Software-Verifikation und ihre Anwendungen

Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Reif, Wolfgang. 1997. "Software-Verifikation und ihre Anwendungen." *it - Information Technology* 39 (3): 34–40. https://doi.org/10.1524/itit.1997.39.3.34.

Software-Verifikation und ihre Anwendungen

Wolfgang Reif, Abteilung Programmiermethodik, Universität Ulm

Prof. Dr. Wolfgang Reif studierte Informatik an der Universität Karlsruhe und war anschließend dort Assistent und Projektleiter am Institut für Logik, Komplexität und Deduktionssysteme. Er ist Professor für Informatik an der Universität Ulm, Leiter der KIV-Arbeitsgruppe und Sprecher der GI-Fachgruppe 1.2.1 Deduktionssysteme. Seit zwei Jahren ist er mit seiner Arbeitsgruppe in der Abteilung Programmiermethodik in Ulm. Forschungsschwerpunkte: Softwaretechnik, Software-Sicherheit, Formale Methoden, Logik und maschinelles Beweisen.

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 die verschiedenen Einsatzmöglichkeiten formaler Spezifikations- und Verifikationsmethoden. Am Beispiel von KIV, einem fortgeschrittenen Spezifikations- und Verifikationswerkzeug, wird der aktuelle Leistungsstand und die Wirtschaftlichkeit der Technologie erläutert.

Software Verification and its Applications

In the spectrum of quality assuring Software-Engineering methods formal specification and verification techniques offer the maximal correctness guarantee that can be achieved today. The correctness of the software is established by mathematical proof. Formal methods are appropiate for applications with very high reliability and quality requirements. This article gives an overview of where and how to apply formal specification and verification methods. The current state of affairs is illustrated with KIV, an advanced support tool for formal specification and verification.

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 US-Dollar [6].
- **Airbusabsturz Toulouse, 1994.** Software-Fehlverhalten ist mitverantwortlich für den Absturz eines A340 bei Toulouse. 7 Tote [9].
- **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 [9].
- **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 [4].

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, Entwurf 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 Sicherheitsga-

rantien bzw. Fehlerfreiheit von Progammen mit mathematischen Mitteln beweisen kann (Verifikation). Der folgende Abschnitt illustriert die verwendeten Sprachen und das methodische Vorgehen an einem realen Beispiel. 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 [13; 14]. Eine Reihe verwandter Arbeiten und weitere Referenzen finden sich in [3].

2 Von informellen Anforderungen zu beweisbar korrekter Software

Das folgende Beispiel beruht auf einer gemeinsamen Pilotstudie[1] der KIV-Gruppe 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.

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 und
- Implementierung und deren Verifikation.

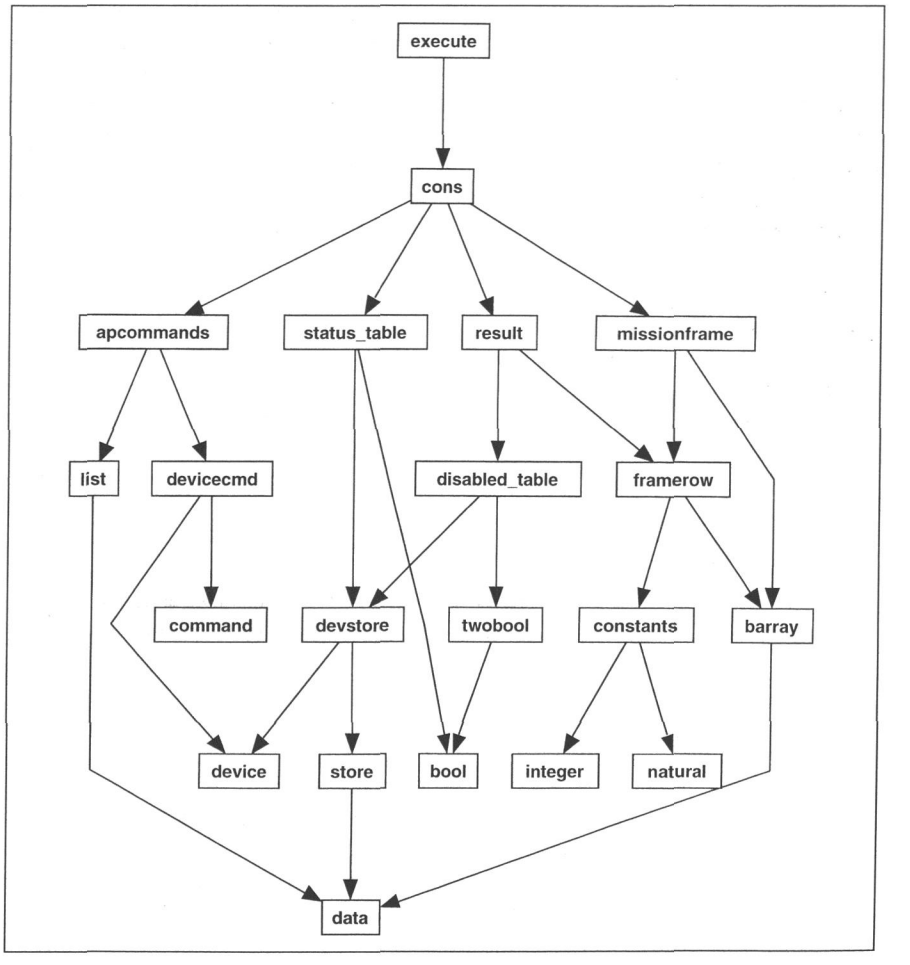
2.1 Beschreibung des funktionalen Verhaltens

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*):

Bild 1: Graphische Darstellung der Spezifikationsstruktur.



[1] im Rahmen des BSI-Projekts VSE (Verification Support Environment).

```
modify_row
     cons(k, frame, status, disable, aps)
   ∧ bad = bad_devices (disable, status, aps)
   ∧ r = row(k, frame)
→ result.1 = overwrite (bad, r, no_op)          (1)

update_disable
     cons(k, frame, status, disable, aps)
   ∧ k < maxcycle
→ result.2 = update (disable, aps, k)           (2)
     cons(k, frame, status, disable, aps)
   ∧ k = maxcycle
   ∧ tmp = temporarily_disabled(aps)
→ result.2 = update(disable,aps - tmp, k)       (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 *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.

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 (siehe Gl. (4)).

```
   dev ∈ bad_devices (disable, status, aps)
↔     device_faulty(dev, status)
   ∨  disable_cmd_sent (dev, aps)
   ∨ (device_disabled (dev, disable)
      ∧¬enable_cmd_sent (dev, aps))             (4)

regular_transfer:
     cons(k, frame, status, disable, aps)
   ∧ result = execute_cycle (k, frame, status,
     disable, aps)
   ∧ ¬ device_faulty(dev, status)
   ∧ ¬ device_disabled(dev, disable)
   ∧ ¬ disable_cmd_sent (dev, aps)
→ cmd_transfer(dev, result.1)                   (5)

no_transfer:
     cons(k, frame, status, disable, aps)
   ∧ result = execute_cycle (k, frame, status,
     disable, aps)
   ∧ device_faulty(dev, status)
→ ¬ cmd_transfer(dev, result.1)                 (6)
```

Die gesamte, vom allgemeinen zum Detail hin strukturierte Spezifikation des gewünschten funktionalen Verhaltens von SCCT ist in Bild 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 [17; 13].

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 den Formeln (siehe Gl. (5) und (6)) übersetzt.

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:

```
SCCT-F ⊨ regular_transfer
∧ no_transfer                                   (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 ist, (siehe Bild 1) 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 (siehe Gl. oben).

```
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 [12]. Der zu beweisende Satz lautet:

$$\text{Import}(M) \cup \text{IND} \models \text{VC}(M) \quad (8)$$

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 mit 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 [7; 13; 14]. 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 Bild 1 als Entwicklungsgraphen visualisiert [5], 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 (siehe Bild 2). 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.

Die eigentliche Beweisstrategie des Systems basiert auf Induktion, effizient implementierten Simplifikationstechniken (z.B bedingte Termersetzung und Eliminationstechniken für definierte Funktionssymbole) und Heuristiken zur intelligenten Quantorenbehandlung.

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. Die Beweisstrategie basiert auf den schon genannten Beweistechniken, ergänzt um spezielle Beweisregeln für Programme. Bei der Verifikation von Software-Moduln findet die KIV-Beweisstrategie üblicherweise ca. 80–90% der Beweisschritte automatisch [14].

Da ein Programm nur im seltensten Fall im ersten Versuch korrekt ist, 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 [15].

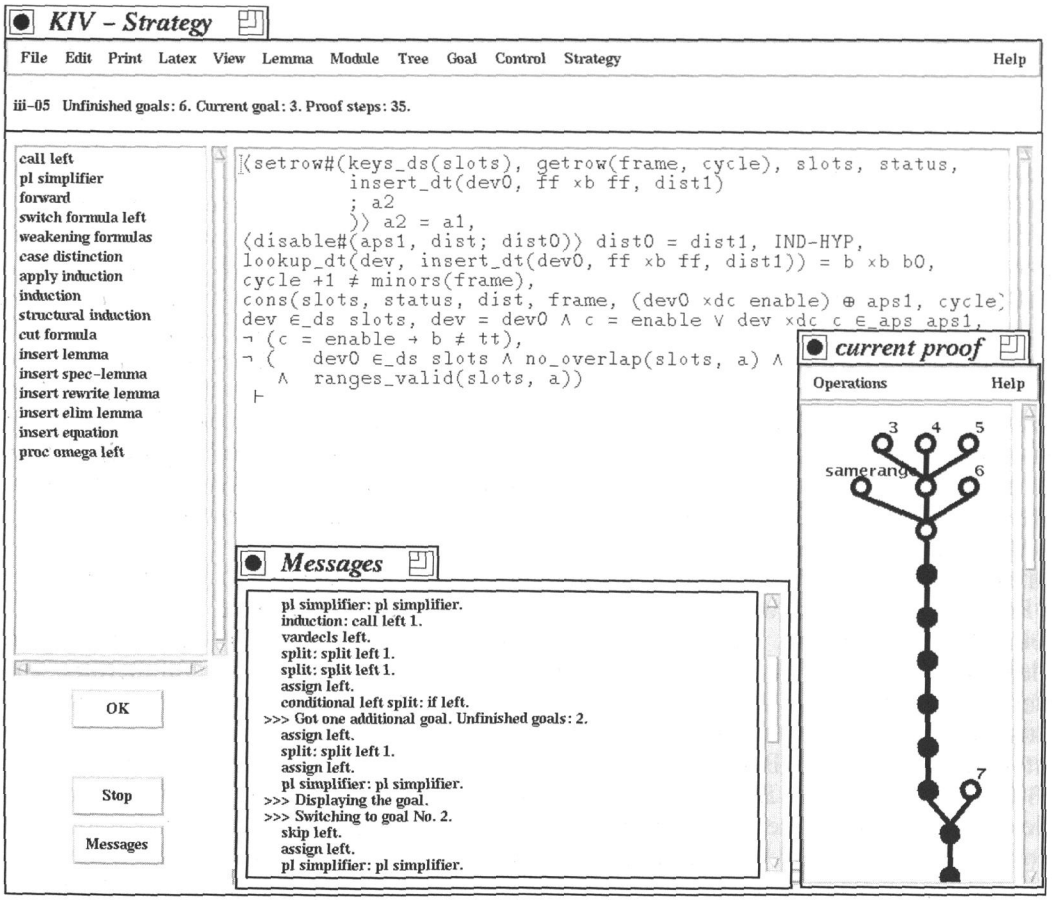


Bild 2: Beweisen im KIV-System.

3.4 Anwendungen und Erfahrungen

Bisher wurden mit dem KIV-System weit über 15000 Zeilen formaler Spezifikationen erstellt und validiert (bzw. Sicherheitseigenschaften bewiesen). Etwa ebensoviele Zeilen Implementierung wurden verifiziert. Dabei wurden mehrere hundert Fehler in Spezifikationen und Programmen entdeckt. Die Technologie wurde 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). KIV wird auch in der Compilerverifikation eingesetzt. In einer Pilotstudie[2] wird die Übersetzungskorrektheit von der Programmiersprache Prolog in den Instruktionssatz der Warren-Abstract-Machine (WAM) bewiesen (Aufwand ca. 1 Personenjahr) [1; 16].

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. „Harte" 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 Werkzeugunterstützung 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. Nqthm (CL Inc., Austin, Texas, USA [2]), PVS (SRI, Stanford, CA, USA [10]) und Isabelle (Cambridge University, GB [11]).

4 Stand und Perspektiven

Mit dem KIV-System kann aktuell eine Produktivität von ca. 1000 – 2000 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 mit 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 Vervielfältigung. Fehlerhafte Compiler zum Beispiel könnten selbst für verifizierte Anwendungsprogramme fehlerhaften Objektcode erzeugen. Daher ist es wichtig, auch Systemsoftware formal zu verifizieren (siehe dazu auch [8]).

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. Einer der nächsten Schritte in KIV ist die Erweiterung auf verteilte, reaktive Systeme sowie der Anschluß an zustandsendliche Systeme. Damit steht dann eine leistungsfähige Technologie zur Verfügung, die ein sehr breites Spektrum industriell relevanter Systeme abdeckt.

[2] im DFG-Schwerpunktprogramm Deduktion.

Literatur

[1] *Börger, E., Rosenzweig, D.:* The WAM – definition and compiler correctness. In: Beierle, C., Plümer, L. (Ed.): Logic Programming: Formal Methods and Practical Applications. Volume 11 of Studies in Computer Science and Artificial Intelligence. North-Holland, Amsterdam, 1995.

[2] *Boyer, R. S., Moore, J. S.:* A Computational Logic. Academic Press, 1979.

[3] *Broy, M., Jähnichen, S.* (Ed.): KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report. Volume 1009 of LNCS. Springer, Heidelberg, Nov 1995.

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

[5] *Fröhlich, M., Werner, M.:* Demonstration of the interactive graph visualization system davinci. In: Tamassia, R., Tollis, I. (Ed.): DIMACS Workshop on

Graph Drawing '94. Proceedings, Springer LNCS 894. Princeton (USA), 1994.

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

[7] *Heisel, M., Reif, W. Stephan, W.:* Tactical Theorem Proving in Program Verification. In: Stickel, M. (Ed.): 10th International Conference on Automated Deduction. Proceedings, Springer LNCS 449. Kaiserslautern, Germany, 1990.

[8] *Langmaack, H.:* Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit. In: it+ti 39 (1997) 3 Oldenbourg Verlag, 1997.

[9] *Neumann, P. G.:* Computer Related Risks. Addison Wesley, 1995.

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

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

[12] *Reif, W.:* Correctness of Generic Modules. In: Nerode, Taitslin (Ed.): Symposium on Logical Foundations of Computer Science. Springer LNCS 620. Logic at Tver, Tver, Russia, 1992.

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

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

[15] *Reif, W., Stenzel, K.:* Reuse of Proofs in Software Verification. In: Shyamasundar, R. (Ed.): Foundation of Software Technology and Theoretical Computer Science. Proceedings, Springer LNCS 761. Bombay, India, 1993.

[16] *Schellhorn, G., Ahrendt, W.:* Verification of a Prolog Compiler – First Steps with KIV. Ulmer Informatik-Berichte 96-05, Universität Ulm, Fakultät für Informatik, 1996.

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

Professor Dr. Wolfgang Reif
Universität Ulm,
Fakultät für Informatik,
Abteilung Programmiermethodik,
89069 Ulm,
Email: reif@informatik.uni-ulm.de