Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1–23
November 1991

Versuche der Compiler–Validierung

J. Blieberger, G.H. Schildt

Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

Versuche der Compiler–Validierung

J. Blieberger und **G.–H. Schildt**
Technische Universität Wien
Treitlstraße 3/4
A–1040 Wien
Österreich

Zusammenfassung. Verschiedene Verfahren der Compiler–Validierung werden gegenübergestellt. Die Spanne bewegt sich dabei von Black–Box–Validierung bis zur White–Box–Validierung. Ein halb–formales Verfahren wird vorgestellt, das die Lücke zwischen formalen mathematischen und weniger formalen Methoden schließen soll. Die Anwendungsbereiche des vorgestellten Verfahrens liegen vor allem in Bereichen der Validierung, die (bis jetzt) einer exakten mathematischen Behandlung nicht zugänglich sind. Das trifft insbesondere auf Systeme mit Multitasking zu.

Keywords. Compiler–Validierung, Black–Box–Validierung, White–Box–Validierung, Sicherheitsnachweise

1 Einleitung

Prozeßsteuerungen mit Sicherheitsverantwortung erfordern vor Inbetriebnahme die Durchführung eines Sicherheitsnachweises. Dies trifft z.B. auf chemischen Anlagen, Kernkraftwerke und automatisierte Verkehrssysteme zu. Auf der Basis der Relaistechnik war in der Vergangenheit auf der Grundlage der klassischen Fail–Safe–Technik oftmals noch ein umfassender Sicherheitsnachweis möglich, obgleich man bei komplexen Anlagen bei dieser Nachweistechnik auch schnell an Aufwandsgrenzen stieß. Vor allem der Nachweis der Vollständigkeit aller zu betrachtender Testfälle bereitete oftmals Schwierigkeiten.

Seit dem Übergang zur Mikroprozessortechnik und gleichzeitiger Abkehr von der klassischen, einkanaligen Fail–Safe–Technik, verbunden mit dem Übergang zu einer "Sicherheit per Verfahren" durch mehrkanalige Systemstrukturen mit Vergleichern bzw. Mehrheitsentscheidern entstanden eine Reihe neuer Herausforderungen [Schi80]:

- o die Überprüfung der (meist) einkanaligen Spezifikation,
- o sowie beim Übergang zu Hochsprachen die Notwendigkeit der Validierung des für die Anwendersoftware zugehörigen Compilers,
- o die Validierung des Echtzeitbetriebssystems,
- o die Validierung der Anwendersoftware.

Bezüglich der Compilervalidierung wurden für verschiedene Programmiersprachen bereits zahlreiche Validierungen unternommen. Im wesentlichen kann man drei verschiedene Arten von Compiler–Validierungen unterscheiden. Es ist aber dabei in allen Fällen zu beachten, daß über die korrekte Implementierung der Zielsprache keinerlei Aussagen gemacht werden. Die Zielsprache kann dabei sowohl eine höhere Programmiersprache *(Precompiler)*, eine Assemblersprache als auch die Maschinensprache eines Prozessors sein, dessen Korrektheitsnachweis von den Chip–Designern zu erbringen ist.

1. *Black–Box–Validierung.* Darunter versteht man die Validierung eines Compilers, wobei weder der Source– und der Objekt–Code des Compilers, noch der vom Compiler erzeugte Code inspiziert wird; es wird lediglich an Hand von Testprogrammen überprüft, ob der Compiler korrekt arbeitet und gegebenenfalls die übersetzten Programme korrekt ablaufen.

2. *White–Box–Validierung.* Hierbei wird der Source–Code, in dem der Compiler geschrieben worden ist, inspiziert und auf seine Korrektheit untersucht, d.h., es wird untersucht, ob zu einem gegebenen Source–Code der richtige Objekt–Code erzeugt wird. Eine verschärfte Art dieser Inspektion benutzt formale Methoden der Semantik–Beschreibung, um exakte mathematische Beweise bezüglich der Korrektheit des Compilers zu führen.

3. *Gray–Box–Validierung.* Darunter versteht man die Validierung eines Compilers, wobei weder der Source– noch der Objekt–Code des Compilers, jedoch der vom Compiler erzeugte Code auf seine Korrektheit überprüft wird.

In den folgenden Kapiteln werden wir für diese drei Arten der Validierung Beispiele angeben und Vor– und Nachteile der einzelnen Methoden erörtern.

2 Die Validierung von Ada–Compilern

Die Überlegungen bezüglich der Validierung von Ada–Compilern wurden bereits vor der Fertigstellung des Ada–Standards initiiert. Dabei wurden sowohl die Grenzen einer möglichen Validierung festgelegt als auch die signifikante Aspekte einer Validierung ausgearbeitet [Goo81]. Die letztgenannte Aktivität mündete darin, ein Dokument zu erstellen, das den Erstellern eines Compilers als Unterstützung dienen soll. Diese Unterlagen – der *Ada Compiler Validation Implementor's Guide* [Sof80] – ist kein ein für allemal feststehendes Dokument, sondern wächst dynamisch mit jeder neuen Implementierung und mit den dabei auftretenden Problemen.

Der Aufbau des *Implementor's Guide* ist ähnlich dem des *Ada Reference Manuals* [Ich83]. Jedes Kapitel enthält einige oder alle der folgenden fünf Unterkapiteln.

1. *Unklare semantische Beschreibungen im Ada Reference Manual.* Hier werden – oft an Hand von Beispielen – Teile des *Reference Manuals* näher erläu-

tert, deren semantische Beschreibung zu Mißverständnissen führen kann oder geführt hat. Dadurch beinhaltet dieses Unterkapitel natürlich wertvolle Hinweise, welche Testfälle für Validationszwecke verwendet werden sollten.

2. *Fehler, die zur Compilier–Zeit erkannt werden müssen.* Dieses Unterkapitel enthält kontextsensitive syntaktische und semantische Einschränkungen, die ein korrektes Programm erfüllen muß. Im einzelnen werden alle Einschränkungen aufgezählt, die nicht durch die Syntax allein erfaßt werden können. Daraus ergeben sich einerseits alle Prüfungen, die ein Ada–Compiler durchführen muß, bevor er ein Programm als korrekt übersetzt angibt, und andererseits wertvolle Hinweise für die Zusammenstellung von Validationstestfällen.

3. *Bedingungen zur Auslösung von Exceptions.* Dieses Unterkapitel beinhaltet alle Bedingungen, die erfüllt sein müssen, damit ein zu einem Compiler passendes Laufzeitsystem eine *Fehlermeldung (Exception)* auslöst, die zu einem bestimmten Sprachkonstrukt gehört. Wie im vorigen Unterkapitel folgen auch hieraus sowohl für den Compiler–Bauer als auch für den Compiler–Validierer wertvolle Hinweise.

4. *Testziele und Entwurfsrichtlinien für Testprogramme.* Hier werden die Testfälle zur Validierung spezifiziert, Probleme und Hinweise aufgezählt, die man beim Erstellen der Testfälle kennen sollte, und (falls notwendig) Programmstrukturen skizziert, um bestimmte Testziele zu erreichen. Dabei wird die Information aus den obigen drei Unterkapiteln genutzt.

   Testfälle werden dabei nicht nur für Sprachkonstrukte entworfen, deren Implementierung Probleme bereitet, sondern auch um zu verhindern, daß der Compiler Unter– oder Übermengen des Standards akzeptiert. Den letzten Fall betreffend wird vor allem geprüft, ob der Compiler fälschlicherweise reservierte Symbole anderer Sprachen akzeptiert.

5. *Inkonsistenzen und Mehrdeutigkeiten.* Hier werden Testfälle konstruiert, um festzustellen, welche Interpretation ein Compiler bezüglich Mehrdeutigkeiten des Sprachstandards gewählt hat. Die Ergebnisse solcher Testfälle führen natürlich nicht dazu, den Compiler für nicht validiert zu erklären.

Bei der Erstellung von Testfällen kann man zwei grundlegende Linien unterscheiden, die die Größe der Testfälle betreffen. Vor– und Nachteile sollen im folgenden kurz aufgezählt werden.

a. Der Vorteil weniger, dafür aber umfangreicher Testfälle besteht vor allem darin, daß für die Vorbereitung einer Validierungsprozedur und für die anschließende Analyse wenig Zeit und Arbeit erforderlich ist.

b. Andererseits besteht ein nicht unwesentlicher Nachteil darin, daß bei Auftreten eines Fehlers am Anfang eines Testfalls unter ungünstigen Umständen keinerlei

Aussagen über den Rest des Testfalls gemacht werden können.

c. Für ein kleines Testprogramm, das sich auf einen einzigen Aspekt der Sprache beschränkt, ist es viel wahrscheinlicher, korrekt übersetzt zu werden, als für ein Programm, das den gesamten Sprachumfang überprüft.

d. Neu hinzukommende Testfälle können leichter integriert werden, wenn viele, kleine Testprogramme existieren.

Für Ada wurde entschieden, möglichst kleine Testprogramme zu erstellen, wobei aber diese viele kleinen Testprogramme bei weiteren Validierungsversuchen zu größeren Einheiten zusammengefaßt werden können, um den Aufwand beim Compilieren zu verringern.

Die Testfälle für Ada sind in sechs Klassen unterteilt worden, die im folgenden erläutert werden sollen.

- Ein Testprogramm der Klasse A gilt als korrekt durchlaufen, wenn während des Compilierens keine Fehler aufgetreten sind. Obwohl diese Programme so erstellt worden sind, daß sie ablauffähig wären, werden keine Laufzeittests vorgenommen.

- Ein Testprogramm der Klasse B ist ein "falsches" Programm. Ein solches Programm besteht den Test, falls bei der Übersetzung alle absichtlich eingestreuten Fehler gefunden werden und sonst kein Fehler auftritt.

- Testprogramme der Klasse L sind Programme, deren Fehler erst beim Linken offensichtlich werden.

- Testprogramme der Klasse C sind selbstprüfende Programme, die den Test bestehen, falls sie korrekt ablaufen und keinen Fehler produzieren.

- Testprogramme der Klasse D überprüfen Beschränkungen, wie etwa die Anzahl von Identifiern oder die Anzahl von Einträgen in Libraries.

- Testprogramme der Klasse E stellen fest, wie ein bestimmter Compiler Mehrdeutigkeiten des Sprachstandards interpretiert. Dies führt zu keiner Aussage betreffend die Validierung, kann aber als Information für Benutzer herangezogen werden.

Die große Anzahl von Testfällen zur Validierung eines Ada-Compilers macht eine zumindest teilweise Automatisierung des Analysevorganges notwendig. Dies wird unter anderem dadurch erreicht, daß fehlerhafte Zeilen eines Programmes mit einem standardisierten Kommentar gekennzeichnet werden. Solche Zeilen können dann mit einem kleinem Programm automatisch gefunden und daraufhin überprüft werden, ob der Fehler gefunden wurde, falls der Compiler gefundene Fehler in das Compiler-Listing

einstreut. Auch exekutierbare Programme sind so aufgebaut, daß ihr Output automatisch auf Korrektheit geprüft werden kann.

Abschließend soll darauf hingewiesen werden, daß eine einmalige Validierung eines Ada-Compilers nicht ausreicht. Vielmehr muß sich auch ein bereits validierter Compiler in regelmäßigen Abständen neuerlich diesem Prozeß unterziehen. Dieser Umstand resultiert aus den dynamisch sich entwickelten Testprogrammen.

Betreffend unsere einleitende Klassifizierung handelt es sich daher bei der Validierung eines Ada-Compilers um eine *Black-Box-Validierung*. Ein Vorteil dieses Verfahrens ist sicherlich der relativ geringe Aufwand des Validierungsprozesses. Der Nachteil dieses Verfahrens ist, daß die dadurch gewonnene Sicherheit bezüglich der Korrektheit eine relativ geringe ist, da keinerlei Code-Inspektion stattfindet, weder in Bezug auf den Code des Compilers, noch in Bezug auf den erzeugten Code.

3 Die Validierung von Pascal-Compilern

3.1 Die PASCAL-Validation Suite

Die PASCAL-Validation-Suite ist (im wesentlichen) ein Paket aus etwa 750 Quellprogrammen der Gesellschaft für Mathematik und Datenverarbeitung (GMD). Das Hauptziel der Suite ist es, einen Compiler auf Einhaltung der Norm DIN 66256 zu prüfen. Jedes Prüfprogramm versucht, eine bestimmte Eigenschaft des Compilers oder des Laufzeitsystems zu testen, indem es entweder eine JA/NEIN-Entscheidung herbeiführt (z.B.: übersetzt/nicht übersetzt, gelaufen/nicht gelaufen) oder eine Maßzahl berechnet [Lan87].

Die so gewonnenen Informationen über den Compiler und sein Laufzeitsystem können als Grundlage für die Erstellung von Programmierrichtlinien für sicherheitstechnische Anwendungen dienen. Diese Informationen sind teilweise auch für andere Anwender des Compilers nützlich, so z.B. Maßzahlen über die Genauigkeit der Gleitkommaoperationen.

Jedes Testprogramm kann einer der folgenden acht Testklassen zugeordnet werden, wobei jede Klasse einen Aspekt der Sprache PASCAL untersucht:

- *Conformance*
  Diese Klasse prüft die Norm-Konformität (hier zu DIN 66256 [DIN85]) in einem positiven Sinn, d.h. alle Programme dieser Klasse sind in korrektem PASCAL geschrieben. Alle Testprogramme sollten daher übersetzbar und ablauffähig sein.

- *Deviance*
  Diese Klasse prüft die Norm-Konformität in einem negativen Sinn, d.h. alle Programme dieser Klasse enthalten eine Abweichung von der in der Norm DIN 66256 definierten Sprache. *Kein* Programm sollte übersetzbar sein.

- *Errorhandling*
  Diese Klasse von Testprogrammen prüft die Fähigkeiten eines Compilers bei der Fehleraufdeckung. In der DIN-Norm 66256 werden 59 (Laufzeit-)Fehler definiert, zu jedem Fehler existiert mindestens ein Testprogramm. Alle Programme sollten spätestens zur Laufzeit eine Fehlerreaktion auslösen. Die Qualität eines Compilers ist dann als höherwertig einzustufen, wenn möglichst viele Fehler bereits zur Übersetzungszeit erkannt werden.

- *Implementation-defined*
  Diese Klasse betrifft die implementierungsdefinierten Eigenschaften eines Compilers. Solche Eigenschaften können für einen Compiler spezifisch sein, sind jedoch in jedem Fall definiert (so z.B. die größte ganze Zahl, die im Prozessor darstellbar ist u.ä.). Testprogramme, die sich auf solche spezifischen Eigenschaften richten, sind notwendigerweise nicht portabel.

- *Implementation - dependent*
  Diese Klasse von Testprogrammen ermittelt die implementierungsabhängigen Eigenschaften eines Compilers. Solche Eigenschaften können für einen Compiler spezifisch sein, sind aber nicht notwendigerweise definiert, wie z.B. die Auswertung der Indices eines Arrays.

- *Quality*
  Diese Klasse ermittelt Maßzahlen für die Qualität eines Compilers wie z.B. den Genauigkeitsverlust bei der Division von Gleitpunktzahlen, Anzahl der Prozeduren, die in einer Übersetzungseinheit definiert werden können u.s.w. (Für den Sicherheitsnachweis eines Compilers sind diese Angaben jedoch ohne Bedeutung).

- *Level 1*
  Diese Klasse von Testprogrammen prüft den Level 1 der in der Norm DIN 66256 definierten Sprache, das Konzept der Conformant Arrays. Diese Klasse von Testprogrammen zerfällt in Unterklassen, nämlich die 6 vorangegangenen Klassen von Testprogrammen. Wird nur der Level 0 geprüft, beschränkt man sich üblicherweise auf Deviance-Tests.

- *Extension*
  Diese Klasse von Prüfprogrammen prüft einige übliche Erweiterungen von PASCAL-Compilern, so z.B. einen OTHERWISE-Zweig in der CASE-Anweisung (mit nur vier Testprogrammen ist diese Klasse recht klein).

Für die Validierung eines Compilers für einen bestimmten Prozessor sind für einen Sicherheitsnachweis die Error-Handling-Tests am interessantesten, so z.B.: Welche Fehler werden unter welchen Umständen *nicht* entdeckt? Aber auch die Klassen "Conformance" und "Deviance" geben wichtige Anhaltspunkte, weil dort gezielt kritische Stellen der Implementierung eines Compilers berührt werden.

Abschließend kann zu diesem Verfahren des Black-Box-Tests festgestellt werden, daß zwar die Fülle von Testprogrammen den Anwender beruhigt, für sicherheitsrelevante Anwendungen kann jedoch die Vollständigkeit der Testprogramme *nicht* nachgewiesen werden.

3.2 Kombinationsvalidierung

Im Gegensatz zur PASCAL-Validation-Suite, die einen Compiler für einen Prozessor funktionsorientiert, jedoch punktuell, prüft, macht die sogenannte *Kombinationsvalidierung* einen Ansatz in Richtung auf eine *systematische* und *vollständige* Prüfung. In unserer Klassifizierung ist dieses Verfahren eine Gray-Box-Validierung. Der grundlegende Vorteil der Kombinationsvalidierung ist ihre *Automatisierbarkeit*.

Die Kombinationsvalidierung geht davon aus, daß es gewisse *atomare* Sprachelemente (Sprachkonstrukte) in dem Sinn gibt, daß es zu jedem dieser Sprachelemente *genau* eine Folge von Maschinenbefehlen gibt. Alle Programme des Source-Bereiches entstehen durch Verschachteln und Hintereinanderschreiben der *atomaren* Sprachelemente. Unter der Annahme, daß ein Programm nur aus validierten Sprachelementen aufgebaut ist, ergibt sich, daß das Programm als Ganzes dann auch validiert ist. Allerdings muß gezeigt werden, daß ein *atomares* Sprachelement in allen Kontexten stets gleich übersetzt wird, d.h., das Verschachteln muß ein *gedächtnisfreies* Konstruktionsmittel sein.

Damit ergibt sich die folgende Vorgehensweise: Zuerst sind alle *atomaren* Sprachelemente zu identifizieren. Zu jedem dieser Sprachelemente muß ein Programm geschrieben werden, das nur dieses Sprachelement enthält. Der von dem Sprachelement erzeugte Code muß manuell validiert werden. Danach sind (möglichst automatisiert) Programme so zu erzeugen, daß das *atomare* Sprachelement in jedem erlaubten Kontext einmal auftritt. Der zuvor validierte Code ist nun (automatisch) mit dem in allen Kontexten erzeugten Code auf Identität zu vergleichen.

Einen sehr wichtigen Teil einer höheren Programmiersprache stellen die (mathematischen) Ausdrücke dar, die mit ihrer Hilfe formuliert werden können. Aber auch dieser "kleine" Teil einer Programmiersprache kann dazu benutzt werden, unendlich viele verschiedene Ausdrücke zu formulieren. Daher ist eine neuerliche Einschränkung nötig. Diese Einschränkung auf endlich viele Ausdrücke kann natürlich auf verschiedene Weise erfolgen. Zwei naheliegende sind:

1. Beschränkung auf eine endliche Anzahl von Operatoren oder Operanden und

2. Beschränkung der Höhe des Stacks, der zur Auswertung der Ausdrücke nötig ist (die Höhe des Stacks ist gleich groß wie die Schachtelungstiefe des zugehörigen Ausdrucks).

Aus der Kombinatorik weiß man, daß die Anzahl der Ausdrücke mit n Operatoren für große Werte von n asymptotisch die Form

$$c_1 \rho^n n^{-3/2}$$

hat (siehe z.B. [Mei78]), d.h., daß die Anzahl im wesentlichen exponentiell in n wächst und man im ersten Fall mit einer in n exponentiell wachsenden Anzahl von Testfällen rechnen muß.

Ebenso weiß man aus der höheren Kombinatorik, daß die Anzahl der Ausdrücke, die möglich sind, wenn man die Stack–Höhe auf h beschränkt, für große Werte von h asymptotisch

$$c_2 (c_3)^h exp(\sigma^h)$$

erfüllt [Fla84], d.h., daß die Anzahl der zu erwartenden Testfälle im zweiten Fall doppelt exponentiell mit dem Parameter h wächst.

Betrachtet man nur diese Argumente, so scheint es sinnvoll, den ersten Fall vorzuziehen. Für sicherheitstechnische Anwendungen jedoch ist die Beschränkung der Stack–Höhe viel wichtiger, da ein unbegrenzt wachsender Stack zu unvorhersehbaren Ereignissen (z.B. memory overflow) führen kann.

Gewisse Probleme bei der hier vorgestellten Kombinationsvalidierung verursacht der auch bei stark eingeschränktem Eingaberaum relativ große Rechenaufwand. In [Pla87] findet sich hierzu eine erste Rechenzeitabschätzung in Abhängigkeit von der Schachtelungstiefe S. Es wurde abhängig von S die Anzahl der erforderlichen Testprogramme berechnet. Nach dem oben dargestellten Verfahren der Kombinationsvalidierung konnten wichtige Teilbereiche für PASCAL 86 auf einem Intel–Processor 80186 als Target–Prozessor validiert werden, da die vorliegende Aufgabe parallelisierbar ist.

Es ist also zu zeigen, daß der Compiler in jedem Fall einen Ausdruck immer gleich übersetzt und nicht äußere Umstände – eine Art *Gedächtnis* – die Codegenerierung beeinflussen. Dies kann nur durch Überprüfung aller Kombinationen von Verschachtelungen bis zu einer bestimmten Schachtelungstiefe erfolgen. Die große Anzahl von Möglichkeiten bedingt, daß dies nur maschinell durchführbar ist. Folgende Tools sind hierfür erforderlich:

- ein Programm, das es ermöglicht, alle Kombinationen von Verschachtelungen aus einer vorgegebenen Menge von Ausdrücken zu erzeugen und als compilierbares, syntaktisch korrektes PASCAL–Programm zu hinterlegen.

- ein Programm, das in dem Listing eines compilierten Programmes ein bestimmtes Statement sucht (durch Kommentar markiert), dessen Nummer ermittelt und im Assembler–Code für dieses Programm die zugehörigen Anweisungen isoliert.

- o ein Programm, das den gefundenen Assembler–Abschnitt mit dem hinterlegten vergleicht.

Das vorgestellte Verfahren betrachtet nur die Abhängigkeit von der *formalen* Umgebung einer Anweisung, nicht aber der *semantischen*. So wird zwar festgestellt, daß z.B. ein **if–then–else**–Konstrukt immer in der gleichen Weise übersetzt wird, es wird jedoch nicht festgestellt, ob eine Variable im zu testenden Ausdruck nicht schon vorher in den vorangegangenen Konstrukten verwendet bzw. modifiziert wurde. Wollte man diese Möglichkeit noch überwachen wollen, so würde sich die Anzahl der zu generierenden Testprogramme nochmals erheblich erhöhen. Die so entstehende Komplexität würde einem Sicherheitsnachweis entgegenwirken.

Alle Library–Funktionen, die vom erzeugten Objekt–Code aufgerufen werden, sind aufgrund ihres disassemblierten Codes zu prüfen, als wären sie Bestandteil des erzeugten Codes. Betriebsarten des Compilers, die Library–Funktionen vermeiden, sind zu bevorzugen, da die Validierung aufwendig und bei neuen Compilerversionen nicht reproduzierbar ist. Die Verwendung von Library–Funktionen einschließlich Ein–/Ausgabe in der Source ist bis auf Ausnahmefälle zu unterbinden.

Nach den ersten Erfahrungen ist es schwierig, den gesamten Sprachumfang zu validieren. Man muß für den Anwender den Sprachumfang einschränken und weitere beschränkende Parameter finden, so z.B. die maximale Schachtelungstiefe. Weiter muß man sich auch bei den Sprachkonstrukten beschränken, also bei Anweisungen und Ausdrücken.

In [Pla87] finden sich Abschätzungen über die Anzahl der Testprogramme bei der Kombinationsvalidierung. Wird die maximale Schachtelungstiefe von PASCAL–Ausdrücken mit S bezeichnet, so ergeben sich aufgrund der Sprachbeschreibung und unter der Annahme von vier Adressierungsarten ca. $400 \cdot 35^S$ Ausdrücke. Davon sind viele semantisch sinnlos, sodaß hier noch ein Potential zur Verringerung der zu generierenden Testprogramme vorhanden ist. Mit P soll die maximale Anzahl von Paramentern einer Prozedur bezeichnet werden. Dann ergeben sich 30^P verschiedene Möglichkeiten eines Aufrufs einer P–parametrigen Prozedur. Mit Z als maximale Schachtelungsstufe von PASCAL–Anweisungen erhält man $6 \cdot 5^Z$ Kombinationen. Damit ergeben sich insgesamt

$$6 \cdot 5^Z + 30^P + 400 \cdot 35^S$$

zu behandelnde Testprogramme. Mit S=3, P=4 und Z=7 entsteht ein durchaus brauchbarer Sprachumfang. Mit diesen Werten und einer durchschnittlichen Bearbeitung von 1000 Testprogrammen je Stunde folgt eine Rechenzeit von ca. 2,1 Jahren.

4 Die Validierung eines Compilers mit formalen mathematischen Methoden

Bei diesem Verfahren handelt sich bezugnehmend auf unsere einleitende Klassifizierung um eine *White–Box–Validierung*. Genaugenommen wird bei der Realisierung des Compilers, ja eigentlich schon bei der Spezifikation der Semantik der Sprache

eine nachfolgende Validierung berücksichtigt.

Wir wollen diese Art der Validierung kurz an Hand eines Beispieles erläutern [Ste91]. Die grundlegende Idee dabei ist, *denotationelle Semantik* [Gor79] zur Beschreibung der Semantik der Sprache zu verwenden, die anschließend zum Beweis der Korrektheit eines Compilers herangezogen wird.

Mittels denotationeller Semantik ist es möglich, eine Sprache zu definieren, indem man jedem Sprachkonstrukt einen mathematischen Wert – eine "Bedeutung" – zuordnet. Dadurch kann man die abstrakte, maschinenunabhängige Bedeutung von Programmen "berechnen".

Für die Implementierung eines Compilers ist nicht nur die Spezifikation der Quellsprache erforderlich, sondern auch die der Zielsprache. Es ist die Aufgabe eines Compilers, höhere Sprachkonstrukte, wie etwa

if <boolean_expr> **then** <then_stmts> **else** <else_stmts>

in Zielsprachkonstrukte, sogenannte *Schablonen (engl. templates)*, zu übersetzen, wie zum Beispiel

```
    <boolean_expr, label1>
    <then_stmts>
    jump label2
label1:
    <else_stmts>
label2:
```

Dabei ist zu beachten, daß die Teile innerhalb der spitzen Klammern rekursiv ähnlich zu behandeln sind. Wenn man nun sowohl für die Quellsprache, als auch für die Zielsprache eine Beschreibung solcher Konstrukte in denotationeller Semantik zur Verfügung hat, kann man für beide ihre Bedeutung berechnen. Erhält man übereinstimmende Ergebnisse, so ist der Übersetzungsprozeß korrekt.

Wir wollen nun kurz an Hand des obigen Beispiels zeigen, wie so ein Beweis aussieht. Dazu werden wir die Formalismen der denotationellen Semantik verwenden, ohne sie genauer zu definieren. Wir hoffen, daß die Umformungen trotzdem verständlich sind. Wer an den Formalismen interessiert ist, sei auf [Gor79] verwiesen. Außerdem werden wir die Semantik der Zielsprache nicht näher spezifizieren; auch sie sollte intuitiv leicht faßbar sein. Es wird versucht werden, durch verschiedene textuelle Darstellung Klarheit zu schaffen bezüglich der Zugehörigkeit von Ausdrücken zu ihren Sprachen, z.B.:

Sprachmittel	Sprache
if ... then ... else	denotationelle Semantik
if ... then ... else	Quellsprache
if ... then ... else	Zielsprache

Wir geben eine informelle Beschreibung einiger wichtiger Bezeichnungen der denotationellen Semantik:

E	ordnet einem Ausdruck seine Bedeutung zu
C	ordnet einer Anweisung seine Bedeutung zu
P	ordnet einem Programm seine Bedeutung zu
ϕ	sind Sprungziele in der Zielsprache
ρ	stellt Umgebung dar
σ	stellt den Speicher dar
Σ	stellt den Zustand einer Berechnung dar

Die semantische Beschreibung des **if–then–else**–Konstruktes lautet

$$\mathbf{C}\,[\,\textbf{if}\ \varepsilon\ \textbf{then}\ \chi_1\ \textbf{else}\ \chi_2\,]\,\rho\Sigma = (\,\text{if}\ \mathbf{E}\,[\,\varepsilon\,]\,\rho\Sigma = \textit{True}\ \text{then}\ \mathbf{C}\,[\,\chi_1\,]\ \text{else}\ \mathbf{C}\,[\,\chi_2\,]\,)\,\rho\Sigma.$$

Das entsprechende Konstrukt in der Zielsprache sieht folgendermaßen aus

ϕ_0: if $\neg\,\varepsilon$ then goto ϕ_1 else skip
  χ_1
  goto ϕ_2
ϕ_1: χ_2
ϕ_2: skip

Um zu zeigen, daß dies eine korrekte Übersetzung darstellt, müssen wir zeigen, daß die beiden Ausdrücke dieselbe Bedeutung haben. Wir schreiben statt

$\mathbf{P}\,[\phi_0$: if $\neg\,\varepsilon$ then goto ϕ_1 else skip; χ_1; goto ϕ_2
  ϕ_1: χ_2
  ϕ_2: skip $]\,\rho\Theta\sigma = \Theta_0\,\sigma$

die folgenden Ausdrücke

$\Theta_0 = \mathbf{P}\,[$ if $\neg\,\varepsilon$ then goto ϕ_1 else skip; χ_1; goto $\phi_2\,]\,\rho_1\Theta_1$
$\Theta_1 = \mathbf{P}\,[\,\chi_2\,]\,\rho_1\Theta_2$
$\Theta_2 = \mathbf{P}\,[$ skip $]\,\rho_1\Theta$
$\rho_1 = \rho\,[\,\Theta_0/\phi_0, \Theta_1/\phi_1, \Theta_2/\phi_2\,]$

Daher ist die gesamte Bedeutung des Zielsprachenkonstruktes $\Theta_0\,\sigma$. Herausheben des ersten sequentiellen Statements ergibt:

$$\Theta_0\,\sigma = \mathbf{P}\,[\,\text{if}\ \neg\,\varepsilon\ \text{then goto}\ \phi_1\ \text{else skip}\,]\,\rho_1\ \{\mathbf{P}\,[\,\chi_1\,;\ \text{goto}\ \phi_2\,]\,\rho_1\Theta_1\}\,\sigma$$

Herausheben des zweiten führt zu:

$$\Theta_0\sigma = \mathbf{P}\,[\,\text{if } \neg\,\varepsilon \text{ then goto } \phi_1 \text{ else skip }]\,\rho_1 \{\,\mathbf{P}\,[\,\chi_1\,]\,\rho_1\{\,\mathbf{P}\,[\,\text{goto } \phi_2\,]\,\rho_1\Theta_1\}\}\,\sigma$$

Setzt man nun für das goto ein, erhält man:

$$\Theta_0\sigma = \mathbf{P}\,[\,\text{if } \neg\,\varepsilon \text{ then goto } \phi_1 \text{ else skip }]\,\rho_1 \{\,\mathbf{P}\,[\,\chi_1\,]\,\rho_1\{\,\rho_1\,[\,\phi_2\,]\,\}\}\,\sigma$$

Substituiert man für den Label ϕ_2, so ergibt das:

$$\Theta_0\sigma = \mathbf{P}\,[\,\text{if } \neg\,\varepsilon \text{ then goto } \phi_1 \text{ else skip }]\,\rho_1 \{\,\mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\}\,\sigma$$

Wenn man das **if–then–else** herauszieht, erhält man

$$\Theta_0\sigma = (\,\text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,]\,\sigma \text{ then } \mathbf{P}\,[\,\text{goto } \phi_1\,] \text{ else } \mathbf{P}\,[\,\text{skip }]\,)\,\rho_1\{\,\mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\}\,\sigma$$

Indem man nun den Teil, der dem **if–then–else** folgt, in die Klammer hineinzieht, ergibt sich:

$$\Theta_0\sigma = \text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,]\,\sigma \text{ then } \mathbf{P}\,[\,\text{goto } \phi_1\,]\,\rho_1\{\mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\}\,\sigma \text{ else } \mathbf{P}\,[\,\text{skip }]\,\rho_1\{\,\mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\}\,\sigma$$

Jetzt ersetzen wir noch im einen Zweig das goto und im anderen das skip und erhalten:

$$\Theta_0\sigma = \text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,]\,\sigma \text{ then } \rho_1\,[\,\phi_1\,]\,\sigma \text{ else } \mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\sigma$$

Indem wir noch die Bedeutung des Labels ϕ_1 einsetzen, ergibt sich:

$$\Theta_0\sigma = \text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,]\,\sigma \text{ then } \Theta_1\sigma \text{ else } \mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\sigma$$

Nun setzen wir die Bedeutung des Teiles ein, der dem **if–then–else** folgt, und erhalten:

$$\Theta_0\sigma = \text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,]\,\delta \text{ then } \mathbf{P}\,[\,\chi_2\,]\,\rho_1\Theta_2\sigma \text{ else } \mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2\sigma$$

Wenn wir annehmen, daß unsere Quellsprache keine **gotos** aus einem **if–then–else** heraus erlaubt, dann kommen in χ_1 und in χ_2 keine Labels ϕ_0, ϕ_1 und ϕ_2 vor. Wenn dann der Compiler bei der Übersetzung von **if–then–else**–Konstrukten immer neue Namen für die Labels erzeugt, folgt, daß $\mathbf{P}\,[\,\chi_1\,]\,\rho_1\Theta_2 = \mathbf{P}\,[\,\chi_1\,]\,\rho\Theta$, usw., und daher

$$\Theta_0\sigma = (\,\text{if } \mathbf{E}\,[\,\neg\,\varepsilon\,] \text{ then } \mathbf{P}\,[\,\chi_2\,] \text{ else } \mathbf{P}\,[\,\chi_1\,]\,)\,\rho\Theta\sigma$$

Wenn wir jetzt noch die Negation eliminieren und die Programmzweige entsprechend vertauschen, erhalten wir das Endresultat:

$$\Theta_0\sigma = (\text{ if } \mathbf{E}[\,\varepsilon\,] \text{ then } \mathbf{P}[\,\chi_1\,] \text{ else } \mathbf{P}[\,\chi_2\,])\,\rho\Theta\sigma$$

Diese Art von Beweisführung kann im Prinzip für die gesamte Syntax und Semantik einer Programmiersprache durchgeführt werden. In [Ste91] wird auch gezeigt, wie diese Art der Semantikbeschreibung in ein Prolog–Programm umgesetzt und dazu verwendet werden kann, gleichzeitig einen Interpreter für die Sprache zu gewinnen.

Dieses Verfahren der Compiler–Validierung bietet einerseits den Vorteil einer exakten Korrektheitsüberprüfung, hat aber andererseits auch einige Nachteile.

1. Das Verfahren ist relativ aufwendig und möglicherweise nur von Experten in denotationeller Semantik durchführbar.

2. Das in [Ste91] dargestellte Verfahren der Compiler–Erstellung unter Zuhilfenahme von Prolog erscheint zweifelhaft, da zum Korrektheitsbeweis des Compilers auch der Nachweis der Korrektheit des Prolog–Compilers oder Prolog–Interpreters nötig ist.

3. Man kann zwar mittels denotationeller Semantik alle Sprachkonstrukte herkömmlicher, prozeduraler Programmiersprachen beschreiben, doch versagt das Verfahren bei Sprachen, die stark echtzeitorientiert sind, wie etwa Ada oder PEARL, da es *nicht* geeignet ist, auf parallelen Programmfluß einzugehen.

5 Schablonenbasierte Validierung

Im Gegensatz zum im vorigen Kapitel vorgestellten Verfahren der Compiler–Validierung, das nur für Quellsprachen verwendet werden kann, deren "Bedeutung" für alle Anweisungen mittels denotationeller Semantik festgelegt ist, schlagen wir vor, Ideen dieses Verfahrens zu übernehmen, und dadurch Validierungen einerseits auch für Sprachen durchführen zu können, für die keine exakte semantische Beschreibung existiert, und andererseits die Möglichkeit zu schaffen, jene Sprachkonstrukte, deren Semantik (bis jetzt) keiner formalen mathematischen Behandlung zugänglich ist, zu validieren.

Dieses Verfahren benötigt folgende Voraussetzungen:

1. Der Compiler produziert nicht nur Objekt–Code sondern auch ein "lesbares" Listing desselben.

2. Vom Compiler–Bauer wird erwartet, daß er Schablonen in der – im Abschnitt 4 eingeführten – Form für die gesamte Quellsprache spezifiziert.

3. Wenn der Benutzer im Quell–Code eines Programmes ein oder mehrere Anweisungen kennzeichnet (etwa durch Pragmas), so markiert der Compiler in eindeutiger Weise die Anweisungen im Objekt–Code entsprechend der Schablonen aus Punkt (2).

4. Der Compiler darf keinerlei Optimierungen vornehmen.

Die im Compiler für die Punkte (1) und (3) zuständigen Code–Teile müssen "von Hand" validiert werden. Da dies keine allzu komplizierten Vorgänge sind, sollte der dafür erforderliche Aufwand nicht allzu groß sein. Sollten aber in diesen Code–Teilen im Laufe der Validierungsprozedur noch Fehler vorhanden sein, so ist es sehr wahrscheinlich, daß sie während der Validierung zu Tage treten.

Der vierte Punkt ist wahrscheinlich der, der von seiten der Compiler–Bauer am meisten Widerstand entgegengebracht werden wird, da sie um die Performance ihres Produktes fürchten werden, andererseits ist für sicherheitsrelevante Anwendungen die Performance eines Programmes hinter den Sicherheitsaspekten einzuordnen.

Das Verfahren wird folgenderweise durchgeführt:

1. Der Validierer erstellt ein oder mehrere Testprogramme, die den zu validierenden Sprachumfang möglichst abdecken. Dabei werden sowohl syntaktisch korrekte als auch syntaktisch inkorrekte Programme erzeugt. Syntaktisch falsche Testprogramme sollen entweder vom Compiler, oder vom Linker erkannt werden (vgl. das für die Sprache Ada angewandte Verfahren (Kapitel 2)).

2. Im Quell–Code jedes syntaktisch korrekten Testprogrammes kennzeichnet er eine oder mehrere Anweisungen.

3. Er compiliert das Testprogramm (der Compiler darf keine Fehler melden!).

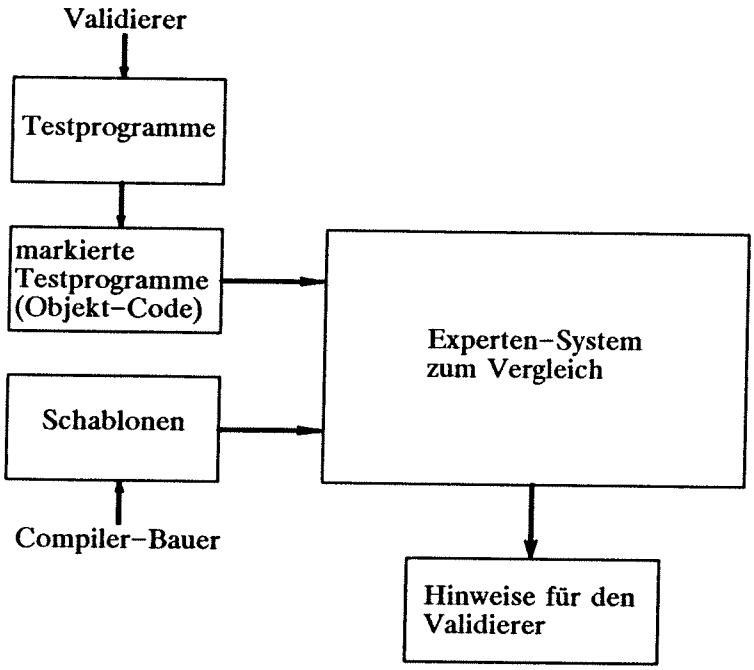
4. An Hand der vordefinierten Schablonen, des Quell– und Objekt–Codes überprüft er die Korrektheit des Übersetzungsvorganges.

Es folgen nun Gedanken, wie dieses Verfahren zu automatisieren ist und welche Teile nicht automatisiert werden können bzw. sollten:

1. Die Erstellung der Testprogramme muß "von Hand" durchgeführt werden; dies bezieht sich vor allem auch auf syntaktisch falsche Programme; diese sind aber ohnehin sprachspezifisch zu erstellen und daher nicht vom zu validierenden Compiler abhängig.

2. Ein Software–Werkzeug, das systematisch eine Anweisung nach der anderen oder zufällig mehrere Anweisungen kennzeichnet und anschließend den Compiler aktiviert, ist sicherlich leicht realisierbar.

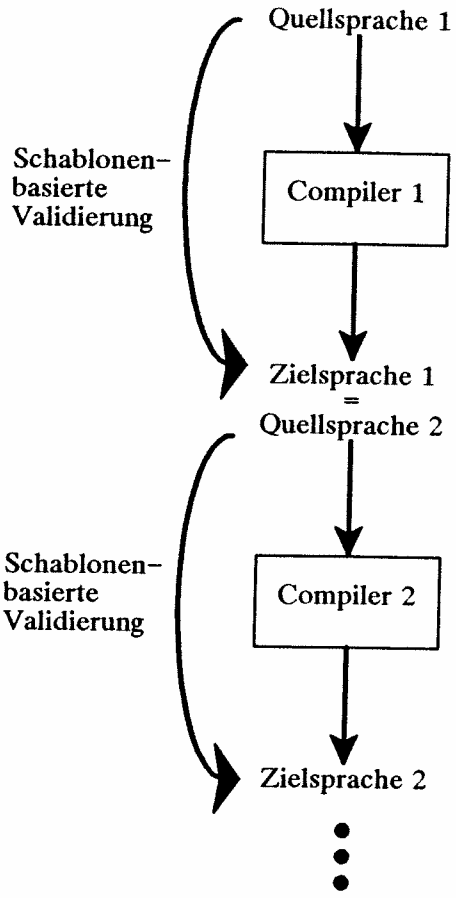
3. Die Überprüfung der Korrektheit der "Schablonenstruktur" im Objekt-Code ist sicherlich zum Teil ebenfalls automatisierbar. Dabei ist insbesondere darauf zu achten, daß ein solches Werkzeug (eine Art Experten-System) alle von ihm nicht durchgeführten Überprüfungen selbsttätig meldet und dadurch die vom Validierer zu leistende Arbeit möglichst eingrenzt.



Experten-System zum Vergleich von markiertem Objekt-Code und Schablonen

Abschließend noch Bemerkungen zum Thema Zwischen-Code-Erzeugung von Compilern. Es ist allgemein übliche Praxis, daß Compiler als Objekt-Code nicht direkt Maschinen-Code oder Assembler-Sprachkonstrukte erzeugen, sondern die Quellsprache zunächst auf eine andere (höhere) Sprache abbilden. Man vergleiche etwa den von manchen Pascal-Compilern erzeugten P-Code (z.B. [Pem82]), der eine Stack-Maschine zur Realisierung erfordert, oder den bei der Firma WERUM für die Sprache PEARL-90 in Entwicklung befindlichen Compiler, der die Quellsprache in C übersetzt [War90]. Allgemein kann man sagen, daß in solchen Fällen die Zielsprache

ein weniger hohes Niveau haben wird als die Quellsprache.



Anwendbarkeit des Verfahrens bei Zwischen-Code-Erzeugung

Die Anwendbarkeit unseres Verfahrens bleibt auch in diesen Fällen voll gegeben, ja es kann sogar für die nächste Stufe im Übersetzungsprozeß erneut eingesetzt werden, sofern der dafür vorgesehene Compiler die von uns vorausgesetzten Bedingungen erfüllt, z.B. Markierung von Anweisungen im Objekt-Code und Bereitstellung der benötigten Schablonen. Man kann dabei erwarten, daß die Syntax einfacher und die Anzahl der Schablonen weniger ist als bei der ursprünglichen Quellsprache.

Darüberhinaus bedeutet unser Verfahren eine wesentliche Arbeitsersparnis, da nur Teile eines Compilers neu validiert werden müssen, falls z.B. der Compiler auf einen neuen Prozessor portiert werden soll.

Der wesentliche Vorteil unseres Verfahrens im Vergleich zu dem im vorigen Kapitel besteht vor allem darin, daß es ohne die Verwendung einer Sprache zur Beschreibung der Semantik auskommt. Dies stellt sicherlich in Hinsicht der Validierung

eines Compilers für eine höhere Programmiersprache dann eine große Erleichterung dar, wenn für die Sprache zwar eine natürlichsprachliche Beschreibung der Semantik existiert (z.B. Ada), aber keine formale, wie etwa in Form von denotationeller Semantik. Andere Semantik-Beschreibungssprachen, wie etwa attributierte Grammatiken (z.B. DIN 66253, Programmiersprache PEARL [DIN82]), müßten erst an das oben dargestellte Verfahren angepaßt werden.

6 Zusammenfassung

Abschließend stellen wir das im Kapitel 5 vorgestellte den anderen erläuterten Verfahren gegenüber. Folgende Vorteile sind zu erwähnen:

1. Das Verfahren ist formaler als das in Kapitel 2 vorgestellte.

2. Da es nicht so formal ist, wie das im Kapitel 4, ist es auch von Personen durchführbar, die keine Experten in denotationelle Semantik sind.

3. Es kann auch dann durchgeführt werden, wenn keinerlei formale Semantik-Beschreibung der Quellsprache vorliegt.

4. Es kann für Sprachkonstrukte verwendet werden, für die (bis jetzt) keine formalen mathematischen Methoden zur Validierung existieren, z.B. Multitasking.

5. Es zwingt den Compiler-Bauer durch die an den Compiler gestellten Anforderungen zur Auseinandersetzung mit den Validierungskonzepten und erhöht die Qualität des Produktes dadurch, daß es eine Testumgebung erzwingt.

Als Nachteile wären zu nennen:

1. Der Verzicht auf Compiler-Optimierung aufgrund sicherheitsrelevanter Argumente führt zu Performance-Einbußen.

2. Der Gewinn an Sicherheit ist gegenüber dem Verfahren aus dem vorigen Kapitel 4 weitaus geringer, da man sich auf eine begrenzte Anzahl von Testfällen stützt und nicht allgemein gültige Aussagen gewinnt. Andererseits sind (bis jetzt) formale Methoden für Multitasking nicht vorhanden.

3. Das Verfahren setzt Bedingungen an den Compiler voraus, die nur dann vernünftig berücksichtigt werden können, wenn der Übersetzer erst entwickelt wird oder wenn intensiv mit der entwickelnden Institution zusammengearbeitet werden kann.

4. Der Automatisierungsgrad (vor allem des angedeuteten Experten-Systems) ist nicht genau festlegbar, da er von Compiler zu Compiler sehr unterschiedlich sein kann und von der Zielsprache abhängt.

Was die Validierung von Programmen, die aus mehreren parallelen Prozessen bestehen, mit formalen mathematischen Methoden betrifft, so kann man festhalten, daß die bisher angewandten Methoden nur dann vernünftig verwendbar sind, wenn einerseits Prozesse Sprachelemente sind, und andererseits Sprachmittel zur Kommunikation und Synchronisation von Prozessen existieren, die zur semantischen Analyse geeignet sind. Man vergleiche z.B. [Dil90], wo selbst die in Ada vorhandenen Mechanismen zum Rendezvous von Tasks noch beschränkt werden, um formale semantische Methoden anwenden zu können. Einer der Gründe, warum eine solche Beschränkung notwendig ist, ist, daß ansonsten in eine entsprechende Validierung das Scheduling des gesamten Systems miteinbezogen werden muß oder alle möglichen zeitlichen Überlappungen der Prozesse betrachtet werden müssen. Vor allem in bezug auf Echtzeitsysteme stellt die Abhängigkeit vom Scheduling des Gesamtsystems, obwohl es einige ermutigende, wahrscheinlichkeitstheoretische Ergebnisse gibt (vgl. [Bli91], [Sch91]), ein wirkliches Problem dar.

Zusammenfassend kann man sagen, daß für Sprachen, die für die Kommunikation und Synchronisation von Prozessen nur Semaphor–Variable vorsehen, in absehbarer Zeit nicht damit gerechnet werden kann, daß Fortschritte bei der Validierung von parallelen Programmen mit formalen, mathematischen Methoden erzielt werden.

Literatur

[Bli91] Blieberger, J., Schmid, U., *Preemptive LCFS Scheduling in Hard Real–Time Applications,* wird erscheinen in 'Performance Evaluation', 1991

[DIN82] DIN 66253, Teil 2, *Programmiersprache PEARL, Full PEARL,* Beuth Verlag GmbH, Berlin, Okt. 1982

[DIN85] DIN 66256, *Programmiersprache Pascal,* Beuth Verlag GmbH, Berlin, Jan. 1985

[Dil90] Dillon, L.K., *Using Symbolic Execution for Verification of Ada Tasking Programs,* ACM Trans. on Prgr. Lang. and Systems, Vol. 12, No. 4, Oct. 1990, p. 643–669

[Fla84] Flajolet, P., Odlyzko, A.M., *Limitdistributions for coefficients of iterates of polynomials with applications to combinatorial enumerations,* Math. Proc. Camb. Phil. Soc., 1984, Vol. 96, p. 237–253

[Goo81] Goodenough, J.B., *The Ada Compiler Validation Capability,* Computer, June 1981, p. 57–64

[Gor79] Gordon, M.J.C., *The Denotational Description of Programming Languages – An Introduction,* Springer, New York, 1979

[Ich83] Ichbiah, J.D, et.al., *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD 1815A, January 22, 1983

[Lan87] Lange, H., Hahn, W., *Kombinationsvalidierung,* (interner technischer Bericht), Braunschweig, 1987

[Mei78] Meir, A., Moon, J.W., *On the Altitude of Nodes in Random Trees,* Canad. J. Math. 30, 1978, p. 997-1015

[Pem82] Pemberton, S., Daniels, M., *Pascal Implementation: The P4 Compiler,* John Wiley & Sons, New York, 1982

[Pla87] Platsatoura, E., *Einfluß der Schachtelungstiefe und der Adressierungsarten auf die Anzahl der Testfälle bei der Validierung eines PASCAL-Compilers,* Diplomarbeit, TU Braunschweig, 1987

[Sch91] Schmid, U., Blieberger, J., *Some Investigations on FCFS Scheduling in Hard Real-Time Applications,* wird erscheinen in 'Journal of Computer and System Sciences', 1991

[Schi80] Schildt, G.H., *Grundlagen für Vergleicher mit Sicherheitsverantwortung,* Siemens-Forschungs- und Entwicklungsberichte, Bd. 9, 1980, Nr. 6, p. 347-353

[Sof80] SofTech Inc., *Ada Compiler Validation Implementor's Guide,* 1067 - 2.3, Contract Number MDA 903-79-C-0687, October 1, 1980

[Ste91] Stepney, S., Whitley, D., Cooper, D., Grant, C., *A Demonstrably Correct Compiler,* Formal Aspects of Computing, 1991, 3, p. 58-101

[War90] Warzawa, M., Kneuer, E., *Neue Implementierungswege mit PEARL 90,* Proceedings: PEARL 90 Workshop über Realzeitsysteme, Boppard, Nov. 1990