# Software-Sicherheit mit formalen Methoden

Wolfgang Reif

Universität Ulm
Abt. Programmiermethodik
reif@informatik.uni-ulm.de

**Abstract**

*Im Spektrum der qualitätssichernden Maßnahmen im Software-Entwurf bieten formale Spezifikations- und Verifikationsmethoden das Maximum dessen, was heutzutage an garantierter Fehlerfreiheit möglich ist. Die Korrektheit der Software wird dabei durch das Führen mathematischer Beweise sichergestellt. Diese Methoden eignen sich besonders für Anwendungen, an die traditionell höchste Zuverlässigkeits- und Qualitätsanforderungen gestellt werden. Dieser Artikel gibt einen Überblick über verschiedene Einsatzmöglichkeiten formaler Spezifikations- und Verifikationsmethoden. Am Beispiel von KIV, einem fortgeschrittenen Spezifikations- und Verifikationswerkzeug, wird der aktuelle Leistungsstand und die Wirtschaftlichkeit der Technologie erläutert.*

## 1 Die Qualitätskrise und ihre Ursachen

Trotz der rasanten Entwicklung, die die Softwaretechnik in den letzten 25 Jahren genommen hat, ist die routinemäßige Entwicklung fehlerfreier Software noch immer eine der großen Herausforderungen. Wie die folgenden, in der Literatur beschriebenen Beispiele zeigen, kann Software durch Spezifikations-, Design- und Programmierfehler zu einem erheblichen Risikofaktor in technischen Anwendungen werden:

- **Flughafen Denver, 1994.** Durch Softwarefehler kommt es zu einem vollständigen Fehlverhalten der neuen Gepäckverteilungsanlage. Die erforderliche Überarbeitung verzögert die Eröffnung um 14 Monate. Es entsteht ein Ausfallschaden von ca. 0,5 Mrd USD, [9].

- **Airbusabsturz Toulouse, 1994.** Software-Fehlverhalten ist mitverantwortlich für den Absturz eines A340 bei Toulouse. 7 Tote. [17].

- **Raumfahrt, Space Shuttle, 1992.** Aufgrund einer Fehlberechnung verpaßt die Raumfähre Endeavor ein Rendezvous mit dem Intelsat Satelliten. Ursache ist ein Rundungsfehler in der Arithmetik. Die NASA rechnet mit Software-Fehlern bei etwa jeder fünften Mission, [17].

- **Deutsche Telekom, 1996**. Durch Spezifikations- und Software-Fehler werden in 550 Vermittlungsstellen falsche Gebühren berechnet. Die dadurch verursachten Kosten werden auf 70 Mio DM geschätzt, [7].

Diese Beispiele zeigen zum einen typische Anwendungsbereiche formaler Methoden, zum anderen die praktische, gesellschaftliche und wirtschaftliche Relevanz des Problems. Sie weisen auch auf Schwächen in der aktuellen 'Software-Konstruktionslehre' in bezug auf die Produktion fehlerfreier Software hin. Eine der Hauptursachen dieser Qualitätskrise ist der weitverbreitete phasenorientierte Herstellungsprozeß, bei dem ca. 75% der Fehler, die in den frühen Phasen (Analyse, Enwurf und Codierung) gemacht werden, erst in den späten Phasen (Integrationstest, Systemtest und Einsatz) entdeckt werden. Innerhalb der Softwaretechnik existieren verschiedene Ansätze, diesen Mangel zu beseitigen. Allen gemeinsam ist das Ziel, möglichst wirkungsvolle qualitätssichernde Maßnahmen in die frühen Phasen des Entwurfs zu verlagern. Die vorherrschenden, 'konventionellen' Ansätze beruhen z.B. auf der Anwendung von CASE-Methoden, organisatorischen Maßnahmen zur Qualitätssicherung oder Prozeßzertifizierung nach ISO 9000. Bei diesen Techniken bleibt jedoch stets die Gefahr versteckter Fehler. Formale Spezifikations- und Verifikationstechniken bieten hier die Möglichkeit eines qualitativen Sprungs: die Dokumente der frühen Phasen werden zu formalen Spezifikationen, auf deren Basis man Sicherheitsgarantien bzw. Fehlerfreiheit von Progammen mit mathematischen Mitteln beweisen kann (Verifikation).

Neben der Anwendungsprogrammierung ist zunehmend auch der Entwurf und die Analyse moderner Programmiersprachen ein lohnendes Einsatzgebiet formaler Methoden. Das Spektrum reicht von der klärenden, formalen Fundierung der Semantik [1], [6], [2], [27], über den Nachweis von Spracheigenschaften wie z.B. der Typsicherheit, [16], [18], bis hin zur Verifikation von Compilern für unterschiedliche Zielmaschinen [26], [15], [10], [3]. Dieser Zweig der formalen Methoden hat nicht zuletzt durch das Aufkommen von JAVA und der damit verbundenen Sicherheitproblematik einen starken Aufschwung erfahren.

Formale Methoden haben mittlerweile auch Eingang in die europäischen bzw. weltweit harmonisierten Kriterienkataloge ITSEC [12], bzw. Common Criteria [5] zur Evaluierung und Zertifizierung informationstechnischer Produkte gefunden. Für eine Prüfung nach den hohen Vertrauensklassen in ITSEC und Common Criteria, die z.B. für online-banking oder digitale Unterschriften in Frage kommen, wird u.a. eine formales Sicherheitsmodell gefordert. Solche Prüfungen werden von Herstellern initiiert und von unabhängigen, behördlichen Stellen (z.B. dem Bundesamt für Sicherheit in der Informationstechnik, BSI) oder zugelassenen, privaten Einrichtungen durchgeführt. Insbesondere die jüngsten Gesetzesregelungen zur digitalen Signatur haben in Deutschland eine Reihe von Evaluierungen und Zertifizierungen angestoßen.

Der folgende Abschnitt illustriert die Verwendung formaler Methoden an einem realen Beispiel aus der Anwendungsprogrammierung. Abschnitt 3 stellt das KIV-System vor, Abschnitt 4 faßt den Stand der Technologie zusammen. Grundlagen und Details des hier beschriebenen Ansatzes finden sich in [22], [24], [23]. Eine Reihe verwandter Arbeiten und weitere Referenzen finden sich in [4].

## 2 Von informellen Anforderungen zu beweisbar korrekter Software

Das folgende Beispiel beruht auf einer gemeinsamen Pilotstudie[1] der Universität Ulm mit der Firma Intecs Sistemi im Auftrag der ESA (European Space Agency) und des BSI (Bundesamt für Sicherheit in der Informationstechnik). Es geht dabei um die Softwarekomponente SCCT, die für den sicheren Befehlstransfer im GNC System eines Raumfahrzeugs verantwortlich ist (Guidance and Navigation Control).

SCCT (Secure Command and Control Transfer) überträgt periodisch alle 2 Sekunden eine Tabelle von Steuerbefehlen (mission frame) an eine Reihe von Geräten an Bord des Raumfahrzeugs (Hauptyzklus). Die Tabelle wird dabei in 100 Unterzyklen (1 ... *maxcycle*) abgearbeitet. Die Übertragung der Steuerbefehle an ein Gerät unterbleibt jedoch, wenn es im aktuellen Unterzyklus defekt ist, oder von einer anderen Softwarekomponente (Anwendungssoftware) ein Deaktivierungsbefehl für das Gerät vorliegt. Deaktivierungen sind temporär oder permanent und können explizit aufgehoben werden. Temporäre Deaktivierungen werden spätestens zu Beginn des nächsten Hauptzyklus automatisch aufgehoben, permanente Deaktivierungen müssen von der Anwendungssoftware explizit aufgehoben werden. SCCT muß sicherstellen, daß niemals Steuerbefehle an ein defektes oder deaktiviertes Gerät übertragen werden.

[1] im Rahmen des Projekts VSE (Verification Support Environment). VSE ist eine gemeinsame Entwicklung des DFKI Saarbrücken, der IST GmbH München und der Universität Ulm im Auftrag des BSI

An diesem Beispiel soll das typische Vorgehen bei einer formalen Entwicklung illustriert werden:

- formale Spezifikation des funktionalen Verhaltens
- formale Spezifikation der Sicherheitsanforderungen
- Verifikation der Sicherheitsanforderungen und Validierung von Spezifikationen
- Implementierung und deren Verifikation

### 2.1 Beschreibung des funktionalen Verhaltens

Die informelle Beschreibung der zentralen Funktion in SCCT, *execute_cycle*, lautet:

*Eingabe von execute_cycle sind der aktuelle Unterzyklus k, der mission frame (frame), eine Statustabelle defekter Geräte (status), die Tabelle der bisher deaktivierten Geräte (disable) sowie die Liste der im Unterzyklus k anliegenden Aktivierungs- bzw. Deaktivierungsbefehle (aps) der Anwendungssoftware. Execute_cycle modifiziert die k-te Zeile des mission frame (Zuordnung von Geräten zu Steuerbefehlen). Für defekte oder deaktivierte Geräte wird statt der Originalsteuerbefehle, die Leeroperation no_op eingetragen (result.1). Weiterhin wird die disable-Tabelle gemäß der aktuellen Befehle in aps für die Folgezyklen modifiziert (result.2).*

Diese Beschreibung wird in eine formale Spezifikation übersetzt, deren wesentlichster Teil die folgenden Formeln sind. Die Funktion *execute_cycle* gibt als Resultat ein Paar von zwei Werten zurück, dessen erster mit .1 und dessen zweiter mit .2 selektiert wird. *Result* steht für das Ergebnis von $execute\_cycle(k, frame, status, disable, aps)$:

modify_row

$$\begin{aligned} & \text{cons(k, frame, status, disable, aps)} \\ & \wedge \text{bad} = \text{bad\_devices(disable, status, aps)} \\ & \wedge \text{r} = \text{row(k, frame)} \\ \rightarrow\ & \text{result.1} = \text{overwrite(bad, r, no\_op)} \end{aligned} \tag{1}$$

update_disable

$$\begin{aligned} & \text{cons(k, frame, status, disable, aps)} \\ & \wedge \text{k} < \text{maxcycle} \\ \rightarrow\ & \text{result.2} = \text{update(disable,aps, k)} \end{aligned} \tag{2}$$

$$\begin{aligned} & \text{cons(k, frame, status, disable, aps)} \\ & \wedge \text{k} = \text{maxcycle} \\ & \wedge \text{tmp} = \text{temporarily\_disabled(aps)} \\ \rightarrow\ & \text{result.2} = \text{update(disable,aps} - \text{tmp, k)} \end{aligned} \tag{3}$$

Bis auf zwei kritische Punkte lassen sich diese Formeln mit etwas Übung direkt aus der informellen Beschreibung ablesen. Ein kritischer Punkt ist die Fallunterscheidung in *update_disable*: Für $k = maxcycle$

muß die neue disable-Tabelle anders berechnet werden als für $k < maxcycle$, da temporäre Deaktivierungen im letzten Unterzyklus im nächsten Hauptzyklus nicht mehr beachtet werden. Dies folgt nicht direkt aus der informellen Beschreibung von *execute_cycle* sondern aus der allgemeinen Problembeschreibung zu Beginn von Abschnitt 2. Der zweite kritische Punkt ist die Einführung des Datenkonsistenzprädikats *cons* in allen drei Formeln. Dieses Prädikat macht eine Reihe von nirgends beschriebenen impliziten Annahmen explizit, z.B. daß der mission frame keine Leeroperationen vorsieht, daß *aps* keine widersprüchlichen Befehle für ein und dasselbe Gerät enthält usw. Dieser Zwang, alle impliziten Annahmen zu explizieren, führt in aller Regel zu einem besseren Verständnis des Problems und eliminiert Fehlerquellen.

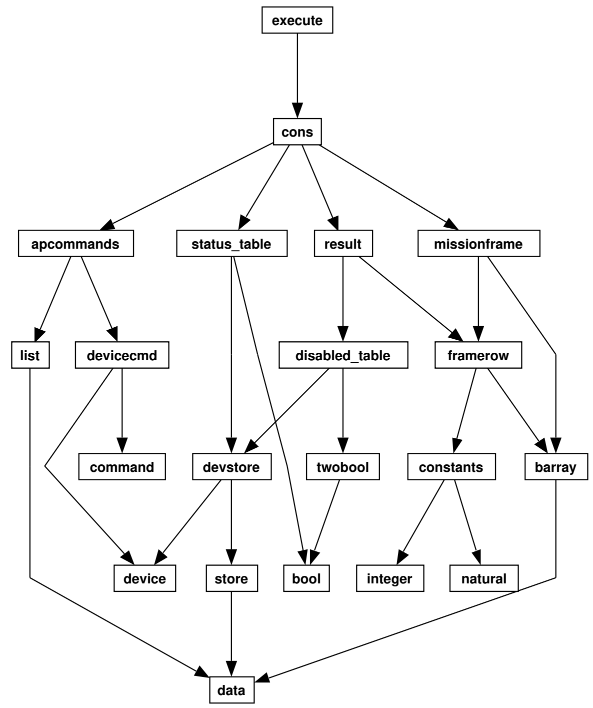


Abbildung 1: Graphische Darstellung der Spezifikationsstruktur

Um eine vollständige Spezifikation zu erhalten, müssen das Konsistenzprädikat ebenso wie alle anderen in den Formeln verwendeten Hilfsfunktionen axiomatisiert werden. Die folgende Formel spezifiziert z.B. die Hilfsfunktion *bad_devices* zur Berechnung der Menge aller Geräte, an die keine Steuerbefehle transferiert werden sollen:

$$
\begin{array}{ll}
& \text{dev} \in \text{bad\_devices(disable, status, aps)} \\
\leftrightarrow & \quad \text{device\_faulty(dev, status)} \\
& \lor \text{disable\_cmd\_sent(dev, aps)} \\
& \lor (\quad \text{device\_disabled(dev, disable)} \\
& \quad \land \neg\, \text{enable\_cmd\_sent(dev, aps))}
\end{array} \tag{4}
$$

Die gesamte, vom allgemeinen zum Detail hin strukturierte Spezifikation des gewünschten funktionalen Verhaltens von SCCT ist in Abbildung 1 gezeigt. Im folgenden wird sie mit SCCT-F bezeichnet. Die Rechtecke stehen für zusammengehörige Mengen von Axiomen, die Kanten beschreiben, wie die einzelnen Spezifikationskomponenten aufeinander aufbauen. Die theoretischen Grundlagen solcher strukturierter Spezifikationen finden sich in [29], [22].

### 2.2 Beschreibung der Sicherheitsanforderungen

Der nächste Entwicklungsschritt ist die Formalisierung der gestellten Sicherheitsanforderungen. Die informelle Formulierung von zwei der sieben Sicherheitsanforderungen für SCCT lautet:

- *Falls ein Gerät weder defekt noch deaktiviert ist und auch kein aktueller Deaktivierungsbefehl vorliegt, werden die zugehörigen Steuerbefehle übertragen.*
- *Falls ein Gerät defekt ist, werden keine Steuerbefehle übertragen*

Diese werden in folgende Formeln übersetzt:

regular_transfer:

$$
\begin{array}{ll}
& \text{cons(k, frame, status, disable, aps)} \\
& \land\ \text{result} = \text{execute\_cycle(k, frame, status,} \\
& \qquad\qquad\qquad \text{disable, aps)} \\
& \land\ \neg\, \text{device\_faulty(dev, status)} \\
& \land\ \neg\, \text{device\_disabled(dev, disable)} \\
& \land\ \neg\, \text{disable\_cmd\_sent(dev, aps)} \\
\rightarrow & \text{cmd\_transfer(dev, result.1)}
\end{array} \tag{5}
$$

no_transfer:

$$
\begin{array}{ll}
& \text{cons(k, frame, status, disable, aps)} \\
& \land\ \text{result} = \text{execute\_cycle(k, frame, status,} \\
& \qquad\qquad\qquad \text{disable, aps)} \\
& \land\ \text{device\_faulty(dev, status)} \\
\rightarrow & \neg\, \text{cmd\_transfer(dev, result.1)}
\end{array} \tag{6}
$$

### 2.3 Nachweis der Sicherheit und Validierung von Spezifikationen

Der nächste Schritt nach der Spezifikation des funktionalen Verhaltens und der Sicherheitsanforderungen ist der Nachweis, daß jede Implementierung, die SCCT-F erfüllt, tatsächlich die Sicherheitsanforderungen einhält. Konventionelle Qualitätstechnik ist hierbei auf Plausibilitätsbetrachtungen, Simulation oder Testen prototypischer Implementierungen angewiesen. Die Verwendung formaler Spezifikationen eröffnet hingegen die Möglichkeit, das Einhalten der Sicherheitsanforderungen als mathematischen Satz zu formulieren und zu beweisen. Es muß gezeigt werden, daß z. B. *regular_transfer* und *no_transfer* aus den Axiomen von SCCT-F gefolgert werden können:

$$\text{SCCT-F} \models \text{regular\_transfer} \land \text{no\_transfer} \tag{7}$$

Gelingt dieser Nachweis, dann ist sichergestellt, daß jede korrekte, künftige Implementierung von SCCT - wie immer sie aussehen mag - die Sicherheitsanforderungen einhält. Dieser Nachweis kann bereits zu einem Zeitpunkt geführt werden, zu dem noch keine Code-Zeile existiert. Schlägt der Nachweis fehl, so weist das auf Fehler oder Lücken im funktionalen Verhalten oder den Sicherheitsanforderungen hin. Durch die Analyse fehlgeschlagener Beweise erhält man wertvolle Korrekturhinweise.

Dasselbe Vorgehen kann auch zur Validierung formaler Spezifikationen eingesetzt werden. Durch den erfolgreichen Nachweis erwarteter Eigenschaften (Testformeln) gewinnt man Vertrauen, daß das formalisierte funktionale Verhalten die ursprüngliche Intention korrekt wiedergibt. Auch hier dient die Analyse fehlgeschlagener Beweisversuche zur Entdeckung von Modellierungsfehlern zum frühestmöglichen Zeitpunkt im Entwicklungsprozeß.

Der Erfolg und die Wirtschaftlichkeit dieses Vorgehens hängen natürlich in hohem Maße davon ab, wie effizient die anfallenden Beweisverpflichtungen gelöst werden können und wie sehr die Beweisfindung durch intelligente Werkzeuge unterstützt werden kann. Darauf gehen wir in Abschnitt 3 ein.

### 2.4 Implementierung und deren Verifikation

Erst jetzt sollte sich ein Software-Entwickler mit der Implementierung der funktionalen Spezifikation SCCT-F beschäftigen. Dabei wird deren Struktur ausgenutzt und jede Spezifikationskomponente separat implementiert. Dadurch entstehen stark modularisierte Systeme. Das folgende Programm zeigt einen Ausschnitt der Implementierung von *execute_cycle*. Das Programm *execute_cycle#* sowie eine Reihe weiterer Prozeduren bilden zusammen einen Implementierungsmodul, dessen Exportschnittstelle die Spezifikationskomponente *execute* aus SCCT-F (siehe Abbildung 1) ist, und dessen Importschnittstelle ebenfalls eine formale Spezifikation ist. Die Aufgabe besteht nun darin, zu zeigen, daß *execute_cycle#* und die anderen Prozeduren des Moduls die entsprechenden Axiome aus SCCT-F erfüllen.

```
execute_cycle#(k, frame, status, disable, aps
               : var result)
  var r = copy(getrow(k, frame)) in
    disable#(disable, aps : disable);
    setrow#(r, status, disable : r);
    if k = maxcycle then
      ctmp#(disable : disable)
    fi;
    result := r × disable
```

Die konventionelle Qualitätstechnik ist hier auf Tests bzw. Simulationen angewiesen, während durch die Verwendung formaler Spezifikationen die Korrektheit von Implementierungen als mathematischer Satz formuliert und bewiesen werden kann. Zu jedem Modul $M$ wird automatisch eine Menge von Beweisverpflichtungen $VC(M)$ erzeugt, deren Nachweis für die Korrektheit von $M$ notwendig und hinreichend ist, [21]. Der zu beweisende Satz lautet:

$$\mathrm{Import(M)} \cup \mathrm{IND} \models \mathrm{VC(M)} \tag{8}$$

Er besagt, daß die Beweispflichten VC(M) aus den Axiomen der Importschnittstelle (Import(M)) induktiv (IND) gefolgert werden können. Der erfolgreiche Nachweis aller Formeln aus VC(M) garantiert die Korrektheit der Implementierung. Die Analyse fehlgeschlagener Beweisversuche hilft wiederum, Fehler aufzudecken und zu korrigieren. Die Verifikation eines Moduls ist bereits zu einem Zeitpunkt möglich, zu dem noch keine Implementierung der Importschnittstelle existiert.

Durch methodische Einschränkung der Freiheitsgrade im Entwurf wird erreicht, daß die Korrektheit des Gesamtsystems kompositional ist, d.h. vollständig auf die Korrektheit der einzelnen Moduln zurückgeführt werden kann. Dadurch wird erreicht, daß der Verifikationsaufwand für ein modulares System lediglich linear in der Anzahl der Moduln wächst. Dies ist der Schlüssel zur Beherrschbarkeit der anfallenden Beweisaufgaben und eine Voraussetzung für die Skalierbarkeit und Wirtschaftlichkeit formaler Verifikation.

## 3 Das KIV-System

KIV ist ein fortgeschrittenes Unterstützungswerkzeug zur Entwicklung korrekter Software, [22], [24], [23], [11]. Unter einer einheitlichen, graphischen Oberfläche bietet KIV Unterstützung für alle genannten Phasen des Entwicklungsprozesses, von der Spezifikationsentwicklung und deren Dokumentation über den Nachweis von Sicherheitseigenschaften bis hin zur Implementierung und deren Verifikation. Insbesondere im Bereich der Beweisentwicklung (Proof-Engineering) bietet KIV weitreichende Unterstützung und einen hohen Automatisierungsgrad.

### 3.1 Spezifikationsentwicklung

Aufbauend auf elementaren algebraischen Spezifikationen können mit den Operationen Vereinigung, Anreicherung, Parametrisierung, Aktualisierung und Umbenennung große modulare Spezifikationen entwickelt werden. Deren syntaktische Korrektheit (insbes. Typkorrektheit) wird automatisch überprüft, und bei Änderungen inkrementell nachgeführt. Strukturierte Spezifikationen werden wie in Abbildung 1 als Entwicklungsgraphen visualisiert (daVinci, [8]), über die graphische Schnittstelle direkt angesprochen, bearbeitet und verwaltet. KIV verfügt über eine umfangreiche Bibliothek wiederverwendbarer Standardspezifikationen.

### 3.2 Beweisen in strukturierten Spezifikationen

Der Nachweis von Sicherheitseigenschaften, die Validierung von Spezifikationen (vgl. Abschnitt 2.3) oder der Beweis allgemein interessanter Eigenschaften erfolgt mit der interaktiven Beweiskomponente in KIV. Die Beweisfindung ist weitgehend automatisiert, in vielen Beweisen ist jedoch an wichtigen Stellen die Interaktion mit dem Benutzer erforderlich (z.B. zur Auswahl des nächsten Schritts aus einer angebotenen Auswahl oder zur Formulierung strategischer Lemmata). Diese Interaktion setzt voraus, daß der Benutzer mit Logik und mathematischer Beweisführung vertraut ist. Der Beweisgang wird als Baum graphisch visualisiert, über die graphische Schnittstelle direkt angesprochen, bearbeitet, modifiziert und dokumentiert. Das sogenannte Korrektheitsmanagement verwaltet Beweispflichten, Beweise, aber auch - wo dies sinnvoll ist - unfertige oder fehlgeschlagene Beweisversuche (zur Wiederverwendung). Es verhindert, daß Lemmata zyklisch verwendet werden (etwa Lemma A im Beweis von Lemma B, und Lemma B im Beweis von Lemma A). Das Korrektheitsmanagement sorgt außerdem dafür, daß bei den häufigen Änderungen von Lemmata nur diejenigen Beweise ungültig werden, die das geänderte Lemma verwenden.

### 3.3 Implementierung und deren Verifikation

Die Entwicklung modularer Programme wird in gleicher Weise wie die Spezifikationserstellung unterstützt. Die für den Korrektheitsnachweis eines Moduls $M$ erforderlichen Beweisverpflichtungen $VC(M)$ (siehe 2.4) werden vom System automatisch erzeugt und vom Korrektheitsmanagement verwaltet. Den eigentlichen Nachweis führt wiederum die Beweiskomponente von KIV. Bei der Verifikation von Software-Moduln findet die KIV-Beweiskomponente üblicherweise ca. 80 - 90 % der Beweisschritte automatisch, [23].

Da Programme nur im seltensten Fall im ersten Versuch korrekt sind, bietet KIV umfangreiche Unterstützung bei der Fehlersuche und -korrektur. Für die Fehlersuche kann zunächst eine Strategie zur Erzeugung von Gegenbeispielen eingesetzt werden. Tritt während der Beweissuche ein unbeweisbares Unterziel auf, so berechnet diese Beweisstrategie in vielen Fällen automatisch ein Gegenbeispiel für die Ausgangsbehauptung. Falls dies nicht möglich ist, unterstützt KIV die interaktive Fehlersuche. Durch einfaches Klicken auf den entspechenden Knoten des Beweisbaums kann jedes Zwischenziel angezeigt werden, und so der Beweisgang analysiert werden. Bei der Korrektur von Programmfehlern sorgt zunächst wieder das Korrektheitsmanagement dafür, daß nur die von einer Änderung echt betroffenen Beweise und Beweisversuche ungültig werden. Diese müssen anschließend neu bewiesen werden. Dabei bietet KIV eine ausgereifte Strategie zur automatischen Wiederverwendung früherer Beweise und Beweisversuche. Die Erfahrung hat gezeigt, daß durch diese Strategie im Durchschnitt etwa 90 % des Aufwands gegenüber einem Neubeweis gespart werden können, [25].

### 3.4 Anwendungen und Erfahrungen

Bisher wurde KIV in Zusammenarbeit mit über 10 Firmen, Behörden und Institutionen in Pilotanwendungen u.a. in den Bereichen Raumfahrt, Medizintechnik, Automobil- und Bahntechnik mit Erfolg erprobt. Beispielsweise wurde die Crash-Control-Software für die Airbags eines europäischen Automobilherstellers verifiziert. Bei der in Abschnitt 2 beschriebenen Raumfahrtfallstudie wurde durch die Verifikation eine bis dahin unbekannte Lücke im Sicherheitsmodell von SCCT entdeckt. Die größte Einzelanwendung ist ein Rundfunkdispositionssystem mit ca. 5000 Zeilen formaler Spezifikation, ca. 7000 Zeilen Code und einem Verifikationsaufwand von ca. 2 Personenjahren (VSE-Projekt).

Im Bereich der Zertifizierung wurde im Auftrag des Bundesamts für Sicherheit in der Informationstechnik ein formales Sicherheitsmodell zur Erstellung und Prüfung digitaler Unterschriften entwickelt und verifiziert. Insbesondere wurden dabei formale Anforderungen an die modellierten Komponenten isoliert, und die daraus folgende Integrität und Nicht-Rückweisbarkeit digitaler Unterschriften formal bewiesen.

KIV wird auch in der Analyse von Programmiersprachen insbesondere der Compilerverifikation eingesetzt. In einer Pilotstudie[2] wurde die Semantik der Programmiersprache Prolog sowie der Warren-Abstract-Machine (WAM) formal spezifiziert und die Übersetzungskorrektheit von Prolog in den Instruktionssatz der WAM bewiesen (Aufwand ca. 1 Personenjahr), [1], [26].

Im Zuge dieser Anwendungen wurde eine große Zahl von Fehlern entdeckt, die von einfachen Programmierfehlern bis hin zu konzeptuellen Anforderungs- und Modellierungsfehlern reichen. Folgende Erfahrungen haben sich im Laufe der Anwendungen herauskristallisiert:

- Der Einsatz formaler Spezifikationen führt durch den Zwang zur akribischen Analyse und Ausarbeitung der Anforderungen zu einer deutlichen Qualitätsverbesserung.

- Formale Spezifikationen ohne den Kontrollmechanismus des Beweisens führen jedoch nur zu einer graduellen Qualitätssteigerung. Die 'harten' Fehler werden erst beim Beweisen entdeckt. Für den angestrebten qualitativen Sprung ist die Verifikation von zentraler Bedeutung.

- Aufgrund der enormen Komplexität der Beweisprobleme, die im Rahmen einer realistischen Entwicklung anfallen, ist die Werkzeugun-

[2] im DFG-Schwerpunktprogramm Deduktion

terstützung beim Beweis-Engineering einer der wichtigsten, erfolgsbestimmenden Parameter.

- Verifikation ist nicht nur eine Frage des automatischen Beweisens, sondern auch eine Frage der Software-Entwicklungsmethodik. Kompositionalität der Korrektheit ist eine Grundvoraussetzung für eine effiziente Verifikationstechnologie.
- Die zentrale Fragestellung ist nicht der affirmative Korrektheitsnachweis eines Moduls oder einer Sicherheitseigenschaft, sondern das Aufspüren von Fehlern in Programmen und Spezifikationen, deren Korrektur und Re-Verifikation, bis schließlich korrekte Versionen entstehen.

Diese Erfahrungen werden auch durch die Resultate anderer Arbeitsgruppen bestätigt. Verwandte, leistungsfähige Systeme sind z.B. ACL2 (University of Texas at Austin, USA, [13]), PVS (SRI International, Stanford, CA, USA, [19]) und Isabelle (Cambridge University, GB, [20]).

## 4 Stand und Perspektiven

Mit dem KIV-System können in der Anwendungsprogrammierung aktuell mehrere tausend Zeilen verifizierten Codes pro Personenjahr erreicht werden. Dies umfaßt die Problemanalyse, die Erstellung der formalen Spezifikationen, Schreiben der Implementierungen sowie die Durchführung aller Beweise. Durch die spezielle Entwicklungsmethodik wächst der Aufwand bei großen Systemen lediglich linear in der Anzahl der Moduln. Ein ausgefeiltes Korrektheitsmanagement und die Möglichkeit zur Wiederverwendung von Beweisen erleichtern den Umgang mit Fehlern und erlauben eine effiziente Verzahnung von Verifikation und Fehlerkorrektur. Das Herzstück von KIV ist eine leistungsfähige interaktive Beweiskomponente mit hohem Automatisierungsgrad. Die in KIV und anderen Systemen verwendete Technologie steht heute an der Schwelle zum industriellen Einsatz. Sie eignet sich nicht nur für sicherheitskritische Systeme, sondern für alle Anwendungen, bei denen die potentiellen Ausfallschäden besonders gravierend sind. Dies gilt z.B. für Software in 'Massenprodukten', deren Fehler zu kostspieligen Rückrufaktionen führen können. Ein weiteres Einsatzgebiet formaler Verifikation sind zentrale Software-Dienste, Generatoren und Systemprogramme, über die besonders viele Anwendungen abgewickelt werden. Fehler in solchen Systemen bergen die Gefahr der unkontrollierten Verfielfältigung. Fehlerhafte Compiler, zum Beispiel, könnten selbst für verifizierte Anwendungsprogramme fehlerhaften Objektcode erzeugen. Daher ist es wichtig, auch Systemsoftware formal zu verifizieren (siehe dazu auch [14], [28]).

Produktivitätssteigerungen sind in der Zukunft vor allem von der Wiederverwendung von Spezifikationen, Beweisen und bereits verifizierten Komponenten zu erwarten. Auch die Beweiskomponente birgt weiteres Entwicklungspotential. Die Erweiterung auf verteilte, reaktive Systeme sowie der Anschluß an zustandsendliche Systeme ist in Entwicklung. Damit steht dann eine leistungsfähige Technologie zur Verfügung, die ein sehr breites Spektrum industriell relevanter Systeme abdeckt.

## Literatur

[1] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam, 1995.

[2] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java (tm)*, LNCS. Springer, 1998. To appear.

[3] J. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog, and A. P. Ravn. A ProCoS II Project Final Report: ESPRIT Basic Reserach Project 7071. *EATCS-Bulletin*, 1996.

[4] M. Broy and S. Jähnichen, editors. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer, Heidelberg, Nov 1995.

[5] CCIB-98-026. *Common Criteria for Information Technology Security Evaluation, Version 2.0*. ISO/IEC JTC 1, May 1998. availiable at http://csrc.nist.gov/cc.

[6] R. Cohen. An acl2 formalization of the java virtual machine. Technical report, Computational Logic Inc., 1997.

[7] Deutsche Telekom. Südwestpresse 11. Januar, 1996.

[8] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system *daVinci*. In R. Tamassia and I. Tollis, editors, *DIMACS Workshop on Graph Drawing '94. Proceedings*, Springer LNCS 894. Princeton (USA), 1994. http://www.informatik.uni-bremen.de/~davinci/.

[9] W.W. Gibbs. Software's Chronic Crisis. *Scientific American*, September 1994.

[10] W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Rueß, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. Technical Report LiTH-IDA-R-96-12, Linkoping University, 1996.

[11] M. Heisel, W. Reif, and W. Stephan. Tactical Theorem Proving in Program Verification. In M. Stickel, editor, *10th International Conference on Automated Deduction. Proceedings.* Springer LNCS 449, 1990.

[12] ITSEC. *Information Technology Security Evaluation Criteria, Version 1.2.* Office for Official Publications of the European Communities, June 1991.

[13] M. Kaufmann and J Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4), April 1997.

[14] Hans Langmaack. Software Engineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzungskorrektheit . In M. Broy, editor, *it+ti Themenheft: Formale Entwurfsmethoden – Software Correctness.* Oldenbourg Verlag, 1997.

[15] J Moore. PITON: A Verified Assembly Level Language. Technical report 22, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.

[16] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm w in isabelle/hol. In C. Paulin-Mohring, editor, *Proceedings of the International Workshop TYPES'96.* Springer LNCS, 1997. To appear.

[17] Peter G. Neumann. *Computer Related Risks.* Addison Wesley, 1995.

[18] T. Nipkow and D. von Oheimb. Java light is type-safe — definitely. In *25th ACM Symposium on Principles of Programming Languages.* ACM, 1998.

[19] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction - CADE-11. Proceedings*, Springer LNAI 607. Saratoga Springs, NY, USA, 1992.

[20] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.

[21] W. Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science.* Springer LNCS 620, 1992.

[22] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.

[23] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg (MD), USA, 1995.

[24] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, 1998.

[25] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science. Proceedings.* Springer LNCS 761, 1993.

[26] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering. Kluwer Academic Publishers, 1998.

[27] D. von Oheimb and T. Nipkow. Machine-checking the java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java (tm)*, LNCS. Springer, 1998. To appear.

[28] U. Hoffmann W. Goerigk and H. Langmaack. Rigorous compiler implementation correctness: How to prove the real thing correct. Technical report, CAU Kiel, January 1997.

[29] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, 1990.