEICH DER PROGRAMMBEWEISER FPP, NPPV UND SPARK

Carsten Freining, Stefan Kauer*, Jürgen F H Winkler

Friedrich-Schiller-Universität, Institut für Informatik, D-07740 Jena, Germany
http://psc.informatik.uni-jena.de
*: Eurocopter Deutschland GmbH, München

FPP (Frege Program Prover) [KW 99; Win 97], NPPV (New Paltz Program Verifier) [Gum 99] und SPARK (Spade Ada Kernel) [Bar 2000] sind drei Programmbeweiser zum Beweisen der Konsistenz zwischen einer Spezifikation und einem Programm. Bei allen drei Beweisern ist die Spezifikation in Form von Kommentaren in das Programm eingefügt. FPP beweist Ada-Programmfragmente, SPARK Ada-Programme und NPPV Pascal-Programmfragmente.

FPP und NPPV arbeiten vollautomatisch und SPARK arbeitet teilautomatisch und interaktiv. Für den Vergleich, über den hier berichtet wird, stand nur der automatisch arbeitende Teil von SPARK zur Verfügung, der aus dem Examiner und dem Simplifier besteht. Im folgenden wird dieser Teil von SPARK als SPARK-aut bezeichnet.

Beim Beweis der Konsistenz zwischen dem Programm und der Spezifikation sind Theoreme zu beweisen, die aus dem mit den Spezialkommentaren annotierten Programm abgeleitet werden und Verifikationsbedingungen genannt werden. NPPV und SPARK-aut verwenden recht einfache Beweiser in Form von Reduktionsregeln, während FPP einen anspruchsvolleren Beweiser verwendet.

In dem Aufsatz werden zuerst die Grundlagen der zusicherungsorientierten Programmverifikation dargestellt, und zwar bezogen auf die Methodik der schwächsten Vorbedingung, die in allen drei Beweisern verwendet wird. Anschließend werden die Arbeitsweise und Benutzungseigenschaften der drei Beweiser beschrieben.

Der Vergleich erfolgt anhand einer Serie von 26 Programmbeispielen, die zum größten Teil aus der Begleitinformation von NPPV stammen. Die restlichen Beispiele stammen von S. Kauer. Die meisten Beispiele betreffen den Konsistenzbeweis, bei einigen Beispielen stellen die Verifikationsbedingungen das Ergebnis dar, und bei einem Beispiel sind alle Verifikationsbedingungen falsch. Von den 26 Beispielen sind jeweils 25 für FPP und NPPV geeignet. Für SPARK-aut sind alle 26 Beispiele geeignet.

Das Ergebnis dieses Vergleiches ist, daß FPP 19 von 25, NPPV 7 von 25 und SPARK-aut 7 von 26 Beispielen richtig verarbeitet, d.h. die Konsistenz beweist, die richtigen Verifikationsbedingungen berechnet oder die falschen Verifikationsbedingungen als falsch nachweist.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—assertion checkers, *correctness proofs*; F.3.1 [**Logics and Meanings of Programs**] Specifying and Verifying and Reasoning about Programs—*assertions, mechanical verification, pre- and post-conditions*

General Terms: Verification

Additional Key Words and Phrases: program correctness, verification conditions, automatic program prover

1 Einleitung

FPP (Frege Program Prover) [KW 99; Win 97], NPPV (New Paltz Program Verifier) [Gum 99] und SPARK (Spade Ada Kernel) [Bar 2000] sind drei Programmbeweiser, die nach dem Prinzip arbeiten, daß die Korrektheit eines Programmes bezüglich einer Spezifikation überprüft wird, die als spezielle Kommentare in das Programm eingefügt ist. FPP beweist Ada-Programmfragmente, SPARK Ada-Programme und NPPV Pascal-Programmfragmente. Alle drei Beweiser verwenden zum Aufstellen der Verifikationsbedingungen den wp-Kalkül, SPARK allerdings in einer eingeschränkten Form, da es z.B. nicht die einschlägigen Verifikationsschemata für Schleifen benutzt.

FPP und NPPV arbeiten vollautomatisch und SPARK arbeitet teilautomatisch und interaktiv. Für den Vergleich, über den hier berichtet wird, stand nur der automatisch arbeitende Teil von SPARK zur Verfügung, der aus dem Examiner und dem Simplifier besteht. Im folgenden wird dieser Teil von SPARK als SPARK-aut bezeichnet.

Die Verifikationsbedingungen, welche die Konsistenz von Spezifikation und Programm ausdrücken, sind Theoreme, die aus dem mit den Spezialkommentaren annotierten Programm abgeleitet werden und dann bewiesen werden müssen. NPPV und SPARK-aut verwenden recht einfache Beweiser in Form von Reduktionsregel, während FPP einen anspruchsvolleren Beweiser verwendet.

In dem Aufsatz werden zuerst die Grundlagen der zusicherungsorientierten Programmverifikation dargestellt, und zwar bezogen auf die Methodik der schwächsten Vorbedingung, die in allen drei Beweisern verwendet wird. Anschließend werden die Arbeitsweise und Benutzungseigenschaften der drei Beweiser beschrieben.

Der Vergleich erfolgt anhand einer Serie von 26 Programmbeispielen, die zum größten Teil aus der Begleitinformation von NPPV stammen. Die restlichen Beispiele stammen von S. Kauer. Die meisten Beispiele betreffen den Konsistenzbeweis, bei einigen Beispielen stellen die Verifikationsbedingungen das Ergebnis dar, und bei einem Beispiel sind alle Verifikationsbedingungen falsch. Von den 26 Beispielen sind jeweils 25 für FPP und NPPV geeignet. Für SPARK-aut sind alle 26 Beispiele geeignet.

Das Ergebnis dieses Vergleiches ist, daß FPP 19 von 25, NPPV 7 von 25 und SPARK-aut 7 von 26 Beispielen richtig verarbeitet, d.h. die Konsistenz beweist, die richtigen Verifikationsbedingungen berechnet oder die falschen Verifikationsbedingungen als falsch nachweist.

Der Aufsatz ist wie folgt gegliedert. Kap. 2 enthält eine kurze Einführung in die Programmverifikation, speziell die auf dem wp-Kalkül basierende Verifikation. In den Kap. 3, 4 und 5 werden die wesentlichen Eigenschaften von FPP, NPPV und SPARK-aut beschrieben. Die Ergebnisse bezüglich der 26 Beispiele enthält Kap. 6. Ein Ausblick in Kap. 7 schließt den Aufsatz ab.

2 Programmverifikation

Programme sind heute ein wichtiger und häufig entscheidender Teil von technischen Systemen. Dies gilt von kleinen Systemen, wie z.B. Herzschrittmachern, bis hin zu großen Systemen, wie z.B. Flugzeugen. Beides, Herzschrittmacher und Flugzeuge, sind Beispiele von Systemen, welche einen hohen Grad an Zuverlässigkeit und Sicherheit erfordern. Dies bedeutet, daß die Konstruktion in allen ihren Teilen sorgfältig daraufhin überprüft werden muß, ob sich die gestellten Anforderungen erfüllt. Für elektrotechnische Systeme oder Systeme des Maschinenbaus existieren Methoden, mit welchen sich geforderte Eigenschaften exakt überprüfen lassen, z.B. die Schaltungsanalyse [Unb 93] für elektrische Schaltungen oder statische Berechnungen in der Mechanik [GW 93]. Solche exakten und quantitativen Verfahren werden von Ingenieuren der Elektrotechnik und des Maschinenbaus routinemäßig angewendet.

Im Bereich der Programmkonstruktion existieren ebenfalls Methoden zur quantitativen Überprüfung von geforderten Eigenschaften, wie z.B. der Funktionalität oder Wirkung eines Programms. Diese Methoden sind aber noch nicht so ausgereift wie die Methoden in den anderen Ingenieurdisziplinen und werden daher weder ausreichend gelehrt noch in der Berufspraxis routinemäßig angewendet. Während jeder Elektroingenieur die Berechnungen durchführen kann, um eine einfache Schaltung durchzurechnen, kann der durchschnittliche SW-Entwickler in der Regel nicht die Berechnungen durchführen, um den Effekt eines Programms oder Programmfragmentes zu berechnen. Beispiele für solche Berechnungen sind:

- Berechnung der schwächsten Vorbedingung (weakest precondition, wp) für ein Programmfragment PF und eine Nachbedingung Post
- Berechnung der stärksten Nachbedingung für ein Programmfragment PF und eine Vorbedingung Pre
- Überprüfung, ob das Tripel {Pre} PF {Post} konsistent ist
- Berechnung von Gegenbeispielen, falls das Tripel {Pre} PF {Post} nicht konsistent ist

FPP, NPPV und SPARK-aut unterstützen den SW-Entwickler bei solchen Berechnungen.

Programmverifikation im engeren Sinne betrifft die Frage, ob das Tripel

$$\{Pre\} \quad PF \quad \{Post\} \tag{1}$$

konsistent ist. Das kann formal definiert werden durch

$$[\ Pre \Rightarrow wp(PF, Post)\ ] \qquad (2)$$

wobei hier der wp-Kalkül zur Definition der Semantik von Sprachelementen zugrunde gelegt ist. Die eckigen Klammern ("[...]") sind hier eine Abkürzung für die Allquantifizierung über die Programmvariablen ("in all states") [DS 90]. Es gibt andere Verfahren zur Definition der Semantik, z.B. die Kombination von wp und wlp (weakest liberal precondition) [DS 90] oder pre- und post-sets [Win 96]. Da die Werkzeuge, die Gegenstand dieses Aufsatzes sind, alle wp verwenden, erfolgt die weitere Diskussion der Programmverifikation ebenfalls auf der Basis von wp. Es ist hier nicht die Absicht die Theorie der Programmverifikation darzustellen. Dazu gibt es eine Vielzahl von Quellen [Bes 95; DS 90; Fut 89; Gri 83; Kau 99; Win 96].

Im allgemeinen ist (2) ein Theorem. Durch den Beweis von (2) wird dann die Konsistenz von (1) gezeigt. Hierzu einige kleine und einfache Beispiele:

$\{v>0\}\ v:=v+1;\ \{v>4\}$
$\equiv\ [\ v>0 \Rightarrow wp(\text{"}v:=v+1;\text{"}, v>4)\ ]$
$\equiv\ [\ v>0 \Rightarrow v+1 \in type(v) \wedge v \geq 4\ ]$
$\equiv$ False.

$\{v>0 \wedge v+1 \in type(v)\}\ v:=v+1;\ \{v>1\}$
$\equiv\ [\ v>0 \wedge v+1 \in type(v) \Rightarrow wp(\text{"}v:=v+1;\text{"}, v>1)\ ]$
$\equiv\ [\ v>0 \wedge v+1 \in type(v) \Rightarrow v+1 \in type(v) \wedge v+1 > 1\ ]$
$\equiv$ True.

$\{ -127 \leq x \leq 127 \wedge x = x\_i \}$
IF $x < 0$ THEN $x := -x$; END IF;
$\{ 0 \leq x \leq 127 \wedge (x = x\_i \vee x = -x\_i) \}$

$\equiv\ [-127 \leq x \leq 127 \wedge x = x\_i \Rightarrow$
$wp(\text{"IF } x < 0 \text{ THEN } x := -x; \text{ END IF;"}, \quad 0 \leq x \leq 127 \wedge (x = x\_i \vee x = -x\_i))\ ]$

$\equiv\ [-127 \leq x \leq 127 \wedge x = x\_i \Rightarrow$
$x < 0 \wedge 0 \leq -x \leq 127 \wedge (-x = x\_i \vee -x = -x\_i) \vee$
$0 \leq x \wedge 0 \leq x \leq 127 \wedge (x = x\_i \vee x = -x\_i)\ ]$

$\equiv\ [-127 \leq x \leq 127 \wedge x = x\_i \Rightarrow$
$x < 0 \wedge -127 \leq x \leq 0 \wedge (x = x\_i \vee x = -x\_i) \vee$
$0 \leq x \leq 127 \wedge (x = x\_i \vee x = -x\_i)\ ]$

$\equiv\ [-127 \leq x \leq 127 \wedge x = x\_i \Rightarrow$
$(-127 \leq x < 0 \vee 0 \leq x \leq 127) \wedge (x = x\_i \vee x = -x\_i)\ ]$

$\equiv\ [-127 \leq x \leq 127 \wedge x = x\_i \Rightarrow$
$(-127 \leq x \leq 127 \wedge x = x\_i) \vee (-127 \leq x \leq 127 \wedge x = -x\_i)\ ]$

$\equiv$ True.

In den Zusicherungen treten zwei Arten von Variablen auf: (1) Programmvariable die auch im Programm vorkommen und (2) Spezifikationsvariable, die nur in den Zusicherungen auftreten. In [Bac 86: 86] werden die Spezifikationsvariablen auch als "ghost variables" bezeichnet. Spezifikationsvariable werden typischerweise dafür verwendet, um in der Nachbedingung (Post) auf den Initialwert von Programmvariablen Bezug zu nehmen, oder umgekehrt in der Vorbedingung (Pre) auf den Finalwert. In diesem Aufsatz wird die Konvention benutzt, die Bezeichner der Initialwerte durch anhängen von „_i" und die der Finalwerte durch anhängen von „_f" an die Bezeichner der Programmvariablen zu bilden. x_i ist dann der Bezeichner für den Initialwert von x und abc_f der Bezeichner für den Finalwert von abc. Zur besseren Unterscheidung könnte

man für diese Zwecke auch gezielt Großbuchstaben einsetzen oder auch Bezeichner bilden, die in Ada unzulässig sind, wie z.B. "X_i\", "X\i" für den Initialwert und "X_f\", "X\f" für den Finalwert von x.
Dieser Einsatz von Spezifikationsvariablen macht die Zusicherungen etwas komplexer aber auch genauer. In informelleren Darstellung wird manchmal darauf verzichtet [Bac 86: 150 f.; Gri 83: 100]. Häufig kommt es vor, daß Autoren relativ schwache Spezifikationen für die Verifikation wesentlich stärkerer Programme verwenden. Ein typischer Fall dafür ist, daß nicht ausgedrückt wird, daß die Werte bestimmter Programmvariablen durch PF nicht geändert werden sollen. Im allgemeinen ist eine solche Verifikation erfolgreich. Dennoch sind wir der Meinung, daß solch schwache Spezifikationen für die praktische Programmkonstruktion nicht adäquat sind. Bei der SW-Entwicklung wird die Spezifikation in der Regel zuerst entwickelt und dient dann als Vorgabe für die Programmkonstruktion. Wenn die (schwache) Spezifikation ein sehr einfaches Programm zuläßt, warum sollte der Programmkonstrukteur dann ein komplizierteres Programm entwickeln?

Programmverifikation beinhaltet viel Formelmanipulation. Diese per Hand durchzuführen ist ziemlich mühsam und auch fehleranfällig, ähnlich wie dies G: W. Leibniz vor einigen hundert Jahren für die Zahlenrechnung festgestellt hat[1]. Wie bei numerischen Berechnungen kann der Rechner auch Formelmanipulationen schneller und zuverlässiger ausführen als der Mensch. Solche Werkzeuge können dann „Programmverifizierer" oder „Programmbeweiser" genannt werden. Wenn das dritte Beispiel von oben an FPP gegeben wird, ergibt sich folgendes Resultat:

Input to FPP

```
   --!Pre: -127 <= x  and  x <= 127  and  x = x_i;
IF x < 0 THEN x := -x; END IF;
   --!Post: x >= 0  and  x <= 127  and  ( x = x_i  or  x = -x_i );
```

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.08.30, 16:04
The answer to your query is:

--!pre: (-127 <= x AND x <= 127 AND x = x_i)
--> wp:    -127 <= x AND x <= 127 AND x = x_i  OR
-->        -127 <= x AND x <= 127 AND x = -x_i
--> vc:      -127 <= x AND x <= 127 AND x = x_i)
-->      ==>
-->          -127 <= x AND x <= 127 AND x = x_i  OR
-->          -127 <= x AND x <= 127 AND x = -x_i
--> Result: proved
IF x < 0 THEN
   x := -x;
END IF;
--!post: (x >= 0 AND x <= 127 AND (x = x_i OR x = -x_i))
```

Die automatische Verifikation eines solchen annotierten Programms besteht aus zwei Schritten:

- Erzeugung der Verifikationsbedingungen. Das ist ein relativ einfacher mechanischer Vorgang
- *automatischer* Beweis der erzeugten Verifikationsbedingungen.

Wenn diese Beweise durch ein Werkzeug W durchgeführt werden, dann enthält W als eine seiner Komponenten einen automatischen Theorembeweiser ATB. Dieser ATB versucht dann die VC zu beweisen, die

[1] "Denn es ist ausgezeichneter Menschen unwürdig, gleich Sklaven Stunden zu verlieren mit Berechnungen."
G.W. Leibniz

ein Theorem in der Prädikatenlogik erster Stufe ist. Die Verfahren für das Theorembeweisen sind wesentlich komplizierter als die für das Berechnen der VC. Daher ist der ATB der schwierige Teil der Programmverifikation.

Das letzte Beispiel in diesem Abschnitt zeigt die Verifikation einer FOR-Schleife mit FPP [KW 97].

Input to FPP:

```
--!pre : r = 1 AND 0 <= n AND n <= 20 AND n = n_i;
--!post: r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i;
--!inv : r = Factorial(i) AND 0 <= n AND n <= 20 AND n = n_i;
for i in 1 .. n  loop
   r := r * i;
end loop;
```

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.08.30, 16:24
The answer to your query is:

--!pre       : r = 1 AND 0 <= n AND n <= 20 AND n = n_i
--!post      : r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
--!inv       : r = Factorial(i) AND 0 <= n AND n <= 20 AND n = n_i
-->functionality ------------------------------------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :     1 <= n AND r = 1 AND 0 <= n AND n <= 20 AND n = n_i
-->            ==> r = 1 AND 0 <= n AND n <= 20 AND n = n_i
-->Result    : proved
-->induction :     r = Factorial(i-1) AND 1 <= n AND n <= 20 AND n = n_i
-->            ==> i*r = Factorial(i) AND 0 <= n AND n <= 20 AND n = n_i
-->Result    : proved
-->final     :     r = Factorial(n) AND 1 <= n AND n <= 20 AND n = n_i
-->            ==> r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
-->Result    : proved
-->null loop :     n = 0 AND r = 1 AND n = n_i
-->            ==> r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
-->Result    : proved
FOR i IN 1 .. n  LOOP
   r := r * i;
END LOOP;
```

Das Beispiel zeigt die Verifikation einer realistischen Berechnung der Fakultätsfunktion, d.h. einer Berechnung, bei welcher der korrekte Wert unter Berücksichtigung des endlichen Wertebereiches für das Resultat berechnet wird. Die meisten Fakultätsprogramme in Informatikbüchern berechnen ja für fast alle zulässigen Argumentwerte ein falsches Ergebnis oder liefern eine Ausnahme [KW 97; NW 89].

3 FPP: Der Frege Program Prover

Der Frege Program Prover (FPP; http://psc.informatik.uni-jena.de/FPP/FPP-main.htm) ist ein Programmanalysewerkzeug, welches Berechnungen im Rahmen der Programmverifikation durchführen kann [KW 99; Win 97]. FPP basiert auf dem wp-Kalkül und einer kleinen Teilmenge von Ada (die Typen integer und boolean; die Zuweisung, IF, CASE, FOR und WHILE). Zusicherungen werden als spezielle Kommentare geschrieben (--! ...) und sind inhaltlich gesehen Boole'sche Ausdrücke. Zusätzlich zu Ada sind Quantoren und die Implikation zugelassen. Die Syntax der akzeptierten Sprache findet sich unter

http://psc.informatik.uni-jena.de/FPP/fpp-synt.htm. Für FOR-Schleifen muß eine Invariante angegeben werden und für WHILE-Schleifen eine Invariante und eine Terminierungsfunktion. Man kann bei WHILE-Schleifen daher sowohl die totale wie die partielle Korrektheit berechnen.

Da wir selbst nicht auf dem Gebiet des automatischen Theorembeweisens arbeiten, benutzt FPP den Theorembeweiser Analytica von Clarke und Zhao [CZ 92]. Dieser Beweiser wurde von S. Kauer modifiziert und erweitert. Da Theorembeweisen schwierig ist, kann der Benutzer von FPP eine Situation erleben, in welcher FPP eine bestimmte VC nicht beweisen kann, Speichermangel meldet oder sehr lange läuft, ohne eine Antwort zu geben. Dieses Verhalten unterscheidet sich prinzipiell nicht von einem menschlichen Beweiser, bei dem es auch vorkommen kann, daß er ein Theorem nicht beweisen kann. Bezüglich der Beweiskraft spiegelt FPP die Beweiskraft des eingebauten ATB wider.

Ein Beispiel hierfür ist Beispiel 8 (fastmultty). Der Beweiser formt die VC in DNF (disjunktive Normalform) um und spaltet diese transformierte VC in eine äquivalente Konjunktion von Klauseln auf. Eine Klausel hat hier dieselbe Form wie eine Klausel in Prolog: eine Implikation bestehend aus einer Prämisse bestehend aus einer Konjunktion von Atomen und einer Konklusion bestehend aus einem Atom. Im ungünstigsten Falle ist die Transformation in DNF exponentiell sowohl in der Zeit als auch im Raum (Speicherplatz) [Sch 95: 30].

Eine weitere Beschränkung der Beweiskraft liegt im Entwicklungsstand des wp-Kalküls begründet. Für Schleifen wird ein Korrektheitsbeweis üblicherweise mit Hilfe einer Invarianten und bei einer WHILE-Schleife zusätzlich mit einer Terminierungsfunktion geführt. Invariante und Terminierungsfunktion müssen dabei in der Regel vom Benutzer aufgestellt und angegeben werden. Wenn die Invariante zu scharf ist, kann es vorkommen, daß [Pre $\Rightarrow$ wp(PF, Post)] nicht bewiesen werden kann, auch wenn {Pre} PF {Post} konsistent ist. Ein Programmbeweiser kann daher auch durch die Entwicklung besserer Beweisregeln verbessert werden. Dies wurde von S. Kauer in [Kau 99] gemacht, wo zwei neue Beweismethoden für bestimmte Schleifen entwickelt wurden. Die erste berechnet automatisch eine Invariante für bestimmte FOR-Schleifen und die zweite berechnet automatisch die schwächste Vorbedingung für bestimmte WHILE-Schleife allein aus der Schleife und der Nachbedingung.

FPP ist auch in Form einer netzfähigen Anwendung implementiert und kann daher interaktiv über das Netz benutzt werden.

FPP hat zwei Hauptfunktionalitäten:

a) *Berechnen der schwächsten Vorbedingung*: wp(PF, Post). Berechne die schwächste Vorbedingung für ein gegebenes Programm (-fragment) PF und eine gegebene Nachbedingung Post

   Beispiel 3.1: wp(”v:=v+1;“, v>4)
   $\equiv$ v+1 $\in$ type(v) $\wedge$ v+1 > 4
   $\equiv$ v+1 $\in$ type(v) $\wedge$ v $\geq$ 4.

   type(v) ist dabei die Wertemenge des Typs der Variablen v.

b) *Überprüfe die Korrektheit eines Programm* (-fragments) PF bezüglich einer Spezifikation (Pre, Post), d.h. versuche zu beweisen, daß das Hoare-Tripel {Pre} PF {Post} konsistent ist. Im Rahmen des wp-Kalküls ist das äquivalent zu [Pre $\Rightarrow$ wp(PF, Post)]. Daran sieht man, daß das Berechnen der schwächsten Vorbedingung eine Teilaufgabe des Korrektheitsbeweises ist.

   Beispiel 3.2: {v>0} v:=v+1; {v>4}
   $\equiv$ [v>0 $\Rightarrow$ wp(”v:=v+1;“, v>4)]
   $\equiv$ [v>0 $\Rightarrow$ v+1 $\in$ type(v) $\wedge$ v $\geq$ 4]
   $\equiv$ False.

Die Anwendung von FPP auf diese beiden Beispiele ergibt:

Beispiel 3.1 *Input to FPP:*

```
v := v+1;
   --!Post: v > 4 and -100 <= v  and v <= 100;
```

Die aktuelle Version von FPP erwartet, daß die wertmäßige Beschränkung einer Variablen explizit in den Zusicherungen ausgedrückt wird. Die Wertemenge {-100 .. +100} ist daher in der Nachbedingung durch `-100 <= v  and v <= 100` ausgedrückt.

Output from FPP:

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1998.03.06, 14:48

The answer to your query is:

--> wp    : (1+v >= 5 AND -100 <= 1+v AND 1+v <= 100)
v := v + 1;
--!post   : (v >= 5 AND -100 <= v AND v <= 100)
```

Zeilen, die mit „-->“ beginnen, enthalten von FPP berechnete Resultate. Diese werden jeweils an die entsprechende Stelle in das Programm eingefügt. Wegen der Variabilität der Programmvariablen ist es wichtig und entscheidend, daß eindeutig klar ist, an welcher Programmstelle eine Aussage, welche Programmvariablen enthält, gilt. In der Ausgabe erkennt man die von FPP berechnete schwächste Vorbedingung wp unmittelbar vor der Zuweisung, für welche wp berechnet worden ist.

Beispiel 3.2 *Input to FPP:*

```
   --!Pre: v > 0;
v := v+1;
   --!Post: v > 4 and -100 <= v  and v <= 100;
```

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1998.03.06, 15:00
The answer to your query is:

--!pre : (v >= 1)
--> wp : (1 + v >= 5 AND -100 <= 1 + v AND 1 + v <= 100)
--> vc : (v >= 1 ==> 1+v >= 5 AND -100 <= 1+v AND 1+v <= 100)
--> Result: not proved
--> Incorrectness condition: (3 >= v AND v >= 1)
v := v + 1;
--!post: (v >= 5 AND -100 <= v AND v <= 100)
```

In diesem Beispiel sind das Programmfragment PF und die durch Pre und Post gegebene Spezifikation *nicht konsistent*. FPP gibt die Antwort „not proved“ aus. Zusätzlich wird eine „incorrectness condition“ angegeben, welche die Zustände z charakterisiert, in welchen zwar Pre gilt, für welche aber gilt: PF(z) $\not\subseteq$ Post.

In solch einem Fall ist es zusätzlich nützlich, wenn ein konkretes Gegenbeispiel angegeben wird. Eine Methode zu Berechnung von Gegenbeispielen in bestimmten Fällen wurde in [Kau 99] entwickelt, sie ist aber noch nicht in FPP eingebaut. Im vorliegenden - sehr einfachen - Beispiel sieht man leicht, daß v=1 ein Ge-

genbeispiel ist. In komplizierteren Fällen ist es natürlich nicht so leicht, aus der incorrectness condition ein konkretes Gegenbeispiel abzuleiten.

Das folgende Beispiel zeigt, wie Schleifen annotiert werden, insbesondere wie die Invariante und die Terminierungsfunktion angegeben werden. Das Programmfragment berechnet den größten gemeinsamen Teiler zweier Zahlen mittels des Euklidischen Algorithmus.

Beispiel 3.3 *Input to FPP:*

```
   --!pre: i>0 and j>0 and i=i_i and j=j_i;
   --!post: i=j and i=GGT(i_i,j_i);
   --!inv: i>0 and j>0 and GGT(i,j) = GGT(i_i,j_i);
   --!term: i+j;
WHILE i /= j LOOP
   IF i>j
   THEN i := i-j;
   ELSE j := j-i;
   END IF;
END LOOP;
```

Für Schleifen werden alle Annotationen vor dem Anfang der Schleife angegeben. In diesem Beispiel sind dies Vorbedingung, Nachbedingung, Invariante und Terminierungsfunktion. FPP versucht dann, die Gültigkeit von

$$[\ pre \Rightarrow inv\ ] \wedge [\ cond \wedge inv \Rightarrow wp(body, inv)\ ] \wedge [\ \neg cond \wedge inv \Rightarrow post\ ] \wedge$$
$$[\ cond \wedge inv \Rightarrow term > 0\ ] \wedge [\ cond \wedge inv \Rightarrow wp(\text{"T:=term; body"}, term < T)\ ]$$

zu zeigen, welches die klassische Verifikationsbedingung für WHILE-Schleifen ist.

Output from FPP (slightly edited):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.01, 16:39
The answer to your query is:

--!pre       : (i >= 1 AND j >= 1 AND i = i_i AND j = j_i)
--!post      : (i = j AND i = GGT(i_i,j_i))
--!inv       : (i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
--!term      : (j + i)
-->functionality ---------------------------
-->initial   :     (i >= 1 AND j >= 1 AND i = i_i AND j = j_i)
-->            ==> (i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->Result    : proved
-->induction :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==>          (i > j) AND (j >= 1) AND (GGT(i,j) = GGT(i_i,j_i))
-->                 OR      (i < j) AND (i >= 1) AND (GGT(i,j) = GGT(i_i,j_i))
-->Result    : proved
-->final     :     (i = j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (i = j AND i = GGT(i_i,j_i))
-->Result    : proved
-->termination ---------------------------
-->initial   :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (j + i >= 1)
-->Result    : proved
-->induction :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (i >= 1 + j AND j + i >= 1 + i OR i < 1 + j AND j + i >= 1+j)
```

```
-->Result    : proved
WHILE i /= j LOOP
   IF i > j THEN
      i := i - j;
   ELSE
      j := j - i;
   END IF;
END LOOP;
```

Alle fünf Konjunkte der Verifikationsbedingung wurden bewiesen, und daher sind die Schleife und die Spezifikation konsistent. Wegen der Konjunktivität von wp können die Konjunkte separat bewiesen werden und werden auch in der Ausgabe von FPP separat dargestellt.

4 NPPV: Der New Paltz Program Verifier

NPPV [GS 2002; Gum 2001] ist ein automatischer Programmbeweiser für eine Teilmenge von Pascal. Die Eingabe besteht wie bei FPP aus einem annotierten Programm. NPPV berechnet die Verifikationsbedingungen und versucht sie dann zu beweisen. Die VC werden entsprechend dem wp-Kalkül berechnet (für Zuweisung, IF und Sequenz). Für FOR- und WHILE-Schleifen werden die aus der Literatur bekannten Induktionsschemata verwendet [Gri 83; Hoa 72]. Der endliche Wertebereich der Datentypen wird nicht berücksichtigt. Die Ausgabe besteht aus den erzeugten VCen, wobei diese nicht direkt in Bezug zum Programmtext gesetzt werden. Wenn eine VC bewiesen werden kann, dann wird „Proof succeeded“ vermerkt im anderen Falle „Remains to prove“. Überraschend war, daß NPPV die meisten Beispiele, die mit dem System mitgeliefert wurden, nicht beweisen konnte (s.a. Tabelle 6.2). Die Ausgabe, die standardmäßig am Bildschirm erscheint, kann auch in einer Datei „session.log“ gespeichert werden. Für die praktische Benutzung ist dieser feste Name unpraktisch, denn wenn man mehrere Beispiele bearbeiten läßt, dann möchte man in der Regel die Ergebnisse in Dateien haben, deren Name zur Eingabedatei in einem erkennbaren Bezug steht.

Der unterstützte Sprachumfang enthält die Typen integer mit dem Bereich -32768..32767 , Boolean und Arrays mit diesen als schlußendlichen Elementtypen. Die Zusicherungen sind Boolesche Ausdrücke im Sinne von Pascal, so daß weder Quantoren noch die Implikation zugelassen sind. Für Schleifen ist eine Invariante anzugeben. In WHILE-Schleifen kann auch eine Terminierungsfunktion angegeben werden, und zwar in eckigen Klammern unmittelbar nach dem Schlüsselwort „DO“. Das bedeutet, daß in diesem Falle das annotierte Programm kein legales Pascal-Programm ist. Wenn keine Terminierungsfunktion angegeben ist, dann werden auch keine VC für die Terminierung berechnet. In der Literatur wird das auch als partielle Korrektheit bezeichnet.

Bezeichner, die mit einem Kleinbuchstaben beginnen, sind Variable. Bezeichner, die mit einem Großbuchstaben beginnen, sind Konstanten. Das bedeutet, daß die Zuweisung n := 5; zulässig ist, N := 5; hingegen nicht zulässig ist. Dies wird von NPPV überprüft. Dadurch weicht der von NPPV akzeptierte Sprachumfang an einer weiteren Stelle von Pascal ab. Es gibt keine explizite Unterscheidung zwischen Programm- und Spezifikationsvariablen, die eben erwähnten Konstanten können aber als Spezifikationsvariablen zur Bezeichnung von Initial- bzw. Finalwerten von Programmvariablen verwendet werden. Dies wird in den Beispielen so praktiziert.

Die Ausgabe der VCen erfolgt separat vom Programm ohne eine Bezugnahme auf den Programmtext. Bei etwas größeren Beispielen ist dann die Zuordnung der VCen zum Quellprogramm nicht leicht ersichtlich. Für den praktischen Einsatz ist dies eine ungünstige Eigenschaft.

Beispiel:

Input to NPPV:

```
{v > 0}
BEGIN
   v := v+1
END
{v > 4}
```

Output from NPPV:

```
Generating verification conditions...
O.K.
====================================
=== Verification Condition No.: 1 ===
v>0
          ==>
                 v+1>4
--------- Remains to prove  ---------
0<v
          ==>
                 4<v+1
====================================
```

NPPV kann von http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html erhalten werden. Es läuft auf DOS, in einem DOS-Fenster von Windows NT oder auf OS/2.

5 SPARK

Mit SPARK werden hier auch die mit SPARK verbundenen Werkzeuge bezeichnet. SPARK 95 ist die Abkürzung für „SPADE Ada95 Kernel“ und SPADE steht für „Southampton Program Analysis Development Environment“. SPARK 95 ein Subset von Ada95, welches zur Programmierung von sicherheitskritischer SW gedacht ist, die mit vertretbarem Aufwand verifiziert werden soll. Zur Unterstützung der Verifikation wurden mehrere Werkzeuge entwickelt: der Examiner, der automatische Simplifier und der interaktive Proof Checker. Für die vorliegende Untersuchung standen nur der Examiner und der automatische Simplifier zur Verfügung, so wie sie dem Buch [Bar 2000] beigefügt sind. Diese werden zusammen als SPARK-aut bezeichnet, da sie den automatisch arbeitenden Teil der SPARK-Werkzeuge darstellen.

SPARK-aut unterscheidet sich von FPP und NPPV in zweierlei Hinsicht

- es werden nur vollständige Programme verarbeitet
- außer der Verifikation im engeren Sinne, wie sie bisher dargestellt wurde, werden noch eine Reihe anderer Analysen durchgeführt, wie z.B. data flow analysis, information flow analysis. Darauf wird in diesem Aufsatz nicht weiter eingegangen.

Die Verifikation kann ähnlich wie bei FPP und NPPV durchgeführt werden: die Spezifikation wird in Form von Annotationen, die auch hier die Form von speziellen Kommentaren haben (--#), in das Program eingefügt. Der Examiner berechnet dann die Verifikationsbedingungen. Die VCen werden auch auf der Basis des wp-Kalküls berechnet, allerdings wird dies nicht durchweg explizit so dargestellt, sondern ist eher aus einzelnen Bemerkungen zu erschließen. Die Darstellung in [AH 2000; Bar 2000] erfolgt hauptsächlich anhand von Beispielen. Ein wesentlicher Unterschied besteht bei der Verifikation von Schleifen: der Examiner kennt die üblichen Verifikationsschemata für FOR- und WHILE-Schleifen nicht. Daher ist die Verifikation von Schleifen umständlicher als in FPP und NPPV.

Für die Durchführung des Vergleichs wurden die Programmfragmente zu vollständigen Hauptprogrammen (Ada-Prozedur) erweitert.

Die Verifikation mit SPARK-aut erfolgt in zwei Schritten: der Examiner analysiert das Programm und berechnet die VCen. Die Ergebnisse werden in bis zu 7 verschiedene Dateien ausgegeben, von welchen hier die Dateien <prog>.lst (Listing) und <prog>.vcg (VCen) wichtig sind.

Beispiel 5.1 *Input to SPARK Examiner:*

```
-- pap02vc   Examiner: verification = vc

 --# main_program;
```

```
procedure pap02vc (v: in out integer)
   --# derives v from v;
is
begin
      --# assert v > 0;
   v := v+1;
      --# assert v > 4 and -100 <= v  and v <= 100;
end pap02vc;
```

Der für den Vergleich mit FPP und NPPV relevante Teil des Programms ist oben eingerahmt.

Output from SPARK Examiner (pap02vc.lst):

```
          ******************************************************
                          Listing of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 5.01 / 08.00
                          Demonstration Version
          ******************************************************
                       DATE : 01-JAN-2002 11:43:10.80
Line
   1    -- pap02vc   Examiner: verification = vc
   2
   3     --# main_program;
   4  procedure pap02vc (v: in out integer)
   5     --# derives v from v;
   6  is
   7  begin
   8        --# assert v > 0;
   9     v := v+1;
  10        --# assert v > 4 and -100 <= v  and v <= 100;
  11  end pap02vc;

+++         Flow analysis of subprogram pap02vc performed:
            no errors found.
--End of file-------------------------------------------------
```

Diese Datei ist wichtig, da hierin die Zeilennummern festgelegt sind, die in <prog>.vcg und dann auch in <prog>.siv verwendet werden.

Output from SPARK Examiner (pap02vc.vcg):

```
          ******************************************************
                       Semantic Analysis of SPARK Text
       SPARK95 Examiner with VC and RTC Generator Release 5.01 / 08.00
                          Demonstration Version
          ******************************************************
                       DATE : 01-JAN-2002 11:43:10.61

                             procedure pap02vc

For path(s) from start to assertion of line 8:
procedure_pap02vc_1.
H1:    true .
H2:    v >= integer__first .
H3:    v <= integer__last .
        ->
```

```
C1:    v > 0 .
```

```
For path(s) from assertion of line 8 to assertion of line 10:
procedure_pap02vc_2.
H1:    v > 0 .
        ->
C1:    v + 1 > 4 .
C2:     - 100 <= v + 1 .
C3:    v + 1 <= 100 .
```

```
For path(s) from assertion of line 10 to finish:

procedure_pap02vc_3.
*** true .          /* trivially true VC removed by Examiner */
```

Im zweiten Schritt wird die Datei pap02vc.vcg vom automatischen Simplifier bearbeitet. Beim Namen dieses Werkzeuges bedenke man, daß Beweisen in einer formalisierten Theorie Reduktion zu True bedeutet. Diese Reduktion kann man auch als Vereinfachung sehen.

Output from SPARK Simplifier (pap02vc.siv):

```
      ******************************************************
                    Semantic Analysis of SPARK Text
      SPARK95 Examiner with VC and RTC Generator Release 5.01 / 08.00
                         Demonstration Version
      ******************************************************

      CREATED 01-JAN-2002, 11:43:10  SIMPLIFIED 01-JAN-2002, 11:43:54
               (Simplified by SPADE Simplifier, Version 1.4)

                              procedure pap02vc

For path(s) from start to assertion of line 8:

procedure_pap02vc_1.
H1:    v >= integer__first .
H2:    v <= integer__last .
       ->
C1:    v > 0 .
```

```
For path(s) from assertion of line 8 to assertion of line 10:

procedure_pap02vc_2.
H1:    v > 0 .
       ->
C1:    v > 3 .
C2:    v <= 99 .
```

```
For path(s) from assertion of line 10 to finish:

procedure_pap02vc_3.
*** true .          /* all conclusions proved */
```

Wie man sieht, wurde die VC vereinfacht, da sie aber nicht wahr ist, kann sie auch nicht bewiesen werden.

6 Vergleich von FPP, NPPV und SPARK-aut

FPP, NPPV und SPARK-aut sind ähnliche Systeme. Sie basieren auf einer Teilmenge einer imperativen Programmiersprache, welche in einer Erweiterung die Formulierung von Zusicherungen erlaubt, mittels derer eine Programmspezifikation formuliert werden kann. Alle drei Werkzeuge bestehen aus einem Generator, der aus einem annotierten Programm Verifikationsbedingungen erzeugt. Anschließend versucht ein automatischer Beweiser, die VCen zu beweisen.

Im Detail und bei der praktischen Benutzung unterscheiden sich die drei Werkzeuge mehr oder minder stark. Diese Unterschiede sind in Tabelle 6.1 dargestellt.

Property	FPP	NPPV	SPARK-aut
programming language	subset of Ada	subset of Pascal	subset of Ada
assertion language	subset of Ada expressions extended with *quantifiers*, *implication* and the additional functions abs, min, max, ggt, sum, factorial, fib	subset of Pascal expressions, enclosed in { }; true is expressed by {}	subset of Ada expressions extended with *quantifiers*, *implication*, equivalence, proof variables and functions, and update of structured objects
form of assertions	special comments: --!	{}comments and [...]	special comments: --#
multiline assertions	supported	supported	supported
supported types	integer and Boolean	integer, array with integer index type and integer component type	all, except tagged, access, task, and exception
supported statements	NULL, assignment, IF, CASE, FOR and WHILE loop	assignment, IF, FOR- and WHILE loop	all, except goto and tasking
proof of loops	precondition, postcondition, invariant and for WHILE loops a termination function has to be supplied	only invariant required, termination function for WHILE loops optional	invariant can be inserted in body, termination has to be expressed by assertions
output	in a file that has the same name as the input file, but a different extension; output contains the statements, the VCs and the result together	optional in a file: session.log; output contains only verification conditions and results	in up to 8 different files
usage	local or via WWW	local	local
pretty printing	supported	not supported	not supported
simplification of expr.	performed to a certain extent	not performed	performed to a certain extent
computing wp	possible	not possible	not possible
theorem proving	possible e.g. with null statement	possible e.g. with x := x	possible e.g. with null statement
implementation language	Ada, C and Mathematica	Visual Prolog	Prolog (?)
proving power	higher than NPPV and SPARK-aut	only trivial	rather limited
automatic theorem prover	mexana, an extension of Analytica	simple rewrite system	automatic Simplifier

Tabelle 6.1. Merkmale von FPP, NPPV und SPARK-aut

Tabelle 6.2 enthält die Ergebnisse, die bei der Verarbeitung von 26 Beispielen mit den drei Werkzeugen erhalten wurden. Die Beispiele stammen aus zwei Quellen. Der Hinweis „Gumm“ in der Spalte „Source“ bedeutet, daß das Beispiel mit NPPV mitgeliefert wurde und „Kauer“ bedeutet, daß das Beispiel von S. Kauer stammt. Die vollständigen Protokolle aller Beispiele finden sich im WWW unter http://psc.informatik.uni-jena.de/Themen/Results-spark.pdf und /Results-fpp-nppv.pdf.

In den Spalten „FPP“, NPPV“ und „SPARK-aut“ finden sich die Ergebnisse für die Verarbeitung der Beispiele durch die drei Programmbeweiser. Einige Beispiele sind in der Unterspalte „proved“ mit „n.a.“ (nicht anwendbar) markiert, wenn das betreffende Werkzeug nicht auf das Beispiel anwendbar ist. Beispiel 2 („array“) ist für FPP nicht geeignet, da FPP Arrays nicht unterstützt. Beispiel 16 („linsearch“) ist für NPPV nicht geeignet, da in NPPV Quantoren nicht verwendet werden können. Die geeigneten Beispiele sind von dreierlei Art:

- Programm enthält genügend Information, um seine Korrektheit zu beweisen (z.B. Bsp 1, 4)
- Programm enthält genügend Information, um seine Inkorrektheit zu beweisen (Bsp. 23)
- Programm enthält nicht genügend Information, um seine Korrektheit oder Inkorrektheit zu beweisen. Hier wird der Programmbeweiser dann nur verwendet, um die VCen aufzustellen (Bsp. 2, 3, 15, 18, 25)

Die Angaben in den Spalten „proved“, „#VC“ und „#OK“ haben folgende Bedeutung:

„proved“

- PROV: Beispiel als korrekt oder nicht korrekt bewiesen
- NOP: Beispiel nicht als korrekt oder nicht korrekt bewiesen
- SOLV: Beispiel nicht beweisbar und VCen richtig berechnet
- NOS: Beispiel nicht beweisbar und nicht alle VCen richtig berechnet
- n.a.: Beispiel nicht anwendbar

„#VC“: Anzahl der berechneten VCen
Bei SPARK-aut werden hier nur die VCen gezählt, die auf den Teil des Programmes entfallen, der dem ursprünglichen Programmfragment entspricht.

„#OK“: falls „proved“ = PROV / NOP dann Anzahl der bewiesenen bzw. widerlegten VCen
falls „proved“ = SOLV / NOS dann Anzahl der korrekten VCen

Die letzte Zeile von Tabelle 6.2 enthält eine quantitative Zusammenfassung der Ergebnisse für die 26 Beispiele. Etwas überraschend dabei ist, daß NPPV die Beispiele, die mit dem Programm mitgeliefert werden, nicht beweisen kann. Insgesamt zeigt sich, daß FPP wesentlich mehr Beispiele beweisen kann als NPPV oder SPARK-aut. Der wesentliche Grund dafür scheint der eingebaute ATP zu sein, denn die VCen werden von allen Beweisern jeweils generiert, was man auch daran sieht, daß NOS selten vorkommt. Bis auf kleine Unterschiede verhalten sich NPPV und SPARK-aut im wesentlichen gleich. Wenn NPPV oder SPARK-aut ein Programm nicht beweisen können, dann ist die Situation meist so, daß einige der VCen zu schwierig sind, denn die Anzahl der VCen mit OK ist bei FPP nur etwa 25% höher als bei NPPV und SPARK-aut.

Bei Beispielen mit WHILE-Schleife, in welchen keine Terminierungsfunktion angegeben ist, wird in FPP die Konstante “1” als Terminierungsfunktion verwendet, die natürlich nicht erlaubt, die Terminierung zu beweisen. Die VCen, welche die Terminierung betreffen, werden in diesen Fällen nicht mitgezählt.

Wird erscheinen in: Ada-Deutschland-Tagung 2002 6.- 8. März 2002, Jena

No.	Example	Remark	Source	FPP			NPPV			SPARK-aut		
				proved	#VC	#OK	proved	#VC	#OK	proved	#VC	#OK
1	abs	no abs function in NPPV	Kauer	PROV	1	1	NOP	2	1	NOP	2	0
2	array	no arrays in FPP	Gumm	n.a.			SOLV	1	1	NOS	1	0
3	assrek	no termination function	Gumm	NOS	0	0	SOLV	4	4	SOLV	4	4
4	factfor		Gumm	PROV	5	5	NOP	5	2	NOP	4	1
5	factforty		Gumm / Kauer	PROV	5	5	NOP	5	1	NOP	4	1
6	fastmul	no termination function	Gumm	NOP	6	5	NOP	6	4	NOP	9	7
7	fastmult		Gumm	NOP	10	7	NOP	14	7	NOP	9	7
8	fastmultty	too many clauses (FPP)	Gumm / Kauer	NOP	0	0	NOP	14	7	NOP	10	7
9	fibo	no termination function	Gumm	PROV	3	3	NOP	4	2	NOP	4	2
10	fibot		Gumm	PROV	6	6	NOP	6	3	NOP	5	3
11	fibotty		Gumm / Kauer	NOP	6	5	NOP	6	3	NOP	5	2
12	gauss	no termination function	Gumm	PROV	4	4	NOP	4	3	NOP	4	3
13	gausst		Gumm	PROV	6	6	NOP	6	5	NOP	5	4
14	gausstty		Gumm / Kauer	PROV	6	6	NOP	6	4	NOP	5	3
15	linrek	no termination function	Gumm	SOLV	7	7	SOLV	7	7	SOLV	8	8
16	linsearch	no quantifiers in NPPV	Kauer	PROV	5	5	n.a.			NOP	4	1
17	nested_for		Kauer	PROV	9	9	NOP	8	5	NOP	9	7
18	proof		Gumm	SOLV	4	4	SOLV	4	4	SOLV	4	4
19	quad		Kauer	PROV	5	5	NOP	5	4	NOP	5	4
20	root		Kauer	PROV	5	5	NOP	5	4	NOP	4	3
21	swap1		Gumm	PROV	1	1	PROV	1	1	PROV	1	1
22	swap2	infinite ranges	Gumm	PROV	1	1	PROV	1	1	PROV	1	1

23	swap2ty	fin ranges, prog incorrect	Gumm	NOP	3	0	NOP	3	0	NOP	3	0
24	swap2ty2		Gumm / Kauer	PROV	3	3	NOP	3	1	PROV	3	3
25	swap3		Gumm	SOLV	1	1	SOLV	1	1	SOLV	1	1
26	cube		[GH 99]	PROV	5	5	NOP	5	3	NOP	5	3
	Summary			19 (25)	107	99	7 (25)	126	78	7 (26)	119	80

Tabelle 6.2. Ergebnisse für die 26 Beispiele

7 Zusammenfassung und Ausblick

Im vorliegenden Aufsatz werden drei Programmbeweiser anhand ihrer Merkmale und der Ergebnisse von 26 kleinen Programmbeispielen verglichen. Neben Unterschieden in der Handhabung wurden vor allem große Unterschiede in der Beweisfähigkeit beobachtet. FPP bearbeitet 19 von 25 Beispielen korrekt, während NPPV und SPARK-aut 7 von 25 bzw. 26 Beispielen korrekt bearbeiten. Die meisten der korrekt bearbeiteten Fälle bei NPPV und SPARK-aut betreffen allerdings das Erzeugen der VCen, was in der Regel einfacher ist als das Beweisen oder Widerlegen von Theoremen.

Aber auch FPP kann nicht alle Beispiele korrekt verarbeiten und unterstützt nur eine kleine Teilmenge von Ada. Verbesserungen sind in folgenden Punkten möglich:

a) Unterstützung weitere Sprachelemente von Ada. Dies ist in FPP-2 geplant (Array, Record, Prozedur und automatische Überprüfung von Wertebereichen).

b) Benutzung neuer Methoden zur Berechnung von VCen, wie z.B. die Berechnung von Invarianten für FOR-Schleifen [KW 2000], die direkte VC für bestimmte WHILE-Schleifen [Kau 99] oder eine schwächere VC für FOR-Schleifen [Win 98].

c) Benutzung eines stärkeren Theorembeweisers.

d) Einführung eines interaktiven Modus, um in schwierigen Fällen dem Beweiser einen Hinweis geben zu können.

Literatur

AH 2000 Hammond, Jonathan; Amey, Peter: Generation of VCs for SPARK Programs. V 4.0, Praxis Critical Systems, Bath, 2000.Jul.10

Bac 86 Backhouse, Roland C.: Program Construction and Verification. Prentice Hall, New York etc., 1986. 0-13-729146-9

Bar 2000 Barnes, John: High Integrity Ada - The SPARK Approach -. Addison-Wesley, Harlow etc., 2000. 0-201-17517-7

Bes 95 Best, Eike: Semantik. Friedr. Vieweg & Sohn, Braunschweig, 1995. 3-528-05431-X.

CZ 92 Clarke, E.; Zhao, X.: Analytica - A Theorem Prover in Mathematica. International Conference on Artificial Intelligence and Symbolic Mathematical Computation. AISM C-3, Steyr 1996, 21 - 37

DS 90 Dijkstra, Edsger W.; Scholten, Carel S.: Predicate Calculus and Program Semantics. Springer, New York etc., 1990. 0-387-96957-8

Fut 89 Futschek, Gerald: Programmentwicklung und Verifikation. Springer, New York etc., 1989. 0-387-81867-7

GH 99 Hehner, Eric C. R.; Gravell, Andrew M.: Refinement Semantics and Loop Rules. In: FM'99, Vol. II, 1497..1510. LNCS 1709, Springer, Berlin etc., 1999. 3-540-66588-9

Gri 83 Gries, David: The Science of Programming. Springer, New York. etc., 1983. 0-387-90641-X

GS 2002 Gumm, Heinz-Peter; Sommer, Manfred: Einführung in die Informatik. R. Oldenbourg Verlag. München 2002. 3-486-25635-1

Gum 99 Gumm, Heinz-Peter: Generating algebraic laws from Imperative Programs. Theoretical Computer Science 217(2): 385-405 (1999)

Gum 2001 http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html, 2001.Sep.04

GW 93 Göldner, Hans; Witt, Dieter: Lehr- und Übungsbuch Technische Mechanik. Fachbuchverlag Leipzig-Köln, 1993. 0-343-00803-6

Hoa 72 Hoare, C.A.R.: A Note on the FOR Statement. BIT 12,3 (1972) 334..341

Kau 99 Kauer, Stefan: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich-Schiller-Universität Jena, 1999

KW 97 Winkler, Jürgen F. H.; Kauer, Stefan: Proving Assertions is also Useful. SIGPLAN Notices 32,3 (1997) 38..41

KW 99 Kauer, Stefan; Winkler, Jürgen F. H.: FPP: An Automatic Program Prover for Ada Statements. GI FG 2.1.5. Ada: Workshop "Objektorientierung und sichere Software mit Ada". Karlsruhe 1999.Apr.21-22

KW 2000 Kauer, Stefan; Winkler, Jürgen F. H.: Automatic Generation of Invariants for FOR-Loops Based on an Improved Proof Rule. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 2000 / 26, 2000.Sep.28

NW 89 Winkler, J.F.H.; Nievergelt, J.: Wie soll die Fakultätsfunktion programmiert werden? Informatik-Spektrum 12,4 (1989) 220..221.

Sch95 Schöning, Uwe: Logik für Informatiker. Spektrum Akademischer Verlag. Heidelberg 1995. 3-86025-684-x

Unb 93 Unbehauen, Rolf: Netzwerk- und Filtersynthese. 4. Aufl. R. Oldenbourg München, Wien, 1993. 3-486-22158-2

Win 96 Winkler, Jürgen F. H.: Some Properties of the Smallest Post-Set and the Largest Pre-Set of Abstract Programs, Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 96 / 32 1996.Oct.23

Win 97 Winkler, J. F. H.: The Frege Program Prover FPP. 42. Internationales Wissenschaftliches Kolloquium, TU Ilmenau, 1997, 116 .. 121

Win 98 New Proof Rules for FOR-loops. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 98 / 13 1998.Nov.07