Validation und Verifikation von Software durch symbolische Ausführung

Ralf Kneuper
Wilhelm-Leuschner-Straße 2, 6100 Darmstadt

1 Einführung

Die Grundidee der symbolischen Ausführung ist es, ein Programm auszuführen, ohne Werte für die Eingabevariablen anzugeben. Auf diese Weise erhält man ein Verfahren zur Validation und Verifikation von Programmen, das zwischen formalem Korrektheitsbeweis und Testen liegt.

Als Ausgabe der symbolischen Ausführung erhält man einen Term, der die Eingabevariablen als freie Variablen enthält. Man spricht dann von *symbolischen* Ein- und Ausgabewerten, die aus Variablen oder Termen anstelle konkreter Werte bestehen und auf diese Weise einen ganzen Bereich konkreter Werte abdecken.

Seien T_{symb} die symbolischen Werte, also Terme und Variablen, seien T die tatsächlichen Werte, und sei σ eine Substitution von symbolischen nach tatsächlichen Werten. Dann ist symbolische Ausführung eine Funktion, bei der das folgende Diagramm kommutiert:

$$\begin{array}{ccc} T_{symb} & \xrightarrow{\text{symb. Ausführung}} & T_{symb} \\ {\scriptstyle\sigma}\downarrow & & \downarrow{\scriptstyle\sigma} \\ T & \xrightarrow[\text{Ausführung}]{} & T \end{array}$$

Inzwischen hat sich die symbolische Ausführung zu einem etablierten Verfahren zur Validation und Verifikation von Software entwickelt, vor allem für imperative Programmiersprachen, wo symbolische Ausführung am leichtesten anwendbar ist. Es wurden verschiedene Systeme zur symbolischen Ausführung entwickelt, die eine Reihe unterschiedlicher Ansätze und Techniken verwenden, um symbolische Ausführung für die Validation und Verifikation von Software in jeweils einer bestimmten Sprache einzusetzen, insbesondere auch zur automatischen Erzeugung von Testfällen. Dabei variiert auch die Bedeutung des Begriffes symbolische Ausführung immer wieder in Abhängigkeit von der beabsichtigten Verwendung und der verwendeten Sprache. Formal und weitgehend unabhängig von der verwendeten Sprache (durch Parametrisierung über die denotationale bzw. operationale Semantik der Sprache) wurde der Begriff der symbolischen Ausführung in [Kne91] definiert. Diese Definition ist nicht mehr auf Programme beschränkt, sondern auch auf eine große Klasse von Spezifikationssprachen anwendbar. Grundidee dabei ist es, die Sprache der symbolischen Werte zu erweitern und auch Prädikate zuzulassen.

In diesem Zusammenhang werde ich mich aber auf den oben eingeführten anschaulichen Begriff von symbolischer Ausführung beschränken, und stattdessen eine Übersicht über die verschiedenen auf symbolischer Ausführung basierenden Ansätze zur Validation und Verifikation von Software geben. Aufgrund der unterschiedlichen Zielsetzung der verschiedenen Systeme zur symbolischen Ausführung spricht man dabei oft auch von symbolischem Testen (z.B. [Lig90, Kap. 4]), symbolischer Auswertung oder symbolischer Interpretation. Soweit diese aber auf der gleichen Grundidee aufbauen, wird in diesem Artikel nicht zwischen den verschiedenen Begriffen unterschieden.

[Lig90, Abschnitt 4.4] gibt auch einen Überblick über die wichtigsten Systeme zur symbolischen Ausführung von (imperativen) Programmen. Daneben gibt es noch Systeme zur symbolischen Ausführung von Spezifikationen. Hierzu gehört insbesondere GIST, siehe [Coh83, CSB82], mit dem Spezifikationen in der Sprache GIST symbolisch ausgeführt werden können. Eine andere Verallgemeinerungsmöglichkeit wird u.a. in [Dil90] beschrieben, wo es um die symbolische Ausführung von parallelen Programmen geht.

Das vom Autor entwickelte System SYMBEX (siehe [JJLM91, Kap. 9, Anhang D]) wurde leider aus Zeitgründen nur ansatzweise implementiert. Im Gegensatz zu anderen Systemen ist SYMBEX nicht auf eine bestimmte Sprache zugeschnitten, sondern kann durch Angabe der operationalen Semantik der Sprache in einer bestimmten Form parametrisiert werden.

Im folgenden Teil dieser Einführung werden die Begriffe Validation und Verifikation sowie ein viel benutztes Konzept bei der symbolischen Ausführung, nämlich die Pfadanalyse, beschrieben. Anschließend wird der Einsatz symbolischer Ausführung für die Verifikation (Abschnitt 2) und Validation (Abschnitt 3) von Software beschrieben. In Abschnitt 4 werden einige der auftauchenden Probleme sowie Lösungsmöglichkeiten beschrieben. Abschnitt 5 schließlich enthält eine kurze Zusammenfassung.

1.1 Validation und Verifikation

Unter Validation verstehen wir die Prüfung von Software-Komponenten wie Programm, Spezifikation, etc., auf Erfüllung der Benutzeranforderungen. Validation ist also ein prinzipiell informeller Schritt, da man dabei gegen die zunächst nur im Kopf des Benutzers existierenden Anforderungen prüfen muß. Ziel der Validation ist ein besseres Verständnis der validierten Komponente. Hier geht es also in erster Linie darum zu zeigen, daß das System bestimmte erwartete Eigenschaften hat, oder es zu 'animieren', also in irgendeiner Form sein Verhalten zu modellieren, z.B. durch einen Prototypen.

Verifikation dagegen ist die Prüfung von Software auf Übereinstimmung mit der Spezifikation. Hier ist es also prinzipiell möglich, die Korrektheit einer Implementierung mit Bezug auf die Spezifikation zu beweisen. Dafür muß es als Basis immer eine Spezifikation geben, gegen die geprüft werden kann.

Laut [Zav84] besteht der Unterschied darin, daß Validation sich damit beschäftigt, das richtige System zu bauen, während Verifikation sich damit beschäftigt, das System richtig zu bauen. Validation und Verifikation beziehen sich also auf verschiedene Aspekte der Korrektheit eines Softwaresystems und ergänzen sich dadurch gegenseitig. Insbesondere ist die Verifikation eines Systems nur sinnvoll, wenn sie auf einer Spezifikation beruht, die tatsächlich die Benutzer-Anforderungen beschreibt. Dies läßt sich aber nur durch Validation überprüfen.

Bisher war von Spezifikation und Programm immer als zwei grundsätzlich verschiedenen Komponenten die Rede. Diese Unterscheidung hängt jedoch ganz wesentlich von der Abstraktionsebene ab, schließlich kann man sogar ein Assembler-Programm als Spezifikation eines Maschinenprogrammes ansehen. Im folgenden wird daher ein Programm immer als Spezialfall einer Spezifikation angesehen.

1.2 Symbolische Ausführung und Pfadanalyse

Ein häufig verwendetes Prinzip der symbolischen Ausführung ist die *Pfadanalyse*. Wird ein Programm entlang eines Pfades symbolisch ausgeführt, so assoziiert man mit jedem Punkt in dieser Ausführung einen Pfadwert und eine Pfadbedingung. Der Pfadwert besteht aus den symbolischen Werten der Variablen, während die Pfadbedingung angibt, unter welcher Bedingung dieser Pfad durchlaufen wird. Während also z.B. Zuweisungen den Pfadwert verändern, verändern Verzweigungen die Pfadbedingungen. Im einfachsten Fall beginnt man mit der Pfadbedingung `true`. Erreicht die symbolische Ausführung eine Verzweigung im Programm, dann wird zuerst versucht, aus der derzeitigen Pfadbedingung abzuleiten, welcher Zweig durchlaufen werden muß. Ist dies möglich, so kann der Code in den übrigen Zweigen nicht durchlaufen werden, jedenfalls nicht, wenn man die Verzweigung entlang des betrachteten Pfades erreicht. Im allgemeinen wird es aber möglich sein, verschiedene

Zweige zu durchlaufen. Hat man beispielsweise die Verzweigung `if x<y then ... else ...`, dann ist aus den symbolischen Werten der Variablen `x` und `y` im allgemeinen nicht ersichtlich, ob der `then`-Zweig oder der `else`-Zweig der Bedingung durchlaufen werden soll.

In diesem Fall muß man sich dann bei der symbolischen Ausführung für einen der möglichen Zweige entscheiden. Die entsprechende Bedingung ($x < y$ bzw. $x \not< y$) wird dann mit in die Pfadbedingung aufgenommen. Kriterium für eine solche Entscheidung für einen Zweig kann z.B. die Entscheidung des Benutzers sein (interaktiv oder auch schon im voraus). Manchmal versucht man auch, alle möglichen Pfade (mit gewissen Einschränkungen) zu durchlaufen und betrachtet daher alle möglichen Alternativen nacheinander.

Notwendig für den Einsatz der Pfadanalyse ist aber natürlich, daß es ein angemessenes Pfadkonzept in der betreffenden Sprache gibt. Für imperative Programmiersprachen ist dies offensichtlich der Fall, während es für Spezifikationssprachen individuell verschieden ist.

2 Verifikation durch symbolische Ausführung

2.1 Symbolische Ausführung plus Induktion über Berechnung

Dieser, zuerst von [HK76] beschriebene, Ansatz dient zum Beweis der partiellen Korrektheit (Terminierung wird also nicht behandelt) von Programmen. Dazu werden Programme mit Zusicherungen versehen, unterschieden nach `ASSUME`-, `PROVE`- und `ASSERT`-Zusicherungen. `ASSUME`-Zusicherungen stellen die Vorbedingungen eines Programmes oder Programmteiles dar, das heißt, diese Zusicherung kann im weiteren Verlauf der Ausführung vorausgesetzt werden. `PROVE`-Zusicherungen entsprechen den Nachbedingungen, ihre Erfüllung ist zu beweisen. `ASSERT`-Zusicherungen schließlich sind eine Kombination aus beidem, sie sind sowohl (zu beweisende) Nachbedingung des an der Zusicherung endenden als auch Vorbedingung des dort startenden Programmteiles. Meist werden sie an einem Schnittpunkt einer Schleife angebracht und stellen die Schleifeninvarianten dar.

Wird nun in jeder Schleife durch eine `ASSERT`-Zusicherung ein sogenannter *Schnittpunkt* gesetzt, so ist der Pfad zwischen zwei aufeinanderfolgenden Zusicherungen immer endlich und führt von einer Vorbedingung, also einer `ASSUME`-Zusicherung oder einer als Vorbedingung interpretierten `ASSERT`-Zusicherung, zu einer Nachbedingung, also einer `PROVE`-Zusicherung oder einer als Nachbedingung interpretierten `ASSERT`-Zusicherung. Zusätzliche `ASSERT`-Zusicherungen können dabei an jeder Stelle des Programmes gesetzt werden.

Um nun die Korrektheit eines Programmes zu beweisen, betrachtet man alle solchen Pfade zwischen zwei Zusicherungen. Ausgehend von der Vorbedingung als Pfadbedingung, führt man das entsprechende Programmstück symbolisch aus. Am Ende sollte sich dann aus den symbolischen Werten der Variablen zusammen mit der Pfadbedingung die geforderte Nachbedingung ableiten lassen.

Das Problem der symbolischen Ausführung von Schleifen wird also hier umgangen, indem man nicht ganze Programme symbolisch ausführt, sondern immer nur Teilstücke von Programmen, die keine Schleife enthalten, sondern höchstens den Körper einer Schleife.

Ein einfaches Beispiel hierfür ist die Division ganzer Zahlen:

```
1  procedure DIV(X,Y)
2  ASSUME Y≠0
3  R:=X
4  Q:=0
5  WHILE (Y <= R) DO
6      ASSERT X=Q*Y+R
7      R := R - Y
8      Q := Q + 1
9  END
10 PROVE X=Q*Y+R and 0<=R<Y
```

Die Zeilen 2, 6 und 10 enthalten die verwendeten Zusicherungen.[1] Die zu betrachtenden Pfade laufen also von 2 nach 6 über 3,4, von 6 zurück nach 6 über 5, 7, 8, und von 6 nach 10 über 5, 9, 10. Die Korrektheit des ersten Pfades ist leicht zu verifizieren. Für den zweiten Pfad wird die Zusicherung 6 als Pfadbedingung initialisiert. Der Körper der Schleife wird nun gemäß Zeile 5 durchlaufen, wenn `Y<=R` gilt. Also wird diese Bedingung mit in die Pfadbedingung aufgenommen und anschließend der Schleifenkörper (Zeilen 7 und 8) symbolisch ausgeführt. Aus den sich daraus ergebenden symbolischen Werten der Variablen läßt sich nun zeigen, daß auch anschließend wieder die Bedingung 6 gilt.

Für den dritten Pfad schließlich wird wiederum Zusicherung 6 als Pfadbedingung initialisiert. Da der Schleifenkörper bei diesem Pfad nicht durchlaufen wird, muß die Negation der Bedingung `Y<=R` in die Pfadbedingung aufgenommen werden. Versucht man nun, die Nachbedingung 10 abzuleiten, so stellt man fest, daß dies nicht möglich ist, die Bedingung $0 \leq R$ ist nicht ableitbar. Mit anderen Worten, bzgl. der gegebenen Vor- und Nachbedingung ist die Prozedur DIV nicht korrekt. Um eine korrekte Prozedur zu erhalten, kann man je nach Anforderungen die Vorbedingung modifizieren (z.B. nur positive Werte für `X` und `Y` erlauben), die Nachbedingung modifizieren (z.B. die Bedingung `0<=R` weglassen), oder das Programm selbst entsprechend modifizieren.

Anders betrachtet handelt es sich bei einem solchen Korrektheitsbeweis von Schleifen um einen Induktionsbeweis über die Anzahl der Schleifendurchläufe, also über die Berechnung. Die Induktionsvoraussetzung ergibt sich dabei aus der Interpretation der `ASSERT`-Zusicherung als Vorbedingung: Wenn die Zusicherung zu Anfang der Schleife gilt, dann gilt sie auch nach null Schleifendurchläufen. Der Induktionsschritt besteht aus der symbolischen Ausführung der Schleife, also aus dem Nachweis, daß die `ASSERT`-Zusicherung eine Invariante der Schleife darstellt.

Nicht berücksichtigt bei diesem Ansatz zur Verifikation von Programmen wird die Termination, es kann also nur partielle, nicht totale Korrektheit nachgewiesen werden. Auch dies ist am Beispiel der Prozedur DIV ersichtlich, denn diese terminiert i.a. nicht für negative Werte von `Y`. Analog dem Vorgehen bei Benutzung von Hoare-Logik kann man die Terminierung von Programmen verifizieren, indem man eine Funktion vom Pfadwert auf die natürlichen Zahlen definiert und nachweist, daß der Wert dieser Funktion mit jedem Schleifendurchlauf geringer wird. Ein tatsächlicher Einsatz dieses Verfahrens im Zusammenhang mit symbolischer Ausführung ist dem Autor allerdings nicht bekannt.

2.2 Symbolische Ausführung plus Induktion über Daten

Der wesentliche Unterschied zwischen diesem, in [Bur74] beschriebenen, Ansatz und dem im vorigen Abschnitt beschriebenen wird beim Beweis der Korrektheit von Schleifen deutlich. Während bisher die Korrektheit durch Induktion über die Anzahl der Schleifendurchläufe gezeigt wurde, benutzt man nun Induktion über die Eingabedaten. Dazu verwendet man eine andere Form von Zusicherungen: Beim oben beschriebenen Ansatz hatte eine Zusicherung P die Bedeutung "Wenn die Programmausführung diese Stelle erreicht, dann gilt garantiert P". Dadurch war es nicht möglich, Terminierung von Programmen direkt zu berücksichtigen. Stattdessen werden nun Zusicherungen der Form $x : P$ verwendet, mit der Bedeutung, daß garantiert die Stelle x im Programm erreicht wird in einem Zustand, in dem P gilt. Insbesondere kann man auf diese Weise totale Korrektheit formulieren.

Im Beispiel der oben definierten Prozedur `DIV` könnte man dann wie folgt vorgehen. Dabei beschränken wir uns jetzt aber auf *positive* Werte der Variablen `X` und `Y`, da die totale Korrektheit der Prozedur gezeigt werden soll. Die totale Korrektheit der Zuweisungen wird genauso gezeigt wie bisher auch. Für die Schleife zeigt man nun durch Induktion über den Wert der Variablen R, daß gilt:

"Wenn $6 : X = Q * Y + R$ dann $6 : (X = Q * Y + R \wedge R < Y)$"

Ist der Wert r der Variablen R gleich 0, so ist die Bedingung offensichtlich erfüllt, da $Y > 0$. Gelte die Bedingung nun für alle Werte kleiner als r. Außerdem gelte $6 : X = Q * Y + R$, wobei

[1] Da sich die Werte von `X` und `Y` im Laufe der Ausführung nicht ändern, wird hier der Einfachheit halber nicht zwischen den Anfangswerten und den jeweils derzeitigen Werten dieser Variablen unterschieden, wie dies normalerweise notwendig ist.

R den Wert r habe. Durch symbolische Ausführung einer Iteration der Schleife erhält man dann, daß $6 : X = (Q + 1) * Y + (R - Y)$, und aus der Induktionsannahme folgt die Behauptung. Durch Kombination der einzelnen Komponenten ergibt sich nun die totale Korrektheit des Programmes.

2.3 Partitionsanalyse

Partitionsanalyse ist der Ansatz zur Verifikation von Programmen, der am deutlichsten auf symbolische Ausführung Bezug nimmt und nicht auch ohne diese machbar ist. Hier werden Spezifikation und Implementierung symbolisch ausgeführt und die Ergebnisse verglichen, siehe [CR84]. Die Gesamtmenge der möglichen Eingabewerte wird dabei entsprechend den Pfadbedingungen der verschiedenen Pfade durch die Spezifikation sowie durch die Implementierung partitioniert. Jedem Element einer dieser Partitionierungen entspricht also genau ein Pfad, mit einer Pfadbedingung, durch die Spezifikation bzw. Implementierung. Diese beiden Partitionierungen der möglichen Eingabewerte werden zusammengefaßt und ergeben eine weitere, feinere Partitionierung der Eingabewerte.

Für jedes Element dieser gemeinsamen Partitionierung werden nun die erhaltenen Pfadwerte verglichen. Das genaue Verfahren zum Vergleich der Pfadwerte hängt teilweise von der geforderten Erfüllungsrelation zwischen Spezifikation und Implementierung ab. Im einfachsten Fall werden die Ergebnisse durch die Spezifikation eindeutig determiniert. In diesem Fall müssen beide Pfadwerte gleich sein, wie in [CR84] beschrieben.

Oft wird eine gute Spezifikation aber unter-determiniert sein, d.h. die geforderten Ergebnisse sind nicht eindeutig bestimmt. Abgesehen von den dadurch entstehenden Problemen, symbolische Ausführung überhaupt durchzuführen (vgl. [JJLM91, Kap. 9, Anhang D]) kann man in diesem Fall auch nicht mehr Gleichheit der Ergebnisse fordern, sondern die symbolischen Pfadwerte der Implementierung müssen ein Spezialfall der Pfadwerte der Spezifikation sein. I.a. wird diese Beziehung aber unentscheidbar sein, wenn man nicht geeignete Einschränkungen macht.

Hat eine Spezifikationssprache allerdings kein Pfadkonzept, sondern werden Systeme durch eine Relation zwischen Eingabe- und Ausgabedaten beschrieben, dann ist dieses Konzept kaum noch anwendbar. Dazu kommt noch, daß Schleifen normalerweise zu einer unendlichen Menge von Elementen in der Partitionierung führen, man also nicht direkt für jedes Element die Ergebnisse überprüfen muß. Stattdessen muß das Ergebnis der Schleife entsprechend umformuliert und in eine geschlossene, endliche Form gebracht werden (vgl. Abschnitt 4.2).

3 Validation durch symbolische Ausführung

3.1 Manuelle Überprüfung der Ergebnisse

Die wahrscheinlich am weitesten verbreitete Methode zur Validation durch symbolische Ausführung ist die manuelle Überprüfung der Ergebnisse der symbolischen Ausführung gegen die erwarteten Ergebnisse, analog der manuellen Prüfung von Testergebnissen. Diese manuelle Überprüfung bezieht sich dabei i.a. einerseits auf die Pfadwerte, d.h. es wird geprüft, ob die durch symbolische Ausführung erzeugten Pfadwerte mit den erwarteten Werten übereinstimmen. Andererseits kann auch die Pfadbedingung überprüft werden, d.h. es wird geprüft, ob der Pfad unter den erwarteten Bedingungen durchlaufen wird.

Bei dieser Überprüfung ist es wichtig, daß die durch symbolische Ausführung erzeugten Werte und die Programmstruktur sich nicht zu sehr ähneln, da sonst ein Fehler, der im Programm nicht bemerkt wurde, auch durch symbolische Ausführung nicht bemerkt wird. Dies kann man z.B. erreichen durch eine geeignete Unterstützung für die Vereinfachung von symbolischen Ausdrücken, oder auch durch eine Übersetzung in eine andere Sprache. Dieser Ansatz wurde bei GIST gewählt, wo ein *Paraphraser* die Ergebnisse der symbolischen Ausführung in eine einfache Form natürlicher Sprache übersetzt (siehe [Swa83]). Eine solche Übersetzung von formaler Sprache in natürliche Sprache ist (im Gegensatz zur umgekehrten Richtung) nicht allzu schwierig und wurde bei GIST

sowohl für Spezifikationen als auch für die Ergebnisse symbolischer Ausführung benutzt. Ein solcher Paraphraser erlaubt eine andere Sicht auf die gleichen Ergebnisse und macht es dadurch einfacher, Fehler zu finden. Insbesondere erlaubt er es auch dem Benutzer, der die Notation der symbolischen Werte nicht kennt, die Ergebnisse der symbolischen Ausführung zu überprüfen.

Verwendet man die manuelle Überprüfung der Ergebnisse der symbolischen Ausführung bei Spezifikationen, so erhält man eine Form des Prototypings, die sehr effektiv sein kann bei der Validation von Spezifikationen. Ein Vergleich der Effektivität dieser Form der symbolischen Ausführung mit verschiedenen Formen des Testens ist in [How78] beschrieben.

Eine andere Einsatzmöglichkeit existiert beim Debugging von Programmen. Unterstützt man bei der symbolischen Ausführung das Setzen von Breakpoints oder Trace-Funktionen, dann kann man ähnlich wie mit einem gewöhnlichen Debugger arbeiten, aber es werden jeweils symbolische Werte statt tatsächlicher Werte angezeigt.

3.2 Nicht ausführbare Programmteile

Eine andere Einsatzmöglichkeit symbolischer Ausführung ist die Identifizierung nicht ausführbarer Programmteile, entweder zur manuellen Überprüfung oder auch zur Programm-Reduktion.

Mögliche Gründe für die Existenz nicht ausführbarer Programmteile sind eine schlechte, unübersichtliche Programmstruktur, die Wiederverwendung von Programmteilen, die automatische Erzeugung von Programmcode, z.B. in einem Compiler, oder aber auch Fehlertoleranz. Prüft man nämlich im Programm auch auf Fehlerbedingungen, die "eigentlich" nicht auftreten können, dann wird das Programm dadurch u.U. sicherer, gerade weil es nicht ausführbaren Code enthält. Allerdings ist zu berücksichtigen, daß solcher Code grundsätzlich dazu führt, daß beim Test nicht einmal Statement-Abdeckung möglich ist.

Um nicht ausführbaren Code mit Hilfe von symbolischer Ausführung zu identifizieren, werden Verzweigungen daraufhin überprüft, welche Zweige möglich sind. Wird ein Zweig von keinem Pfad durchlaufen, so handelt es sich um nicht ausführbaren Code. Schwierigkeiten bereitet dabei wieder die Behandlung von Schleifen, da sie zu einer unendlichen Anzahl von Pfaden führen können. Die erste Alternative zur Lösung dieses Problems ist die Aufspaltung der Pfade durch Verwendung von Zusicherungen und Ausführung der (endlich vielen) Pfade zwischen zwei aufeinanderfolgenden Zusicherungen, wie in Abschnitt 2.1 beschrieben.

Eine andere Möglichkeit ist es, die in einer Schleife modifizierten Werte auf neue, nicht definierte Werte zu setzen durch Einführung neuer Variablen, anstatt die Schleife selbst zu durchlaufen. Hängt die Bedingung einer Verzweigung nicht von den in der Schleife modifizierten Variablen ab, dann kann man auch so feststellen, ob alle Zweige durchlaufen werden können, ohne dabei eine unbekannte Anzahl von Iterationen der Schleife berücksichtigen zu müssen. Diese Methode ist korrekt, aber nicht vollständig, da nicht ausführbarer Code nicht immer erkannt wird.

Ist auch dies nicht möglich, so muß man die Ergebnisse von symbolischer Ausführung der Schleife auf andere Weise geeignet formulieren, um danach folgende Verzweigungen analysieren zu können. Mögliche Ansätze dafür sind in Abschnitt 4.2 beschrieben.

Bei dieser Beschreibung wurde davon ausgegangen, daß man aus der Betrachtung einer Verzweigung schon erkennen kann, ob der folgende Code ausgeführt wird oder nicht. Dies funktioniert aber nur, wenn man auch sicher erkennen kann, ob die Kontrolle von einer anderen Stelle in den Code in einem der Zweige springen kann, z.B. mit einer `goto`-Anweisung. Eine Alternative, bei der dieses Problem nicht in dieser Form auftritt, ist die Markierung von ausgeführtem Code bei der symbolischen Ausführung. Code, der nicht markiert wird, ist auch nicht ausführbar.

3.3 Testfall-Erzeugung

Eine der am weitesten verbreiteten Einsatzmöglichkeit von symbolischer Ausführung ist die (automatische) Erzeugung von Testfällen durch symbolische Ausführung von Spezifikation oder Implementierung. Ziel dabei ist es, Testfälle zu erzeugen, die einen bestimmten Abdeckungsgrad, z.B.

Zweigabdeckung, erreichen.

Dazu geht man in mehreren Schritten vor. Im ersten Schritt werden die gewünschten Pfade durch das Programm erzeugt, entweder automatisch nach bestimmten, vorgegebenen Kriterien, oder manuell, d.h. der Benutzer entscheidet jeweils, welcher Zweig einer Verzweigung genommen wird. Mögliche Kriterien für die Auswahl von Pfaden sind z.B. Statement-Abdeckung, Zweig-Abdeckung, oder die Auswahl aller Pfade bis zu einer bestimmten Länge bzw. bis zu einer bestimmten Anzahl von Schleifen-Iterationen.

Für die ausgewählten Pfade werden dann die jeweiligen Pfadbedingungen generiert. Im nächsten Schritt sucht man nach Lösungen dieser Pfadbedingungen, d.h. man sucht Eingabewerte, die die Pfadbedingung erfüllen. Je nach Form der Pfadbedingung kann dies relativ einfach sein, i.a. ist es aber ein sehr komplexes Problem. Gemäß dem Satz von Matijasevic (siehe [Dav73]) ist es sogar schon für Pfadbedingungen, die nur aus einer Gleichung über beliebigen Polynomen bestehen, i.a. unentscheidbar, ob eine solche Lösung überhaupt existiert.

Andererseits gibt es natürlich eine Vielzahl algebraischer Verfahren, um Spezialfälle von Pfadbedingungen zu lösen. ATTEST beispielsweise beschränkt sich auf Systeme linearer Gleichungen und Ungleichungen und kann dadurch immer eine Lösung der Pfadbedingung finden [CR84, S. 148].

Hat man schließlich eine Lösung der Pfadbedingung und damit einen Testfall für einen bestimmten Pfad gefunden, so bleibt noch als letzter Schritt die Berechnung des erwarteten Ausgabewertes aus dem Pfadwert. Solange es sich hierbei um eine imperative Programmiersprache handelt, besteht der Pfadwert einer Variablen aus einem expliziten Term und es ergibt sich dabei kaum ein Problem. Anders muß man vorgehen, wenn auch Prädikate als symbolische Werte zugelassen sind, z.B. bei der symbolischen Ausführung von Spezifikationen. In diesem Fall ist das erwartete Ergebnis nicht mehr allgemein berechenbar, aber das würde auch kaum nutzen, da das Ergebnis oft nicht eindeutig durch den Pfadwert bestimmt sein wird. Hier besteht dann die Möglichkeit, die durch den Test erzeugten Ausgabewerte daraufhin zu überprüfen, ob sie die durch die Pfadbedingung gegebenen Prädikate erfüllen. In der Praxis ist dieses Problem wesentlich einfacher zu lösen, auch wenn man es weiterhin mit einem theoretisch unentscheidbaren Problem zu tun hat, nämlich der Überprüfung eines beliebigen Prädikates auf Wahrheit.

3.4 Analyse von Fehlerbedingungen

Bei diesem Verfahren werden vorgegebene Fehlerbedingungen wie z.B. Division durch Null auf Konsistenz mit der Pfadbedingung untersucht. Sind die Bedingungen konsistent, so kann der Fehler bei tatsächlicher Ausführung auftreten. Verwandt damit ist die Überprüfung auf *Clean termination*, also Überprüfung, daß partielle Funktionen nur auf Argumente angewandt werden, die in ihrem Definitionsbereich liegen, sowie auf Overflow- und Underflow-Bedingungen [CH79].

Auch wenn [CH79] mit einer modifizierten Hoare-Logik und nicht mit symbolischer Ausführung arbeitet, so lassen sich die dort geforderten Prüfungen doch auch mit symbolischer Ausführung durchführen. Eine derartige Prüfung auf Fehlerbedingungen wird z.B. in ATTEST unterstützt, siehe [CR84, §4.1].

Ein wesentliches Problem beim Einsatz dieser Methode ist allerdings (sowohl bei Verwendung einer Hoare-Logik als auch bei Verwendung von symbolischer Ausführung) die Konsistenzprüfung. Hierbei benötigt man erhebliche Beweisunterstützung, nach Möglichkeit automatisch. I.a. ist die Konsistenzprüfung allerdings ein unentscheidbares Problem, so daß die Prüfung auf Fehlerbedingungen mit diesem Verfahren nicht unbegrenzt möglich ist.

Eine weitere Einschränkungen dieses Ansatzes ist, daß nur bestimmte, vorgegebene Fehler gefunden werden können. Diese werden allerdings für den jeweiligen Pfad immer gefunden, vorausgesetzt der benutzte Beweiser ist stark genug. Außerdem bezieht sich eine einzelne Prüfung immer nur auf einen einzigen Pfad, nicht auf alle Pfade, die zu einem bestimmten Anweisung führen. Hier tritt also wieder das Problem auf, eine Aussage über eine unendliche Anzahl von Pfaden zu machen, das auch wieder auf ähnliche Weise gelöst werden kann.

Eine Variation dieses Verfahrens ist die Prüfung von Zusicherungen ohne echten Korrektheitsbeweis mit Bezug auf Vor- und Nach-Bedingungen. Wann immer symbolische Ausführung eine Zusicherung erreicht, wird überprüft, daß die Pfadwerte die Zusicherung erfüllen. Ist dies nicht der Fall, so wird eine Fehlermeldung ausgegeben. Auf diese Weise kann man die Korrektheit der gemachten Annahmen oder Zusicherungen überprüfen, ohne allerdings dadurch einen Beweis zu erhalten, da sich die Überprüfung jeweils nur auf einen Pfad bezieht. Im Extremfall kann man dieses Verfahren jedoch bis zu einem allgemeinen Beweis ausbauen, indem man eine Zusicherung am Ende des Programmes überprüft, also eine Nachbedingung, und dabei *alle* möglichen Pfade dorthin berücksichtigt.

4 Auftretende Probleme und Möglichkeiten ihrer Lösung

Beim Einsatz der beschriebenen Methoden zur Verifikation und Validation treten natürlich eine Reihe von Schwierigkeiten auf, die den Einsatz von symbolischer Ausführung erschweren. Im folgenden werden die wichtigsten dieser Probleme beschrieben (einige davon wurden auch schon oben erwähnt) und Lösungsmöglichkeiten vorgeschlagen.

4.1 Auswertung von Parametern

Hantler und King [HK76] beschreiben den Aufruf von Prozeduren mit Parametern wie folgt: An der Stelle des Aufrufs wird der Text der Prozedur eingesetzt, wobei die Parameter jeweils durch die übergebenen symbolischen Werte ersetzt werden. Dadurch erhält man eine *call-by-reference*-Semantik beim Aufruf von Prozeduren etc., auch wenn die verwendete Sprache eigentlich mit *call-by-value* arbeitet. [HK76] gibt als Beispiel hierzu die folgende Prozedur, die die Werte zweier Variablen vertauscht:

```
procedure EXCHANGE(X,Y)
X := X-Y;
Y := X+Y;
X := Y-X;
```

Ruft man nun in symbolischer Ausführung diese Prozedur auf mit dem gleichen Argument für beide Parameter, so erhält man `Z := Z-Z; Z := Z+Z; Z := Z-Z`, d.h. Z erhält den Wert Null. Das Problem entsteht dadurch, daß nicht nur die Werte der Variablen X und Y, sondern die Variablen selbst gleichgesetzt wurden. Hantler und King [HK76] schlagen als Lösung dieses Problems eine Fallunterscheidung vor, d.h. man arbeitet mit unterschiedlichen Vor- und Nachbedingungen für `EXCHANGE` je nachdem, ob $X = Y$ oder $X \neq Y$. Bei Prozeduren mit vielen Variablen kann dieses Verfahren aber sehr aufwendig werden, da die Anzahl der Fallunterscheidungen exponentiell mit der Anzahl der Variablen ansteigt. Außerdem wird das Problem auf diese Weise nicht wirklich gelöst, sondern nur umgangen.

Eine mögliche Alternative besteht darin, zwischen einer Variablen (als Adresse) und einem Symbol für ihren Wert zu unterscheiden. Im obigen Beispiel wird also ein Symbol, das den *Wert* der Variablen Z repräsentiert, beim symbolischen Aufruf an `EXCHANGE` übergeben. In diesem Fall erhält man wie gewünscht `X := Z-Z; Y := (Z-Z)+Z; X := ((Z-Z)+Z) - (Z-Z)`.

Um also eine call-by-value-Semantik zu erhalten, muß man, wenn man beim Aufruf einer Prozedur den Text der Prozedur an der Stelle des Aufrufs einsetzt, den Variablen am Anfang der Prozedur die übergebenen Werte zuweisen und den Text der Prozedur unverändert lassen.

4.2 Implizite, rekursive und iterative Definitionen

Wie schon wiederholt erwähnt, gibt es erhebliche Schwierigkeiten bei der Anwendung symbolischer Ausführung auf implizite, rekursive und iterative Definitionen, insbesondere auch bei Schleifen. Je nach Anwendung wurden dafür eine Reihe von Lösungsmöglichkeiten vorgeschlagen, die allerdings das

Problem immer nur unter bestimmten Voraussetzungen oder in einem bestimmten Zusammenhang lösen.

Implizite Definitionen treten in erster Linie bei der symbolischen Ausführung von Spezifikationen auf, in Programmiersprachen sind meist nur explizite Definitionen erlaubt. Gelegentlich gibt es bei der symbolischen Ausführung einer impliziten Definition die Möglichkeit, diese zu *lösen*, also explizit zu machen. Im allgemeinen ist dies aber nicht möglich, in diesem Fall muß die Sprache der symbolischen Werte entsprechend erweitert werden, z.B. durch Prädikate.

Eine erste Möglichkeit ist die Betrachtung von Spezialfällen. Dazu gehört beispielsweise die Beschränkung der Anzahl der Iterationen einer Schleife oder der Pfadlänge, oder sogar die explizite Angabe eines Variablenwertes anstelle des symbolischen Wertes der Variablen. Dieser Ansatz ist vor allem relevant für die Behandlung von iterativen oder rekursiven Programmkonstrukten. Die Gefahr bei diesem Ansatz ist allerdings, daß die Eingabewerte zu sehr eingeschränkt werden, so daß die Vorteile von symbolischer Ausführung gegenüber Testen verloren gehen und nur die Nachteile (z.B. andere Laufzeitumgebung, zusätzlicher Aufwand) erhalten bleiben.

Andererseits kann man durch Betrachtung von Spezialfällen evtl. allgemeine Ergebnisse erkennen und in einem Satz formulieren, den man dann beweist und anschließend als Zusicherung mit berücksichtigt, oder, falls die Sprache der symbolischen Werte auch Prädikate erlaubt, als symbolischen Wert einer Variablen nach Ausführung der betrachteten Schleife etc.. In manchen Fällen ist es sogar möglich, direkt einen geschlossenen Ausdruck als symbolischen Ergebniswert anzugeben.

Die Verwendung von passenden Prädikaten oder geschlossenen Ausdrücken als symbolische Werte ist natürlich unabhängig von der Betrachtung von Spezialfällen, diese ist nur ein mögliches Hilfsmittel, um passende Werte zu finden. Manche Systeme versuchen, derartige Werte automatisch zu finden, in ATTEST beispielsweise wird durch symbolische Ausführung einer Schleife eine rekursive Relation *recurrence relation* generiert und dann versucht, diese analytisch zu lösen [CR84, Kap. 2].

Ob eine solche Lösung gefunden wird, ja ob sie überhaupt existiert, hängt ganz wesentlich von der verwendeten Sprache der symbolischen Werte ab. Ist die Sprache expressiv genug, so kann die Lösung immer ausgedrückt werden (notfalls durch Einführung einer neuen Funktion $Loesung(\ldots)$), aber eine derartig expressive Sprache wird kaum je zur Verfügung stehen (abgesehen davon, daß es kaum hilft zu erfahren, die Lösung des Problems P sei $Loesung(P)$). Beim Einsatz von symbolischer Ausführung ist es daher aber wichtig zu definieren, welche Sprache man für symbolische Werte benutzen will, da diese Definition wesentlichen Einfluß auf die Ergebnisse hat. In [Kne91] wurden beispielsweise allgemeine Prädikate als symbolische Werte erlaubt, um auch Spezifikationen mit impliziten Definitionen symbolisch ausführen zu können.

Eine andere Möglichkeit, die bei manchen Anwendungen möglich ist, wurde schon in Abschnitt 2.1 beschrieben, nämlich der Einsatz von Zusicherungen, um Pfade in eine endliche Anzahl endlich langer Pfadsegmente zu zerlegen. Dieser Ansatz ist angemessen, wenn man bestimmte Eigenschaften des Programmablaufes überprüfen oder nachweisen will, ist aber nicht geeignet zur symbolischen Ausführung eines kompletten Programmes, um z.B. die Ergebnisse manuell zu überprüfen oder um Testfälle zu erzeugen.

Eine weitere Möglichkeit, die einsetzbar ist, wenn man mit Prädikaten als symbolischen Werten arbeitet, ist die *schwache* symbolische Ausführung [Kne91, §3.1]. Hier wird nicht mehr gefordert, daß die Ergebnisse symbolischer Ausführung die Ergebnisse tatsächlicher Ausführung vollständig beschreiben, sondern die erhaltenen Prädikate müssen diese nur noch korrekt beschreiben. Ohne dies explizit zu erwähnen, wird dieser Ansatz z.B. auch in GIST [BGW82, Coh83] verwendet, wo symbolische Ausführung als eine Form des Ableitens von Eigenschaften des beschriebenen Systems gesehen wird.

4.3 Semantik der benutzten Programmiersprache

In den letzten beiden Abschnitten war ein wesentlicher Aspekt die korrekte Umsetzung der Semantik der Programmiersprache für symbolische Ausführung. Explizit oder implizit wird zur symbolischen Ausführung eine Definition der Semantik der benutzten Programmiersprache in einer sonst nicht

vorhandenen Form notwendig. Diese Tatsache wird allerdings meist übergangen, da man sich jeweils auf eine einzige Sprache beschränkt. Indirekt ist dies aber der Grund dafür, warum man meist nur von einer kleinen Sprache mit wenigen Konstrukten ausgeht — eine vollständige Beschreibung der Semantik einer der verbreiteten Programmiersprachen im für symbolische Ausführung geeigneten Format wäre sehr aufwendig zu erstellen.

Explizit beschrieben wird der Zusammenhang zwischen symbolischer Ausführung und der Semantik der benutzten Sprache in [Kne91]. Hier werden Definitionen der denotationalen und operationalen Semantik von symbolischer Ausführung gegeben, die jeweils durch die Semantik der Sprache parametrisiert werden. Allerdings ist dieser Ansatz nur für Sprachen geeignet, deren Semantik sich durch Zustände und Zustandstransformationen beschreiben läßt, also u.a. imperative Programmiersprachen sowie modell-orientierte Spezifikationssprachen. Für diese Gruppe von Sprachen liefert die Definition der denotationalen Semantik dann einen Korrektheitsbegriff für symbolische Ausführung, während die operationale Semantik als Grundlage eines sprachunabhängigen Systems für symbolische Ausführung dienen kann (vgl. [JJLM91]).

4.4 Vereinfachung der Ergebnisterme

Insbesondere für die manuelle Weiterverarbeitung der Ergebnisse ist eine sehr flexible, vom Benutzer steuerbare Vereinfachungsstrategie wichtig. Diese Flexibilität ist besonders auch deshalb wichtig, weil verschiedene Benutzer verschiedene Auffassungen darüber haben, welche Terme einfacher zu verstehen sind als andere. Vereinfachung im Sinne einer Transformation in eine Normalform ist dagegen für die automatische Weiterverarbeitung meist sehr nützlich, für den menschlichen Benutzer ist das Ergebnis aber meist kaum noch lesbar. Je nach Einsatz gibt es also bei der symbolischen Ausführung erheblich unterschiedliche Anforderungen an die Vereinfachung von Termen.

Dazu gehört insbesondere die Frage, welche Funktionssymbole in einem Ergebnisterm expandiert werden sollten. Flexibilität in dieser Frage erlaubt, auch unvollständig beschriebene Systeme symbolisch auszuführen (durch Nicht-Expansion nicht definierter Komponenten).

Gegeben sei beispielsweise die folgende Spezifikation eines Sortierprogrammes:

$$SORT\ (l : \mathbf{N} - seq)\ r : \mathbf{N} - seq$$
$$\mathbf{post}\ is - permutation(l, r) \wedge is - sorted(r)$$

In einem frühen Stadium der Entwicklung sind die Funktionen $is - permutation$ und $is - sorted$ möglicherweise noch nicht definiert. Trotzdem will man die schon fertigen Teile der Spezifikation schon frühzeitig validieren. Mit symbolischer Ausführung ist dies möglich, indem man die Symbole $is - permutation$ und $is - sorted$ in den Ergebnistermen der symbolischen Ausführung beläßt und sie noch nicht expandiert. Auf diese Weise kann man sogar eventuelle Anforderungen an diese Funktionen feststellen (weil z.B. vom Ergebnis von $SORT$ bestimmte Eigenschaften erwartet werden) und bei der Definition berücksichtigen.

Dies setzt aber voraus, daß bei symbolischer Ausführung der Benutzer die Kontrolle darüber hat, wann eine Definition in einem gegebenen Term expandiert wird und wann nicht, wobei unter "Definition" hier auch eine Prozedur oder Funktion zu verstehen ist. Diese Kontrolle wird in Systemen zu symbolischen Ausführung üblicherweise nicht unterstützt. Bei Definitionen mit Nebeneffekten ist eine solche Möglichkeit allerdings auch nur von zweifelhaftem Nutzen, da die tatsächlichen Ergebnisse kaum zu erkennen sind aus symbolischen Ergebniswerten, die noch nicht-expandierte Definitionen enthalten.

Die Vereinfachung der Ergebnisterme geht aber noch weiter und umfaßt nicht nur Expansion von Definitionen im oben genannten Sinne, sondern auch andere Definitionen in der Sprache der Ergebnisterme sowie die Verwendung von Gleichheiten über Termen zur Substitution. Um dies voll ausnutzen zu können, benötigt man allerdings auch entsprechende Beweisunterstützung, insbesondere zum Beweis der verwendeten Gleichheiten. Will man z.B. $f(a, b)$ durch $f(b, a)$ substituieren, so muß man zuerst die Kommutativität von f beweisen. Im praktischen Einsatz wird aus diesem Grund aber meist die Ersetzung gleicher Terme nicht unterstützt.

4.5 Beweisen und Entscheidbarkeit

Wie wiederholt erwähnt, wird zur symbolischen Ausführung häufig (automatische) Beweisunterstützung benötigt, so z.B. zur Prüfung der Konsistenz von Fehler- und Pfadbedingung, oder zur Verifikation von Zusicherungen.

Während manche Systeme nur eine sehr eingeschränkte Form von Beweisen anbieten, dadurch dann auch nur bestimmte Ansätze der Verifikation und Validation unterstützen, sind andere direkt mit einem Werkzeug zur allgemeinen Beweisunterstützung verbunden. Ein Mittelweg zwischen diesen beiden Formen der Beweisunterstützung besteht darin, jeweils für die speziell auftretenden Probleme auch spezielle Unterstützung zu bieten. So ist z.B. eine Konsistenzprüfung einerseits möglich durch direktes Beweisen der entsprechenden Sätze (axiomatischer Ansatz), andererseits aber auch durch Suchen nach einer Lösung der Bedingungen (vor allem bei Systemen von (Un-) Gleichungen etc., algebraischer Ansatz). Dieser zweite Weg wurde z.B. bei ATTEST gewählt.

Unabhängig von der gewählten Form der Beweisunterstützung stößt man aber auf die Schwierigkeit, daß die meisten der betroffenen Probleme im allgemeinen Fall unentscheidbar sind. Hier bleibt nur die Möglichkeit zu versuchen, für möglichst viele der tatsächlich auftauchenden Spezialfälle eine Lösung zu finden, bzw. sich auf bestimmte, lösbare Klassen von Problemen zu beschränken (z.B. lineare Pfad- und Fehlerbedingungen bei der Konsistenzprüfung).

4.6 Andere Laufzeitumgebung

Bei der symbolischen Ausführung eines Programmes eine andere Laufzeitumgebung benutzt wird als bei der tatsächlichen Ausführung des gleichen Programmes. Umgebungsabhängige Probleme können daher normalerweise durch symbolische Ausführung nicht entdeckt werden. Prinzipiell können diese natürlich schon in der Beschreibung der Semantik der Programmiersprache berücksichtigt werden, in der Praxis ist dies allerdings nur in beschränktem Rahmen möglich, da selten alle Einschränkungen bekannt und bewußt sind. Außerdem wird das Ergebnis der symbolische Ausführung bei Berücksichtigung aller Einschränkungen sehr komplex, so daß ihr Nutzen zweifelhaft wird.

Bestimmte Umgebungsprobleme, wie z.B. Rundungsfehler, sind auf diese Weise auch sehr schwierig in den Griff zu bekommen, da viele Vereinfachungen des Ergebnisses, beispielsweise $x+y-x=y$, für Fließpunktzahlen nicht gelten.

Neben der bei der manuellen Überprüfung der Ergebnisse anderen Sichtweise auf die gleichen Ergebnisse ist die andere Laufzeitumgebung der wichtigste Grund dafür, daß man Programme auch nach ihrer symbolischen Ausführung noch testet, teilweise sogar die Testfälle erst durch symbolische Ausführung erzeugt.

5 Zusammenfassung

Es gibt eine Reihe von verschiedenen Ansätzen und Systemen zur symbolischen Ausführung, die jeweils einzelne der beschriebenen Ansätze zur Validation oder Verifikation unterstützen und dadurch auch zur Entdeckung unterschiedlicher Fehler geeignet sind.

Eingesetzt wir symbolische Ausführung dabei regelmäßig, um allgemeinere Ergebnisse als durch reines Testen zu bekommen, teilweise auch (bei symbolischer Ausführung von Spezifikationen), um diese Ergebnisse schon frühzeitig zu bekommen, wenn Testen im engeren Sinne noch nicht möglich ist. Symbolische Ausführung läßt sich also einordnen zwischen reinem Testen einerseits und Korrektheitsbeweisen andererseits, wobei beide Extreme stellenweise auch durch symbolische Ausführung erreicht werden.

Eine erste Abschätzung der Effektivität und Effizienz von symbolischer Ausführung ist in [How78] beschrieben. Allerdings hängen diese sehr stark von einer Reihe von sowohl technischen als auch benutzerorientierten Aspekten ab. Zur ersten Gruppe gehören dabei in erster Linie die Unterstützung aller relevanten Konstrukte der betrachteten Sprache, eine mächtige Unterstützung bei Beweisen

und bei der Lösung von Systemen von Pfadbedingungen etc., sowie eine expressive Sprache für symbolische Werte, die z.B. für viele Anwendungen Prädikate erlauben sollte. Diese Funktionen müssen aber durch den Benutzer gesteuert werden, um die gewünschten Ergebnisse tatsächlich zu erzielen, da eine vollautomatische Lösung meist prinzipiell unmöglich ist. Auch bei der Vorgabe der Vor- und Nachbedingungen oder der Fehlerbedingungen eines Programmes, der manuellen Überprüfung der Ergebnisse, oder z.T. auch der Wahl der zu betrachtenden Pfade, ist der Benutzer gefordert. Eine allgemeine Antwort auf die Frage der Effizienz und Effektivität von symbolischer Ausführung für die Validation und Verifikation von Software ist daher nicht möglich.

Literatur

[BGW82] R.M. Balzer, N.M. Goldman, and D.S. Wile. Operational specification as the basis for rapid prototyping. *ACM Sigsoft Software Engineering Notes*, 7(5):3–16, 1982.

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*. North-Holland Publ. Co., 1974.

[CH79] D. Coleman and J.W. Hughes. The clean termination of Pascal programs. *Acta Informatica*, 11:195–210, 1979.

[Coh83] Donald Cohen. Symbolic execution of the GIST specification language. In *Proc. 8th Int. Joint Conference on Artificial Intelligence '83 (IJCAI-83)*, pages 17–21, 1983.

[CR84] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation — an aid to testing and verification. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 141–166. North-Holland, 1984.

[CSB82] D. Cohen, W. Swartout, and R. Balzer. Using symbolic execution to characterize behavior. *ACM Sigsoft Software Engineering Notes*, 7(5):25–32, December 1982.

[Dav73] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80:233–269, 1973.

[Dil90] Laura K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12:643–669, 1990.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[How78] William E. Howden. An evaluation of the effectiveness of symbolic testing. *Software — Practice and Experience*, 8:381–397, 1978.

[JJLM91] C.B. Jones, K.D. Jones, P.A. Lindsay, and R.C. Moore. *mural — A Formal Development Support System*. Springer-Verlag, 1991. With contributions from J. Bicarregui, M. Elvang-Gøransson, R. Fields, R. Kneuper, B. Ritchie, A.C. Wills.

[Kne91] Ralf Kneuper. Symbolic execution: a semantic approach. *Science of Computer Programming*, 16:207–249, 1991.

[Lig90] Peter Liggesmeyer. *Modultest und Modulverifikation. State of the Art.* BI Wissenschaftsverlag, 1990.

[Swa83] William R. Swartout. The GIST behavior explainer. Research report ISI/RS-83-3, USC/Information Sciences Institute, July 1983.

[Zav84] Pamela Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.

Inhaltsverzeichnis