Software-Sicherheitsprüfung mit VALSOFT

Jens Krinke, Gregor Snelting, Torsten Robschink
Abteilung Softwaretechnologie
Technische Universität Braunschweig

Zusammenfassung

Die Physikalisch-Technische Bundesanstalt muß alle eichpflichtigen Meßgeräte prüfen. Da heute fast jedes Meßgerät durch Software gesteuert wird, muß sichergestellt werden, daß der Datenpfad vom Sensoreingang zur Anzeige (*Eichpfad*) nicht durch externe Faktoren beeinflußt werden kann.

Das VALSOFT-Werkzeug erkennt, analysiert und visualisiert Beeinflussungen des Eichpfades. Grundlage der Analyse sind *Program-Slicing* und *Constraint-Solving*. Zu beliebigen Programmpunkten (z. B. Meßwertausgaben) können diejenigen Anweisungen bestimmt werden, die diesen Punkt beeinflussen (sog. Slice). Zusätzlich können genaue Bedingungen berechnet werden, unter denen verdächtige Datenflüsse stattfinden (sog. Pfadbedingungen). Anwendungen in anderen sicherheitskritischen Bereichen sind ohne weiteres möglich.

1 Einleitung und Übersicht

Die Physikalisch-Technische Bundesanstalt (PTB) ist verantwortlich für die Baumusterprüfung aller eichpflichtigen Meßgeräte. Jedes kommerzielle Meßgerät, sei es ein Stromzähler oder ein Blutalkoholtester, muß gesetzlich vorgeschriebenen Standards im Hinblick auf Meßgenauigkeit und Robustheit genügen. Hersteller, deren Produkte im Bereich des gesetzlichen Meßwesens eingesetzt werden, müssen dabei die Resistenz gegenüber mißbräuchlicher oder unbeabsichtigter Fehlbedienung nachweisen.

Da heutzutage fast jedes Meßgerät durch Software gesteuert wird, ist die Software-Validierung Teil des Meßgeräte-Prüfprozesses. Insbesondere muß sichergestellt werden, daß der Datenpfad vom Sensoreingang zur Anzeige des gemessenen Wertes (der sog. *Eichpfad*) nicht durch externe Faktoren beeinflußt werden kann. Jedwede Manipulation des Eichpfades muß aufgedeckt werden, wohingegen an nicht eichpflichtige Teile der Software (z. B. die Benutzerschnittstelle) auch weniger strenge Maßstäbe angelegt werden.

Zur Zeit wird Quellcode von Meßgerätesoftware in der PTB manuell inspiziert – ein zeitaufwendiger und fehleranfälliger Prozeß. Die PTB ist deshalb an Werkzeugunterstützung für die Softwarevalidierung interessiert. Auch Hersteller von Meßgeräten oder anderer sicherheitskritischer Software (z. B. in der Verkehrsleittechnik) können von einem Validierungswerkzeug profitieren und evtl. sogar Sicherheitsteilprüfungen selbst durchführen.

Vor diesem Hintergrund entwickelten die Abteilung Softwaretechnologie der TU Braunschweig, das Braunschweiger Softwarehaus LINEAS und die PTB gemeinsam ein Werkzeug zur Analyse von sicherheitskritischer Software (Verbundprojekt VALSOFT: „Validierung softwaregesteuerter Meßsysteme durch Program-Slicing und Constraint-Solving“). Das Werkzeug analysiert Datenflüsse zur Meßwertausgabe und kann so Beeinflussungen des Eichpfades aufdecken. Da zur Beurteilung von Software auf menschliches Expertenwissen nicht verzichtet werden kann, bietet das Werkzeug interaktiven Zugang sowie eine Visualisierung der gewonnenen Information an.

Grundlage der Analyse sind *Program-Slicing* und *Constraint Solving*. Aus einem C-Programmquelltext wird ein Programmabhängigkeitsgraph (PDG) erzeugt. Dieser erlaubt es, zu beliebigen Programmpunkten (insbesondere Meßwertausgaben, 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 verfälscht“.

In diesem Artikel wollen wir die Grundlagen des Analyseverfahrens skizzieren und von ersten Erfahrungen berichten. Wir wollen auch aufzeigen, daß das Verfahren in andere sicherheitsrelevante Bereiche übertragen werden kann. Immer wenn verdächtige Datenflüsse aufgedeckt und analysiert werden müssen, kann VALSOFT zum Zuge kommen.

2 Program-Slicing

Wir beginnen mit einer Beschreibung des Slicing. Unsere Darstellung ist informal; formale Definitionen finden sich z. B. in dem Übersichtsartikel von Frank Tip [Tip95].

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 [Wei84].

Slicing läßt sich natürlich und effizient mit Programmabhängigkeitsgraphen (PDGs) implementieren. Der PDG hat Anweisungen und Ausdrücke als Knoten und enthält zwei Arten von Kanten:

- *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).

- *Datenabhängigkeitskanten* gehen von Anweisungen, in denen eine Variable definiert wird (links in einer Zuweisung o. ä.), zu Anweisungen, in denen dieselbe Variable verwendet wird (rechts in Zuweisungen oder in anderen Ausdrücken); ohne daß im Programmablauf zwischen Definitions- und Verwendungspunkt eine Umdefinition stattfindet.

Den Rückwärtsslice zu einer Anweisung s erhält man nun einfach dadurch, daß man alle Knoten aufsammelt, von denen aus ein Pfad zur Zielanweisung s führt. Dies kann man durch einfaches Rückwärtsverfolgen der PDG-Kanten – ausgehend von s – implementieren.

Als Beispiel betrachten wir den Quelltext in Abbildung 1, der einer realen Meßgerätesoftware nachempfunden wurde. 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 einer Anzeige 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 2 zeigt den PDG zu Abbildung 1. Kontrollabhängigkeitskanten sind fett gezeichnet. Ergänzend zu den üblichen Anweisungs- und Prädikatsknoten wurden die Datenquellen und Datensenken (Hardwareregister `p_ab, p_cd, LCD`) hinzugefügt. Die Initialisierung und Fortschaltung der FOR-Schleife führte zum Einfügen zweier zusätzlicher Knoten. Die Nummern in den Knoten entsprechen den Zeilennummern in Abbildung 1. Man beachte, daß einige Kanten, die `idx` betreffen, der Übersichtlichkeit halber weggelassen wurden.

Berechnet man nun den Rückwärtsslice zu `u_kg` (Anweisung 14), 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.

```
(1)   while(TRUE) {
(2)     if ((p_ab[CTRL2] & 0x10)==0) {
(3)       u = ((p_ab[PB] & 0x0f) << 8) + 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;
(11)              case '-': kal_kg *= 0.9; break;
                  }
              }
            }
(12)      e_puf[idx] = '\0';
          }
(13)    printf("Artikel: %07.7s\n",e_puf);
(14)    printf("      %6.2f kg    ",u_kg);
        }
```

Abbildung 1: Fragment einer Meßgerätesoftware

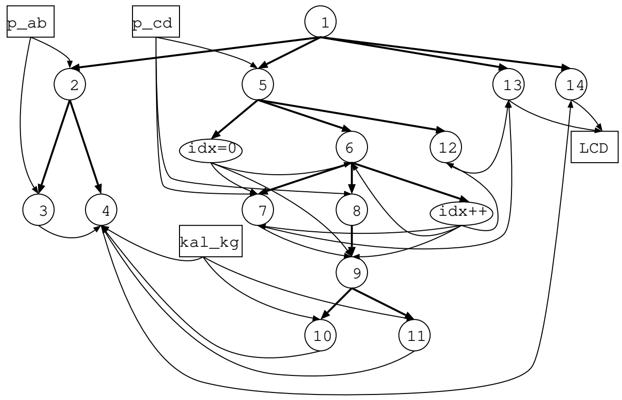

Abbildung 2: PDG zu Abbildung 1 (leicht vereinfacht)

Komplizierter wird die Bestimmung des PDG, wenn nicht nur strukturierte Anweisungen, sondern beliebige GOTOs möglich sind; in diesem Fall kommen neue Kontrollflußabhängigkeiten hinzu [Agr94]. Prozeduren erfordern größere Erweiterungen, denn eine Prozedur kann in verschiedenen Kontexten aufgerufen werden. Dazu werden spezielle Kanten zwischen den Knoten der aktuellen und formalen Parameter eingefügt. Mit Hilfe dieser Kanten werden die transitiven Abhängigkeiten zwischen den Prozedurparametern berechnet und an den Aufrufstellen werden entsprechende *Summary*-Kanten eingefügt. Der entstehende Graph heißt *Systemabhängigkeitsgraph* (SDG). Der Slice wird dann unter Verwendung der neuen Kanten in zwei Phasen berechnet (auf- und absteigen in der Aufrufkette) [HRB90, RHSR94].

Nichttriviale Erweiterungen sind auch für komplexe Datenstrukturen wie Arrays, Records, 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: die Analyse von Pointerstrukturen ist generell unentscheidbar, selbst konservative Approximation kann exponentiellen Aufwand haben [Ram94].

3 Pfadbedingungen

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). 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 Datenabhängigkeitskante 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. VALSOFT reichert deshalb die Analyse mit deduktiven Verfahren an, was nicht nur die Analysegenauigkeit erhöht, sondern auch die Berechnung expliziter Bedingungen erlaubt, unter denen ein Datenfluß stattfinden kann.

Zu jedem Pfad im PDG werden Pfadbedingungen aufgesammelt, 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 Datenabhängigkeitskante 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 Datenabhängigkeitskante 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 bzw. Datenflüsse stattfinden: Anweisung (7) wird nur ausgeführt, wenn $i > 10 \wedge j \geq 5$.[1]

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

Datenabhängigkeiten führen weitere Nebenbedingungen (sog. Abhängigkeitsgleichungen) ein. Im letzten Beispiel führt etwa die Datenabhängigkeitskante (2)→(3) zur Bedingung $x_2 = x_3$, denn eine Datenabhängigkeitskante verbindet Definition und Verwendung desselben Wertes. Datenabhängigkeiten machen also die Unterscheidung von Variablen nach Programmpositionen teilweise wieder rückgängig.

Allgemein kann die Konstruktion der Pfadbedingungen folgendermaßen beschrieben werden. 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. Dieser gibt eine notwendige Bedingung dafür an, daß der Knoten überhaupt ausgeführt werden kann. Der regierende Ausdruck eines Knotens x besteht deshalb aus allen Prädikatsknoten $p_x^1, \ldots, p_x^l$, die man ausgehend von x durch Rückwärtswandern entlang Kontrollabhängigkeitskanten erhalten kann; diese werden konjunktiv verknüpft: $R(x) = p_x^1 \wedge \ldots \wedge p_x^l$. Die regierenden Ausdrücke eines Pfades werden konjunktiv zur Pfadbedingung verknüpft: $PC(P) = R(k_1) \wedge \ldots \wedge R(k_n)$. Gibt es mehrere Pfade $P_1, P_2, \ldots, P_m$ von x nach y, werden deren Pfadbedingungen disjunktiv verknüpft. Mithin ist

$$PC(x, y) = \bigvee_{\mu} \bigwedge_{\nu} R(k_\nu^\mu)$$

wobei μ über die Pfade läuft und ν über die Knoten eines Pfades.[2]

Pfadbedingungen sind notwendige Bedingungen, d. h. wenn sie nicht erfüllbar sind, kann kein Datenfluß stattfinden. Aus ihrer Erfüllbarkeit kann aber noch nicht auf einen tatsächlichen Datenfluß geschlossen werden (obwohl diese Vermutung naheliegt). Die Erzeugung von

[1] Zur Analyse von Array-Abhängigkeiten wurden bereits dedizierte Verfahren entwickelt (z. B. [PW94]); diese können leicht in VALSOFT integriert werden.

[2] Falls die Kontrollabhängigkeitskanten keinen Baum bilden, z.B. wg. GOTOs, muß für die $R(k_\nu^\mu)$ selbst eine ähnliche Formel angewendet werden [Sne96].

Pfadbedingungen wurden in [Sne96, KS98] genau untersucht. Es wurden exakte Formeln für ihre Generierung angegeben, die im Fall GOTO-freier Programme sogar redundanzfrei sind. Auch bei beliebigem Kontrollfluß kann durch Ausklammern gemeinsamer Teilpfade weitgehende Redundanzfreiheit erreicht werden. Ferner stellte sich überraschenderweise heraus, daß Zyklen im PDG die Pfadbedingungen nicht beinflussen und ignoriert werden können. Denn Pfadbedingungen sind möglichst allgemeine notwendige Bedingungen, und Iterationen durch einen Zyklus machen die Bedingungen nur spezieller. Die Pfadbedingungen samt Abhängigkeitsgleichungen müssen allerdings in jedem Fall vereinfacht werden, da sie viele redundante Teilausdrücke enthalten können. Ziel ist es, Pfadbedingungen so zu vereinfachen, daß lokale Hilfsvariablen möglichst verschwinden und die Bedingungen nach den Eingabevariablen (Datenquellen) aufgelöst sind.

Als Beispiel betrachten wir wiederum das Programm aus Abbildung 1. Die möglichen Pfade von `p_cd` nach `u_kg` müssen weiter analysiert werden. Die Berechnung der Pfadbedingung liefert nach Vereinfachung folgenden Ausdruck (siehe [Sne96] für Details):

```
p_cd[CTRL2] & 0x01 ≠ 0 ∧
p_cd[CTRL2] & 0x10 ≠ 0
∧ (e_puf[idx]='+' ∨
e_puf[idx]='-')
```

Man beachte, daß in diesem Beispiel die beiden Auftreten von `p_cd` nicht unterschieden werden müssen, da zwischen (5) und (8) keine Umdefinition des Registers stattfindet. Einsetzen von Abhängigkeitsgleichungen aufgrund der Kante (7)→(9) führt wegen `e_puf[idx]=p_cd[PA]` zur vereinfachten Bedingung, die den Eingabepuffer nicht mehr enthält:

```
  p_cd[CTRL2] & 0x01 ≠ 0
∧ p_cd[CTRL2] & 0x10 ≠ 0
∧ (p_cd[PA]='+' ∨ p_cd[PA]='-')
```

Hieraus ist ersichtlich, daß eine Manipulation des Eichpfades durch Eingabe von '+' oder '–' erfolgen kann, sofern gleichzeitig das Bit für „Paper Out" gesetzt ist. Für größere Programme lassen sich derartige Manipulationen des Eichpfades kaum noch manuell entdecken!

4 Aufbau von VALSOFT

Gegenstand einer VALSOFT-Analyse ist immer eine Menge von ANSI-C-Quelltexten zusammen mit ihrer Konfigurationsinformation (s.u.). Der Hersteller muß ferner von vornherein angeben, welche Teile der Software zum Eichpfad gehören. Nur Datenflüsse in den Eichpfad hinein werden von der PTB als sicherheitskritisch betrachtet und müssen untersucht werden. Die Analyse läuft auf UNIX bzw. Linux Workstations, während als Zielplattform zur Zeit nur Standard-PCs unterstützt werden.

Der Grobaufbau des VALSOFT-Kerns ist in Abbildung 3 dargestellt. Der Analysekern umfaßt Front-End, PDG-Erzeugung, Common-Subexpression-Elimination, Constant-Propagation, Slicer und Solver. Schnittstellen zur Visualisierung, zum Benutzer, zur Konfigurationsinformation sowie zu externen Constraint-Solvern dienen der Integration zum Gesamtsystem.

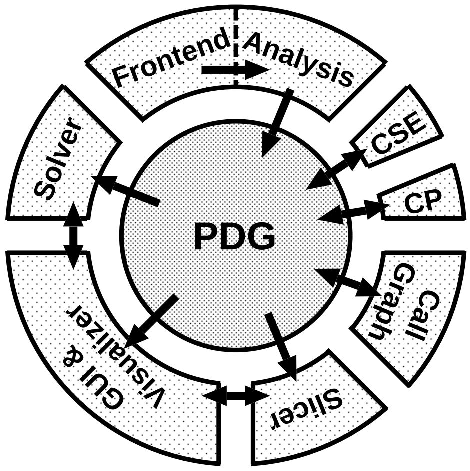

Abbildung 3: Aufbau des Prüfwerkzeuges

4.1 Front End

Das Valsoft-Werkzeug verarbeitet ANSI-C-Programme, die durch den Präprozessor vorverarbeitet sind. Diese Programme müssen syntaktisch korrekt und ablauffähig sein. Das C-Frontend überführt die C-Module in einen attributierten abstrakten Syntaxbaum. Dieser bildet die Grundlage für die Datenflußanalyse und den Aufbau des Abhängigkeitsgraphen. In der Link-Phase wird aus den Symboltabellen der einzelnen Module eine gemeinsame Symboltabelle für Funktionen und Objekte mit externer Linkage erzeugt. Extern bzw. global verwendeten Identifiern gleichen Namens und gleichen Typs wird dabei ein gemeinsames Symbol zugeordnet.

4.2 PDG und Slicer

Aus dem Syntaxbaum wird der Programmabhängigkeitsgraph (PDG) aufgebaut. Unter Verwendung der besten zur Zeit bekannten Verfahren (insbesondere [RHSR94]) werden alle Sprachkonstrukte von ANSI-C analysiert und in entsprechende Knoten und Kanten des PDG umgesetzt. Für die interprozedurale Analyse werden die Kanten des Systemabhängigkeitsgraphen bestimmt. Gegenüber ANSI-C gibt es zur Zeit noch folgende Einschränkungen:

- Pointer: es wird nur eine flußunabhängige Alias-Analyse durchgeführt.
- Switches: diese werden nur bei strukturierter Benutzung behandelt.
- Gotos: entsprechende Algorithmen müssen noch integriert werden.
- Unions: diese werden im Moment wie Structs behandelt.

Nur aus Zeitgründen war die Behandlung dieser Punkte nicht mehr möglich. Die C-typischen Konstrukte setjump/longjump sind derzeit weder von unserem noch von einem der international erhältlichen Programanalysatoren behandelbar.

Abweichend von traditionellen Ansätzen enthält unser PDG Knoten nicht nur für Anweisungen, sondern auch für (Teil)ausdrücke. Diese Erweiterung ist notwendig, da Ausdrücke Seiteneffekte erzeugen können, was später bei der Erzeugung der Pfadbedingungen berücksichtigt werden muß. Da sie aber zu einer stark erhöhten Anzahl von Knoten führt, wurde ein Möglichkeit geschafen, Knoten in sog. Units zusammenzufassen, wobei innerhalb einer Unit maximal ein Seiteneffekt erlaubt ist. Dieses Vorgehen hat den Vorteil, daß keine Programmtransformation zum Entfernen von Seiteneffekten notwendig ist und der erzeugte PDG strukturell sehr ähnlich zum Syntaxbaum und damit zum Quelltext gehalten wird.

Der feingranulare PDG ist die Basis für alle weiteren Analyseschritte und wird persistent in einer Datei gespeichert. Zusätzliche Werkzeuge können den PDG vereinfachen:

- eine Analyse des Aufrufgraphen („Call Graph“ in Abb. 3) entfernt überflüssige Aufrufkanten und die durch sie erzeugten Datenabhängigkeiten (nur bei Verwendung von Funktions-Zeigern);
- Auswertung und Propagation von Konstanten („CP“ in Abb. 3) führt zur Reduzierung von Daten- und Kontrollabhängigkeiten;
- Elimination von gleichen Teilausdrücken („CSE“ in Abb. 3) faßt gleiche Teilgraphen zusammen.

Der PDG ist auch Basis für den Slicer, der als selbständiges Werkzeug Chops, Vorwärts- und Rückwärtsslices berechnen kann[3]. Aufgrund der feineren Struktur unserer PDGs konnten die bekannten Slicing-Algorithmen nicht ohne Änderungen angewandt werden, da innerhalb von Ausdrücken die Kontroll- und Datenabhängigkeiten Zyklen erzeugen. Die klassischen Algorithmen würden deshalb immer komplette Anweisungen in den Slice einfügen, was zwar zu gleichen Ergebnissen wie traditionelle PDGs führt, aber unsere feingranularen PDGs konterkariert. Die feingranularen PDGs haben nur einen linearen Mehrbedarf an Zeit und Platz zur Folge.

Der tatsächlich berechnete, feingranulare PDG zum vollständigen Programm aus Abbilung 1 ist zusammen mit dem Quelltext des Beispiels in Abbildung 4 zu sehen. Der Chop zwischen `p_cd` und `u_kg` ist im Programmtext invertiert; entsprechende Knoten sind im PDG markiert. Das Beispiel vermittelt einen Eindruck von der Komplexität realer PDGs sogar für kleine Programme.

[3]Ein Rückwärtsslice umfaßt alle Anweisungen, die einen bestimmten PDG-Knoten beeinflussen. Ein Vorwärtsslice umfaßt alle Anweisungen, die von einem bestimmten PDG-Knoten beeinflußt werden. Ein Chop zu zwei Knoten ist der Durchschnitt von Vorwärtsslice des ersten und Rückwärtsslice des zweiten Knotens.

4.3 Erzeugung und Vereinfachung der Pfadbedingungen

Neuartig an VALSOFT ist das Erzeugen von Pfadbedingungen, die kritische Slices eingrenzen und genau charakterisieren. Der sog. Solver erzeugt und vereinfacht diese Pfadbedingungen. Die Anwendung des Solvers erfolgt in der Regel dann, wenn der Verdacht auf eine Manipulation des Eichpfades besteht. In diesem Fall werden die Pfadbedingungen des verdächtigen Pfades berechnet. Der Solver kann aber auch auf beliebig ausgewählten PDG-Knoten arbeiten. Die berechneten minimalen Pfadbedingungen werden ausgegeben und können durch einen Constraint-Solver weiterverarbeitet werden.

Zunächst werden die Pfadbedingungen aufgestellt, wobei es zwischen zwei PDG-Knoten eine große Menge an Pfaden geben kann. Gemäß der in Abschnitt 3 skizzierten Formeln werden die regierenden Bedingungen der Anweisungen im Pfad bestimmt; bei mehreren Pfaden werden gemeinsame Teilpfade soweit wie möglich „ausgeklammert“. Komplexe Ausdrücke (die aus IF-, WHILE- usw. Anweisungen stammen) werden in elementare Bedingungen zerlegt; für jeden Elementarausdruck wird eine symbolische Variable eingeführt. Die so transformierten Pfadbedingungen werden schließlich einem Standardalgorithmus zur Minimierung boolescher Ausdrücke übergeben. Um eine annehmbare Ausführungszeit zu gewährleisten, wurde ein mehrstufiges Optimierungsverfahren entwickelt, das parallel zum Aufbau der Pfadbedingungen prädikatenlogisch widersprüchliche bzw. implizierende Pfade erkennt und von vornherein ausschließt; so werden von vornherein weitgehend redundanzfrei Bedingungen erzeugt. Pfadbedingungen können auch für interprozedurale Slices berechnet werden; dabei werden die zusätzlichen Kanten im Systemabhängigkeitsgraphen herangezogen.

Abbildung 5 zeigt die minimierte Pfadbedingung aus Abschnitt 3 in der Form, in der sie zur Zeit tatsächlich berechnet und ausgegeben wird. Das letzte Wort bezüglich Benutzerschnittstelle ist hier aber noch nicht gesprochen.

4.4 Anschluß externer Solver

Der “Solver” im VALSOFT-Kern generiert lediglich aussagenlogisch minimierte Pfadbedingungen. Diese sind für viele Anwendungen zwar schon sehr aussagekräftig, wünschenswert ist aber eine Auflösung der Pfadbedingungen nach den Eingabevariablen. Dazu hat der Solver eine (in Abbildung 3 nicht sichtbare) Schnittstelle zum Anschluß externer Constraint Solver, Deduktionsbeweiser o.ä. Diese werden über eine Textschnittstelle betrieben. Von vornherein ist an den Anschluß mehrerer Systeme gedacht, die jeweils dedizierte Teilaufgaben erfüllen:

- Constraint Solver, z.B. das System CLP(BNR) [vHS96] können Bedingungen über Booleans, Integers und Reals vereinfachen und berechnen numerische Intervalle für die Zielvariablen.
- Systeme zur symbolischen Mathematik, z.B. Mathematica, haben nach unseren bisherigen Erfahrungen

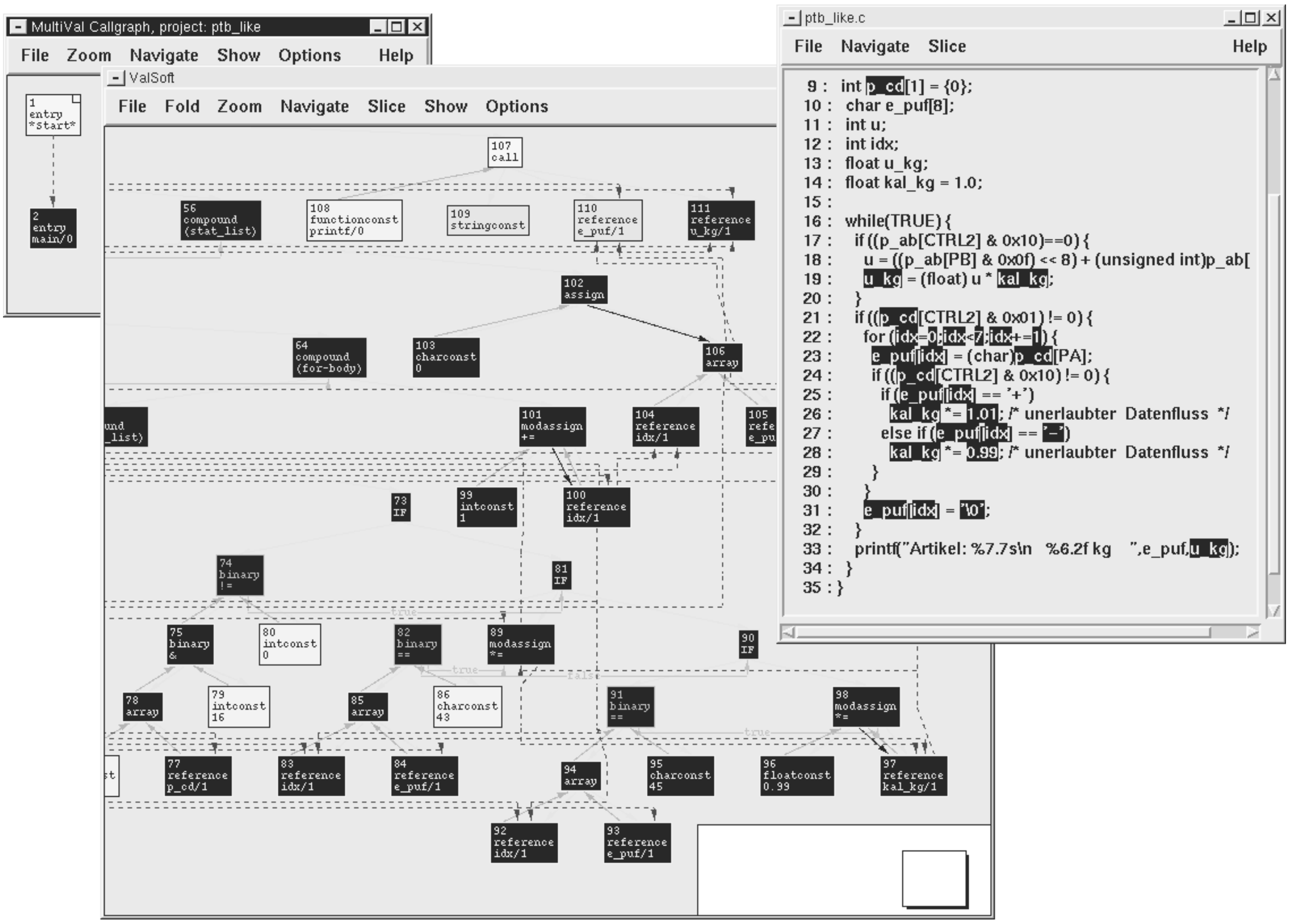

Abbildung 4: Benutzerschnittstelle

mindestens denselben Leistungsumfang wie Constraint Solver.

- Spezialverfahren wie z.B. der Omega-Test zur Abhängigkeitsanalyse von Array-Elementen [PW94] dienen zur Unterstützung differenzierter programmiersprachlicher Analysen.

Im Moment ist nur Mathematica wirklich angeschlossen; der Anschluß der anderen Systeme wird vorbereitet. Man beachte, daß aufgelöste Pfadbedingungen auch juristisch relevant sind, denn sie dienen gerade als Zeugen für einen beanstandeten Datenfluß. Wenn der Hersteller eine Eichpfadbeeinflussung nicht zugeben will, so wählt man einfach Eingabewerte aus den errechneten Intervallen, und der Prüfling wird das beanstandete Verhalten reproduzieren. Abzuwarten bleibt, ob hierbei wirklich 100%tige Präzision erreicht werden kann, oder ob manche Pfadbedingungen nur näherungsweise bestimmt werden können.

4.5 Visualisierung und Benutzerschnittstelle

Der vollständige PDG kann sehr viele Knoten und Kanten enthalten. Als Einstieg in die Graphansichten wird deshalb zunächst ein „Aufrufgraph“ gezeigt. Dieser enthält Knoten für jede Funktion des PDG und Kanten zum Anzeigen der Aufrufstruktur. Durch einfachen Doppelklick auf einen solchen Funktionsknoten kann der intraprozedurale Programmabhängigkeitsgraph (PDG) dieser Funktion in einem eigenen Grapheditorfenster betrachtet werden.

Im Grapheditor besteht generell die Möglichkeit, Knoten zu selektieren, den Graphen durch Zoomen in eine Detail- oder Übersichtsdarstellung zu bringen, oder durch die Faltmechanismen Teilbäume zu einem Knoten zu verdichten. Das Navigieren im Graphen wird durch Such- und Anzeigefunktionen unterstützt. Mittels eines zweidimensionalen Panners kann der sichtbare Ausschnitt verschoben werden. Zu selektierten Einheiten im Graph oder im Quelltextfenster können Berechnungen von Vorwärts- und Rückwärts-Slices, Chops, Pfad- oder Ausführungsbedingungen angestoßen werden. Das Ergebnis ist eine Menge von Knoten, die sowohl im Graph als auch im Quelltext durch Selektion deutlich angezeigt wird. Im Aufrufgraph sind alle betroffenen Prozeduren markiert. Selektierte Knoten können in eine Knotenmenge übernommen und auch gespeichert werden. Über diese Knotenmengen können verschiedene Werkzeuge, z. B. der Slicer und der Solver, mit dem GUI kommunizieren.

Interessanterweise hat sich inzwischen herausgestellt, daß schon für kleine Programme die PDGs so groß werden, daß ihre graphische Darstellung dem Anwender nicht mehr weiterhilft. Zur Zeit wird deshalb eine alternative Benutzerschnittstelle entwickelt, die sich nur auf die Text-

```
Path Condition
----------------------------------------
((e_puf$25[idx$25] == 43) OR
  (e_puf$27[idx$27] == 45))
AND ((p_cd$21[0] & 1) != 0)
AND ((p_cd$24[0] & 16) != 0)
AND (idx$22 < 7)
AND ((p_ab$17[0] & 16) == 0)
-----------------------------------------

Substitution
-----------------------------------------
e_puf$25[idx$25] = e_puf$10
OR
(e_puf$25[idx$25] = p_cd$23[1]
AND idx$25 == idx$23)
OR
(e_puf$25[idx$25] = 0)
AND idx$25 == idx$31)

e_puf$27[idx$27] = e_puf$10
OR
(e_puf$27[idx$27] = p_cd$23[1]
AND idx$27 == idx$23)
OR
(e_puf$27[idx$27] = 0
AND idx$27 == idx$31)
...
```

Abbildung 5: Ausgabe des Solvers für Abbildung 1

darstellung des Programms stützt und Slices etc. im Programmtext farbig markiert. Die PDG-Visualisierung bleibt aber unverzichtbar zur Erprobung von VALSOFT selbst.

4.6 Modellierung der Zielsystem-Umgebung

Meßgerätesoftware ist hardwarenahe Software. Die Programme enthalten Zugriffe auf Hardwareregister (z. B. Ein-Ausgabeschnittstelle), verwenden Bibliotheksfunktionen zur Gerätesteuerung oder enthalten sogar eingebetteten Assemblercode. Für die Analyse ist allerdings das tatsächliche Verhalten des Assemblercodes, der Bibliotheksfunktionen usw. ohne Belang. Entscheidend ist, welche Kontroll- und Datenabhängigkeiten sie erzeugen. Dazu reicht eine Simulation der Bibliotheksfunktionen usw. durch rudimentäre C-Prozeduren. Diese müssen dieselbe Schnittstelle und dieselben Datenabhängigkeiten (bezüglich Parametern und globalen Objekten) haben wie die echten Bibliotheksfunktionen. Der Vorteil ist, daß die PDGs dieser „Stümpfe“ mit dem Werkzeug selbst erzeugt werden können.

Als Zielsystem wurde ausschließlich ein Standard-PC auf der Basis des Betriebssystems MS-DOS betrachtet, der Programme in der Programmiersprache ANSI-C ausführt. Die Modellierung der Software-Umgebung umfaßt die Modellierung der MS-DOS- und BIOS-Betriebssystemfunktionen und die Standard-C-Programmbibliothek.

Um VALSOFT für andere Zielplattformen einzusetzen, ist wiederum eine entsprechende Modellierung notwendig. Der VALSOFT-Kern kann auch stets ohne Zielplattformmodellierung betrieben werden, weiß dann aber nichts über Hardwareschnittstellen und Bibliotheksfunktionen.

4.7 Performance

Viele Möglichkeiten zur Optimierung (z. B. Binärformat für die PDG-Dateien) konnten aus Zeitmangel nicht mehr realisiert werden. Dennoch können sich die bisherigen Performance-Messungen (auf einem P200 Linux-System) durchaus sehen lassen:

LOC	PDG-Knoten	PDG-Zeit	Solver-Zeit
31	113	0.1s	1s
830	3800	1.9s	1s - 10s
2107	6090	3.3s	5s - 3min

Da für eine Software-Sicherheitsprüfung nur einige wenige Solver-Aufrufe notwendig sind (nämlich für Pfade, die die Grenze des Eichpfades überqueren), sind Analysezeiten von einigen Minuten durchaus akzeptabel.

5 Andere Arbeiten und Ausblick

Es gibt eine Fülle von Arbeiten, die Verfahren zum Slicing und zur Abhängigkeitsanalyse darstellen (siehe dazu den erwähnten Übersichtsartikel von Tip). Diese wurden überwiegend in den USA entwickelt. Es wurden eine ganze Reihe von Forschungsprototypen realisiert, während die amerikanische Industrie erst in letzter Zeit in Slicing-Technologien investiert hat.

Die bekanntesten Systeme seien hier kurz aufgeführt:

- Das amerikanische National Institute of Standards entwickelte das System UNRAVEL, das explizit für Sicherheitsprüfungen konzipiert wurde. UNRAVEL deckt den vollen C-Sprachumfang ab; lediglich „Long Jumps“ und komplexe Pointerstrukturen können nicht analysiert werden [LWG$^+$95].

- Das Wisconsin Slicing Project ist eines der ältesten Slicing-Werkzeuge von den Pionieren S. Horwitz und T. Reps. Es ist sehr leistungsfähig, aber erst seit kurzem öffentlich verfügbar [HKF96].

- IBM entwickelte zwei Slicing-Tools für Cobol und PL/1, die insbesondere zum Software-Reengineering (z. B. „Jahr 2000“-Problem) gedacht sind. Die Werkzeuge werden kommerziell vertrieben[4]. IBM hat demnach die Priorität bei kommerziellen Slicing-Anwendungen.

Wenn man bestimmte Eingabewerte kennt, lassen sich mit dynamischem Slicing [ADS91] die Slices viel stärker eingrenzen. Dies hilft aber nicht bei Sicherheitsprüfungen, bei denen die kritischen Eingabewerte ja gerade nicht bekannt sind – in VALSOFT werden sie erst durch Constraint Solving deduziert. Stellt man sich allerdings vor, daß VALSOFT schon beim Hersteller für entwicklungsbegleitende Qualitätssicherung eingesetzt wird, so kann die Sicherheitsprüfung mit dem Funktionstest kombiniert werden, und dann wird auch dynamisches Slicing möglich: die Präzision der Analyse wird dann nochmals besser.

[4] das Cobol-Tool z.B. als Teil der „Visual Age for Cobol Professional Edition“

6 Schluß

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“. VALSOFT ist es erstmals gelungen, Slicing mit deduktiven Verfahren zu koppeln. Ergebnis ist der Prototyp eines Werkzeugs, das mittelgroße ANSI-C Programme analysieren kann und zu verdächtigen Datenflüssen genaue Pfadbedingungen berechnet. Erste Fallstudien zeigen, daß VALSOFT seinem hohen Anspruch gerecht wird; es sind aber noch einige Anpassungen an reale Meßgeräte erforderlich sowie die Integration von Analyseverfahren für komplexe Pointer- und GOTO-Strukturen.

VALSOFT bietet die Chance, die Slicing-Technologie in Deutschland zu verankern. Es bleibt abzuwarten, ob die Industrie nunmehr den VALSOFT-Prototypen aufgreift.

Danksagung. VALSOFT wurde vom BMBF unter dem Förderkennzeichen 01 IS 513 A-C und wird von der DFG unter dem Kennzeichen Sn11/5-1 gefördert. K.-U. Bloem, U. Grottker, E. Dorok, B. Keitel, M. Hauser, F. Ehrich, J. Diederich und O. Pfohl leisteten hervorragende Arbeit bei der Implementierung von VALSOFT.

Literatur

[ADS91] Hiralal Agrawal, Richard A. DeMillo und Eugene H. Spafford. Dynamic slicing in the presence of pointers, arrays and records. In *Proc. ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*, S. 60–73, Oktober 1991.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, Band 29(6), S. 302–312, Orlando, FL, Juni 1994.

[Ber95] Valdis Berzins. *Software Merging and Slicing*. IEEE Computer Society Press, Los Alamitos, 1995.

[GGS96] M. Goldapp, U. Grottker und G. Snelting. Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving. In *Statusseminar des BMBF Softwaretechnologie*, S. 405–425, Berlin, 1996. Auch als Informatik-Bericht 96–02 (März 1996), TU Braunschweig, FB Informatik.

[HKF96] Tommy Hoffner, Mariam Kamkar und Peter Fritzson. Evaluation of program slicing tools. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, S. 51–69, Saint Malo, France, Mai 1996.

[HRB90] Susan B. Horwitz, Thomas W. Reps und David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Januar 1990.

[KS98] Jens Krinke und Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. Informatikbericht 98-01, TU Braunschweig, Januar 1998. Zur Veröffentlichung eingereicht.

[LWG+95] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole und David W. Binkley. Unravel: A case tool to assist evaluation of high integrity software. Bericht 5691, National Institute of Standards and Technology, August 1995.

[PW94] W. Pugh und D. Wonnacott. Static analysis of upper and lower bounds on dependency and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, Januar 1994.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RHSR94] Thomas W. Reps, Susan B. Horwitz, Mooly Sagiv und Genevieve Rosay. Speeding up slicing. In *SIGSOFT'94: Proc. Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, S. 11–20, New Orleans, LA, Dezember 1994. ACM Software Engineering Notes 19(5).

[Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis; Third International Symposium, SAS'96*, *LNCS*, Band 1145, Aachen, September 1996. Springer Verlag.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[vHS96] Pascal van Heytenryck und Vijay Saraswat. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701–726, Dezember 1996.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, Juli 1984.