

SOFTWARE – DISASTER,
UND WIE MAN SIE
VERHIDERN KANN

Proseminar

Thema : Der Pentium Bug (1994)

Wintersemester 2002/2003

11 Dezember 2002

Iliya Iliev

iliev@in.tum.de

Inhaltsverzeichnis

I. Einleitung

- 1. Der Anfang
- 2. Was eigentlich schief ging ?
- 3. Der Algorithmus (und warum er gewählt wurde)
- 4. Warum es funktioniert ?
- 5. Anatomie des Bugs

II. Formale Methoden

- 1. Was bedeutet der Begriff "Formale Methoden" ?
- 2. Warum formale Methoden ? (Beispiele)
- 3. Was ist PVS ?
- 4. Die PVS-Sprache
- 5. Der PVS-Beweiser
- 6. Das PVS-Interface
- 7. Der Beweis

III. Zusammenfassung

IV. Quellenangabe

I. Einleitung

1. Der Anfang

Erste Beschreibung des Bugs in der Öffentlichkeit, erschien als E-Mail von einem Mathematikprofessor (Originalnachricht):

FROM: Dr. Thomas R. Nicely, Professor of Mathematics,
Lynchburg College, Lynchburg, Virginia ,30 October 1994
TO: Whom it may concern RE: Bug in the Pentium FPU

It appears that there is a bug in the floating point unit (numeric co-processor) of many, and perhaps all, Pentium processors. In short, the Pentium FPU is returning erroneous values for certain division operations. For example,

1/824633702441.0

is calculated incorrectly (all digits beyond the eighth significant digit are in error)...even the Windows calculator (use the scientific mode), by computing

$(824633702441.0) * (1/824633702441.0)$,

which should equal 1 exactly (within some extremely small rounding error; in general, coprocessor results should contain 19 significant decimal digits). However, the Pentiums tested return

0.999999996274709702

for this calculation.

.....The bug has not been observed on any 486 or earlier system, even those with a PCI bus. If the FPU is locked out (not always possible), the error disappears; but then the Pentium becomes a "586SX", and floating point must run in emulation, slowing down computations by a factor of roughly ten. I encountered erroneous results which were related to this bug ... I had eliminated all other likely sources of error (software logic, compiler, chipset, etc.). ...the bug was observed on a 66-MHz system at Intel, but there was no further information or explanation, other than the fact that no such bug had been previously reported or observed.

Prof. Thomas Nicely war aber nicht der Erste, der den Bug (Intel nennt ihn 'flaw') entdeckt hat. Eigentlich wurde er schon früher von Intel bemerkt, es wurden aber keine Massnahmen (in der Öffentlichkeit) getroffen. Der Mathe-Professor teilte zusätzlich seine 'Entdeckung' dem „Electronic Engineering Times“ mit. Der FDIV-Bug (FDIV steht für Floating-Point-Division) entwickelte sich schnell zum Lieblingsthema in Internet und in verschiedenen news-groups (z.B. comp.sys.intel, comp.soft-sys.matlab, etc.).

Viele Fachleute zeigten grosse Interesse an diesem Problem. Einer von denen war Tim Coe, Ingenieur bei Vitesse Semiconductor – eine Konkurrentfirma von Intel. Er analysierte den Bug und veröffentlichte einige Beispiele, die die Fehleranfälligkeit des Pentium-Prozessors zeigten.

In MATLAB berechnete er den Ausdruck

$$x = 4195835 ; y = 3145727 ; z = x - (x/y)*y$$

Man erwartet, dass $z = 0$ auskommt. So war auch das Ergebnis, das von verschiedenen Rechnern (mit Intel 286,386,486 chips) geliefert wurde. Aber der Pentium zeigte $z=256$, was einem relativen Fehler $z/x = 2^{-14}$ oder $6.1e-5$ entspricht. Der berechnete Quotient x/y bleibt korrekt nur bis zu der 14-ten Bitstelle.

2. Was eigentlich schief ging ?

Der Pentium®-Prozessor wurde Mai '93 vorgestellt. Das Ziel war durch niedrige Kosten und hohe Effizienz die anderen Chipproduzenten ('Rivalen' von Intel) im Bereich der Wissenschaftlichen- und Ingenieurwissenschaften zu schlagen.

(wie z.B.: Digital Equipment Corporation(DEC), Hewlett-Packard, IBM, Sun Microsystems)

Intel hat das teuerste Test- und Werbeprogramm (mit dem Logo „Intel Inside“) in ihrer Geschichte eingesetzt. Die CPU-Architektur ist 3mal komplexer als die der 486-CPU und enthält verbesserte Floating-Point-Algorithmen. Eigentlich war der Prozessor noch Ende '92 'produktionsbereit', man wartete aber bis mehr Tests gemacht werden. Im Sommer '94 wurde ein Fehler in der FPU entdeckt und Intel startete ein separates Projekt, gemeinsam mit zusammenarbeitenden Mathematikern und Wissenschaftlern, die sich mit dem Problem beschäftigten. Der Fehler wurde als 'sehr selten' kategorisiert und aus dem Herstellungsprozess beseitigt. Ein Monat später kam die Nachricht von Prof. Th. Nicely ins Netz. Aber zusammen mit der fehlerhaften Public-Relations- und Kunden-Politik kostete Intel insgesamt der Bug \$475,000,000 (laut eigene Angaben).

3. Der Algorithmus (und warum er gewählt wurde)

Die FPU des Prozessors benutzt zur Floating-Point-Division den radix-4-SRT-Algorithmus, (entwickelt von Sweeney, Robertson, Tocher im 1958), er gehört der Gruppe der 'non-restoring'-Algorithmen, d.h. er kann sich die 'früheren' Rechenschritten nicht merken. Die Implementation des Algorithmus beim Pentium erlaubt die Berechnung von zwei Quotientenbits pro Takt. Die 486-CPU benutzte den klassischen „shift-and-subtract“-Algorithmus und erzeugte nur 1 Quotientenbit pro Takt. Durch verschiedene Tricks (Kombination von Compiler und Hardware-Technologie) konnte der neue Prozessor im Vergleich zu 486-Prozessor mit Gleitkommazahl-Skalar- und Gleitkommazahl-Vektor-Kode 3- bzw. 5-mal schneller rechnen.

Hier ist eine kurze Beschreibung der einzelnen Schritten :

- S1. Sammle die signifikanten Stellen von Divisor und Dividend.
- S2. Lese aus der Lookup-Tabelle eine Schätzung für die nächste Stelle des Quotienten.
(in unserem Fall $\{-2,-1,0,1,2\}$)
- S3. Multipliziere den Divisor mit dem Schätzungsquotienten und subtrahiere das Ergebnis von dem Dividend.
(Die Operation ist Addieren, falls der Schätzungsquotient negativ ist.)
- S4. Speichern der Quotienten-Zahlen ins Quotientenregister.
- S5. Der Rest und das Quotientenregister um 2 Stellen nach links verschieben.
- S6. Sammle signifikante Stellen des neuen (verschobenen) partiellen Rests.
- S7. Wieder nach Schritt S2 gehen, falls weitere Nachkommastellen berechnet werden sollen, sonst gehe nach S8.
- S8. Der binäre Quotient durch Zusammenhängen (assembling) der Werten im Quotientenregister bilden.
- S9. Falls der letzte partielle Rest negativ war, dann subtrahiert man sein Wert von dem Quotienten, um korrektes Ergebnis zu erhalten.

Oder (mehr mathematisch) :

p (divisor) initialisiert den partiellen Rest, der immer wieder neu berechnet wird, und die q 's sind die entsprechend gewählten Quotienten, die Multiplikation mit 4 entspricht der Verschiebung von Rest und Quotientenregister je einmal nach links.

$p_0 := p$
 for $k = 0, 1, \dots$
 „Lookup“ a digit $q_k \in \{-2, -1, 0, 1, 2\}$ in such way that
 $p_{k+1} := 4 * (p_k - q_k * d)$

satisfies $|p_{k+1}| \leq (8/3) * d$

und am Ende gilt (also man ein korrektes Ergebnis bekommen)

$$p/d = \sum_{i=0}^{\infty} q_i / 4^i$$

Man nennt ihn radix-4, weil man mit Basis 4 rechnet (Rest und Quotientenregister je einmal nach links verschieben – $2 * 2 = 4$). Mehr dazu kann man in [Pra95] und [WPI] finden.

4. Warum es funktioniert ?

Da man mit Floating-Point-Zahlen arbeitet und der Algorithmus non-restoring ist, können Fehler im Ergebnis auftreten. Deshalb braucht man eine Lookup-Tabelle mit positiven und negativen Einträge, die bei der Quotientenwahl benutzt wird. Es hängt von dem Ergebnis, ob es unter- bzw. überschätzt wird, was für einen Quotient nötig ist, damit der Fehler bei der nächsten Operation durch Addieren (negativer Eintrag in der Tabelle) bzw. Subtrahieren (positiver Eintrag in der Tabelle) korrigiert wird. Auf diese Weise wird viel an Geschwindigkeit gewonnen. Ein Beweis des Algorithmus kann man in [AC95] finden.

Das Prinzip des Algorithmus beruht auf die Tatsache, dass der absolute Wert des Neuberechneten Rests nie größer als $(8/3) * \text{Divisor}$ (die beiden Zahlen sind in Form von „normalized fractions“) ist, da der absolute Wert von $(\text{Rest} - \text{Quotient} * \text{Divisor})$ kleiner gleich $(2/3) * \text{Divisor}$ ist.

Also für:

$$p_{k+1} = 4(p_k - q_k * d) \text{ gilt } p_{k+1} \leq (8/3) * d \quad (1)$$

Definiert man $Q_k \equiv \sum_{i=0}^{k-1} (q_i / 4^i)$ und die Gleichung

$$(p/d) = Q_k + (p_k / d) * 4^{-k} \quad (2)$$

ist equivalent zu

$$(p_k * 4^{-k}) = (p_{k+1} / 4^{k+1}) + ((q_k * d) / 4^k)$$

Es wurde benutzt, dass (p_k / d) zu einem konstanten Wert (in unserem Fall $(8/3)$) gebunden werden kann. Als Folgerung von (2) bekommen wir die folgende Formel

$$p_k = d * \sum_{i=k}^{\infty} q_i / 4^{i-k} = d * (q_k + (q_{k+1} / 4) + (q_{k+2} / 4^2) + \dots)$$

Wenn wir die geometrische Progression aufsummieren (für alle „extrem“ gewählte $q_i = +2$ oder $q_i = -2$), ergibt sich, dass die Bedingung $|p_k| \leq (8/3)*d$ nicht zufällig gewählt worden ist, sondern ein wichtiger Teil des Algorithmus ist. Bei dem Pentium wird für bestimmte Werte von d und für bestimmte Werte von p_k als Quotient $q_k = 0$ statt $q_k = 2$ (wegen die falschen Einträge in der Look-up Tabelle) gewählt.

5. Anatomie des Bugs

Die Algorithmusimplementation bei dem Pentium benutzt eine Lookup-Tabelle (division table), die im PLA (Programmable-Logic-Array) des Prozessors gespeichert ist. Diese Tabelle enthält insgesamt 1066 Einträge mit verschiedenen Werten $\{-2, -1, 0, 1, 2\}$. Beim ersten Laden dieser Tabelle ist ein Fehler aufgetreten (oder die Tabelle ist falsch berechnet worden), und richtig wurden nur 1061 Werte gespeichert, die übrigen 5 Zellen der Tabelle hatten einen Wert 0 statt 2, was natürlich später zu falschen Ergebnissen führte. Die offizielle Position von Intel (wie der Fehler verursacht wurde) ist:

“...a script was written to download the entries into a hardware PLA[i.e. division table]. An error was made in this script that resulted in a few lookup entries...being omitted from the PLA.”

Also der Fehler konnte durch einen Programmierfehler oder fehlerhafte Compilierung des Skripts verursacht werden. Manche Leute glauben, dass die Tabelle einfach falsch berechnet war, die falschen Einträge für unerreichbar gedacht waren und somit keine Tests mit diesen Werten notwendig sind. Diese Werte treten zwar sehr selten auf, bleiben aber erreichbar (meistens für wissenschaftliche Anwendungen). Man kann sagen, dass die Testmethoden unglücklich gewählt worden sind.

Die am breitesten eingesetzten Methoden zur Entdeckung und Vermeidung solcher Fehlern sind die formalen Methoden. Diese Methoden gewinnen an Bedeutung, werden immer beliebter und öfter in der Industrie eingesetzt.

II. Formale Methoden

1. Was bedeutet der Begriff 'Formale Methoden' ?

Formale Methoden ist ein zusammenfassender Begriff aus der theoretischen Informatik für eine große Vielfalt verschiedener Methoden, die nicht nur zur Untersuchung und Modellierung (grundlegender) Strukturen und Prozesse, sondern auch dem Richtigkeitsbeweis von Systemen und Programmen dient. Wenn man über die formale Beschreibung und formale Spezifikation des Wunschverhaltens eines Systems verfügt, können formale Methoden dazu benutzt werden, dass man mathematisch beweist, ob das System seiner Spezifikation entspricht. Wenn es die Software gemeint ist, oft wird auch der Begriff formale Verifikation benutzt.

2. Warum Formale Methoden ?

Der Pentium-Bug hat gezeigt, wie wichtig, komplex und teuer die Verifikation der Prozessor-Arithmetik sein kann. Formale Methoden sind ein mächtiges und sicheres Mittel zum Richtigkeitsbeweis. Formale Methoden müssen in formalen Systemen umgesetzt werden.

Die formale Spezifikation ist die Benutzung von logischen Notationen zur Beschreibung von:

- Annahmen, Hypothesen, Voraussetzungen (assumptions) über die Arbeitsumgebung des Systems
- Bedingungen, Anforderungen (requirements), die das System erfüllen muss
- Konstruktion, Darstellung (design), die die Anforderungen erfüllt.

Die Logik bietet Berechnungsregel, die es erlauben, gültige Folgerungen aus den Prämissen (Voraussetzungen) zu machen. Eine solche Berechnung nennt man Beweis. Falls die Prämissen gültig für die Umgebung sind (true statements), dann können die Logiktheoreme garantieren, dass die Folgerung auch für diese Umgebung zutrifft. Die Annahmen (Prämisse) für die Umgebung werden explizit gemacht, und zwar getrennt von den Ableitungsregeln.

Techniken (Beispiele):

Was macht eine Methode formal (allgemeine Formel für Hard- und Software) ?

Formale Methode = Logik + mathematische Beschreibung

oder (Softwarerealisation)

Formale Methode = Logik + Programmiersprache

Die Formalen Methoden, realisiert in Theorem-Beweisern bringen viele Vorteile:

- sicherere Beweise als das „Handbeweisen“ – Theoreme neigen zu Flachheit, sind aber zahlreich, enthalten viele Details, die aber nicht alle relevant sind,
- Beweise werden automatisiert – Hilfe bei der Entdeckung von Beweisen, speichern von Beweisen, Wissensbasiserweiterung,
- es gibt zwei Arten von Beweisern: vollautomatische (keine Menschintervention) und interaktive (benötigt menschliche Hilfe/Hilfsanweisungen/ beim Beweisprozess, teuer als Menscheneinsatz)

- Spezifikation mit Z – Notation zur Beschreibung von Systemen, basierend auf Prädikatenlogik der ersten Stufe, wird auch bei der Entwicklung von Hardware- und Softwaresystemen benutzt, mehr in [ZRM].
- Binäre Entscheidungsdiagramme - Die Entscheidungsdiagramme finden in vielen Bereichen der Informatik Anwendung, mehr in [BDD].
- Model checking (Modellprüfung) – formale Methode zur automatischen Verifikation von verteilten Hard- und Softwaresystemen, sehr verbreitet. Das zu prüfende System wie auch relevante Aspekte des Verhaltens aber müssen formal modellierbar sein. Damit stellt das Verfahren eine Alternative oder mögliche Ergänzung zum Einsatz von Theorembeweisern sowie von Simulations- und Testverfahren dar. Findet die Modellprüfung keine Fehler, so ist die Korrektheit des Systems in Bezug auf seine Spezifikation nachgewiesen. Wenn ein Teil einer Spezifikation nicht erfüllt ist, dann liefert das Verfahren einen Zeugen (einen Gegenbeispiel), bei dem das Versagen des Systems zutage tritt. Test- und Simulationsverfahren garantieren dagegen lediglich die Korrektheit in den getesteten bzw. simulierten, stichprobenartig gewählten Einzelfällen und bei der Anwendung eines Theorembeweisers muss die Verifikation manuell gesteuert werden. Mehr zu diesem Thema kann man in [EL] finden.

Formale Systeme (Beispiel):

- Analytica (A Theorem Prover for Mathematica), ein automatischer Theorem-Beweiser, geschrieben in Mathemacsprache, ist auch der erste, der die Technik der symbolischen Berechnung benutzt, braucht weniger Benutzerinteraktion, ist aber sehr aufwendig. Es gelang E.M. Clarke, S.M.German und Xudong Zhao (Carnegie Mellon University) den radix-4-SRT-Divisionsalgorithmus mit Analytica zu beweisen.
 - Isabelle – allgemeiner Theorembeweiser, entwickelt an Cambridge University und TU München (Prof. Larry Paulson (CU), Prof. Tobias Nipkow und Mitarbeiter(TUM))
 - GENEVIEVE – GENeration EnVironment for Effective Verification, ein gemeinsames Project der Europäische Union und IBM mit Schwerpunkt Design-Verification.
 - PVS - Theorembeweiser, entwickelt von SRI(Stanford Research Institute), wird hier vorgestellt.
- weitere:
- HOL, INKA, RRL (Rewrite Rule Laboratory), SPIN (Bell Labs), SMV (Symbolic Model Verifier), ...

3. Was ist PVS ?

PVS ist ein formales Verifikationssystem zur Entwicklung und Analyse von formalen Spezifikationen, das aus den folgenden Teile besteht:

- eine integrierte Spezifikationsprache,
- einen Typprüfer (typechecker erzeugt typecorrectness conditions/TCCs/),
- einen Theorembeweiser (theorem prover),
- viele vordefinierte „Sätze“ (predefined theories, specification libraries),
- verschiedene Dienstprogramme (utilities),
- Dokumentation.

PVS (Prototype Verification System) wurde von SRI (Stanford Research Institute) entwickelt und bietet mechanisierte Unterstützung für die formale Spezifikation und formale Verifikation. Das System kombiniert eine ausdrucksstarke Spezifikationsprache und mächtige automatisierte Ableitung (deduction), die es erlauben viele Beispiele, die früher schwer für viele Verifikationssysteme gedacht waren, verständlich und effizient bearbeitet zu werden.

4. Die PVS-Sprache

Die Spezifikationsprache basiert auf klassische, einfache typorientierte Logik höherer Ordnung. Eine PVS-Spezifikation besteht aus einer Sammlung von Theorien, jede Theorie besteht aus einer Signatur für die Typnamen und Konstanten (die in der Theorie eingeführt sind), Axiomen, Definitionen und andere Theoremen, die mit der Signatur verbunden sind. Eine Theorie kann aus selbstdefinierten Typen bestehen und/oder andere Theorien benutzen, dabei können einzelne Typen und Werte beschränkt werden – Maximum/Minimum der Länge einer Liste usw. Zur Beschränkung von Parametern einer Theorie werden Hypothesen eingesetzt (assumptions) – z.B. wenn man eine geordnete Relation totalgeordnet machen will.

Als Typen unterscheidet man:

- Typen, die vom Anwender eingeführt sind, und
- eingebaute Typen - `bool, real, nat, rational` und mehr

Neue Sprachterme können durch Anwendungsfunktionen, Lambda-Abstraktion, record- und Tupel-Konstruktionen eingeführt werden. z.B.:

- der Typkonstruktor hat den Typ $[A \rightarrow B]$,
- oder wenn man alle reelle Zahlen, die ungleich 0 braucht, würde man sie so definieren
 $\{x:\text{real} \mid x \neq 0\}$

Die Typkonstruktoren beinhalten Funktionen, Sätze (sets), Tupel, records (wie Tabellen), Aufzählungen und rekursiv definierte abstrakte Datentypen (wie Listen und Binärbäume). Prädikattypen (predicate subtypes) und abhängige Typen können zur Einführung von Beschränkungen (Nebenbedingungen-constraints) benutzt werden. Die PVS-Ausdrücke verfügen über typische arithmetischen und logischen Operatoren, Anwendungsfunktionen, Lambda-Abstraktion, Quantoren.

Hier wird die Syntax der Sprache kurz aufgelistet. Ergänzung zu der BNF-Notation:

- In UPPERCASE geschriebene Wörter sind reserviert,
- [a] - bedeutet „a ist optional“,
- | - illustriert, das eine alternative Wahl gemacht werden kann,
- B₊ - B kann ein- oder mehrmals wiederholt werden,
- B* - B kann beliebig viel wiederholt werden(auch 0-mal),
- B₊₊ ' , ' bzw. B** ' , ' – wie oben, aber die einzelne B's müssen mit Kommas getrennt werden .

Achtung: Bei "{ " und "}" auf der rechten Seite muss zwischen Literale der Sprache und BNF-Klammern unterschieden werden.

Die logische Notation in PVS kann kurz in dieser Tabelle aufgelistet werden.

\neg	NOT
\wedge	AND, &
\vee	OR
\supset	IMPLIES, =>
\iff	IFF, <=>
\forall	FORALL
\exists	EXISTS
λ	LAMBDA

und es folgen die Spezifikationsdeklaration, Typdeklarationen, Beispiele für Namen, PVS-Prädikate usw. Mehr zur Syntax der Sprache gibt es in [LR].

Specification

<i>Specification</i>	::= { <i>Theory</i> <i>Datatype</i> } ₊
<i>Theory</i>	::= <i>Id</i> [<i>TheoryFormals</i>] : THEORY [<i>Exporting</i>] BEGIN [<i>AssumingPart</i>] [<i>TheoryPart</i>] END <i>Id</i>
<i>TheoryFormals</i>	::= [<i>TheoryFormal</i> ₊₊ , ']
<i>TheoryFormal</i>	::= [(<i>Importing</i>)] <i>TheoryFormalDecl</i>
<i>TheoryFormalDecl</i>	::= <i>TheoryFormalType</i> <i>TheoryFormalConst</i>
<i>TheoryFormalType</i>	::= <i>Ids</i> : { TYPE NONEMPTY_TYPE TYPE+ } [FROM <i>TypeExpr</i>]
<i>TheoryFormalConst</i>	::= <i>IdOps</i> : <i>TypeExpr</i>

Type Expressions

<i>TypeExpr</i>	::= <i>Name</i> <i>EnumerationType</i> <i>Subtype</i> <i>TypeApplication</i> <i>FunctionType</i> <i>TupleType</i> <i>RecordType</i>
<i>EnumerationType</i>	::= { <i>IdOps</i> }
<i>Subtype</i>	::= { <i>SetBindings</i> <i>Expr</i> } (<i>Expr</i>)
<i>TypeApplication</i>	::= <i>Name Arguments</i>
<i>FunctionType</i>	::= [FUNCTION ARRAY] [{ [<i>IdOp</i> :] <i>TypeExpr</i> } ₊₊ , ' -> <i>TypeExpr</i>]
<i>TupleType</i>	::= [{ [<i>IdOp</i> :] <i>TypeExpr</i> } ₊₊ , ']
<i>RecordType</i>	::= [# <i>FieldDecls</i> ₊₊ , ' #]
<i>FieldDecls</i>	::= <i>Ids</i> : <i>TypeExpr</i>

<i>IdOp</i>	::= <i>Id</i> <i>Opsym</i>
<i>Opsym</i>	::= <i>Binop</i> <i>Unaryop</i> IF TRUE FALSE [[]] (()) {{{}}
<i>Binop</i>	::= ○ IFF <=> IMPLIES => WHEN OR \∨ AND ∧ & XOR ANDTHEN ORELSE ^ + - * / ++ ~ ** // ^^ - = < > = /= == < <= > >= << >> <<= >>= # @@ ##
<i>Unaryop</i>	::= NOT ~ □ <> -
<i>FormulaName</i>	::= AXIOM CHALLENGE CLAIM CONJECTURE COROLLARY FACT FORMULA LAW LEMMA OBLIGATION POSTULATE PROPOSITION SUBLEMMA THEOREM

Datatypes

<i>Datatype</i>	::=	<i>Id</i> [<i>TheoryFormals</i>] : DATATYPE [WITH SUBTYPES <i>Ids</i>] BEGIN [<i>Importing</i> [;]] [<i>AssumingPart</i>] <i>DatatypePart</i> END <i>Id</i>
<i>InlineDatatype</i>	::=	<i>Id</i> : DATATYPE [WITH SUBTYPES <i>Ids</i>] BEGIN [<i>Importing</i> [;]] [<i>AssumingPart</i>] <i>DatatypePart</i> END <i>id</i>
<i>DatatypePart</i>	::=	{ <i>Constructor</i> : <i>IdOp</i> [: <i>Id</i>] } +
<i>Constructor</i>	::=	<i>IdOp</i> [({ <i>IdOps</i> : <i>TypeExpr</i> } ++ ', ')]

5. Der PVS-Beweiser

Der PVS-Theorembeweiser benutzt eine Sammlung von mächtigen Folgerungsprozeduren, die interaktiv mit Führung des Benutzers angewendet werden. Ihre Implementation ist für grosse Beweise optimiert: z.B. die Aussagenvereinfachung benutzt binäre Entscheidungsdiagramme. Die benutzerdefinierte Prozeduren können diese Folgerungen benutzen, um die Beweisstrategien zu verbessern. Ausführlichere Informationen in [PG].

6. Das PVS-Interface

PVS benutzt Gnu-/ X-Emacs, um ein integriertes Interface für seine Sprache und seinen Beweiser bereitzustellen. Die Kommandos können mit den Pull-Down-Menüs oder mit erweiterten Emacs-Kommandos gewählt werden. Dazu gibt es ausführliche Hilfs-, Status-Reporting- und Browsing-Tools. Es besteht die Möglichkeit, Typspezifikationen (mit benutzerdefinierter Notation) unter LaTeX zu erstellen. Beweisbäume und die hierarchische Theoriestruktur können mit Tcl/Tk graphisch dargestellt werden.

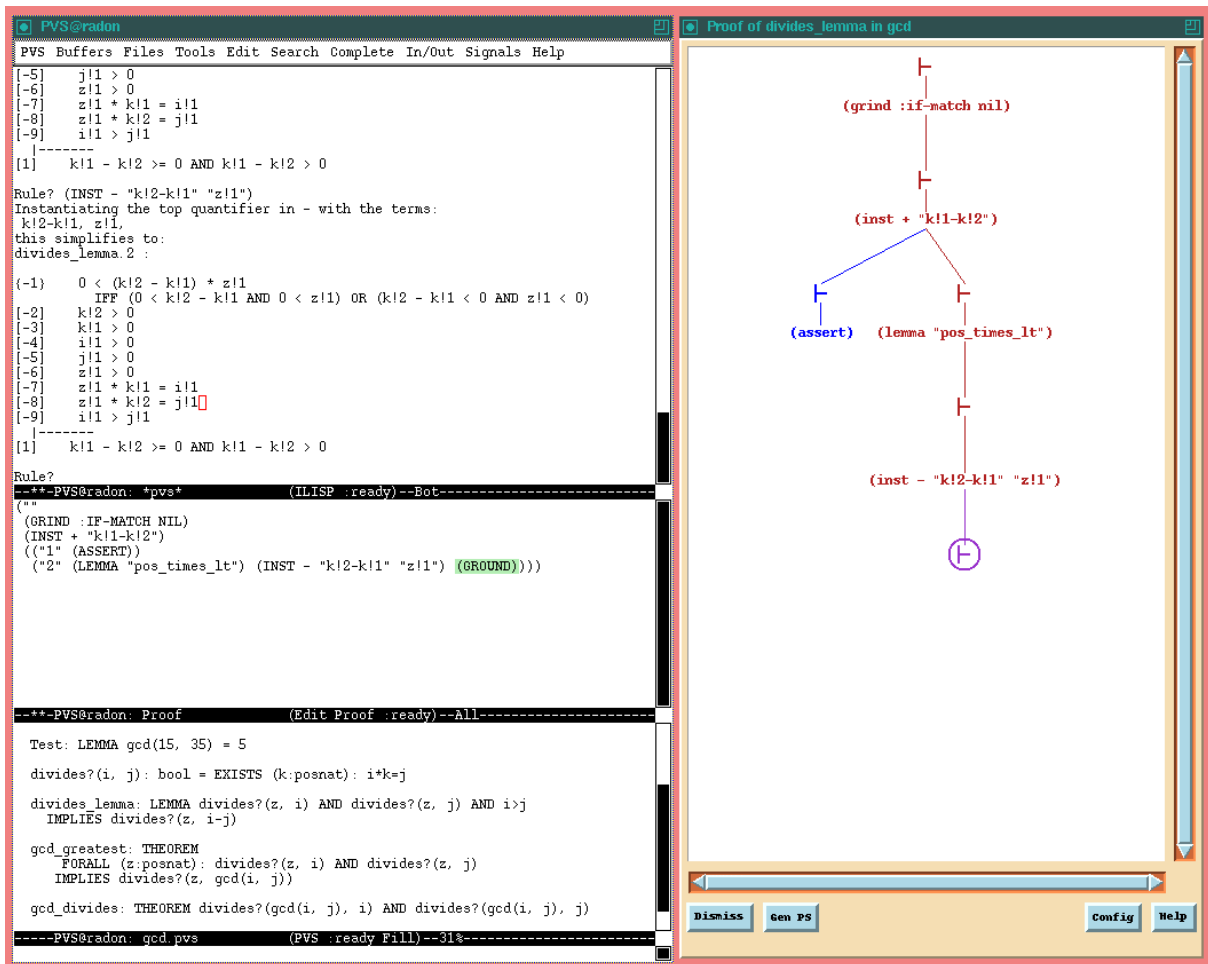
Unten steht ein Screenshot des Systems. Erläuterungen zum Bild:

PVS-Spezifikation - unten links,

Beweisverlauf - mitte links,

der Teil, der bearbeitet wird – oben links,

der Beweisbaum – rechts.



7. Der Beweis

Der Beweis wird zuerst allgemein (für beliebiger Radix) vorgestellt. Für ein gegebenes radix r , Dividend $p:\text{rational}$, Divisor $d:\text{posrat}$, wird bei jeder Iteration des radix- r -SRT-Algorithmus einen Quotient q und neuer partieller Rest $p_{\text{new}}:\text{rational}$ erzeugt. Man definiert eine Hilfsfunktion

```
recurrence?(p_new, p, q, d): bool = (p_new = r * (p - q * d))
rho: rational = a / (r - 1)
remainder_bound?(d, p): bool = (-r*rho <= p/d & p/d <= r*rho)
```

und die Zahlenmenge für Quotientwahl definiert durch

$$\text{subrange}(-a, a) \quad \text{mit} \quad a \geq \text{ceiling}(r/2)$$

Diese Werte müssen vorher spezifiziert (ersetzt) werden - in unserem Fall mit

$$r = 4 \qquad a = 2$$

Der Quotient wird bei jeder Iteration neu gewählt, und die Quotientensequenz wird gesammelt, wie von der rekursive Funktion `valq` bestimmt

```
valq(i, qq) : RECURSIVE rational =
  IF i = 0 THEN 0 ELSE qq(i-1)*1/r^(i-1) + valq(i-1,qq) ENDIF
MEASURE i
```

Hier ist `qq` vom Typ `subrange(-a, a)` und $i:\text{nat}$.

recurrence? und remainder_bound? (der letzte ist an der Partiellerrestliste gebunden pp:sequence[rational]), können korrekt die Fraktion p/d durch eine Sequenz von Quotienten $qq(0), \dots, qq(k)$ berechnen. Der Beweis unten benutzt Induktion für die Anzahl von Iterationen k und setzt Benutzerintervention und eine zusätzliche Lemma voraus.

```
approximation : THEOREM
  (FORALL i: recurrence?(pp(i+1),pp(i),qq(i),d)
    & remainder_bound?(d,pp(i)))
  IMPLIES abs(pp(0)/d - valq((k+1),qq)) <= 1/r^k*rho
```

Quotientenwahl: Eine schwere Aufgabe für jede Iteration ist die Wahl von q , so dass der nächste partielle Rest p_new die Bedingung remainder_bound? erfüllt. Deshalb werden die Lemma remainder_bound_equiv_legitimate und die Bedingung legitimate? eingeführt, die den Interval für den Quotienten charakterisiert, und seine Abhängigkeit von d benutzt.

```
legitimate?(q,d,p): bool = (q-rho <= p/d AND p/d <= q+rho)
```

```
remainder_bound_equiv_legitimate: LEMMA
  (FORALL (p_new,p,q,d): recurrence?(p_new,p,q,d)
    IMPLIES (remainder_bound?(d,p_new)
      IFF legitimate?(q,d,p))
```

und werden mit dem folgendem PVS-Kommando bewiesen (für alle reelle Zahlen)

```
(grind :theories "real_props")
```

Für bestimmte Paaren von (d,p) hängt die Wahl des Quotienten von ρ , deshalb können wir zeigen, dass mit dem gerundeten partiellen Rest $P:rational$ und dem gerundeten Divisor $D:posrat$ der Quotient sich berechnen lässt. Falls P unterschätzt p bzw. D unterschätzt d , man kann zeigen dass P an D gebunden ist.

```
delta ,eps:posrat
estimation_bound?(D, P): bool =
  (- eps - r * rho * (D + delta) < P & P < r * rho * (D + delta))
remainder_bound_implies_estimation_bound: LEMMA
  (P <= p & p < P + eps AND D <= d & d < D + delta AND
  remainder_bound?(d, p))
  IMPLIES estimation_bound?(D, P)
```

und die folgende Bedingung garantiert zusammen mit der Lemma die Quotientenwahl

```
lookup_legitimate?(q,D,(P:rational|estimation_bound?(D, P))):
bool =
COND
q = 2 -> 4/3 * (D + delta) <= P,
q = 1 -> 1/3 * (D + delta) <= P & (P + eps) <= 5/3 * D,
q = 0 -> -2/3 * D <= P & (P + eps) <= 2/3 * D,
q = -1 -> -5/3 * D <= P & (P + eps) <= -1/3 * (D + delta),
q = -2 -> (P + eps) <= -4/3 * (D + delta)
ENDCOND
```

```
lookup_is_legitimate: LEMMA
(P <= p & p < P + eps AND D <= d & d < D + delta AND
remainder_bound?(d, p) AND lookup_legitimate?(q, D, P))
IMPLIES legitimate?(q, d, p)
```

und die untenstehende Kommando beweist das Theorem

```
(grind :theories "real_props")
```

Die Implementierung dieser Quotientenwahlregel braucht nur Speicherung von q in einer Look-up-Tabelle für die verschiedenen Paaren (P, D) , so dass man die Bedingung $lookup_legitimate?(q, D, P)$ kontrollieren kann. Da D normalerweise begrenzt ist (die Werte, die es annehmen kann, sind endlich), ist auch die Lookup-Tabelle endlich.

Quotientenvorhersage: Durch die Berechnung des nachfolgenden partiellen Rests $pp(i+1)$ und „vorhersagen“ des Quotienten $qq(i+1)$ kann man parallel noch eine Stelle $pp(i+2)$ berechnen. Die Schätzung $PP(i)$ ist eine (Unter)Schätzung des nächsten partiellen Rests $pp(i+1)$. Wenn wir die Ergebnisse von oben kombinieren, können wir leicht die Invariante des SRT-Division-Algorithmus beweisen.

```
invariant: THEOREM
remainder_bound?(d, pp(0)) AND legitimate?(qq(0), d, pp(0)) AND
(FORALL j: recurrence?(pp(j + 1), pp(j), qq(j), d)
AND
PP(j) <= pp(j + 1) & pp(j + 1) < PP(j) + eps &
D <= d & d < D + delta AND
(estimation_bound?(D, PP(j))
IMPLIES lookup_legitimate?(qq(j + 1), D, PP(j))))
IMPLIES
remainder_bound?(d, pp(i)) AND legitimate?(qq(i), d, pp(i))
```

Das Theorem wird mit Induktion für i bewiesen, und für den nicht trivialen Fall in der Induktion werden die folgenden Implikationen durchgeführt

```
legitimate?(qq(i), d, pp(i))
->remainder_bound?(d, pp(i + 1))
->estimation_bound?(D, PP(i))
->lookup_legitimate?(qq(D, P(i)), D, PP(i))
->legitimate?(qq(i + 1), d, pp(i + 1))
```

Allgemein, um einen SRT-Division-Algorithmus zu beweisen, braucht man zu zeigen, dass:

- die arithmetische Interpretationen der beiden Sequenzen (partieller Rest und Quotienten) die Relation `recurrence?` erfüllen,
- es gibt Konstanten `delta` und `eps`, für die der Divisor und die partiellen Reste gebunden werden können,
- die Logik der Quotientenwahl erfüllt den Prädikat `lookup_legitimate?`.

Wenn diese Bedingungen zutreffen, können wir die Theoreme `invariant` und `approximation` anwenden.

Die Implementation der Lookup-Tabelle kann besonders kompakt sein. Dazu braucht man den Konstruktor TABLE. Die Tabelle enthält aber nicht alle Werte (denn es müssen eigentlich 1066 Einträge geben). Der Beweiser teilt P in 5-bit und D in 3-bit mögliche Werte und man bekommt insgesamt 256 Fälle, die zu beweisen sind ($2^5+2^3=256$).

Und es gilt auch das folgende Theorem.

(es testet für alle D, P -Paaren, ob die Kondition lookup_legitimate? zutrifft)

```
(FORALL (D,(P: bvec[7] |
  estimation_bound?(valD(D), valP(P)))):
  lookup_legitimate?(q(D, P), valD(D), valP(P)))

q(D:bvec[3],
  (P: bvec[7] | estimation_bound?(valD(D),valP(P)))):
  subrange(-2, 2) =
LET  a = -(2 - P(1) * P(0)), b = -(2 - P(1)), c = 1 + P(1),
     d = -(1 - P(1)), e = P(1),
Dp:nat = bv2pattern(D),
Ptruncp:nat = bv2pattern(P^(6,2))
IN TABLE Ptruncp, Dp
  |[ 000| 001| 010| 011| 100| 101| 110| 111]|
%-----%
|01010|      |      |      |      |      |      |      |      |      |      |
|01001|      |      |      |      |      |      |      |      |      |      |
|01000|      |      |      |      |      |      |      |      |      |      |
|00111|      |      |      |      |      |      |      |      |      |      |
|00110|      |      |      |      |      |      |      |      |      |      |
|00101|      |      |      |      |      |      |      |      |      |      |
|00100|      |      |      |      |      |      |      |      |      |      |
|00011|      |      |      |      |      |      |      |      |      |      |
|00010|      |      |      |      |      |      |      |      |      |      |
|00001|      |      |      |      |      |      |      |      |      |      |
|00000|      |      |      |      |      |      |      |      |      |      |
|11111|      |      |      |      |      |      |      |      |      |      |
|11110|      |      |      |      |      |      |      |      |      |      |
|11101|      |      |      |      |      |      |      |      |      |      |
|11100|      |      |      |      |      |      |      |      |      |      |
|11011|      |      |      |      |      |      |      |      |      |      |
|11010|      |      |      |      |      |      |      |      |      |      |
|11001|      |      |      |      |      |      |      |      |      |      |
|11000|      |      |      |      |      |      |      |      |      |      |
|10111|      |      |      |      |      |      |      |      |      |      |
|10110|      |      |      |      |      |      |      |      |      |      |
|10101|      |      |      |      |      |      |      |      |      |      |
%-----%
ENDTABLE
```

Man kann sich auch $P = g7g6g5g4.g3g2g1$ und $D = f1.f2f3f4$ vorstellen, dann sehen die Tabelleneinträge so aus - $g7g6g5g4.g3$ und $.f2f3f4$.

Die Tabelle lässt sich so komprimieren, weil nur wenige Werte von $g2g1$ abhängen – für diese Werte werden die symbolische Werte „a“, „b“, „c“, „d“, „e“ eingeführt.

$$/a = -(2 - g2 * g1) ; b = -(2 - g2) ; c = 1 + g2 ; d = -1 + g2 ; e = g2 /$$

III. Zusammenfassung

PVS-Beweise sind sicher, sie können durch leichte Modifikationen wieder in anderen Beweisen benutzt werden. Die Verifikation oben zeigt eine praktische Anwendung des Systems. Formale Methoden werden immer beliebter, da auch die neuen Hard- und Softwaresysteme komplexer werden. Ohne formale Methoden würde bald die menschliche Grenze bei der Verifikation erreicht. Man arbeitet an einer weiteren Entwicklung der Implementation des Systems, damit alles noch schneller funktioniert (der Tabellen-Beweis oben z.B. ca. 3 Stunden braucht).

Das System ist hauptsächlich für Formalisation von High-Level-Spezifikationen und für die Analyse von Problemen gedacht. Es wurde auf Algorithmen und Architekturen von Flug-Steuerung-Systemen, Problemen im Hardwaregebiet und real-time-Systemdesign angewendet. Es wird heute von verschiedenen Luft- und Raumfahrtfirmen (auch von NASA) für Mikroprozessoranwendungen, Diagnostik von Algorithmen für fault-tolerant-Architekturen benutzt (z.B. bei Space Shuttle flight-control system). Formale Methoden werden auch für Korrektheitsbeweis des Verhaltens von Schaltungen benutzt. Die heutige Entwicklung des Systems visiert die Anwendung im Gebiet der concurrent und real-time-Systemen.

Was heute der Einsatz formaler Methoden noch schwer macht – die Definition und Anwendung von passenden Sprachen, Formulierung von Eigenschaften, Motivation der Designers zum Gebrauch von diesen neuen Methoden.

IV. Quellenangabe:

<http://www4.in.tum.de/~isabelle> - Isabelle-Home

<http://www.inf.ethz.ch/~biere/talks/kakol2001.pdf> - Formale Methoden zur Lösung von Komplexitäts- und Qualitätsproblemen, State-of-the-Art und Ausblick, Informatik Kolloquium, Universität Karlsruhe, Prof. Dr. Armin Biere, ETH Zürich

<http://pvs.csl.sri.com> - PVS homepage

<http://www.csl.sri.com/projects/pvs/> - PVS-Project

<http://www.cafm.sbu.ac.uk/> - Centre for Applied Formal Methods, South Bank University, London

- [AC95] Parallel Scientific Computing, Lecture 3: The Pentium bug 1995, Anthony Caola, MIT
- [Pra95] “Anatomy of the Pentium Bug”, CA 94305-2140, Vaughan Pratt, Stanford, 11 June 1995
- [WPI] <http://www.intel.com/support/processors/pentium/fdiv/> - Official Flaw White Papers, Intel Corp.
- [RB95] “Bit-level Analysis of an SRT Divider Circuit”, Technical Report: CMU-CS-95-140, Randal E. Bryant, Carnegie Mellon University
- [MV] “Mechanizing Verification of Arithmetic Circuits: SRT Division” Deepak Kapur (Comp. Science Department, State University of New York), M. Subramaniam (Functional Verification Group, Silicon Graphics Inc.)
- [EL] „Gerichtete Modellprüfung mit Explorationsverfahren der Künstliche Intelligenz“, Stefan Edelkamp, Alberto Lluch-Lafuente, Universität Freiburg
- [ZRM] “The Z Notation: A Reference Manual”, Second Edition, by J.M. Spivey, Programming Research Group, University of Oxford, auch im Internet unter <http://spivey.orient.ox.ac.uk/~mike/zrm/zrm.pdf>
- [SG] “PVS System Guide”, Computer Science Laboratory, SRI International
- [LR] “PVS Language Reference”, Computer Science Laboratory, SRI International
- [PG] “PVS Prover Guide”, Computer Science Laboratory, SRI International
- [BDD] “Binäre Entscheidungsdiagramme”, Prof. Ulrich Herzog, aus dem Skript zur Vorlesung “Organisation und Technologie von Rechensystemen I”, Universität Erlangen