Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving*

Michael Goldapp
LINEAS GmbH
Braunschweig

Ulrich Grottker
Physikalisch-Technische Bundesanstalt
Braunschweig

Gregor Snelting
Technische Universität
Braunschweig

Zusammenfassung

Meßgeräte, die zur Abwicklung von Geschäften, in der Verkehrsüberwachung oder im Umweltschutz eingesetzt werden, müssen besonderen Sicherheitsanforderungen genügen. Da heutzutage fast jedes Meßgerät durch Software gesteuert wird, muß sichergestellt werden, daß nicht der Datenpfad vom Sensoreingang zur Anzeige des gemessenen Wertes (*Eichpfad*) durch externe Faktoren beeinflußt werden kann. Es soll deshalb ein Werkzeug entwickelt werden, das Beeinflussungen des Eichpfades erkennt, analysiert und in Form einer Mängelliste zusammenstellt und visualisiert. Damit soll die *Manipulationssicherheit* und *Robustheit* der Meßgerätesoftware validiert werden.

Grundlage der Analyse sind *Program Slicing* und *Constraint Solving*. Aus dem Programmquelltext wird ein Programmabhängigkeitsgraph (PDG) erzeugt. Dieser erlaubt es, zu beliebigen Programmpunkten (z.B. Meßwertausgaben) diejenigen Anweisungen zu bestimmen, die diesen Punkt beeinflussen. Zusätzlich werden die Pfadbedingungen der verdächtigen Pfade aufgesammelt und durch Constraint Solving nach den Datenquellen aufgelöst. Ergebnis ist eine Liste von Aussagen der Form „Falls keycode=Esc und mousecode=left, wird die Meßgeräteanzeige um 10% verfälscht“.

1 Einleitung und Übersicht

Meßgeräte, die zur Abwicklung von Geschäften, in der Verkehrsüberwachung oder im Umweltschutz eingesetzt werden, müssen besonderen Sicherheitsanforderungen genügen. Hersteller von Meßgeräten wollen deshalb Mängel in den Geräten erkennen und sie robust gegen Fehlbedienung und manipulationssicher konstruieren. Da heutzutage fast jedes Meßgerät durch Software gesteuert wird, ist die Software-Validierung Teil des Meßgeräte-Prüfprozesses. Insbesondere muß sichergestellt werden, daß nicht der Datenpfad vom Sensoreingang zur Anzeige des gemessenen Wertes (*Eichpfad*) durch externe Faktoren beeinflußt werden kann.

Im Rahmen des Vorhabens “Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving” soll ein Werkzeug entwickelt werden, das fehlerhafte Datenflüsse erkennt, analysiert und in Form einer *Mängelliste* zusammenstellt. Damit soll

*Gefördert vom BMBF, FKZ 01 IS 513 C9

die *Manipulationssicherheit* und *Robustheit* der Software, also die Resistenz gegen beabsichtigte oder unbeabsichtigte Beeinflussung der Ergebnisse validiert werden. Insbesondere für Meßgerätesoftware können so Beeinflussungen des Eichpfades aufgedeckt werden. Da zur Beurteilung von Meßgerätesoftware auf menschliches Expertenwissen nicht verzichtet werden kann, soll das Werkzeug interaktiven Zugang sowie eine Visualisierung der gewonnenen Information anbieten. Das Werkzeug hat einen sprachunabhängigen Analysekern, der zur Software-Qualitätssicherung auch in anderen Bereichen eingesetzt werden kann.

Grundlage der Analyse sind *Program Slicing* und *Constraint Solving*. Aus dem Programmquelltext wird ein Programmabhängigkeitsgraph (PDG) erzeugt. Dieser erlaubt es, zu beliebigen Programmpunkten (insbesondere Meßwerteausgaben, sog. Datensenken) diejenigen Anweisungen (insbesondere Sensor-/Tastatureingaben oder andere Datenquellen) zu bestimmen, die diesen Punkt beeinflussen. Ein Datenfluß von außen in den Eichpfad hinein bedeutet jedoch noch nicht notwendig eine unerlaubte Beeinflussung und muß deshalb weiter analysiert werden. Dazu werden die Pfadbedingungen der verdächtigen Pfade aufgesammelt und durch Constraint Solving nach den Datenquellen aufgelöst. Ergebnis ist eine Liste von Aussagen der Form „Falls keycode=Esc und mousecode=left, wird die Meßgeräteanzeige um 10% verfälscht".

Das Werkzeug soll so weit entwickelt werden, daß es zu Testzwecken bei ausgewählten Meßgeräteherstellern eingesetzt werden kann. Durch Austausch des Frontends können verschiedene Sprachen (z.B. C, Pascal und Fortran) unterstützt werden. Anwendungen zur Software-Qualitätssicherung in anderen Bereichen sind ohne weiteres möglich.

2 Ausgangssituation im Meßgerätebereich

2.1 Wirtschaftliche Bedeutung eichpflichtiger Meßgeräte

Die Physikalisch-Technische Bundesanstalt ist verantwortlich für die Baumusterprüfung aller eichpflichtigen Meßgeräte. Hersteller, deren Produkte im Bereich des gesetzlichen Meßwesens eingesetzt werden, müssen dabei die Resistenz gegenüber mißbräuchlicher oder unbeabsichtigter Fehlbedienung nachweisen. Welche Schäden tatsächlich durch mangelnde Robustheit und Manipulationssicherheit entstehen, läßt sich nicht exakt angeben. Um welche Größenordnungen es hierbei jedoch allein im gesetzlichen Meßwesen geht, verdeutlichen die folgenden Zahlen:

- Ausgehend von über 140 Milliarden Litern Mineralöl, die jährlich über entsprechende Volumenzähler abgerechnet werden, muß bei einem Anteil von Fehlmessungen von nur 1% und einem Literpreis von 1 DM bereits mit einem Verlust in Milliardenhöhe gerechnet werden.
- Über eine Großwaage zur Schüttgutverwiegung werden ca. 1 Million DM/Wägung abgerechnet, so daß sich bei 100 Wägungen im Jahr ein Umsatz von 100 Mill. DM/Jahr ergibt. Bei einem Anteil von Fehlmessungen von nur 1% wäre somit allein pro Waage ein Verlust in Millionenhöhe entstanden.
- Bei einem Bestand von 4 Millionen elektronischen Elektrizitätszählern, die ab einem Verbrauch von 10 MWh/Jahr eingesetzt werden, beläuft sich der über diese Zähler abgerechnete Umsatz bei 1000 DM/Jahr und Zähler auf insgesamt mindestens 4

Milliarden DM/Jahr, so daß wiederum bei einem Anteil von Fehlmessungen von nur 1% mit Schäden in 2stelliger Millionenhöhe zu rechnen wäre.

Im gesetzlichen Bereich läßt sich die wirtschaftliche Relevanz insbesondere auch anhand der Zulassungszahlen von software-gesteuerten Meßgeräten belegen, die bei 1800 Zulassungen (1993) für 85 verschiedene Gerätearten ein enormes Marktpotential repräsentieren. Wie Erhebungen zeigen, sind im Bereich des gesetzlichen Meßwesens, das ein breites Gerätespektrum (z.B. Volumen-, Gas-, Brennwert-, Temperaturmeßgeräte, Waagen, Choirometer, Verkehrsmeßgeräte, usw.) abdeckt, mehrere tausend Firmen und dabei vorzugsweise kleinere Unternehmen mit Fragen der Software-Sicherheit befaßt. Dabei ist zu beachten, daß heute im gesetzlichen Meßwesen bereits 95% aller Meßgeräte rechnergesteuert sind. Die momentan häufig praktizierte manuelle Validierung der Software kann jedoch mit dem zunehmenden Softwareeinsatz nicht Schritt halten. Auch äußern Hersteller immer häufiger den Wunsch nach Qualitätszertifikaten, um ihre Wettbewerbsposition zu verbessern.

2.2 Validierung von Meßgerätesoftware

Die Software-Sicherheit (Resistenz gegenüber absichtlicher und unbeabsichtigter Fehlbedienung), die bei software-gesteuerten Systemen neben der Korrektheit als wesentliches Qualitätsmerkmal zu werten ist [DGQ 86], wird derzeit mit Hilfe von manuellen, zeitaufwendigen und mit großen Unsicherheiten behafteten Methoden überprüft und bestenfalls durch den Einsatz von Analysatoren unterstützt.

Forschungsanstrengungen auf dem Gebiet der Software-Analyse konzentrierten sich bisher vorzugsweise auf Verfahren zur Fehleraufdeckung in der Entwicklung befindlicher oder fertiggestellter Software, während Aspekte wie die Manipulationssicherheit und Robustheit gegenüber Bedienungsfehlern eher sekundär betrachtet wurden.

Zur Qualitätssicherung und Validierung von Software-Systemen kommen je nach geforderter Prüftiefe folgende Verfahren zum Einsatz [VDI 93], [DGQ 92], [DIN90]:

- Informelle Verfahren (wie Inspektion, Review, Walkthrough), die auf der manuellen Prüfung von Dokumenten (z.B. Architekturentwurf) bzw. der gedanklichen Ausführung von Dokumenten beruhen, werden bei geringen Qualitätsniveaus angewandt.
- Für mittlere bis höhere Prüftiefen werden statische Analysen eingesetzt, die meist rechnergestützt erfolgen (mit Hilfe von Analysatoren) und sichere und reproduzierbare Aussagen über statische Eigenschaften wie Daten- und Kontrollfluß liefern.
- Zur Erfassung der dynamischen System-Eigenschaften und zur Überprüfung der Robustheit werden ergänzend Testverfahren (Black- und White-Box-Tests) eingesetzt, die aufgrund ihres Stichprobencharakters (begrenzte Menge von Eingabedaten) als alleiniger Sicherheitsnachweis jedoch in der Regel als unzureichend anzusehen sind [VDI 93].
- Für höchste Qualitätsanforderungen sind mathematische Korrektheitsbeweise zu erbringen [DIN 90], [DGQ 92]. Diese Verfahren, die in ihrer Leisungsfähigkeit informelle und statische Analysemethoden bei weitem übertreffen, scheiden in der Praxis jedoch häufig aufgrund ihres enormen Aufwandes aus.

- Sehr viel wirtschaftlicher ist der Einsatz von entwicklungsbegleitenden Werkzeugen, die strengen Top-down-Entwurf erzwingen, Dokumentation liefern und zum Teil formale Verifikation ermöglichen (CASE-Tools wie z.B. EPOS, RAISE, VSE). Sie setzen aber bereits bei der formalen Spezifikation an und sind daher für die Überprüfung bereits existierender Systeme ungeeignet.

Zur Validierung von Meßgeräte-Software im Hinblick auf Manipulationssicherheit und Robustheit existieren keine speziell zugeschnittenen Werkzeuge. Software in geeichten Meßgeräten gehört aber ganz klar in den sensiblen Bereich und hat – wenngleich sie nicht so stark im Blickpunkt des öffentlichen Interesses steht – eine mit Fahrzeugsteuerungen durchaus vergleichbare Bedeutung. Methoden und Werkzeuge zur Software-Qualitätssicherung sind notwendig, um auf Dauer den gesetzlichen Anforderungen Genüge zu tun. Mithin besteht ein starkes öffentliches Interesse an Verfahren, die den Prüfprozeß automatisieren und dadurch genauer und effizienter machen können. Denn solche Verfahren haben wiederum verbesserte Sicherheit und Zuverlässigkeit der Software zur Folge, und dies kommt im Bereich des Meßwesens nicht nur den Herstellern, sondern letztlich allen Endverbrauchern zugute.

2.3 Allgemeine Ziele des Forschungsvorhabens

Die beschriebene Ausgangssituation – hohe wirtschaftliche Bedeutung softwaregesteuerter Meßsysteme bei unzureichender Validierungsunterstützung – war Motiv und Ausgangspunkt für das Forschungsvorhaben "Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving". Im Prinzip ist nämlich mit dem Program Slicing, ergänzt durch Constraint Solving, eine Technologie verfügbar, die zur (semi)automatischen Validierung von Meßgerätesoftware geeignet ist. Zwar wird Program Slicing in Deutschland fast nicht angewandt, dies liegt aber lediglich am mangelnden Technologietransfer. Wie kürzlich bekannt wurde, hat auch das amerikanische "National Bureau of Standards" ein Software-Validierungsprojekt aufgelegt, das auf Program Slicing basiert.

Der Transfer neuer Ergebnisse der softwaretechnologischen Grundlagenforschung in den Meßgerätebereich wird aus Anwendersicht viele Vorteile bringen. Unser Vorhaben verbessert die Situation der mittelständischen Hersteller und Abnehmer software-gesteuerter Meßsysteme, die in die Lage versetzt werden, eigene Qualitätsanalysen kostengünstig und effektiv durchzuführen und in Qualitätssicherungssysteme (gemäß DIN EN ISO 9000) zu integrieren. Im gesetzlich geregelten Bereich eröffnet sich den Herstellern zudem die Möglichkeit, bei Nachweis eines zertifizierten Qualitätssicherungssystems selbst für die Qualität ihrer Produkte verantwortlich zu zeichnen und damit Eichungen und zukünftig staatliche Zulassungsteilprüfungen selbst durchzuführen. Neben dem Konformitätsbewertungsverfahren für Industrieprodukte besitzt das zu entwickelnde Werkzeug Bedeutung im Zusammenhang mit dem Produkthaftungsgesetz sowie bei der gesamtwirtschaftlich bedeutsamen Schadensbegrenzung (Sach- und Personenschäden) infolge möglicher Fehlfunktionen. Ein Qualitätssicherungssystem erleichtert den Software-Herstellern, in Verfahren nachzuweisen, daß ein Schaden durch das Produkt nicht auf Fahrlässigkeit in der Produktion zurückzuführen ist. Es wird dabei vermehrt auch mit Gutachten und Zertifikaten geworben, die eine eingehende Überprüfung der Sicherheit voraussetzen.

Im Rahmen der EU kann mit der Entwicklung geeigneter Werkzeuge zugleich ein Beitrag zur Umsetzung der EU-Richtlinien geleistet werden, die für den Bereich des gesetzlichen

Meßwesens (d.h. im geschäftlichen und amtlichen Verkehr sowie im Umweltschutz) den Nachweis gegenüber absichtlicher und unbeabsichtigter Fehlbedienung fordern. Handlungsbedarf auf dem Gebiet der Manipulationssicherheit und Robustheit besteht insofern, als in Europa derzeit eine uneinheitliche Prüfpraxis existiert. Aufgrund fehlender Anforderungsprofile, Prüfwerkzeuge und Spezialisten wird eine Software-Prüfung z.T. nur unzureichend oder gar nicht vorgenommen.

Die im folgenden beschriebenen Verfahren zur automatischen Überprüfung von Manipulationssicherheit und Robustheit lassen sich nicht nur bei der Prüfung von Meßgeräten anwenden, sondern auch in verwandten Bereichen (z.B. Zertifizierungsstellen, Prüflaboratorien, TÜVs). Allgemein ist das Aufdecken verdächtiger Datenflüsse nebst Analyse der Randbedingungen sicher ein nützliches Instrument für die Software-Qualitätssicherung.

3 Wissenschaftliche Grundlagen

3.1 Program Slicing

Wir wollen nun die softwaretechnologischen Grundlagen des zu entwickelnden Werkzeugs beschreiben. Wir beginnen mit einer Beschreibung des Program Slicing. Unsere Darstellung ist informal; formale Definitionen finden sich z.B. in dem ausgezeichneten Übersichtsartikel von Frank Tip [Ti95].

Ein (Rückwärts)Slice eines Programms P zu einem gegebenen Programmpunkt s ist die Menge aller Anweisungen, die diesen Punkt beeinflussen können. Zum Slice gehören Anweisungen, die in s verwendete Variablen verändern, aber auch Anweisungen, die s kontrollierende Prädikate beeinflussen. Slices wurden als Hilfsmittel zum Debugging definiert; Programmierer machen nämlich beim Testen auch nichts anderes als bei fehlerhaften Zwischenergebnissen rückwärts nach verursachenden Anweisungen zu suchen. Man verlangt von den Anweisungen des Slices, daß sie für die Variablen am Slicepunkt dieselben Werte erzeugen wie das ursprüngliche Programm. Abbildung 1 zeigt ein einfaches Beispiel: Wenn man nur wissen will, welche Anweisungen den ausgegebenen Wert von `product` beeinflussen können, so entfallen die Anweisungen in Zeile 3,6 und 9; der Wert von `sum` nämlich geht in keiner Weise in den ausgegeben Wert von `product` ein.

In kleinen Programmen wird ein Ausgabewert o.ä. von fast allen Anweisungen abhängen, so daß ein Rückwärtsslice fast das ganze Programm umfaßt. In großen Programmen mit guter Struktur sind jedoch Slices im Vergleich zur Gesamtprogrammgröße klein [OT89]. Weiser führte den Begriff des Program Slice schon vor über 10 Jahren ein, aber erst Ottenstein erkannte, daß sich Slices natürlich und effizient mit Programmabhängigkeitsgraphen (PDGs) implementieren lassen.

Der PDG hat Anweisungen und Ausdrücke als Knoten und enthält zwei Arten von Kanten:

1. Kontrollabhängigkeitskanten gehen von Prädikaten in IF-, WHILE u.ä. Anweisungen zu all jenen Anweisungen, die davon "regiert" werden, die also nur ausgeführt werden, wenn das Prädikat einen bestimmten Wert hat (`true` für THEN-Zweige, `false` für ELSE-Zweige, spezifische Werte für die Alternativen einer SWITCH-Anweisung).

2. Datenflußabhängigkeitskanten gehen von Anweisungen, in denen eine Variable definiert wird (links in einer Zuweisung o.ä.), zu Anweisungen, in denen dieselbe Variable

```
 (1)  read(n);                          read(n);
 (2)  i:=1;                             i:=1;
 (3)  sum:=0;
 (4)  product:=1;                       product:=1;
 (5)  WHILE i<=n DO                     WHILE i<=n DO
        BEGIN                             BEGIN
 (6)      sum:=sum+i;
 (7)      product:=product*i;                product:=product*i;
 (8)      i:=i+1                             i:=i+1
        END;                              END;
 (9)  write(sum);
(10)  write(product);                  write(product);
```

Abbildung 1: Ein Programm und sein Slice im Hinblick auf `product` in Zeile 10

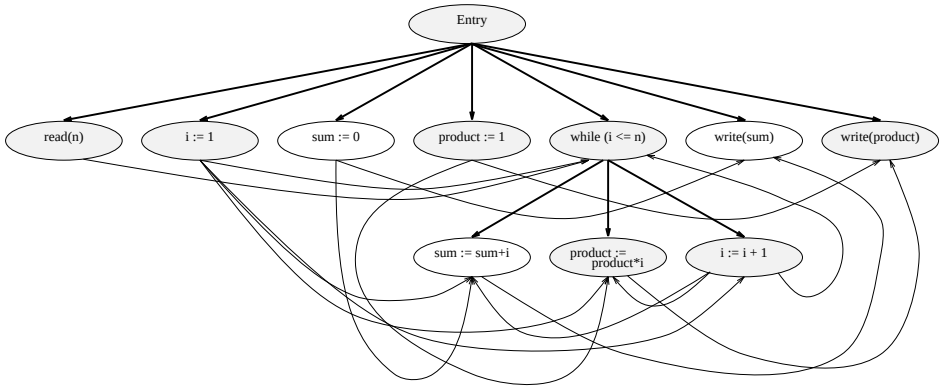

Abbildung 2: PDG zu Abbildung 1

verwendet wird (rechts in Zuweisungen oder in anderen Ausdrücken); ohne daß im Programmablauf zwischen Definitions- und Verwendungspunkt eine Umdefinition stattfinden kann.

Abbildung 2 zeigt als Beispiel den PDG zu Abbildung 1. Der PDG wird stets mit einem Startknoten "angereichert", von dem Kontrollabhängigkeitskanten zu allen "Top-Level"-Anweisungen gehen. Zu jeder Anweisung in Abbildung 1 gibt es einen Knoten. Kontrollabhängigkeiten sind fett eingezeichnet, Datenflußabhängigkeiten normal. Den Slice zu `product` in Zeile 10 erhält man nun ganz einfach, indem man Kanten – ausgehend vom entsprechenden Knoten (ganz rechts) – rückwärts verfolgt. Die dadurch erreichbaren Knoten sind in Abbildung 2 schattiert dargestellt.

Komplizierter wird die Bestimmung des PDG, wenn nicht nur strukturierte Anweisungen, sondern beliebige GOTOs möglich sind; in diesem Fall kommen neue Kontrollflußabhängigkeiten hinzu. Man betrachte etwa das Programm in Abbildung 3, das eine Variante von Abbildung 1 darstellt. Nach dem oben gesagten würden von der GOTO-Anweisung keine Kanten ausgehen, da dort keine Variablen definiert oder Prädikate abgefragt werden. Dadurch würde die GOTO-Anweisung in keinen Slice aufgenommen – was offensichtlich falsch ist, denn das Weglassen der GOTO-Anweisung führt dazu, daß die Schleife nicht mehr terminiert (im Gegensatz zum oben Verlangten, daß nämlich der Slice für die Va-

```
read(n);
i:=1;
sum:=0;
product:=1;
WHILE true DO
  BEGIN
    IF i>n THEN
      GOTO L;
    sum:=sum+i;
    product:=product*i;
    i:=i+1
  END;
L: write(sum);
write(product);
```

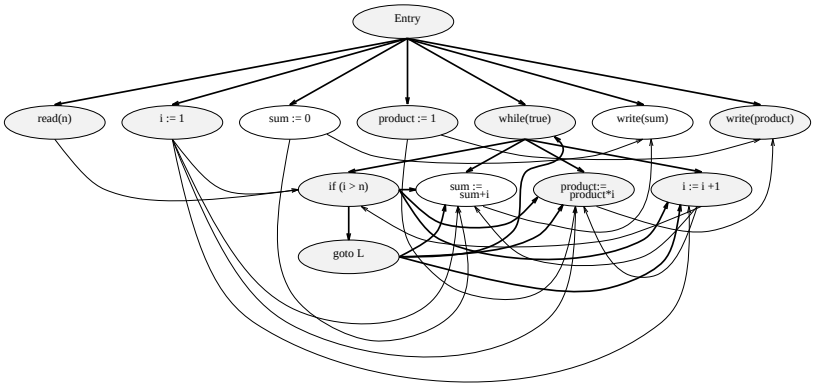

Abbildung 3: Variante von Abbildung 1 mit GOTO

riablen am Slicepunkt dieselben Werte erzeugt wie das ursprüngliche Programm). Es ist deshalb erforderlich, zusätzliche Kontrollabhängigkeitskanten einzuziehen, die zu jenen Anweisungen gehen, die durch das GOTO übersprungen werden können. Denn diese Anweisungen können betrachtet werden als regiert durch den ELSE-Fall eines fiktiven IF mit Bedingung `true`, in dessen THEN-Zweig das GOTO steht (in unserem Beispiel ist das IF sogar real und nicht fiktiv). Das Ergebnis zeigt Abbildung 3.

Prozeduren erfordern größere Erweiterungen. Die naheliegende Idee, jeden Prozeduraufruf durch den Rumpf mit entsprechend substituierten Parametern zu ersetzen und dann das normale Verfahren anzuwenden (Inlining), funktioniert nämlich bei rekursiven Prozeduren nur mit zusätzlicher Fixpunktinduktion und kann zu einem exponentiellen Größenwachstum des Programms führen. Ohne Inlining entsteht aber das Problem, daß eine Prozedur in verschiedenen Kontexten aufgerufen werden kann. Rechnet man beim Slicing einfach "über die Parameter durch die Prozedur hindurch", so würden sich Datenflußabhängigkeiten der Prozedur-Eingabeparameter von *allen* Aufrufstellen ergeben, was den Slice viel zu umfangreich macht. Deshalb wird für jede Prozedur aus ihrem PDG ein "Summary Graph" berechnet, der die transitiven Abhängigkeiten zwischen den Prozedurparametern zusammenfaßt. Dieser Graph wird dann an allen Aufrufstellen in den PDG "eingesetzt", indem spezielle Kanten von den aktuellen Parametern zu den entsprechenden Knoten im Summary Graph gezogen werden. Der entstehende Graph heißt Systemabhängigkeitsgraph (SDG). Der Top-Level Slice wird dann unter Verwendung der Summary-Kanten berechnet; in einer zweiten Phase werden die Anweisungen in Prozedurrümpfen dazugenommen [HRB90, RHS94].

Abbildung 4 und Abbildung 5 zeigen ein einfaches Beispiel, nämlich eine Variante von Abbildung 1 mit Prozeduren. Summary-Kanten sind fett gestrichelt, dünn gestrichelte Kanten stellen die Verbindung zwischen Prozeduraufruf nebst Parameterübergabe und Prozedurrumpf her. Der Top-Level Slice ist hell schattiert, der Gesamtslice umfaßt zusätzlich die Prozedurrümpfe (dunkel schattiert). Man sieht, daß `sum:=0`, `Add(sum,i)` und `write(sum)` nicht im Slice zu `write(product)` liegen.

Nichttriviale Erweiterungen sind auch für komplexe Datenstrukturen wie Arrays, Records,

```
PROGRAM Example;
BEGIN
read(n);
i:=1;
sum:=0;
product:=1;
WHILE i<=n DO
  BEGIN
    Add(sum,i);
    Multiply(product,i);
    Add(i,1)
  END;
write(sum);
write(product);
```

```
PROCEDURE Add(a, b);
BEGIN
a:=a+b
END;

PROCEDURE Multiply(c, d);
j:=1;
k:=0;
WHILE j<=d DO
  BEGIN
    Add(k,c);
    Add(j,1)
  END;
c:=k;
END;
```

Abbildung 4: Variante von Abbildung 1 mit Prozeduren

und Pointer erforderlich. Aus Platzgründen soll auf eine genauere Beschreibung verzichtet werden; es sei jedoch angemerkt, daß insbesondere Pointer sich als extrem unangenehm erwiesen haben. Unter Umständen kann es erforderlich werden, die Verwendung von Pointern einzuschränken.

Datenflußverfahren und PDGs wurden zuerst für Optimierungsaufgaben in Compilern entwickelt. In diesem Zusammenhang entstand eine weitentwickelte Theorie samt effizienten Analysealgorithmen und überzeugenden Anwendungen wie z.B. automatische Parallelisierung [FOW87]; diese Entwicklungen kamen natürlich den oben skizzierten Anwendungen des Program Slicing zugute. Da PDGs aber ursprünglich für einfache imperative Sprachen (Fortran) entwickelt wurden, entstand das Problem, PDGs für komplexere Sprachen mit rekursiven Prozeduren, Pointern usw. zu definieren. In einer Serie von Artikeln der letzten Jahre wurde der Einsatzbereich von PDGs systematisch erweitert, bis schließlich die Anwendbarkeit auf „C“ zu einer realistischen Option wurde [HPR89a, HTB90, Ag94, JZR91]. Horwitz und Reps zeigten nicht nur, daß der PDG die Programmsemantik vollständig erfaßt [HPR88], sondern gaben auch verbesserte Algorithmen zur interprozeduralen Sliceberechnung an [RHS94].

Slicing wurde als Technik zur Unterstützung des Debugging intensiv untersucht [We84, Ot84]. Dynamisches Slicing bezieht bekannte Eingabewerte mit ein und kann so den Slice viel stärker eingrenzen [ADS93]. Neuerdings wurden auch andere Anwendungen im Software Engineering studiert, z.B. zur Programmwartung [GL91] oder zur automatischen Programmintegration [HPR89b].

Slicing ist eine inzwischen ausgereifte Methode, wobei die einzige offene Frage ist, ob man wirklich den ganzen Sprachumfang von C bzw. Fortran behandeln kann oder einige wenige Einschränkungen machen muß, die einige Programme von der automatischen Validierung ausschließen. Falls sich diese Situation ergibt, können jedoch Programmierrichtlinien erlassen werden, die bestimmte Sprachkonstrukte ausschließen – und diese wären im Bereich des gesetzlichen Meßwesens ohne weiteres akzeptabel.

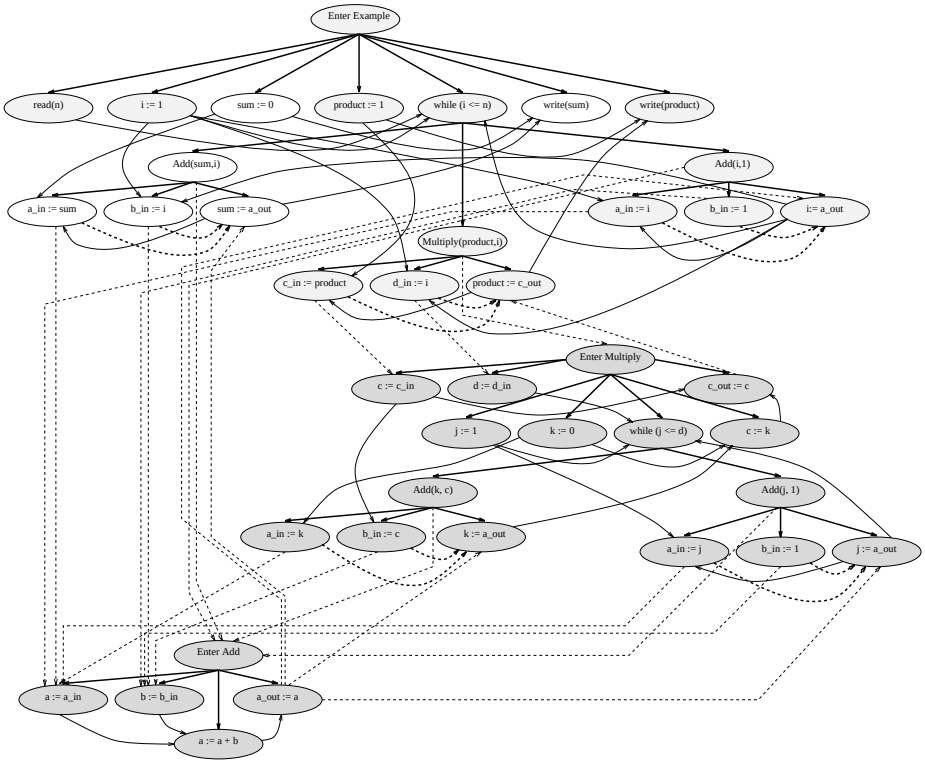

Abbildung 5: Systemabhängigkeitsgraph zu Abbildung 4

3.2 Constraint Solving

In manchen Fällen sind Slices zu grob, d.h. sie umfassen Anweisungen, die in Wirklichkeit auf den Slicepunkt keine Auswirkung haben können (der umgekehrte Fall, daß Anweisungen, die Auswirkungen auf den Slicepunkt haben könnten, nicht im Slice liegen, kann hingegen nicht vorkommen, denn PDGs und Slices werden nach dem Prinzip der konservativen Approximation konstruiert). Als Beispiel betrachten wir die Analyse von Arrays:

```
(1)   read(i,j);
(2)   a[i+3]:=x;
(3)   y:=a[2*j-42];
```

Da die Werte von `i` und `j` zur Analysezeit nicht bekannt sind, muß ("vorsichtshalber", Stichwort konservative Approximation) eine Datenflußkante von Anweisung 2 zu Anweisung 3 gezogen werden – denn es ist nicht ausgeschlossen, daß $i+3 = 2j-42$ sein könnte, und deshalb ist eine Beeinflussung von y durch den Wert von x möglich. Mithin muß jeder Zugriff auf eine Arraykomponente wie ein Zugriff auf das ganze Array behandelt werden – was die Slices i.a. viel zu grob macht und viele falsche Warnungen produziert. In der Literatur wurde deshalb vorgeschlagen, nicht mit Variablen, Arrays, Records und Pointern zu rechnen, sondern mit abstrakten Speicherzellen [AMS91]. Damit können u.U. genauere Ergebnisse erzielt werden. Für uns kommt dieses Vorgehen nicht in Betracht, da es u.U. Kenntnis der Speicherabbildung des Compilers erfordert. Unser Werkzeug soll compilerunabhängig sein.

Wir gehen statt dessen einen anderen Weg. Zu jedem Pfad im PDG sammeln wir Pfadbedingungen auf, die zum Durchlaufen des Pfades erfüllt sein müssen. Z.B. gehört zur Kante (2)→(3) in obigem Beispiel die Pfadbedingung $i + 3 = 2j - 42$ – nur wenn die Bedingung erfüllt ist, kann der Pfad tatsächlich durchlaufen werden. Bedingungen sind i.a. Formeln der Aussagen- oder Prädikatenlogik (inklusive Arithmetik usw.) über den Programmvariablen. Pfadbedingungen können nicht nur durch Arrays entstehen, sondern auch durch bedingte Anweisungen:

```
(1)   read(i,j);
(2)   a[i+3]:=x;
(3)   IF i>10 THEN
(4)     IF j<5 THEN
(5)       y:=a[2*j-42]
(6)     ELSE
(7)       y:=17;
```

Anweisungen (4)-(7) stehen unter der Bedingung $i > 10$, Anweisung 5 unter der Bedingung $i > 10 \wedge j < 5$. Eine Datenflußkante von (2) nach (5) steht also unter der Bedingung $i > 10 \wedge j < 5 \wedge i + 3 = 2j - 42$. Diese Bedingung kann aber nicht erfüllt werden, denn die linke Seite der letzten Gleichung ist auf jeden Fall > 13, die rechte auf jeden Fall < -34 ! Mithin braucht im PDG keine Datenflußkante von (2) nach (5) gezogen zu werden, was die Slices kleiner macht. Selbst wenn Pfadbedingungen nicht zum Löschen von Kanten führen, präzisieren sie doch die genauen Umstände, unter denen Pfade ausgeführt werden – Anweisung (7) wird nur ausgeführt, wenn $i > 10 \wedge j \geq 5$.

Zur weitergehenden Analyse eines Pfades werden demnach die Pfadbedingungen aufgesammelt. Dies geschieht wie folgt: Zu jedem Pfad P von x nach y werden dessen Knoten $k_1, k_2, \ldots, k_n$ bestimmt. Zu jedem Knoten k_i wird sein regierender Ausdruck $R(k_i)$ bestimmt. Der regierende Ausdruck eines Knotens x besteht aus allen Prädikatsknoten $k_x^1, \ldots, k_x^l$, die man ausgehend von x durch Rückwärtswandern entlang Kontrollabhängigkeitskanten erhalten kann (bzw. deren Negation, falls die Kontrollkante zu einem ELSE-Zweig gehört); diese werden konjunktiv verknüpft: $R(x) = k_x^1 \wedge \ldots \wedge k_x^l$. Die regierenden Ausdrücke eines Pfades werden konjunktiv zur Pfadbedingung verknüpft: $R(P) = R(k_1) \wedge \ldots \wedge R(k_n)$. Gibt es mehrere Pfade $P_1, P_2, \ldots, P_m$ von x nach y, werden deren Pfadbedingungen disjunktiv verknüpft: $R(x, y) = R(P_1,) \vee \ldots \vee R(P_m)$.

Es ist auch möglich, für den Startknoten x eines Pfades von vornherein bestimmte Einschränkungen anzugeben: $R(x, y) = E_x \wedge (R(P_1,) \vee \ldots \vee R(P_m))$. Dies erlaubt die Spezifikation zusätzlicher Eingabebedingungen und ähnelt dem Vorgehen in [FRT95].

In den auf diese Weise bestimmten Pfadausdrücken kann nun aber dieselbe Variable verschiedenes bedeuten, wie man an folgendem Beispiel sieht:

```
(1)  IF x=7 THEN
(2)    x:=y+z;
(3)    IF x=8 THEN
(4)      p();
```

Hier ergibt sich als Pfadbedingung für `p()`: $x = 7 \wedge x = 8$, was niemals erfüllt werden kann. Es wird nicht berücksichtigt, daß `x` zwischen den Pfadbedingungen umdefiniert wurde. Im

allgemeinen kann dieselbe Variable in verschiedenen Teilpfadbedingungen verschiedene Werte haben; man muß deshalb alle Vorkommen von Variablen unterscheiden, also sie mit Anweisungsnummern als Indizes versehen. Im Beispiel führt dies zur Pfadbedingung $x_1 = 7 \wedge x_3 = 8$.

Datenflußabhängigkeiten führen weitere Nebenbedingungen ein. Die Datenflußkante (2)→(3) führt etwa zur Bedingung $x_2 = x_3$, denn eine Datenflußkante verbindet Definition und Verwendung dessselben Wertes. Die Datenflußabhängigkeiten machen also die Unterscheidung von Variablen nach Programmpositionen teilweise wieder rückgängig. Seien dazu allgemein k_s^i und k_s^j zwei Prädikatsknoten in der regierenden Bedingung $R(s)$ der Anweisung s. k_s^i und k_s^j gehören selbst zu Anweisungen t und u. Sei weiterhin v eine Variable, die in k_s^i und k_s^j vorkommt. Nach dem oben gesagten ist im allgemeinen $v_t \neq v_u$. Falls aber die Datenflußkanten für v rückwärts aus k_s^i und k_s^j heraus zum selben v-Definitionsknoten r führen, gilt $v_r = v_t = v_u$.

Zuweisungen erzeugen weitere Bedingungen. Im Beispiel muß etwa gelten $x_2 = y_2 + z_2$. Allgemein führt jeder Definitionsknoten p mit Zuweisung `x:= E(y1,...,yn)` zur Bedingung $x_p = E(y1_p, \ldots, yn_p)$. Insgesamt ergibt sich so für jeden Knoten im kritischen Pfad eine Bedingung, und für jede Kante, die zum kritischen Pfad führt, zwei weitere Bedingungen. All diese Bedingungen ergeben insgesamt die Pfadbedingung und grenzen die betrachtete Situation genau ein.

Die Pfadausdrücke samt Nebenbedingungen müssen allerdings in jedem Fall vereinfacht werden, da sie viele redundante Teilausdrücke enthalten. Ziel ist es, Pfadbedingungen so zu vereinfachen, daß lokale Hilfsvariablen möglichst verschwinden und die Bedingungen nach den Eingabevariablen (Datenquellen) aufgelöst sind. Falls dies möglich ist, lassen sich die vereinfachten Bedingungen, die etwa die Randbedingungen einer Eichpfadbeeinflussung angeben, direkt in eine Mängelliste umwandeln. Dazu ist es nur notwendig, diese nach dem Prinzip des "Pretty-Printing" textuell aufzubereiten und dem Benutzer zu präsentieren.

Zunächst sollten Pfadbedingungen in minimale Normalform verwandelt werden. Dazu werden elementare Prädikate durch Variablen ersetzt und einer der bekannten Algorithmen zur Minimierung boolscher Ausdrücke angewandt. Durch Rückersetzung der Variablen entstehen schon wesentlich reduzierte Pfadbedingungen.

Das anschließende Auflösen oder Vereinfachen der reduzierten Pfadbedingungen führt auf das Problem des Constraint Solving. Constraint Solving ist ein Verfahren zum Lösen beliebiger Systeme von gewissen Bedingungen (Constraints). Bedingungen sind boolesche, ganzzahlige oder reelle Gleichungen oder Ungleichungen. Sie zu lösen heißt, für die vorkommenden Variablen die zulässigen Werte(bereiche) zu bestimmen, so daß alle Constraints erfüllt sind.

Für Gleichungen über boolschen Variablen steht mit der Booleschen Unifikation ein Standardverfahren zur Verfügung [MN89], das zwar immer eine eindeutige Lösung liefert, aber in der Praxis sehr teuer sein kann. Bestrebungen, auch Constraints über ganze oder reelle Zahlen zu behandeln, kamen aus Anwendungen in den Ingenieurwissenschaften und führten zum Konzept des Constraint Logic Programming [JJ87, Co92, JM95]. In Erweiterung von Prolog können hier beliebige Constraints über einem gewissen Domain Teil des logischen Programms sein. Beim Programmlauf werden wie üblich Variablenbelegungen berechnet, die ein vorgegebenes Ziel gemäß der Regeln und Constraints erfüllen. In der Praxis wichtig ist insbesondere das System CLP(R), das Constraints (Gleichungen und Ungleichungen) über reellen Zahlen erlaubt [JMSY92]. Da die Prädikatenlogik über den re-

ellen Zahlen im Prinzip natürlich unentscheidbar ist, kommen diverse Spezialverfahren wie Intervallarithmetik, Simplex-Algorithmus, Gauss-Seidel-Iteration usw. zum Einsatz. Auch eine Beschränkung auf Integers verbleibt im unentscheidbaren Bereich (Unentscheidbarkeit der Arithmetik), allerdings können zusätzlich Suchverfahren wie Branch-and-Bound eingesetzt werden. Kürzlich wurde das System CLP(BNR) vorgestellt, das beliebige Kombinationen von Constraints über Booleans, Integers und Reals behandeln kann [OV93, BO95]. Im Gegensatz zu CLP(R) berechnet es nicht symbolische Lösungen, sondern numerische Intervalle. Untersucht wurden auch Erweiterungen wie paralleles oder objektorientiertes Constraint Programming [SHW93], die jedoch für uns keine Rolle spielen.

Da wir einen Constraint Solver nicht selbst entwickeln wollen, soll hier nicht weiter auf die technischen Grundlagen eingegangen werden – hierzu sei auf die Literatur verwiesen, etwa die Sammlung aktueller Arbeiten in [BC93]. Zum Constraint Solving soll ein semikommerzielles Werkzeug (CLP(BNR)) eingesetzt werden, das in den USA erhältlich ist. Dieser Constraint Solver gilt als der im Moment leistungsfähigste. Alternativ soll untersucht werden, ob Algebra-Systeme wie MATHEMATICA eingesetzt werden können.

Anwendungen von Constraint Solving in der Software-Validierung sind bisher nicht bekannt. Da das Problem im Prinzip unentscheidbar ist, kann man auch keine perfekten Lösungen erwarten. Die Praxis im Constraint Logic Programming zeigt aber, daß man sehr oft befriedigende Lösungen erhält. Das Beispiel in Kapitel 5 wird zeigen, daß man sogar ohne Constraint Solver, d.h. nur mit Minimierung von Pfadbedingungen, bereits brauchbare Ergebnisse erhält.

4 Aufbau des Werkzeugs

4.1 Wissenschaftliche und technische Arbeitsziele

Ziel unseres Vorhabens ist es, ein Softwarewerkzeug zur automatischen Validierung von Meßgerätesoftware im Hinblick auf Manipulationssicherheit und Robustheit zu entwickeln. Dazu muß auch Wissen über die Soft- und Hardwareumgebung des Prüflings eingebracht werden. Ziel der Manipulationsschutzanalyse ist die Identifikation sicherheitsrelevanter Pfade und die Erkennung nicht spezifizierter Funktionalitäten, sowie die Überprüfung der Immunität der relevanten Programmteile gegenüber nichtsicherheitsbezogenen Einflüssen. Bei der Robustheitsanalyse werden die aus der Analyse errechneteten Eingabeintervalle mit der Spezifikation verglichen, um zu prüfen, ob nicht spezifizierte Eingabewerte das Meßergebnis beeinflussen können.

Das Werkzeug soll zunächst C-Programme bearbeiten. Es ist geplant, später ein zweites Frontend für eine andere Sprache (z.B. Fortran) zu entwickeln; im Bereich des Meßwesens ist jedoch C dominant, so daß das C-Frontend Priorität hat. Das Werkzeug soll folgende Leistungsmerkmale aufweisen:

- Analyse von mittleren bis größeren C- und Pascal-Programmen
- fast vollständiger Sprachumfang
- Aufbau und Visualisierung des Programmabhängigkeitsgraphen
- Berechnung und Visualisierung von intra- und interprozeduralen Slices

- Modellierung der Software- und Hardwareumgebung des Prüflings
- interaktive Auswahl der Slicepunkte
- Identifikation der sicherheitsrelevanten Pfade
- Bestimmung verdächtiger Slices von außen in den kritischen Pfad hinein
- Weiteranalyse verdächtiger Slices durch Aufsammeln der Pfadbedingungen
- Auflösen (falls möglich) der Pfadbedingungen durch Constraint Solving
- Ausgabe einer Mängelliste
- Ausgabe von Metriken und Statistiken
- leichte Adaptierbarkeit an andere Sprachen und Einsatzumgebungen.

Das Frontend erzeugt die PDGs für die einzelnen Prozeduren, die dann vom Analysekern weiterverabeitet werden. Aus den PDGs wird der SDG berechnet. SDG und PDGs können visualisiert werden, und der Benutzer (Prüfer) kann im Graphen oder im Quelltext Anweisungen anklicken, zu denen er den statischen Rückwärtsslice graphisch oder textuell visualisiert bekommt. Auf Knopfdruck werden die Pfadbedingungen zusammengestellt und aufgelöst. Wissen über die Einsatzumgebung des Prüflings wie z.B. spezielle Hardwareschnittstellen oder Systemfunktionen kann in Form einer Konfigurationsbeschreibung eingelesen werden, um das Werkzeug von der Anwendungsumgebung unabhängig zu machen. Ergänzend können gewisse Metriken berechnet werden, um die allgemeine Qualitätsbeurteilung zu unterstützen.

Ergänzend sollen auch einige theoretische Aspekte untersucht werden. So soll versucht werden, einige der verwendeten Analyseverfahren durch abstrakte Interpretation zu modellieren, wodurch sich eine mathematisch-formale Durchdringung des Problems ergibt, die möglicherweise zu besseren Analyseverfahren führt. Ferner soll die Anwendbarkeit auf andere Gebiete als Meßgerätesoftware untersucht werden.

Abbildung 6 zeigt den Aufbau des Werkzeugs, dessen einzelne Phasen im folgenden detaillierter beschrieben werden.

4.2 Frontend

Das Werkzeug soll zunächst ANSI-C Programme verarbeiten. Dabei wird davon ausgegangen, daß der Präprozessor CPP bereits gelaufen ist, so daß alle Makros aufgelöst, Include-Dateien eingebunden usw sind. Zunächst soll der gesamte Quelltext eines zu validierenden Ssystems auf einmal analysiert werden; später sollen die Techniken des interprozeduralen Slicing verwendet werden, um die Analyse einzelner Module nebst nachfolgender Integration des PDG zu ermöglichen.

Die Syntaxanalyse wurde mit Lex und Yacc aus der ANSI-C Grammatik generiert. Der Parser erzeugt zunächst abstrakte Syntaxbäume, die dann durch eine Symboltabelle erweitert werden. Das Frontend überprüft auch die Einhaltung eventuell notwendiger Restriktionen z.B. für Pointer. Wir gehen davon aus, daß die zu analysierenden Programme statisch korrekt sind; es ist deshalb nur eine rudimentäre Syntaxfehlerbehandlung vorgesehen. Der abstrakte Syntaxbaum muß zusätzliche Querverweise in den Quelltext enthalten,

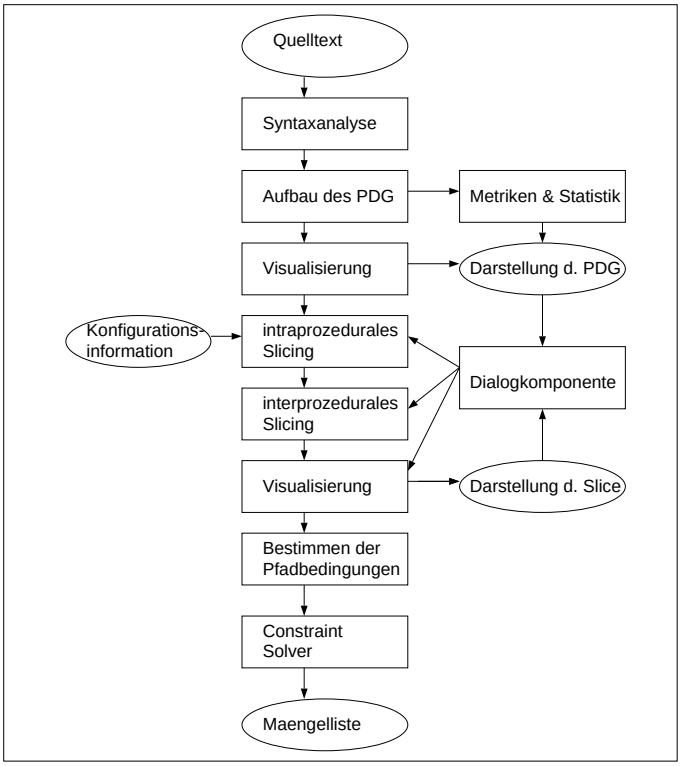

Abbildung 6: Aufbau des Prüfwerkzeuges

um die spätere Visualisierung zu ermöglichen. Die Symboltabelle legt Typinformation ab und erzeugt die Bindungen von Variablen an ihre Deklaration; dazu werden im abstrakten Syntaxbaum Bezeichner durch einen Verweis auf einen Symboltabelleneintrag ergänzt.

4.3 Analysekern

Der Analysekern erzeugt den PDG aus dem Syntaxbaum. Der PDG ist sprachunabhängig und somit die Schnittstelle für eventuelle alternative Frontends. Die Knoten und Kanten des PDG werden wie in Kapitel 3 beschrieben aufgebaut, wobei der volle Sprachumfang von C abgedeckt wird - lediglich für Pointer sind u.U. gewisse Einschränkungen erforderlich. Die Knoten des PDG müssen Rückverweise in den Syntaxbaum enthalten; diese werden zur Visualisierung benötigt. Zunächst wird die intraprozedurale PDG-Berechnung durchgeführt; gemäß dem Algorithmus von Reps erfordert das Berechnen des Systemabhängigkeitsgraphen für die interprozedurale Analyse eine zweite, separate Phase.

In der Regel wird der PDG unter Verwendung des traditionellen Kontrollflußgraphen berechnet, der also zuerst für jede Prozedur aufgebaut werden müßte. Neuere Verfahren erlauben aber die direkte Konstruktion des PDG unter Umgehung des Kontrollflußgraphen, was die Analyse schneller und weniger speicherintensiv macht. Kontrollflußinformation wird aber für andere Zwecke auch benötigt (z.B. für Metriken oder zur Unterstützung der Visualisierung), so daß der PDG entsprechend angereichert werden muß.

Nach Auswahl eines Slicepunktes durch den Benutzer wird der Rückwärtsslice berechnet und visualisiert. Der Benutzer kann kritische Pfade im Slice auswählen und weiteranalysie-

ren lassen; dazu werden die Pfadbedingungen entlang der Kontrollflußabhängigkeitskanten aufgesammelt und durch Constraint Solving vereinfacht. Die entstehenden Ausdrücke werden in textuelle Form umgewandelt und angezeigt.

Zur Ergänzung des Analysevorgangs können Datenflußintensität (Verhältnis Datenflußabhängigkeitskanten zu PDG-Knoten), Kontrollintensität (Verhältnis Kanten zu Knoten im Kontrollflußgraphen), zyklomatische Zahl und andere Metriken berechnet und angezeigt werden.

4.4 Simulation der Soft- und Hardwareumgebung

Meßgerätesoftware ist meist hardwarenahe Software. Die Programme enthalten Zugriffe auf Hardwareregister (z.B. Ein-Ausgabeschnittstelle), verwenden Bibliotheksfunktionen zur Gerätesteuerung oder enthalten sogar eingebetteten Assemblercode. Derartige Features sind natürlich plattformspezifisch, was mit unserem Ansatz eines plattform- und compilerunabhängigen Werkzeugs eigentlich inkompatibel ist. Verzichten kann man aber auf die externen Schnittstellen auch nicht.

Zur Lösung dieses Problems ist vorgesehen, eine Schnittstelle zum Einstöpseln von "Fremd-PDGs" zu realisieren. Für die Analyse ist nämlich das tatsächliche Verhalten des Assemblercodes, der Bibliotheksfunktionen usw. ohne Belang. Entscheidend ist, welche Kontroll- und Datenflußabhängigkeiten sie erzeugen. Inspiriert durch die "Summary Graphs" der interprozeduralen Analyse wird es deshalb möglich sein, PDGs für Bibliotheksfunktionen zum System Dependency Graph des Prüflings hinzuzufügen. Im Gegensatz zu Abbildung 5 kommen die PDGs der Bibliotheksfunktionen selbst aus einer Bibliothek – einer Bibliothek von Betriebssystemfunktions-PDGs.

Zur Erzeugung der Bibliothek sind zwei Vorgehensweisen denkbar:

1. Simulation der Bibliotheksfunktionen durch rudimentäre C-Programme. Diese müssen dieselbe Schnittstelle und denselben Datenfluß haben wie die echten Bibliotheksfunktionen. Der Vorteil ist, daß die PDGs mit dem Werkzeug selbst erzeugt werden können.

2. Verwendung einer Beschreibungssprache für PDGs. Eine spezielle Beschreibungssprache erlaubt die direkte Spezifikation von PDGs anhand von Knoten, Kanten und evtl.zusätzlicher Information (z.B. Markierungen von Knoten und Kanten). Der Vorteil ist, daß sich so auch PDGs für Assemblercodestücke (manuell) erzeugen lassen, die man durch maschinelle Analyse nicht erhalten kann.

In jedem Fall müssen die externen PDGs persistent in Dateien gespeichert werden; bei der Analyse muß der Aufruf von Bibliotheksfunktionen oder die Verwendung von Assembler zum Einbinden der entsprechenden PDGs führen.

4.5 Benutzerschnittstelle

Nach dem Aufbau des PDG muß dieser visualisiert werden. Dazu kann ein Standard-Graph-Layout-Algorithmus wie z.B. der Sugiyama-Algorithmus entsprechend angepaßt werden. In unserem Fall scheint es aber besser, die spezielle Struktur von PDGs auszunutzen. So kann man die Unterscheidung von Kontroll- und Datenflußabhängigkeitskanten

nutzen, um zuerst den "Kern" des PDG, nämlich die Kontrollkanten nebst Prädikaten und Anweisungen darzustellen. Die Datenflußabhängigkeiten können in einer zweiten Phase ergänzt werden. Dies hat den Vorteil, daß das "Groblayout" sich am Quelltext orientiert – es wird ähnlich zum abstrakten Syntaxbaum, was für das Verständnis wesentlich besser ist (vgl. Abbildungen 2 und 9). Es müssen Knoten und Kanten verschiedener Art unterstützt werden (vgl. Abbildung 9). Ferner muß es möglich sein, Prozedur-PDGs im SDG durch ihre Summary-Graphen zu ersetzen, um die Visualisierung überschaubar zu halten (Zoom-in/Zoom-out Funktion).

Durch einfaches Anklicken kann der Quelltext zu einem Knoten angezeigt werden; ebenso Kantenmarkierungen von Kontrollflußkanten (diese sind i.a. mit konstanten Werten markiert, die die verschiedenen Zweige nach IF- oder SWITCH-Anweisungen identifizieren). Ebenso einfach können Slicepunkte ausgewählt werden, wobei Slices farbig dargestellt werden sollen. Innerhalb von Slices (oder auch beliebig im PDG) können Pfade durch Anklicken von Anfangs- und Endpunkt ausgewählt werden. Zu jedem Pfad im PDG können auf Knopfdruck die Pfadbedingungen aufgesammelt und vereinfacht werden; das Ergebnis wird in einem separaten Fenster angezeigt. Zoom-in, Zoom-out sowie das Einstöpseln von Bibliotheks-PDGs ist nach Auswahl einer Referenzstelle einfach möglich.

5 Ein Beispiel

Abbildung 7 zeigt einen Quelltext, der aus einer realen Meßgerätesoftware extrahiert (und modifiziert) wurde. Zwar fehlen einige Programmteile, die Art der Programmierung ist jedoch typisch. Das Programm verwendet zwei Hardware-Eingaberegister: `p_ab` für einen Analog-Digital-Wandler (Meßgeräteeingabe), `p_cd` für Keyboard-Eingabe. Weitere hardwarespezifische Adressen sind als Konstanten definiert. Die Variable `kal_kg` enthält einen Kalibrierungsfaktor, der zur Berechnung des Gewichts aus der Sensoreingabe benötigt wird; das Gewicht `u_kg` wird auf ein LCD ausgegeben (im Programm stehen dazu einfache Print-Anweisungen). Die Haupt-Eventschleife fragt permanent die Eingaberegister ab und veranlaßt entsprechende Aktionen. Das Display zeigt Artikelnummer (aus der Eingabe gepuffert in `e_puf`) und Gewicht in jedem Schleifendurchlauf neu an.

Abbildung 8 zeigt wiederum einen Auszug aus Abbildung 7. Man erkennt nun deutlicher, daß bei Eingabe eines '+' (und gleichzeitigem Herausnehmen des Papiers aus dem Drucker) die Anzeige um 10% erhöht wird, denn der Kalibrierfaktor wird mit 1.1 multipliziert; analog führt Eingabe eine '–' zu einer um 10% zu niedrigen Anzeige. Diese bewußte Manipulationsmöglichkeit wurde von den Autoren absichtlich in den Quelltext eingefügt, um das Verhalten unseres Werkzeuges zu demonstrieren.

Abbildung 9 zeigt den PDG zu Abbildung 8. Kontrollabhängigkeitskanten sind fett gemalt. Ergänzend zu den üblichen Anweisungs- und Prädikatsknoten wurden die Datenquellen und Datensenken (Hardwareregister) hinzugefügt. Die Initialisierung und Fortschaltung der FOR-Schleife führte zum Einfügen zweier zusätzlicher Knoten. Die SWITCH-Anweisung verwendet einen Knoten pro Alternative (was eine reale Implementierung nicht tun würde, es dient nur der vereinfachten Darstellung der Pfadbedingungen). Ansonsten entsprechen die Nummern in den Knoten den Zeilennummern in Abbildung 8. Man beachte, daß einige Kanten, die den Datenfluß von `idx` betreffen, der Übersichtlichkeit halber weggelassen wurden.

```
/* ******************************************************** **
** Prototyp eines Messgeraet-Steuerprogramms                 **
** Stand: 16.02.96                                          **
** ******************************************************** */

#include <stdio.h>

/* ---------------------------------------------------------- */

char
  idx;                  /* Laufvariable                      */
  e_puf[17];            /* Tastatur-Puffer fuer Artikel-Nr.   */

int
  u;                    /* Momentanwert Spannung             */

float
  u_kg,                 /* Gewicht                           */
  kal_kg = 2.664E-3;    /* Kalibrierfaktor                   */

/* ---------------------------------------------------------- */
/* Hardware: 2 Multifunktionsbausteine, deren Register hier   **
** durch zwei unsigned char arrays wiedergegeben werden.      **
** An Port p_ab[PA], p_ab[PB] ist ein Analog-Digital-Wandler  **
** angeschlossen. An Port p_cd[PA] eine Tastatur, die ASCII-  **
** Zeichen liefert (Handshake-Leitungen an p_cd[CTRL2], bits 0 **
** und 1). An Port p_cd[PB] ist ein Drucker angeschlossen     **
** (Kontrolleitung fuer "paper out" an p_cd[CTRL2], bit 4).   */

unsigned char
  p_ab[16], p_cd[16];     /* Port-Register fuer ADU, Tastatur,  */
                          /*   Drucker, ...                     */

#define PB 0              /* Register des 6522                 */
#define PA 1
#define PA2 15
#define DDRB 2
#define DDRA 3
#define TIM1_L 4
#define TIM1_H 5
#define T1L_L 6
#define T1L_H 7
#define TIM2_L 8
#define TIM2_H 9
#define SSR 10
#define CTRL1 11
#define CTRL2 12
#define I_FLAGS 13
#define I_ENABL 14

/*-------------------------------------------------------------*/
```

```
main() {

  u = 0;
  u_kg = 0.0;

  while(TRUE) { /* Schleife Normalbetrieb --> */

    if ((p_ab[CTRL2] & 0x10)==0) { /* ADU fertig --> */
      u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
                                /* 12-bit Momentanwert          */
      u_kg = u * kal_kg;
                                /* Kalibrierung, Fliesskomma-    **
                                **   Wandlung                   */
      } /* ... ADU fertig */
    /* . . . . . . . . . . . . . . . . . . . . . . . . . . . .  */

    if ((p_cd[CTRL2] & 0x01) != 0) { /* Taste gedrueckt --> */

      for (idx=0;idx<7;idx++) { /* max. 7 Zeichen einlesen -->  */
        if (p_cd[PA] == 0x0d) break; /* ENTER-Taste gedrueckt    */
        e_puf[idx] = p_cd[PA]; /* Zeichen holen (ASCII)   */
        p_cd[CTRL2] |=  0x02;         /* ACK-Impuls              */
        p_cd[CTRL2] &= ~0x02;
        if ((p_cd[CTRL2] & 0x10) != 0) {/* Eingebaute Manipula- */
                                       /* tionsmoeglichkeit !!!   */
                                       /* Nur wenn gleichzeitig  */
                                       /* "paper out"            */
          switch(e_puf[idx]) {
            case '+':
              kal_kg *= 1.1;
              idx--;
              break;
            case '-':
              kal_kg *= 0.9;
              idx--;
              break;
            default: break;
            } /* ... switch */
          } /* ... "paper out" */
        } /* ...for(;idx;), BREAK bei ENTER-Taste */

      e_puf[idx] = '\0';        /* String-Abschluss              */

      } /* ... Taste gedrueckt */
    /* . . . . . . . . . . . . . . . . . . . . . . . . . . . .  */

    /* Ausgabe 1. Zeile LCD-Display */
    /*      0123456789012345        */
    /*     [Artikel: NNNNNNN]       */
    printf("Artikel: %07.7s",e_puf);

    /* Ausgabe 2. Zeile LCD-Display */
    /*      0123456789012345        */
    /*     [   -00.00 kg    ]       */
    printf("   %6.2f kg    ",u_kg);

    } /* ... Schleife Normalbetrieb */
  } /* ... main */
/*-------------------------------------------------------------*/
```

Abbildung 7: Beispiel eines Meßgerätecodes

Berechnet man nun den Rückwärtsslice zu `u_kg` (Anweisung 13b), so erkennt man, daß `p_cd` im Slice liegt, und zwar über die Anweisungen 10 und 11. Mithin hat die Tastatureingabe Auswirkungen auf das angezeigte Gewicht – ein äußerst verdächtiger Tatbestand. Gäbe es die Anweisungen 10 und 11 nicht, so würde der Meßwert nur vom Wert am Sensor-Eingaberegister abhängen, und alles wäre in Ordnung.

Die möglichen Pfade von `p_cd` nach `u_kg` müssen weiter analysiert werden. Die Berechnung der Pfadbedingung nach dem in Kapitel 3.2 geschilderten Verfahren liefert nach Vereinfachung folgenden Ausdruck:

$$\texttt{p\_cd[CTRL2] \& 0x01} \neq 0 \wedge 0 \leq \texttt{idx} < 7 \wedge \texttt{p\_cd[CTRL2] \& 0x10} \neq 0 \wedge (\texttt{e\_puf[idx]='+'} \vee \texttt{e\_puf[idx]='-'})$$

Man beachte, daß in diesem Beispiel die beiden Auftreten von `p_cd` nicht unterschieden

```
(1)  while(TRUE) {
(2)    if ((p_ab[CTRL2] & 0x10)==0) {
(3)      u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
(4)      u_kg = u * kal_kg;
         }
(5)    if ((p_cd[CTRL2] & 0x01) != 0) {
(6)      for (idx=0;idx<7;idx++) {
(7)        e_puf[idx] = p_cd[PA];
(8)        if ((p_cd[CTRL2] & 0x10) != 0) {
(9)          switch(e_puf[idx]) {
(10)            case '+': kal_kg *= 1.1; break;     /* unerlaubter */
(11)            case '-': kal_kg *= 0.9; break;     /* Datenfluss  */
                }
            }
          }
(12)     e_puf[idx] = '\0';
         }
(13)   printf("Artikel: %07.7s\n",e_puf);
(14)   printf("     %6.2f kg    ",u_kg);
       }
```

Abbildung 8: Auszug aus Abbildung 7

werden müssen, da zwischen (5) und (8) keine Umdefinition des Registers stattfindet – alle Datenabhängigkeiten von `p_cd` gehen vom selben Knoten aus. Weiteres Einsetzen von Nebenbedingungen aufgrund der Datenflußkante (7)→(9) führt wegen `e_puf[idx]=p_cd[PA]` zur vereinfachten Bedingung, die den Eingabepuffer nicht mehr enthält:

$$\begin{aligned} & \texttt{p\_cd[CTRL2] \& 0x01} \neq 0 \\ \wedge\ & \texttt{p\_cd[CTRL2] \& 0x10} \neq 0 \\ \wedge\ & (\texttt{p\_cd[PA]}=\texttt{'+'} \vee \texttt{p\_cd[PA]}=\texttt{'-'}) \end{aligned}$$

Hieraus ist ersichtlich, daß die Manipulation des Eichpfades nur durch Eingabe eines '+' oder '–' erfolgen kann, sofern gleichzeitig das Bit für "Paper Out" gesetzt ist – genau wie dies die manuelle Analyse vorhergesagt hat. Für größere Programme lassen sich derartige Manipulationen des Eichpfades kaum noch manuell entdecken!

6 Ausblick

Das Projekt "Validierung softwaregesteuerter Meßgeräte mit Program Slicing und Constraint Solving" bringt zwei im Prinzip vorhandene, aber in Deutschland kaum eingesetzte Technologien unmittelbar in einen wirtschaftlich bedeutsamen Anwendungsbereich. Meßgeräteherstellern soll ein einsatzreifes Werkzeug zur Software-Validierung an die Hand gegeben werden. Gleichzeitig hat das Projekt aber auch Grundlagenanteile, da noch Raum für bessere Analyseverfahren und ihre theoretischen Fundamente besteht.

Die Projektpartner verfügen über das notwendige Know-How: Die Abteilung Softwaretechnologie der TU Braunschweig ist kompetent im Bereich Software Engineering, Programmiersprachen und Inferenzverfahren und hat umfangreiche Erfahrung in der Werkzeug-

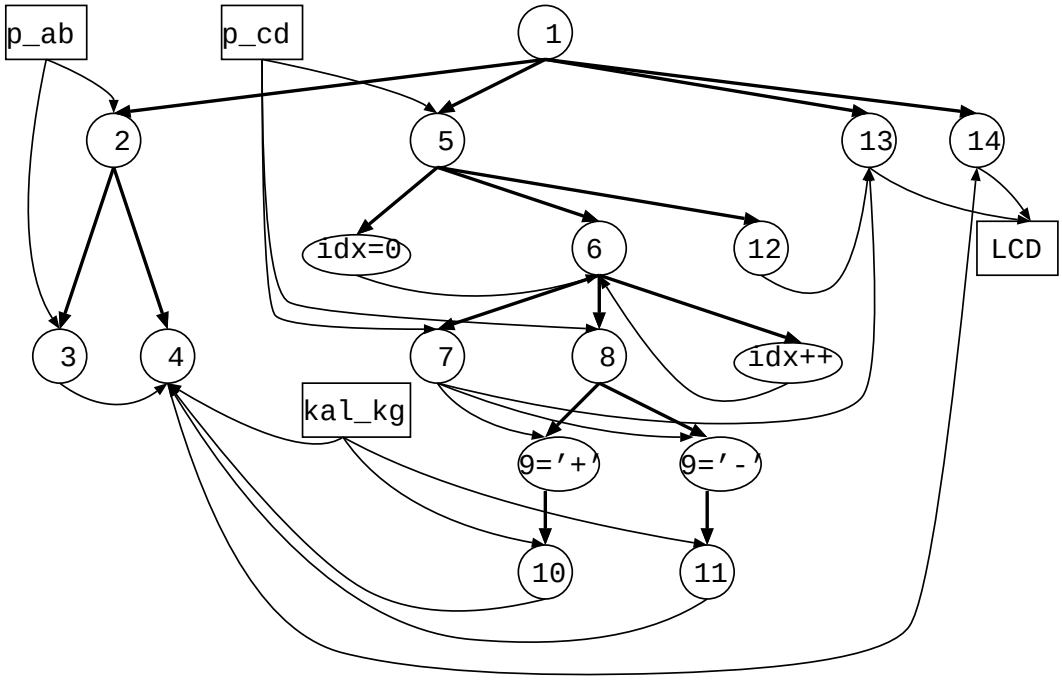

Abbildung 9: PDG zu Abbildung 8

entwicklung. Die PTB bringt das Anwenderwissen ein, und die LINEAS GmbH verfügt über umfangreiche Erfahrungen im Bereich Visualisierung und interaktive Systeme. Die räumliche Nähe der Partner ermöglicht enge Kooperation.

Mithin ist VALSOFT ein Paradebeispiel für den Transfer von neuen theoretischen Ergebnissen in eine relevante Anwendung. Eine Übertragung auf andere sicherheitskritische Anwendungsbereiche ist ohne weiteres möglich. Wir hoffen, in zwei Jahren eine erste Version des Werkzeuges einsatzbereit zu haben, die in Fallstudien bei der PTB eingesetzt werden kann. Über das weitere Vorgehen (evtl. Funktionserweiterungen, Übertragung in andere Anwendungsbereiche, Markteinführung) soll danach entschieden werden.

7 Literatur

[AMS91] H. Agrawal, R. DeMillo, E. Spafford: Dynamic slicing in the presence of unconstrained pointers. Proc. 4th Symposium on Testing, Analysis, and Verification, S. 60 – 73.

[Ag94] H. Agrawal: On slicing programs with jump statements. Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation, S. 302 – 312.

[AMS93] H. Agrawal, R. DeMillo, E. Spafford: Debugging with dynamic slicing and backtracking. Software Practice and Experience 23 (6), S. 589 – 616, Juni 1993.

[BC93] F. Benhamou, A. Colmerauer (Hrsg.): Constraint Logic Programming: Selected Research. MIT Press 1993.

[BO95] F. Benhamou, W. Older: Applying Interval Arithmetic to Real, Integer and Bolean Constraints. Erscheint in Journal of Logic Programming (1995).

[Co92] J. Cohen: Constraint logic programming languages. Communications of the ACM 33(7), S. 52 – 68, 1992.

[DD77] D. Denning, P. Denning: Certification of programs for secure information flow. Communications of the ACM 20(7), S. 504 –513, Juli 1977.

[DGQ86] Deutsche Gesellschaft für Qualität e.V., Nachrichtentechnische Gesellschaft im VDG: Software-Qualitätsicherung. Beuth-Verlag, Berlin 1986.

[DGQ92] Deutsche Gesellschaft für Qualität e.V.: Methoden und verfahren der Software-Qualitätssicherung. DGQ-ITG-Schrift 12–52, Beuth-Verlag, Berlin 1992.

[DIN90] DIN V VDE 0801 Grundsätze für Rechner mit Sicherheitsaufgaben. Ausgabe Januar 1990.

[FOW87] J. Ferrante, K. Ottenstein, J. Warren: The program dependence graph and its uses in optimization. ACM Transactions on Programming Languages and Systems 9(3), S. 319 – 349, Juli 1987.

[FRT95] J. Field, G. Ramalingam, F. Tip: Parametric program slicing. Proc. 21th Symposium on Principles of Programming Languages, ACM 1995, S. 379 –392.

[GL91] K. Gallagher, J. Lyle: Using program slicing in software maintenance. IEEE Transactions on Software Engineering, 17(8), S. 751 – 761, August 1991.

[Gro91] Grottker, U. et al.: Nachweis der Software-Zuverlässigkeit. Technische Zuverlässigkeit, TTZ1991, ITG-Fachberichte 116.

[HPR89a] S. Horwitz, P. Pfeiffer, T. Reps: Dependence analysis for pointer variables. Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation, S. 28 –40.

[HPR89b] S. Horwitz, J. Prins, T. Reps: Integrating noninterfering versions of programs. ACM Transactions on Programming Languages and Systems 11(3), S. 345 – 387, Juli 1989.

[HPR88] S. Horwitz, J. Prins, T. Reps: On the adequacy of program dependence graphs for representing programs. Proc. 15th Symposium on Principles of Programming Languages, ACM 1988, S. 146 – 157.

[HRD90] S. Horwitz, T. Reps, D. Blinkley: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12(1), S. 26 – 60, Januar 1990.

[HR92] S. Horwitz, T. Reps: The use of program dependence graphs in software engineering. Proc. 14th Int. Conference on Software Engineering, IEEE 1992, S. 392 – 411.

[JJ87] J. Jaffar, J. Lassez: Constraint logic programming. Proc. 14th Symposium on Principles of Programming Languages, ACM 1987, S. 111 – 119.

[JMSY92] J. Jaffar, S. Michaylow, P. Stuckey, R. Yap: The CLP(R) language and system. ACM Transactions on Programming Languages and Systems 14(3), S. 339 – 395 (Juli 1992).

[JM95] J. Jaffar, M. Maher: Constraint logic programming: a survey. Erscheint in Journal of Logic Programming (1995).

[JZR91] J. Jiang, X. Zhou, D. Robsons: Program slicing for C – the problems in implementation. Proc. IEEE Conference on Software Maintenance, IEEE 1991.

[MN89] Martin, U., Nipkow, T.: Boolean Unification — the Story so Far. Journal of Symbolic Computation, 7, 275-293 (1989).

[OO84] K. & L. Ottenstein: The program dependence graph in a software development environment. Proc. SIGPLAN Symposium on Practical Programming Development Environments, S. 177 – 184, ACM 1984.

[OT89] L. Ott, J. Thuss: The relationship between slices and module cohesion. Proc. 11th International Conference on Software Engineering, IEEE 1989.

[OV93] W. Older, A. Vellino: Constraint arithmetic on real intervals. In [BC93].

[RHS94] T. Reps, S. Horwitz, M. Saagiv: Speeding up slicing. Proc. SIGSOFT '94 Symposium on Foundations of Software Engineering, erscheint demnächst.

[SHW93] G. Smolka, M. Henz, J. Würz: Object-Oriented Concurrent Constraint Programming in Oz. DFKI Research Report 93–16.

[Ti95] F. Tip: A survey of program slicing techniques. Journal of Programming Languages 3 (1995), S. 121 – 189.

[VDI93] VDI/GIS/AIZ Arbeitsgruppe Software-Zuverlässigkeit: Software-Zuverlässigkeit. VDI Verlag, Düsseldorf 1993.

[We84] M. Weiser: Program Slicing. IEEE Transactions on Software Engineering, 10(4), S. 352 – 357, Juli 1984.