Informatik Forsch. Entw. (1994) 9: 192–200

Informatik
Forschung und Entwicklung

Korrekte Software: Nur eine Illusion?

Maritta Heisel[1] **und Debora Weber-Wulff**[2]

[1] Technische Universität Berlin, FB Informatik, Franklinstraße 28–29, D-10587 Berlin
[2] Technische Fachhochschule Berlin, FB Informatik, Luxemburger Straße 10, D-13353 Berlin

Eingegangen am 13. September 1993/Angenommen am 8. Juni 1994

Zusammenfassung. Es wird oft behauptet, daß es zwar eine nette, akademische Übung sei, die Korrektheit von kleinen Programmen nachzuweisen, es sei aber eine Illusion, daß korrekte Software weite Verbreitung in der Industrie erfahren könne. In diesem Artikel wollen wir uns mit dieser Illusion „Korrektheit" auseinandersetzen und zeigen, warum wir der Meinung sind, daß Korrektheit eine kommende Realität ist.

Schlüsselwörter: Programmverifikation, Softwaretechnik, Schließen über Programme, Deduktion und Automatisches Beweisen, Automatische Programmierung

Abstract. It has often been said that proving the correctness of small programs is a nice academic exercise, but that it is illusory to expect it to be widely accepted in industry. In this article we discuss the so-called illusion of correctness and explain why we are of the opinion that correctness will soon be much more widely used.

Key words: Program verification, software engineering, reasoning about programs, deduction and theorem proving, automatic programming

CR Subject Classification: D.2.4, D.2, F.3.1, I.2.3, I.2.2

Wir wollen der Frage nachgehen, wie das scheinbar illusorische routinemäßige Führen von Korrektheitsnachweisen für Software in der industriellen Praxis erreicht werden kann. Zunächst beschäftigen wir uns mit den gängigsten Argumenten für die Unmöglichkeit der Produktion von beweisbar korrekter Software. Im Anschluß zeigen wir, daß Korrektheit für sichere Software unabdingbar ist, obwohl ein Korrektheitsbeweis nicht alle Probleme lösen kann. Wir stellen einige Thesen zur praktischen Realisierbarkeit von Korrektheitsnachweisen auf. Insbesondere werden Anforderungen an Werkzeuge aufgestellt, die Entwicklerinnen und Entwickler von Software beim Korrektheitsbeweis unterstützen sollen. Zum Schluß diskutieren wir, warum die genannten Voraussetzungen technisch und wirtschaftlich gesehen auch hinreichend sind, selbst wenn die Verwirklichung des Zieles noch einiges an Zeit und Geld in Anspruch nehmen wird.

1 Die Illusion: Korrektheit kann es nicht geben

Es gibt viele Berichte in der Presse – Fachpresse wie populärwissenschaftliche Zeitschriften – die etwas schadenfroh berichten, wie Softwareausfälle unangenehme oder sogar tödliche Folgen haben können. Peter G. Neumann, der Herausgeber der ACM SIGSOFT „Software Engineering Notes", stellt in seiner Kolumne „Risks to the Public" seitenweise die Folgen fehlerhafter Systeme vor. Von Problemen mit rechnergesteuerten medizinischen Systemen über Probleme mit rechnergesteuerten Flugleitsystemen in neueren Flugzeugen bis hin zu Systemen, die einfach abstürzen: Wir sind von unsicheren Systemen umgeben. Ein besonders eklatantes Beispiel sind die Therac-25-Unfälle [24], wo fehlerhafte Software mehrere Menschen das Leben kostete.

Seit Jahren wird ein erbitterter Streit über die Frage der Realisierbarkeit von korrekter Software ausgetragen. DeMillo, Lipton und Perlis [13] haben schon 1979 in Frage gestellt, daß überhaupt die Korrektheit nichttrivialer Programme nachgewiesen werden kann. Sie verstehen den Prozeß des Beweisens (übertragen aus dem mathematischen Kontext) als einen sozialen Prozeß des Überzeugens, nicht als eine streng logische Folgerungskette.

James Fetzer [16] entfachte einen wahren Proteststurm mit seinem provokativen Artikel *„Program Verification: The Very Idea"*. Daraufhin meldeten sich sehr viele Wissenschaftlerinnen und Wissenschaftler, die nicht nur glauben, daß Verifikation möglich sei, sie betreiben sie schon – und das mit gutem Erfolg. Mit vielen Verifikationssystemen wurden bereits beachtliche Korrektheitsnachweise geführt, siehe z. B. [1, 3, 8].

Allerdings ist es trotz dieser im Prinzip ermutigenden Ergebnisse bisher nicht gelungen, Korrektheitsnachwei-

se in größerem Umfang bei der industriellen Produktion von Software einzusetzen. Die Gründe dafür sind vielfältig:

- Beispielsweise ist die Produktion von nachweislich korrekter Software teurer als die bisher übliche Funktionsüberprüfung durch Testen. Wie wir noch ausführen werden, ist dies allerdings eine kurzsichtige Betrachtungsweise, da außer den Produktionskosten auch Wartungskosten entstehen. Seit 1990 besteht außerdem das Risiko von Regreßansprüchen aufgrund von Softwarefehlern (Abschn. 4).
- Erhöhte Produktionskosten entstehen dadurch, daß zum Nachweis der Korrektheit von Software *formale Methoden* eingesetzt werden müssen, da ein „sozialer Prozeß des Überzeugens" die Möglichkeit falscher Beweise nicht ausschließt. Mit einem streng formalen Vorgehen läßt sich die Gefahr fehlerhafter Beweise dagegen auf ein Minimum reduzieren (Abschn. 2).
- Hinzu kommt, daß die mit der Softwareentwicklung betrauten Personen für den Umgang mit formalen Methoden nicht ausgebildet sind und daß es leider noch keine einfach einsetzbaren Werkzeuge für die Erstellung von nachgewiesen korrekter Software gibt.

Die erhöhten Softwareproduktionskosten aufgrund des Einsatzes formaler Methoden, die mangelnde formale Ausbildung von Softwareentwicklerinnen und -entwicklern sowie das Fehlen geeigneter Werkzeuge sind also entscheidende Ursachen dafür, daß korrekte Software höchstens eine akademische Übung, in der industriellen Praxis jedoch zur Zeit in der Tat noch eine Illusion ist.

Bevor wir allerdings darangehen zu überlegen, wie hier Abhilfe geschaffen werden kann, müssen wir uns darüber klar werden, was ein Korrektheitsbeweis leisten kann und was nicht.

2 Korrektheit macht nicht glücklich, aber sie beruhigt

In der Praxis hat jedes Programm Fehler. Wenn durch einen Fehler eine Situation entstehen kann, in der das Risiko eines Unfalles gegeben ist, von dem eine Gefahr für Leben oder Eigentum entstehen kann, sprechen wir von sicherheitskritischer Software. Die Steuerungen von Eisenbahnübergängen, Kernkraftwerken oder auch Flugsicherungssysteme gehören zu dieser Kategorie. Eine weitere Art von Softwaresystemen, die als sicherheitskritisch betrachtet werden müssen, sind solche, bei denen ein unberechtigter Zugriff auf Daten und/oder Programme unbedingt ausgeschlossen werden muß. Die Anwender von Software im sicherheitskritischen Bereich stellen inzwischen sehr hohe Anforderungen u. a. an die Korrektheit ihrer Systeme.

In zunehmendem Maße verbreitet sich in der Bundesrepublik Deutschland die Erkenntnis von der Notwendigkeit sicherer Software. Die Bundesregierung hat z. B. ein Bundesamt für Sicherheit in der Informationstechnik gegründet und es damit beauftragt, Sicherheitskriterien für die Informationstechnik zu entwickeln [39]. Deren oberste Qualitätsstufen (Q6 und Q7) können nur durch formal spezifizierte und verifizierte Systeme erreicht werden.

Von den umfangreichen Forschungen auf diesem Gebiet seien hier stellvertretend das vom Bundesministerium für Forschung und Technologie geförderte Projekt KORSO („Korrekte Software", [38]) und die von ESPRIT geförderten Projekte ProCoS („Provably Correct Systems" [12]) und PROSPECTRA („Program Development by Specification and Transformation") [20] genannt.

Wie auf der Tagung der Gesellschaft für Informatik 1992 in Karlsruhe von M. Lehmann [23] erläutert, läßt außerdem das neue EG-weite Gewährleistungsrecht keinen generellen Haftungsausschluß für Software mehr zu. Deshalb müssen sich Firmen, die Software herstellen, intensiver mit Fragen sicherer Software beschäftigen.

Korrektheit ist eine *notwendige* Voraussetzung für sichere Software. Wir behaupten aber nicht, daß sie auch hinreichend ist. Auf real existierenden Rechnern können immer noch Fehler auftreten. Selbst wenn ein Programm bezüglich einer formalen Spezifikation als korrekt nachgewiesen wurde, kann nicht garantiert werden, daß

- die Spezifikation auch wirklich die Anwendungsbedürfnisse adäquat modelliert,
- der Übersetzer und das Betriebssystem korrekt funktionieren;
- die verwendete Hardware keine Fehler enthält und daß
- kein Stromausfall oder eine sonstige Katastrophe eintritt.

Ein Korrektheitsbeweis gibt zwar die *größtmögliche* Sicherheit, daß eine Implementierung tatsächlich ihrer formalen Spezifikation genügt, aber auch diese Sicherheit kann nicht absolut sein. Aus diesem Grund beinhalten die Anforderungen an sichere Software auch Zuverlässigkeit, Robustheit, Fehlertoleranz, Verfügbarkeit, Wartbarkeit, Portabilität usw., mit denen wir uns an dieser Stelle allerdings nicht beschäftigen wollen.

Im folgenden wollen wir alle Systeme, die ihre Benutzerinnen und Benutzer bei der Entwicklung korrekter Software unterstützen, als „Korrektheitswerkzeuge" bezeichnen. Dieser Begriff ist sehr weit gefaßt und beinhaltet sowohl nachträgliche Verifikation als auch Synthese, deduktives als auch transformationelles Vorgehen. Solche Systeme müssen auch nicht im strengen Sinne Beweissysteme sein: Beispielsweise ist KIDS (Kestrel Interactive Development System) [34] kein reines Beweissystem, es benutzt aber automatische Deduktion[1].

Ein Einwand, der oft gegen formale Korrektheitswerkzeuge vorgebracht wird, ist der folgende: „Was nützt mir ein Korrektheitsbeweis, der von einem System erzeugt wurde, das seinerseits nicht als korrekt nachgewiesen ist?" Auch hier lautet die Antwort: Eine absolute Sicherheit kann es nicht geben! Selbstver-

[1] Mit diesem System ist es – wie Douglas R. Smith in Dagstuhl im März 1994 vortrug – gelungen, einen Algorithmus zur Transportplanung zu entwickeln, der tausend mal schneller ist als die bisher bekannten.

ständlich wird ein nicht als korrekt bewiesenes Korrektheitswerkzeug Fehler enthalten. Jedoch kann die Wahrscheinlichkeit, daß fälschlicherweise ein „Beweis" für ein inkorrektes Programm erzeugt wird, auf ein Minimum reduziert werden.

Dies wird z. B. im Karlsruhe Interactive Verifier (KIV) [18] dadurch erreicht, daß Beweise als *Datenobjekte*, sog. Beweisbäume, repräsentiert werden. Diese Bäume sind eine exakte Repräsentation einer Deduktion mittels logischer Regeln. Beweisbäume können nur durch logische Operationen erzeugt werden, wobei *matching* verwendet wird. Dieser Ansatz hat den Vorteil, daß sich die im KIV-System enthaltenen Fehler beispielsweise in Systemabstürzen oder im Fehlschlagen von eigentlich korrekten Operationen bzw. Aktionen niederschlagen, nicht jedoch im Aufbau „falscher" Beweis-Datenobjekte, da hier immer eine syntaktische Prüfung stattfindet, ob die Teile, aus denen ein Beweisobjekt zusammengesetzt wird, auch zueinander passen. Die Erzeugung eines „falschen" Beweises ist damit zwar nicht im strengen Sinne ausgeschlossen, aber auch nicht wahrscheinlicher, als daß der sprichwörtliche Affe, vor eine Tastatur gesetzt, einen sinnvollen Text erzeugt.

In neuester Zeit werden Anstrengungen unternommen, die Korrektheitswerkzeuge selbst zu verifizieren. Ein Workshop der CADE-12 in Nancy, Frankreich, im Juni 1994 beschäftigt sich mit genau dieser Fragestellung: *Correctness and Metatheoretic Extensibility of Automated Reasoning Systems.* Das neue System von Boyer und Moore, ACL2, ist in seiner eigenen Logik geschrieben; es wird also möglich sein, Theoreme über den Beweiser selbst zu zeigen.

Wir halten fest: Ein Korrektheitsnachweis kann zwar keine Garantie für fehlerfreies Funktionieren eines Programmes im realen Rechnerbetrieb bieten, ist aber ein drastischer Qualitätssprung gegenüber dem bisher nur üblichen Testen.

3 Thesen zur Realisierung korrekter Software

Im folgenden stellen wir einige Thesen bezüglich der Realisierbarkeit korrekter Software auf. Für jede These wird eine kurze Begründung gegeben, warum wir sie wichtig finden. Wir glauben, daß ohne die Verwirklichung der in den Thesen gestellten Forderungen korrekte Software weiter eine Illusion bleiben wird.

These 1 *Allein das Aufstellen einer formalen Spezifikation führt zu einer Qualitätsverbesserung.*

Unabhängig davon, um welche spezielle Methode es sich handelt, kann schon allein durch die Erstellung einer formalen Spezifikation erheblicher Nutzen entstehen. Das genaue Inspizieren des Problembereiches oder die Betrachtung des Problems aus einem anderen Blickwinkel beim Spezifizieren nach einer bestimmten Methode kann Widersprüche aufdecken oder auf bisher nicht behandelte Sonderfälle hinweisen. Auch wenn sich keine formale Weiterverarbeitung (wie z. B. Programmsynthese) anschließt, bringt eine formale Spezifikation viele Vorteile, weil Fehler zu einem sehr frühen Zeitpunkt entdeckt werden können.

Craigen et al. berichten z. B. in [11] über den Einsatz verschiedener Werkzeuge und formaler Methoden in Teilbereichen von zwölf Industrieprojekten. Neben vielerlei Kritik und Anregungen für die Weiterentwicklung der verwendeten Werkzeuge ist häufig von positiven Effekten allein durch den Einsatz der Werkzeuge bei der Spezifikation die Rede.

Houston und King berichten in [21] über positive Ergebnisse bei der Benutzung der Spezifikationssprache Z [35] zur Entwicklung einer neuen Version des IBM *Customer Information Control System* (CICS). Dieses Projekt, für das die verschiedensten Maßzahlen erhoben wurden, zeigte, daß durch die Benutzung einer formalen Methode

„[...] the overall quality of the product had been improved, with a reduction in the number of errors found, and the use of formal methods was a great benefit in causing errors to be found earlier in the development process. ([21], S. 589)."

Die genannte Qualitätsverbesserung und die Tatsache, daß weniger Fehler und diese in früheren Phasen des Software-Lebenszyklus gefunden wurden, führten sogar zu einer Kostenersparnis von 9 % für die formal spezifizierten Teile des Systems.

Um diese These weiter zu erhärten, sollten kontrollierte Experimente über den Nutzen von formalen Spezifikationen durchgeführt werden. Dabei sollten zwei Teams mit vergleichbarem Erfahrungsstand und ohne direkten Kontakt miteinander die Aufgabe bekommen, ein System mit bzw. ohne den Einsatz einer bestimmten formalen Methode zu entwickeln. Eine Begleitung des Experimentes sollte den *Entstehungsprozeß* des Softwareproduktes beobachten. Zum Schluß sollten die Endprodukte unter verschiedenen Aspekten (Umfang, Lesbarkeit, Wartbarkeit, Fehler, Benutzerakzeptanz, etc.) analysiert werden. Eine Anzahl solcher empirischer Experimente würde helfen, den Grad des Nutzens nachzuweisen und evtl. neue Wege für die Zukunft aufzeigen[2].

These 2 *Maschinenunterstützung ist bei Korrektheitsbeweisen unerläßlich.*

Unter der Annahme, daß ein Korrektheitsbeweis notwendig sei, kommt man mit einem handgeführten Beweis nicht allzu weit. Schon bei kleineren nicht-trivialen Programmen können sich sehr leicht Fehler einschleichen, die nicht sofort auffallen. Ein Beispiel hierfür ist der in den *Communications of the ACM* geführte Beweis von Hesselink und Jongejan [19]. Dort wird eine Formalisierung eines von Teuhola und Wegner beschriebenen Verfahrens [36] vorgestellt, mit dem man doppelte Einträge aus Vielfachmengen entfernen kann,

[2] Uns ist kein solches Experiment bekannt – seine Durchführung würde schließlich viel Geld kosten. Allerdings planen wir ein Projekt im Kleinen, bei dem eine Aufzugsteuerung nach zwei verschiedenen Methoden von zwei Gruppen entwickelt werden soll.

um eine Menge zu erhalten. In dem von Hand geführten „Beweis" sind einige Fehler, die Teuhola und Wegner in ihrer im gleichen Heft abgedruckten Erwiderung deutlich herausstreichen.

Streng formale Beweise, d.h. Ableitungen in einem formalen System, z.B. einer Logik, sind gegenüber Hand-Beweisen noch deutlich länger. Allerdings, so wird in [10] argumentiert, handelt es sich dabei um einen *konstanten* Faktor, der zudem durch geeignete Maßnahmen, wie z.B. die Definition von Großschritten (siehe These 3), verkleinert werden kann.

Um ein Gefühl für den Aufwand einer mechanischen Verifikation zu vermitteln, geben wir hier einige Zahlenbeispiele für Verifikationen, die mit zwei Systemen durchgeführt wurden, dem KIV-System und dem Boyer-Moore-Beweiser. In [31] werden folgende Zahlen für den Umfang von Korrektheitsbeweisen mit dem KIV-System angeführt:

- Die Implementierung der natürlichen Zahlen mit Null, Nachfolgerfunktion, Addition, Multiplikation und einem „Teilt"-Prädikat durch Binärwörter ist 135 Zeilen lang. Um sie als korrekt zu beweisen, müssen 21 Sätze bewiesen werden, für deren Beweis weitere 19 Lemmata gezeigt werden müssen. Der Beweis benötigt 7000 Beweisschritte, die nicht etwa Anwendungen einfacher logischer Regeln sind, sondern in etwa der Größe von Beweisschritten entsprechen, wie sie Menschen in Hand-Beweisen machen.
- Um die Implementierung von Mengen durch Listen zu verifizieren (98 Zeilen Code), sind 4278 Beweisschritte nötig.
- Implementiert man Mengen durch ausgewogene (AVL)-Bäume, so benötigt man für 270 Zeilen Code gar 27643 Beweisschritte.

Ähnliche Zahlen werden in [7] über Beweise genannt, die mit dem Boyer-Moore-Induktionsbeweiser geführt worden sind:

- Der Beweis der Eindeutigkeit der Primfaktorzerlegung umfaßt 49 Zeilen für die Formulierung der Behauptung. Weitere 108 Funktionen und Lemmata mit 696 Zeilen sind in dem Beweisgraphen verzeichnet. Die längste Beweistiefe ist 17 Schritte.
- Ein Beweis der Korrektheit eines hypothetischen Chip (des FM8501) benötigt 991 Programmzeilen und 230 zusätzliche Funktionen und Lemmata mit 2171 Zeilen. Dieser Beweis hat eine ähnliche Tiefe, 18 Schritte.
- Der Beweis von Gödels Unvollständigkeitssatz benötigt etwa gleich viele Zeilen für die Formulierung (864), aber wegen der Komplexität des mathematischen Sachverhaltes sind 1741 weitere Lemmata und Funktionen mit 20002 Zeilen auf einer Tiefe von 58 Schritten nötig.

Natürlich sind die Zahlen für die beiden Beweiser nicht direkt vergleichbar, da die in NQTHM verwendeten Lemmata viele KIV-Beweisschritte beinhalten. Außerdem kann in jedem der beiden Systeme ein Sachverhalt auf verschiedene Art und Weise gezeigt werden.

Trotzdem sprechen die genannten Zahlen eine deutliche Sprache. Es erscheint völlig ausgeschlossen, daß Menschen derartig umfangreiche Beweise ohne Maschinenunterstützung führen können. Die oben beschriebenen Korrektheitsbeweise waren auch mit Maschinenunterstützung nur deshalb möglich, weil die Systeme zwischen 70 % und 95 % der Schritte automatisch ausführen konnten. Wenn dieser Sachverhalt schon für Fallstudien zu beobachten ist, die in einem akademischen Umfeld vorgenommen wurden, so gilt er umso mehr bei industriell eingesetzten Systemen, die um ein Vielfaches größer sind.

Vom industriellen Standpunkt aus ist außerdem die Gesamtzeit, die für die Beweise benötigt werden, interessant. In der Literatur [32] werden Zahlen von 5–10 Zeilen verifizierten Codes pro Person und Tag genannt. Nach unserer Erfahrung ist jedoch die benötigte Zeit stark vom Wissens- und Erfahrungsstand der betreffenden Person abhängig[3].

These 3 *Korrektheitswerkzeuge* sollten interaktiv konzipiert werden.

Betrachtet man zur Zeit existierende vollautomatische Beweissysteme [5, 7, 27, 30], so stellt man fest, daß diese hauptsächlich beim Beweis von nicht allzu schwierigen mathematischen Sätzen erfolgreich sind. Derartige Sätze treten häufig als *Verifikationsbedingungen* bei der Synthese oder Verifikation von Programmen auf, so daß diese Systeme zur Realisierung korrekter Software wichtig und nützlich sind.

Im Umgang mit Programmen als solchen ist aber eine vollständige Automatisierung bisher nicht gelungen, obwohl Anstrengungen in dieser Richtung unternommen wurden [4, 6]. Das Problem scheint zu sein, daß beim Finden von Korrektheitsargumenten wie Rekursionsschemata oder Schleifeninvarianten menschliche Kreativität bisher nicht ausreichend ersetzt werden konnte. Beispielsweise ist das automatische Finden von Schleifeninvarianten nach wie vor Forschungsgegenstand. Es scheint daher zumindest für die nähere Zukunft erfolgversprechender zu sein, die beweisführende Person in den Beweisgang mit einzubeziehen.

Ein vielversprechender Ansatz hierzu ist das *taktische Theorembeweisen* [10, 18]. Dabei werden in Form von sog. *Taktiken* Funktionen definiert, die aus Beweiszielen hinreichende Unterziele generieren. Diese Taktiken entsprechen *Großschritten* in einem formalen Beweis und können auf diese Weise menschliche Beweisgänge nachahmen. Zudem sind Beweise mittels Metasprachen *programmierbar*. Die Ausführung eines solchen Beweisprogramms erfolgt weitgehend automatisch; nur an kritischen Punkten muß eingegriffen wer-

[3] Aufgrund ihrer Erfahrungen bei Computational Logic, Inc. (CLInc), hat Debora Weber-Wulff für NQTHM den Begriff des „Matt-Faktors" geprägt: Was Matt Kaufmann, Wissenschaftler bei CLInc, in *n* Zeiteinheiten beweisen kann, kann Yuan Yu (der seine Dissertation über den Beweis des 68020-Chips von Motorola mit NQTHM geschrieben hat) in *2n* der nächsthöheren Zeiteinheit beweisen. Für einen 5-Minuten-Matt-Beweis braucht Yuan 10 Stunden, Debora Weber-Wulff benötigt fast 20 Tage für den gleichen Beweis. Maritta Heisel kann Ähnliches über KIV und KIDS berichten.

den und beispielsweise – durchaus wiederum mit Systemunterstützung – die Generalisierung einer Induktionsbehauptung durchgeführt oder eine Schleifeninvariante entwickelt werden.

Grundsätzlich läßt sich sagen, daß Automatisierungsversuche umso erfolgreicher sind, je eingeschränkter entweder der Anwendungsbereich der betrachteten Programme (z.B. CAD) oder die Form der betrachteten Algorithmen (z.B. Divide-and-Conquer) ist. Beispiele hierfür sind in [26] zu finden.

These 4 *Alle wichtigen Informationen müssen explizit repräsentiert werden.*

Aus These 3, nach der Korrektheitswerkzeuge interaktiv arbeiten sollten, folgt sofort, daß die Benutzerin oder der Benutzer mit systemgenerierten Zwischenzuständen konfrontiert wird. In solchen Situationen müssen Entscheidungen getroffen werden, die dem System ein sinnvolles Weiterarbeiten ermöglichen. Es versteht sich damit von selbst, daß solche Zwischenzustände für Menschen *verständlich* sein müssen. Diese Forderung schließt gewisse Formalismen von vorneherein für Korrektheitswerkzeuge aus, wie z.B. das Resolutionsprinzip, das in vielen vollautomatischen Beweisern verwendet wird. Dagegen ist der Boyer-Moore-Beweiser beispielhaft, da er weitestgehend alle Beweisschritte so darstellt wie in mathematischen Textbüchern üblich. Allerdings sind die in LISP-Notation dargestellten Zwischenergebnisse ein etwas gewöhnungsbedürftiger Anblick[4].

Bei der expliziten Repräsentation wichtiger Informationen erweist sich auch die Benutzung von Spezialformalismen, wie z.B. der dynamischen Logik [17], als vorteilhaft. Bei dieser Logik können die Formeln imperative Programme enthalten. Die Objekte, deren Eigenschaften bewiesen werden, kommen also *explizit* in den zu zeigenden Sätzen vor. Bei Verifikationsbedingungsgeneratoren (wie z.B. der Gypsy Verifikationsumgebung [9]) werden die Programme so schnell wie möglich auf prädikatenlogische Formeln zurückgeführt. Im Gegensatz dazu können in der dynamischen Logik Beweise geführt werden, die auf der Semantik der Kontrollstrukturen der betrachteten Programmiersprache beruhen.

Die dynamische Logik ist ein Formalismus, der speziell auf der Behandlung imperativer Programme zugeschnitten ist. Für andere Zwecke ist die dynamische Logik weniger geeignet. Hieraus läßt sich der Trend zu einer immer breiteren Palette von Spezialformalismen ablesen: Will man Codierungen vermeiden, so kann man nicht hoffen, mit einem einzigen Formalismus, wie z.B. der Prädikatenlogik, eine Vielzahl von Problembereichen erfolgreich behandeln zu können. Statt dessen ist es sinnvoll, eine Vielfalt von Formalismen zu entwikkeln, die für den jeweiligen Zweck maßgeschneidert sind und damit die Beweisarbeit wesentlich erleichtern.

[4] In NQTHM wird sowohl für die verwendete Logik als auch für die Objektsprache, d.h. die Sprache, in der die zu verifizierenden Programme geschrieben sind, ein Pure-LISP-Dialekt verwendet.

These 5 *Ein Korrektheitswerkzeug darf bei der Anwendung keine bestimmten Vorgehensweisen erzwingen.*

Es ist unbestritten, daß verschiedene Individuen verschiedene Programmierstile entwickeln. Diese Tatsache kann zumindest zur Zeit noch als ein großes Hindernis bei der Einführung formaler Methoden zur Etablierung von Programmkorrektheit angesehen werden. Formale Methoden legen ihren Benutzerinnen und Benutzern oftmals ein zu enges Korsett an: Wenn die formale Methode ein genaues Verfahren vorschreibt, das bei der Entwicklung korrekter Software starr eingehalten werden muß, kann dies dafür verantwortlich sein, daß das betreffende Werkzeug nicht akzeptiert wird [37]. Pointiert gesagt, lautet unsere Forderung, daß formale Methoden an menschliche Vorgehensweisen angepaßt werden müssen und nicht umgekehrt. Dies bedeutet, daß Korrektheitswerkzeuge, die auch akzeptiert werden sollen, ein Höchstmaß an *Flexibilität* aufweisen müssen. Es sollte also möglich sein, trotz Benutzung eines Korrektheitswerkzeugs ein einmal eingeführtes Prozeßmodell weitestgehend beizubehalten.

Es wäre sogar wünschenswert, den gleichen Korrektheitsnachweisschritt auf verschiedene Weise ausführen zu können! So wie die Chefköchin eine Vielzahl von Wiege-, Schneide-, Mix- und Kochwerkzeugen parat hat, um aus einer Menge von Zutaten eine Menüfolge herzustellen, so sollte die Benutzerin bzw. der Benutzer einer automatisierten formalen Methode die Kontrolle über den Beweisgang haben. Man wählt je nach Beschaffenheit des aktuellen Problems ein Werkzeug aus. Und ebenso wie es viele Möglichkeiten gibt Zwiebeln zu zerkleinern – mit einem Messer, mit der Küchenmaschine, mit einem Beil[5], etc. – kann auf verschiedene Art und Weise bewiesen werden, daß z.B. Mengen durch Listen darstellbar sind[6].

These 6 *Ein Korrektheitswerkzeug muß offen sein.*

Diese Forderung an ein akzeptables Korrektheitswerkzeug hängt mit der vorherigen eng zusammen: Außer der Tatsache, daß Programmierstile überhaupt existieren, muß auch anerkannt werden, daß Programmierstile sich ändern und neue Programmiermethodiken entwickelt werden. Daraus ergibt sich das Bedürfnis, ein Korrektheitswerkzeug parallel mit der eigenen Expertise weiterzuentwickeln. Dies muß *graduell* möglich sein. Außerdem darf Formalisierungs- und Entwicklungsarbeit, die vorher aufgewandt wurde, um dem System einen gewissen Programmierstil beizubringen, nicht entwertet werden. Man muß sich ein Korrektheitswerkzeug also vielmehr als einen Werkzeug*kasten* vorstellen: Der Erwerb neuer Einzelwerkzeuge erhöht die handwerklichen Fähigkeiten, ohne die vorhandenen unbrauchbar zu machen.

[5] Man sieht, es gibt geeignete und weniger geeignete Werkzeuge!

[6] Oder wie die Köchin Konserven benutzt, kann auch diese Tatsache einer Beweisbibliothek entnommen und weiterverarbeitet werden.

These 7 *Es muß einen Wandel in der Informatik- bzw. Programmierausbildung geben.*

Wir können nicht erwarten, daß die Einsicht in die Nutzbarkeit von formalen Methoden in der Praxis spontan entsteht. Sowohl im Ausbildungsprozeß als auch durch gezielte Fortbildung muß gezeigt werden, warum formale Methoden für sichere Software förderlich sind.

Dies geschieht *nicht* durch die von vielen, allen voran Dijkstra [15], geforderte Beschäftigung mit noch mehr Mathematik an sich. Es reicht nicht aus, Mathematik zu lehren, mit der Begründung, dies fördere das abstrakte Denken. Wir müssen *zeigen,* was wir davon haben, formale Methoden zu verwenden. Als konkrete Maßnahmen schlagen wir vor:

- Schon in der Ausbildung müssen die Grundlagen gelegt werden. Dies bedeutet aber nicht nur das Einpauken von diskreter Mathematik. Es muß gezeigt werden, wie und warum mathematische Konstrukte wie Schleifeninvarianten von Nutzen sind: Weil sie helfen die berüchtigten „off-by-one" -Fehler zu vermeiden.
- Es muß konkrete Übungsmöglichkeiten geben, durch die gelernt werden kann, was ein gewisses Maß an Formalität für die Arbeit bedeutet.
- Es muß mehr Textbücher geben, die sich mit konkreten, nichttrivialen Beispielen beschäftigen. Die Bücher dürfen sich nicht im Formalismus verlieren, sondern sie müssen verständlich machen, was durch den Formalismus gewonnen wird.
- Der *Weg* zur Korrektheit – nicht der polierte Korrektheitsbeweis, sondern die Methode, mit der man diesen Beweis aufstellt – muß lehrbar gemacht werden. Interessanterweise forderte Dijkstra gerade dies und das Erlernen von „pondering" (Erwägen) in einer frühen Schrift [14].
- Wir müssen Situationen anbieten, in denen durch den Einsatz von formalen Methoden *Erfolgserlebnisse* entstehen anstatt der so oft vermittelten Frustrationen.

Dies wird nicht von heute auf morgen geschehen können – wir können nicht erwarten, daß jemand nach drei Wochen völlig kompetent in der Anwendung formaler Methoden wird. Aber nach dieser Zeit kann schon ein Anfang gemacht werden. Es ist nicht mehr nötig in Informatik promoviert zu sein, um formale Methoden verwenden zu können. Uwe Schmidt [33] berichtet von durchweg guten Erfahrungen mit der Verwendung von VDM [22] an einer Fachhochschule.

These 8 *Auch die Spezifikations- und Wartungsphase müssen formal bzw. maschinell unterstützt werden.*

Diese Forderung resultiert aus zwei Tatsachen: Zum einen ist es wohlbekannt, daß die Behebung von Fehlern, die in den frühen Phasen des Software-Lebenszyklus gemacht werden, umso teurer wird, je später sie entdeckt werden. Es ist also von besonderer Wichtigkeit, Fehler in der Spezifikation zu entdecken, bevor mit der Programmierung begonnen wird. Zu diesem Zweck ist Maschinenunterstützung in mehrfacher Hinsicht von Nutzen. *Deduktionssysteme* können dazu benutzt werden, zu zeigen, daß die Spezifikation nicht in sich widersprüchlich ist. Unter Umständen kann auch gezeigt werden, daß alle möglichen Randfälle abgedeckt werden, oder daß die Spezifikation eine Systemfunktion eindeutig festlegt. Durch Vorwärtsschließen sind Konsequenzen zu ermitteln, die sich aus der Spezifikation ergeben, und die den Kunden zur Beurteilung vorgelegt werden können.

Natürlich kann nicht erwartet werden, daß die Auftraggeber von Software formale Spezifikationssprachen beherrschen. Hier leisten *Paraphrasierer* gute Dienste. Dies sind Programme, die Ausdrücke einer formalen Spezifikationssprache in natürliche Sprache übersetzen. Neuerdings gibt es auch Ansätze, halb-formale Spezifikationen wie Entity-Relationship-Diagramme oder Datenflußdiagramme automatisch in formale Spezifikationen zu übersetzen [29]. Damit wird klar, daß Maschinenunterstützung in der Spezifikationsphase sehr hilfreich wäre, obwohl es sich nicht formal fassen läßt, ob eine Spezifikation die Wünsche an das System adäquat wiedergibt.

Zum anderen müssen wir uns vor Augen halten, daß der Übergang von einer Spezifikation zu einem Programm in hohem Maße *unstetig* ist. Das heißt, daß kleine Änderungen in der Spezifikation erhebliche Änderungen im Programm nach sich ziehen können. Daraus folgt, daß eine Spezifikation viel leichter an geänderte Bedürfnisse anzupassen ist als das zugehörige Programm. Was liegt also näher, als Programme in systematischer Weise aus der Spezifikation abzuleiten, so daß ihre Korrektheit garantiert werden kann, diese Ableitungen zu speichern, und die Wartung von Programmen durch (i. d. R. kleinere) Änderungen der Spezifikation und weitestgehende Wiederverwendung der vorhandenen Ableitungen durchzuführen? Eine maschinelle Unterstützung der Wartungsphase bedeutet also die Wiederverwendung nicht nur vom Programmen, sondern auch von Entwicklungsprozessen.

Dieser Ansatz wird inbesondere beim wissensbasierten Software-Engineering zugrunde gelegt [25, 26].

4 Korrektheit ist machbar!

Im vorigen Abschnitt haben wir einige Forderungen aufgestellt, die wir für *notwendig* für die Erstellung von korrekter Software im größeren Stil halten. Nun wollen wir darlegen, warum wir meinen, daß sie im wesentlichen auch *hinreichend* sind. Auf zwei Gründe wollen wir im folgenden näher eingehen: Erstens sind wir der Überzeugung, daß es möglich ist, Erkenntnisse aus Korrektheitsnachweisen für kleine Algorithmen auf wesentlich komplexere Systeme zu übertragen. Zweitens dürfte die erweiterte Produkthaftung, wie sie mit dem neuen EG-Recht eingeführt wurde, zu einem gewaltigen Motivationsschub führen, bei der Produktion von Software die Korrektheit der Produkte sicherzustellen: Korrektheit wird auch wirtschaftlich sein.

DiMillo, Lipton and Perlis haben in ihrem Artikel behauptet [13]:

„The specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. [...] These are not differences in degree. They are differences in kind. Babysitting for a sleeping child for one hour does not scale up to raising a family of ten – the problems are essentially, fundamentally different."

Dieses vernichtende Urteil über die Übertragbarkeit von Verifikationserfahrungen mit kleineren Programmen übersieht, daß, während Babysitten und eine Großfamilie erziehen im direkten Vergleich tatsächlich um Größenordnungen auseinanderliegen, es einen *nachvollziehbaren Pfad* vom einen zum anderen gibt. Nachdem ein schlafendes Kind eine Stunde lang gehütet worden ist, kann schrittweise die Beaufsichtigung für mehrere Stunden, auch eines wachen Kindes, und später für mehrere Kinder übernommen werden. Also müssen wir zuerst lernen, Korrektheitsnachweise von relativ kleinen Programmen vernünftig zu bewältigen. Danach können wir Schritte in Richtung auf Korrektheitsbeweise für komplexere, realistischere Systeme unternehmen.

Dieser Schritt wird schon in den verschiedensten Instituten und Firmen getan. Ein Beispiel haben wir bereits genannt: CICS. Weiterhin ist in [3] ist ein Übersetzer für die Sprache Micro-Gypsy bis hinunter zur Gatterebene verifiziert worden. Dieser Beweis besteht aus zwei zusammenhängenden Beweisen, die ihrerseits aus unzähligen kleineren zusammengesetzt sind. Und ganz tief unten sind auch die sog. trivialen Beweise von Mengen- oder Listentheorie zu finden. Gerade dieses Aufbauen von komplexeren Beweisen auf der Basis von sehr einfachen ist nicht nur denkbar, es wird auch tatsächlich durchgeführt.

Wie in [32] genauer ausgeführt, ist es durch eine geeignete Modularisierung möglich, die Korrektheit eines großen modularen Systems auf die Korrektheit seiner Komponenten zurückzuführen. Voraussetzung dafür ist, daß bereits die Spezifikation des Systems modular aufgebaut ist, d.h. mittels strukturellen Operationen aus einfacheren Spezifikationen gewonnen wurde. Diese strukturierten Spezifikationen werden dann durch schrittweise Verfeinerung in ausführbaren Code überführt. Bei diesem Ansatz weisen Spezifikation und Programm die gleiche Struktur auf. Da dies nicht immer verlangt werden kann, muß auch untersucht werden, inwiefern intermodulare Abhängigkeiten in ihrem Zusammenwirken zusätzlich betrachtet werden müssen. Reif [32] und Moore [28] sprechen in diesen Zusammenhang von „glue", dem Klebstoff, der die Moduln zusammenhält.

Letztendlich wird aber die vorhandene Skepsis nicht durch Machbarkeitsargumente beseitigt werden – Korrektheit wird ganz einfach durch wirtschaftliche Zwänge notwendig werden. Nach dem Vorbild der USA wurde seit den sechziger Jahren in Deutschland begonnen, die Produzentenhaftung für fehlerhafte Produkte zu erweitern [23]. So gilt z.B. gemäß §§ 823 ff. BGB eine Beweislastumkehr hinsichtlich des Verschuldens:

„Wird bei der bestimmungsgemäßen Benutzung eines Produktes eine Person oder eine Sache hinsichtlich des Integritätsinteresses dadurch geschädigt, daß das Produkt fehlerhaft hergestellt war, muß der Hersteller beweisen, daß ihn im Hinblick auf diesen Fehler kein Verschulden trifft" ([23], S. 137).

Diese strenge deutsche Produzentenhaftung wurde als Leitfaden für die EG-Richtlinie 85/374 EWG über die Haftung für fehlerhafte Produkte herangezogen, welche wiederum die Grundlage für das am 1.1. 1990 in Kraft getretene Produkthaftungsgesetz ist.

Für Software besteht also sowohl *Produzentenhaftung* gem. §§ 823 ff. BGB als auch *Produkthaftung* gem. Produkthaftungsgesetz. Bei der Produkthaftung gilt:

„Wird bei der bestimmungsgemäßen Verwendung von Standard-Software und bzw. oder -Hardware eine Person oder eine Sache dadurch geschädigt, daß diese Produkte [...] fehlerhaft sind, so haftet der [...] Hersteller gem. §§ 823 ff. BGB nach den allgemeinen Grundsätzen der verschuldensabhängigen Produzentenhaftung" ([23], S. 139).

Es liegt also bei den Herstellern, zu beweisen, daß sie hinsichtlich etwaiger Fehler kein Verschulden trifft.

Bei der Produkthaftung wurde in der Vergangenheit davon ausgegangen, daß Software kein „körperlicher Gegenstand" und somit keine „Sache" sei, so daß eine Produkthaftung ausgeschlossen werden konnte. Diese Ansicht steht allerdings im Widerspruch zu der genannten EG-Richtlinie. Die neuere Rechtsprechung geht nun davon aus, daß *Software als Produkt* dem Produkthaftungsgesetz unterliegt. Die Folgen dieser veränderten Rechtsauffassung sind klar: Auf die Softwarehersteller können in Zukunft verstärkte Haftungsrisiken zukommen, die bei sicherheitskritischer Software existenzbedrohend sein können. Diese Risiken können nur auf ein vertretbares Maß reduziert werden, wenn verstärkte Anstrengungen bezüglich der Sicherheit von Software unternommen werden. Wie wir gesehen haben, ist Korrektheit ein integraler Bestandteil von Softwaresicherheit.

Korrekte Software bekommt man nicht geschenkt. Dies ist in Anbetracht der Qualitätssteigerungen, die mit Korrektheit verbunden sind, auch nicht zu erwarten. Die Mehrkosten sind jedoch gut angelegt, wenn sie Softwarehersteller vor existenzbedrohenden Regreßansprüchen schützen. Es ist zu erwarten, daß dieses positive Preis-Leistungsverhältnis sich auch bei weniger sicherheitsrelevanter Software einstellen wird, sobald sich die Produkthaftung für Software in größerem Maße eingebürgert hat.

Literatur

1. Aagaard, M., Leeser, M.: Verifying a logic synthesis tool in Nuprl: A case study in software verification. Proceedings of the 4th Workshop on Computer Aided Verification, 1992
2. Barr, A., Cohen, P.R., Feigenbaum, E.A., (eds.): The handbook of artificial intelligence, Vol. 4. Reading, MA: Addison-Wesley 1989

3. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning *5* (4), 411–418 (1989)
4. Bibel, W., Hörnig., K.M.: LOPS – a system based on a strategical approach to program synthesis. In: Biermann, A., Guiho, G., Kodratoff, Y. (eds.) Automatic Program Construction Techniques, pp. 69–89. New York: MacMillan 1984
5. Biundo, S., Hummel, B., Hutter, D., Walther, C.: The Karlsruhe induction theorem proving system. In: Siekmann, J.H. (ed.) Proceedings 8th CADE. (Lect. Notes Comput. Sci. vol. 230, pp. 673–675) Berlin, Heidelberg, New York: Springer 1986
6. Biundo, S.: Automatische Synthese rekursiver Programme als Beweisverfahren. Berlin, Heidelberg, New York: Springer 1992
7. Boyer, R.S., Moore, JS.: A computational logic handbook. New York: Academic Press 1988
8. Cardell-Oliver, R.: Formal verification of real time protocols using higher order logic. Technischer Bericht 206, Computer Laboratory, Universität Cambridge 1990
9. Carranza, M., Young, W.D.: An annotated gypsy bibliography. Technischer Bericht 105, CLInc Internal Note 1989
10. Constable, R.L., Knoblock, T.B., Bates, G.L.: Writing programs that construct proofs. J. Autom. Reasoning *1*, 285–326 (1985)
11. Craigen, D., Gerhart, S., Ralston, T.: Formal Methods Reality Check: Industrial Usage. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME'93: Industrial Strength Formal Methods. (Lect. Notes Comput. Sci., vol. 670, pp.250–267) Berlin, Heidelberg, New York: Springer 1993
12. ProCoS – ESPRIT BRA 3104 Final report: Provably Correct Systems. Technischer Bericht, ProCoS ID/DTH 1991
13. DeMillo, R., Lipton, R., Perlis, A.: Social processes and proofs of theorems and programs. CACM 22, 271–280 (1979)
14. Dijkstra, E.W.: On the teaching of programming, i.e. on the teaching of thinking. In: Bauer, F.L., Samelson, K. (eds.) Language hierarchies and interfaces, (Lect. Notes Comput. Sci., vol. 46, pp. 1–10). Berlin, Heidelberg, New York: Springer 1976
15. Dijkstra, E.W.: On the cruelty of really teaching computing science. CACM *32*, 1398–1404 (1989)
16. Fetzer, J.H.: Program verification: The very idea. CACM *31*, 1048–1063 (1988)
17. Goldblatt, R.: Axiomatising the logic of computer programming. (Lect. Notes Comput. Sci., vol.130). Berlin, Heidelberg, New York: Springer 1982
18. Heisel, M., Menzel, W., Reif, W., Stephan, W.: Der Karlsruhe Interactive Verifier (KIV). In: Kersten, H. (eds.): Sichere Software. Formale Spezifikation und Verifikation vertrauenswürdiger Systeme, pp. 172–193. Heidelberg: Hüthig 1990
19. Hesselink, W., Jongejan, J.: Duplicate deletion derived. CACM *35*, 99–107 (1992)
20. Hoffmann, B., Krieg-Brückner, B. (eds.): PROgram development by SPECification and TRAnsformation, the PROSPECTRA methodology, language family and system. (Lect. Notes Comput. Sci., vol. 680) Berlin, Heidelberg, New York: Springer 1993
21. Houston, I., King, S.: CICS project report. Experiences and results from the use of Z in IBM. In: Prehn, S., Toetenel, H. (eds.) VDM'91: Formal software development methods. (Lect. Notes Comput. Sci., vol. 551, pp. 588–596) Berlin, Heidelberg, New York: Springer 1991
22. Jones, C.B.: Systematic software development using VDM. New York, London: Prentice Hall 1990
23. Lehmann, M.: Produkt- und Produzentenhaftung bei integrierter Produktion. In: Information als Produktionsfaktor. 22. GI-Jahrestagung, Karlsruhe, pp. 133–147. Berlin, Heidelberg, New York: Springer 1992
24. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Comput. *26*, 18–41 (1993)
25. Lowry, M., Duran, R.: Knowledge-based software engineering. In: [2], Kapitel 20, pp. 241–322. Reading, MA: Addison-Wesley 1989
26. Lowry, M., McCartney, R.D. (eds.): Automating software design. Menlo Park: AAAI Press 1991
27. McCune, W.: OTTER 2.0 users guide. Technischer Bericht ANL-90/9, Argonne National Laboratory, 1990
28. Moore, J.S.: Piton: A verified assembly-level language. Technischer Bericht 22, CLInc, 1988
29. Nickl, F., Wirsing, M.: A formal approach to requirements engineering. Technischer Bericht 9314, Ludwig-Maximilians-Universität München, Institut für Informatik, 1993
30. Ohlbach, H.J., Siekmann, J.: The Markgraf-Karl-refutation procedure. Artikel zur Festschrift von Alan Robinsons Geburtstag. Oxford: Oxford Press 1990
31. Reif, W.: Korrektheit von Spezifikationen und generischen Moduln. Dissertation, Universität Karlsruhe, 1991
32. Reif, W.: Verification of Large Software Systems. In: Shyamasundar, R. (ed.) Foundations of Software Technology and Theoretical Computer Science. 12th Conference. New Delhi, India (Lect. Notes Comput. Sci., vol. 652, pp. 241–252) Berlin, Heidelberg, New York: Springer 1992
33. Schmidt, U.: Formale Softwareentwicklungsmethoden in der Ausbildung. In Proceedings. 2. Workshop SEUH – Software Engineering im Unterricht der Hochschulen. Hamburg, 1993
34. Smith, D.R.: KIDS: a semiautomatic program development system. IEEE Trans. Software Eng. *16*, 1024–1043 (1990)
35. Spivey, J.M.: The Z notation – A reference manual, 2. Auflage. New York, London: Prentice Hall 1992
36. Teuhola, J., Wegner, L.: Minimal space average linear time duplication deletion. CACM *34*, 62–73 (1991)
37. Weber-Wulff, D.: Selling formal methods to industry. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME'93: Industrial-strength formal methods. (Lect. Notes Comput. Sci., vol. 670, pp. 671–678) Berlin, Heidelberg, New York: Springer 1993
38. Wirsing, M., Brix, H., Broy, M., Conrad, S., Gastinger, S., Gogolla, M., Hußmann, H., Jatzeck, M., Krieg-Brückner, B., Liu, J., Pepper, P., Peterreins, H., Poigné, A., Reif, W., Reus, B., Schellhorn, G., Wolff, B., von Henke, F.: A framework for software development in KORSO. Technischer Bericht 9205, LMU München, 1992
39. Zentralstelle für Sicherheit in der Informationstechnik. Kriterien für die Entwicklung, Realisierung und Zulassung von Werkzeugen zur formalen Spezifikation und Verifikation, 1990