

Kapitel 2

Software-Fehler

2.1 Lexikalische und syntaktische Fehlerquellen

Sehen wir von den Methoden der grafischen Programmierung ab, so besteht ein typisches Programm auf der lexikalischen Ebene aus einer Aneinanderreihung von Symbolen (*tokens*). Neben den Schlüsselwörtern umfassen die Tokens einer Programmiersprache sämtliche Operatoren und Bezeichner. Die Zerlegung des Zeichenstroms in einzelne Symbole ist Inhalt der *lexikalischen Analyse* – eine heute wohlverstandene Standardaufgabe im Bereich des Compiler-Baus.

Damit ein Programm erfolgreich übersetzt oder korrekt interpretiert werden kann, muss der gelesene Symbolstrom gewissen Regeln genügen, die zusammen die *Grammatik* einer Programmiersprache bilden. Mit anderen Worten: Die Grammatik legt die *Syntax* einer Programmiersprache fest. In den allermeisten Fällen führen Programmfehler auf der lexikalischen Ebene zu einer Verletzung der zugrunde liegenden Grammatikregeln und dadurch entweder zu einer Fehlermeldung während der Übersetzung oder zu einem Laufzeitfehler, falls es sich um eine interpretierte Sprache handelt. Vereinzelt lassen lexikalische Fehler jedoch Programme entstehen, die sich problemlos übersetzen bzw. interpretieren lassen. Die Gründe für das Auftreten solcher Fehler sind vielfältig und reichen von einfachen Tippfehlern bis hin zu mangelndem Verständnis der Sprachsyntax auf Seiten des Programmierers.

Ein Beispiel einer Programmiersprache, die das Auftreten lexikalischer Fehler aufgrund ihrer Struktur und Syntax fast schon provoziert, ist die Programmiersprache C. Obwohl die Sprache bereits Anfang der Siebzigerjahre entwickelt wurde, hat sie heute immer noch eine enorme Praxisbedeutung. Zum einen besteht die Sprache nahezu unverändert als Untermenge in den objektorientierten Erweiterungen C++, C# und Objective-C fort, zum anderen ist C unangefochten die am häufigsten verwendete Sprache im Bereich eingebetteter Systeme und der betriebssystemnahen Programmierung.

Um einen ersten Einblick in die lexikalischen Schwierigkeiten dieser Sprache zu erhalten, betrachten wir die folgende C-Anweisung:

```
c = a---b;
```

Wie wird diese Anweisung durch den Compiler interpretiert? Selbst langjährige C-Programmierer parieren diese Frage nicht immer mit einer schnellen Antwort und greifen voreilig zur Tastatur. Da die Programmiersprache C sowohl den Subtraktionsoperator („-“) als auch den Dekrementierungsoperator („--“) kennt, stehen die folgenden beiden Alternativen zur Auswahl:

```
c = (a--) - b;
c = a - (--b);
```

Glücklicherweise gibt es in C eine einzige klare Regel, mit deren Hilfe sich die Frage eindeutig beantworten lässt. Andrew Koenig spricht in [154] bezeichnend von der *Maximal munch strategy*, hinter der sich nichts anderes als die in der Informatik häufig verwendete *Greedy-Methode* verbirgt [58]. Koenig beschreibt die Regel in treffender Weise wie folgt:

„Repeatedly bite off the biggest piece.“

Mit anderen Worten: Die einzelnen Tokens werden während der Extraktion aus dem Zeichenstrom stets so groß wie irgend möglich gewählt. Abb. 2.1 zeigt den resultierenden Token-Stream, den die Greedy-Strategie für das klassische Hello-World-Programm produziert. Jetzt wird auch auf einen Schlag klar, wie der Ausdruck `c = a---b`; ausgewertet wird. Durch Anwendung der Greedy-Regel wird aus dem Zeichenstrom `---` zunächst das Token `--` und anschließend das Token `-` extrahiert. Der Ausdruck `c = a---b`; ist damit äquivalent zu `c = (a--) - b`;. Da `c = a - (--b)`; ebenfalls ein gültiger C-Ausdruck ist, schleichen sich durch die Missachtung oder die falsche Auslegung der lexikalischen Regeln Fehler ein, die durch den C-Compiler nicht erkannt werden können.

Die nächsten drei C-Anweisungen sind weitere Beispiele von Ausdrücken, die einen handfesten lexikalischen Fehler oder zumindest eine potenzielle Fehlerquelle enthalten (vgl. [154]):

■ Beispiel 1

```
x=-1; /* Initialisiere x mit dem Wert -1 */
```

Alle moderneren C-Compiler verarbeiten diesen Ausdruck korrekt. Zwar kennt die Programmiersprache C den dekrementierenden Zuweisungsoperator `--=`, eine Verwechslung ist aufgrund der unterschiedlichen Position des Minuszeichens aber ausgeschlossen. Ein Programmierer mag sein blaues Wunder trotzdem erleben, wenn er das Programm mit einem älteren C-Compiler oder so manch angestaubtem Compiler der Embedded-Welt übersetzt. In den frühen Tagen der Programmiersprache C enthielt die Sprachdefinition den Operator `=-`, mit exakt derselben Semantik des heute verwendeten ANSI-C-Operators `--=`. In diesem Fall wird die Zeichenkombination `=-` innerhalb des Ausdrucks `x=-1` als Operator interpretiert und die Variable `x` nicht länger mit dem Wert `-1` initialisiert. Stattdessen wird die zu diesem Zeitpunkt potenziell undefinierte Variable `x` schlicht dekrementiert. Folgerichtig verbessert eine saubere syntaktische Aufbereitung,

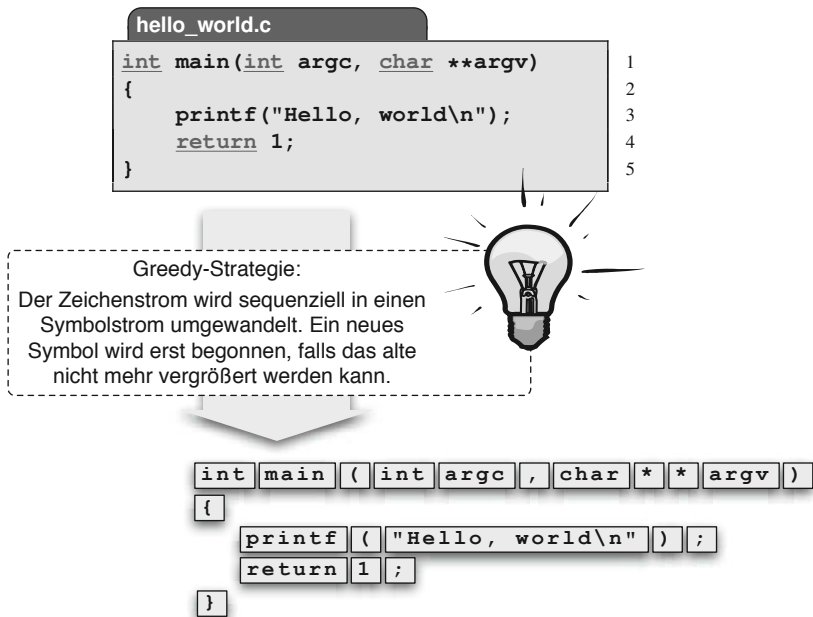


Abb. 2.1 Lexikalische Programmanalyse in der Programmiersprache C

wie sie z. B. die Separierung der einzelnen Elemente durch Leerzeichen leistet, nicht nur die Lesbarkeit, sondern auch die Robustheit des Programmtextes.

Die Verwendung von Leerzeichen ist eine sichere Methode, um in den meisten Programmiersprachen die verschiedenen Tokens voneinander zu trennen. Dass dies nicht in allen Sprachen der Fall ist, wird uns der weiter unten diskutierte und wahrscheinlich prominenteste Software-Fehler der Programmiersprache FORTRAN demonstrieren.

■ Beispiel 2

```

struct { int vorwahl; char *city; }
phone_book[] = { 07071, "Tübingen",
                 0721,  "Karlsruhe",
                 089,   "München" };

```

Das Code-Fragment definiert ein dreielementiges Array `phone_book`. Jedes Element des Arrays ist ein Tupel, das eine Telefonvorwahl mit dem Namen der zugehörigen Stadt assoziiert. Das Beispiel demonstriert einen der vielleicht unglücklichsten lexikalischen Aspekte der Programmiersprache C, der sich aus Gründen der Rückwärtskompatibilität in das einundzwanzigste Jahrhundert hinüberretten konnte und wahrscheinlich auch zukünftige Entwicklergenerationen vor das eine oder andere Rätsel stellen wird.

Erkennen Sie bereits den Fehler? In C wird jede numerische Zeichenfolge, die mit der Ziffer **0** beginnt, als Oktalzahl (Basis 8) gedeutet. Die Zeichenfolge **0721** entspricht damit nicht der Dezimalzahl 721. Stattdessen wird der Ausdruck wie folgt interpretiert:

$$0721 = 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 = 448 + 16 + 1 = 465 \quad (2.1)$$

Selbst der Struktureintrag **089** führt nicht in jedem Fall zu einer Fehlermeldung, obwohl die Ziffern 8 und 9 außerhalb des oktalen Zahlenbereichs liegen. Einige C-Compiler sehen über die Bereichsüberschreitung großzügig hinweg und interpretieren die Zahl folgendermaßen:

$$089 = 8 \times 8^1 + 9 \times 8^0 = 64 + 9 = 73 \quad (2.2)$$

■ Beispiel 3

```
y = x/*p /* p ist Pointer auf Divisor */
```

Wie der Kommentar unmissverständlich andeutet, möchte der Programmierer dieses Beispiels den Wert von **x** zunächst durch den an der Speicherstelle **p** abgelegten Wert dividieren und das Ergebnis anschließend der Variablen **y** zuweisen. Die Speicherstelle wird durch die Dereferenzierung von **p** mit Hilfe des Sternoperators ***** ausgelesen. Unglücklicherweise ist die Kombination **/*** selbst ein Schlüsselwort in C und leitet einen Kommentarblock ein. Nach jedem Auftreten von **/*** ignoriert der lexikalische Parser alle weiteren Zeichen bis zur Endemarkierung ***/**. Da die allermeisten C-Compiler in der Standardeinstellung keine verschachtelten Kommentare berücksichtigen, führt der obige Programmcode nicht zwangsläufig zu einer Fehlermeldung.

Wie schon im ersten Beispiel kann das Problem auf einfache Weise behoben werden. Durch das schlichte Einfügen von Leerzeichen oder das Klammern von Teilausdrücken wird der Token-Strom korrekt aufgelöst:

```
y = x / (*p) /* p ist Pointer auf Divisor */
```

Das in Abb. 2.2 dargestellte Beispiel demonstriert die potenziellen Auswirkungen lexikalischer Fehler auf eindringliche Weise und zeigt zugleich, wie schwer derartige Defekte zuweilen zu erkennen sind. Das Code-Fragment ist Teil einer Funktion zur sequenziellen Suche eines Bezeichners innerhalb einer Symboltabelle und stammt aus der Implementierung eines ANSI-C-Compilers [164]. Im ersten Abschnitt des Programms wird zunächst für die Variable **hashval** ein Hash-Wert ermittelt, der die *wahrscheinliche* Position des Symbols enthält. Von diesem Wert ausgehend, wird die genaue Position des Symbols anschließend durch eine einfache sequenzielle Suche bestimmt.

Ein geschärfter Blick auf den Programmtext zeigt, dass der erste Kommentarblock nicht beendet wurde. Der C-Parser hat keinerlei Chance, den Fehler zu erkennen und ignoriert alle Anweisungen bis zum Ende des zweiten Kommentarblocks.

```
/* PJW hash function from
 * "Compilers: Principles, Techniques, and Tools"
 * by Aho, Sethi, and Ullman, Second Edition.
while (cp < bound)
{
    unsigned long overflow;

    hashval = (hashval << 4) + *cp++;
    if ((overflow=hashval & ((unsigned long)0xF<< 28))!=0)
        hashval ^= overflow | (overflow >> 24);
}
hashval %= ST_HASHSIZE;          /* choose start bucket */

/* Look through each table, in turn, for the name.
 * If we fail, save the string, enter the string's pointer,
 * and return it.
 */
for (hp = &st_ishash; ; hp = hp->st_hnext) {
    int probeval = hashval;      /* next probe value */
```

Abb. 2.2 Ausschnitt aus dem Quelltext eines ANSI-C-Compilers

Folgerichtig wird der gesamte Initialisierungscode dem ersten Kommentarblock zugeschlagen und nicht ausgeführt. Hierdurch beginnt die sequenzielle Suche stets bei 0 – mit dramatischen Geschwindigkeitseinbußen im Falle großer Symboltabellen.

Der demonstrierte Fehler ist in zweierlei Hinsicht der Alptraum eines jeden Entwicklers. Zum einen äußert sich der Fehler ausschließlich in einer Laufzeitverschlechterung – aus funktionaler Sicht ist das Programm vollkommen korrekt. Die meisten Testfälle, die von Software-Entwicklern in der Praxis geschrieben werden, führen jedoch ausschließlich den Nachweis der funktionalen Korrektheit. Nur wenige Testfälle befassen sich mit der Überprüfung der Laufzeitanforderungen. Zum anderen tritt der Fehler nur für Symboltabellen mit einer Größe zum Vorschein, die in typischen Testfällen nicht erreicht wird. Insgesamt ist die Wahrscheinlichkeit damit außerordentlich hoch, dass der Defekt für lange Zeit unentdeckt in den Programmquellen verweilt.

Wäre dieser Fehler vermeidbar gewesen? Die Antwort ist ein klares Ja. Der abgebildete Programmcode stammt aus einer Zeit, in der Quelltexte mit vergleichsweise primitiven Texteditoren eingegeben wurden und lexikalische Fehler dieser Art nur schwer zu erkennen waren. Obwohl wir auch heute nicht gegen Tippfehler gefeit sind, kann der Fehler durch die Verwendung einer handelsüblichen integrierten Entwicklungsumgebung (IDE) bereits während der Eingabe aufgedeckt werden. Moderne Editoren rücken Programmcode automatisch ein und stellen die verschiedenen Sprachkonstrukte in verschiedenen Farben dar. Jeder unbeabsichtigt auskommentierte Code-Block ist dadurch mit bloßem Auge zu erkennen. Das Beispiel unterstreicht erneut, dass die saubere Aufbereitung des Programmcodes mehr ist als reine Ästhetik.

<code>IF (TVAL .LT. 0.2E-2) GOTO 40</code>	1
<code>DO 40 M = 1, 3</code>	2
<code>W0 = (M-1)*0.5</code>	3
<code>X = H*1.74533E-2*W0</code>	4
<code>DO 20 N0 = 1, 8</code>	5
<code>EPS = 5.0*10.0**(N0-7)</code>	6
<code>CALL BESJ(X, 0, B0, EPS, IER)</code>	7
<code>IF (IER .EQ. 0) GOTO 10</code>	8
20 <code>CONTINUE</code>	9
<code>DO 5 K = 1. 3</code>	10
<code>T(K) = W0</code>	11
<code>Z = 1.0/(X**2)*B1**</code>	12
<code>2+3.0977E-4*B0**2</code>	13
<code>D(K) = 3.076E-2*2.0*</code>	14
<code>(1.0/X*B0*B1+3.0977E-4*</code>	15
<code>* (B0**2-X*B0*B1))/Z</code>	16
<code>E(K) = H**2*93.2943*W0/SIN(W0)*Z</code>	17
<code>H = D(K)-E(K)</code>	18
5 <code>CONTINUE</code>	19
10 <code>CONTINUE Y = H/W0-1</code>	20
40 <code>CONTINUE</code>	21

Abb. 2.3 Von der NASA im Rahmen des Mercury-Projekts eingesetzter FORTRAN-Code

Der vielleicht berühmteste lexikalische Fehler der Computergeschichte ist in Abb. 2.3 dargestellt. Der abgebildete FORTRAN-Code entstammt dem Mercury-Projekt, das von der NASA 1958 ins Leben gerufen und 1963 erfolgreich beendet wurde. Mit diesem Projekt wagte die NASA erstmals den Schritt in die bemannte Raumfahrt und legte die Grundlagen für die historische Mondlandung von Apollo 11 am 20. Juli 1969.

Der in Abb. 2.3 dargestellte Code-Abschnitt ist Teil eines Programms zur Berechnung von Orbitalbahnen und wurde für diverse Mercury-Flüge erfolgreich eingesetzt. Erst 1961 entdeckte ein Programmierer der NASA Ungenauigkeiten in der Berechnung. Der Code lieferte zwar annähernd gute Ergebnisse, die Genauigkeit blieb aber hinter der theoretisch zu erwartenden zurück.

Im Rahmen einer genauen Analyse konnte die Ungenauigkeit auf einen simplen lexikalischen Fehler zurückgeführt werden. Der fehlerhafte Programmcode verbirgt sich unterhalb von Zeile 20 in der `DO`-Anweisung – dem Schleifenkonstrukt in FORTRAN:

```
DO 5 K = 1. 3
```

Die allgemeine Form der Anweisung lautet wie folgt:

```
DO [ label ] [ , ] var = first, last [ ,inc]
```

Auf das Schlüsselwort `DO` folgt in FORTRAN ein optionaler Bezeichner (`label`), der das Schleifenende definiert. Fehlt dieses Argument, so muss der Schleifenkörper

mit dem Schlüsselwort **END DO** abgeschlossen werden. **var** bezeichnet die Schleifenvariable. Diese wird zu Beginn mit dem Wert **first** initialisiert und nach jeder Schleifeniteration um den Wert **inc** erhöht. Fehlt die Angabe der Schrittweite, so wird auf **var** nach jeder Iteration eine 1 addiert. Die Schleife terminiert, wenn die Zählvariable einen Wert größer als **last** erreicht.

Ein genauer Blick auf das Schleifenkonstrukt in Zeile 20 zeigt, dass die Intervallgrenzen nicht mit einem Komma, sondern mit einem Punkt separiert wurden. Dieser mit bloßem Auge kaum zu entdeckende Fehler hat dramatische Auswirkungen. Durch das Fehlen des Kommas erkennt FORTRAN den Ausdruck nicht mehr länger als Zählschleife, sondern interpretiert den Ausdruck schlicht als einfache Variablenzuweisung:

```
DO5K = 1.3
```

Dass der Fehler durch den FORTRAN-Compiler nicht als solcher erkannt wird, liegt an zwei Besonderheiten der Sprache, die sie von den meisten anderen Programmiersprachen unterscheidet. Zum einen erlaubten frühe FORTRAN-Versionen, Leerzeichen an beliebiger Stelle einzufügen – insbesondere auch innerhalb von Bezeichnern und Variablennamen. Diese Eigenschaft erscheint aus heutiger Sicht mehr als fahrlässig. Zur damaligen Zeit bot sie jedoch durchaus Vorteile, da die ersten FORTRAN-Programme noch auf Lochkarten gespeichert wurden. Werden alle Leerzeichen ignoriert, kann ein Programm selbst dann noch erfolgreich eingelesen werden, wenn zwischen zwei gestanzten Zeilen versehentlich eine ungestanzte übrig bleibt.

Zum anderen ist es in FORTRAN gar nicht nötig, Variablen vor ihrer ersten Nutzung zu deklarieren. Hier wird besonders deutlich, wie wertvoll die Bekanntmachung von Funktionen und Variablen in der Praxis wirklich ist. Dass eine Variable **DO5K** bereits an anderer Stelle deklariert wurde, ist so unwahrscheinlich, dass jeder Compiler die Übersetzung der mutmaßlichen Zuweisung verweigert hätte. Kurzum: Die Verwendung einer deklarationsbasierten Programmiersprache, wie z. B. C, C++ oder Java, hätte den Software-Fehler des Mercury-Projekts vermieden – der Fehler wäre bereits zur Übersetzungszeit durch den Compiler entdeckt worden.

Es bleibt die Frage zu klären, wie der hier vorgestellte FORTRAN-Bug eine derart große Berühmtheit erlangen konnte, um heute zu den meistzitierten Software-Fehlern der IT-Geschichte zu zählen? Die Antwort darauf ist simpel. Der vorgestellte Fehler demonstriert nicht nur wie kaum ein anderer die Limitierungen der Sprache FORTRAN, sondern zeichnet zugleich für eine der größten Legenden der Computergeschichte verantwortlich. Auf zahllosen Internet-Seiten, wie auch in der gedruckten Literatur, wird der FORTRAN-Bug beharrlich als Ursache für den Absturz der Raumsonde Mariner I gehandelt, die am 22. Juli 1962 von der NASA an Bord einer Atlas-Trägerrakete auf den Weg zur Venus gebracht werden sollte. Kurz nach dem Start führte die Trägerrakete unerwartet abrupte Kursmanöver durch und wich deutlich von der vorbestimmten Flugbahn ab. Alle Versuche, korrigierend einzugreifen, schlugen fehl. Nach 290 Sekunden fällt die Flugkontrolle schließlich die Entscheidung, die Trägerrakete aus Sicherheitsgründen zu sprengen.

Beispiel 1

```
float nrm(float x)
{
    while (abs(x)>0,1)
        x = x / 10;
    return x;
}
```

1
2
3
4
5
6
7



Programm
terminiert
nicht

Beispiel 2

```
int abs(int x)
{
    if (x < 0)
        x = -x,
    return x;
}
```

1
2
3
4
5
6
7



Diesen Fehler
erkennt der
Compiler

Beispiel 3

```
void fill()
{
    int i=0, a[10];

    for (; i<=10; i++)
        a[i] = 0;
}
```

1
2
3
4
5
6
7



Programm-
verhalten ist
unterschiedlich

Abb. 2.4 Ist C ein Fortschritt gegenüber FORTRAN?

Die Ursache für den Absturz der Mariner-Trägerrakete ist weit unspektakulärer als gemeinhin angenommen und geht schlicht auf die falsche Umsetzung der Spezifikation zurück. Obwohl die Anforderungsbeschreibung der Flugsteuerung korrekt vorgab, die Verlaufskurve eines Messwerts geglättet zu verwenden, wurde diese in der Implementierung ungeglättet weiterverarbeitet. Trotzdem: Der FORTRAN-Bug der Mercury-Mission wird heute immer noch so beständig mit dem Mariner-Absturz in Verbindung gebracht, dass er wahrscheinlich auch in Zukunft als spektakuläre Erklärung für das Scheitern dieser Mission herhalten muss.

An dieser Stelle mag sich so mancher Leser zu dem Gedanken hinreißen lassen, dass der Fehler schlicht auf die Eigenarten einer fast schon prähistorischen Programmiersprache zurückgeht – insbesondere haben wir weiter oben herausgearbeitet, dass der Fehler in Sprachen wie C oder C++ nicht unentdeckt geblieben wäre. Doch stellen C bzw. C++ wirklich den erhofften Fortschritt in dieser Richtung dar? Dass auch in diesen Sprachen der Teufel im Detail steckt, demonstrieren die drei in Abb. 2.4 aufgeführten Beispielprogramme.

Das erste Beispielprogramm definiert die Funktion **nrm**. Als einzigen Übergabeparameter nimmt die Funktion eine **float**-Variable entgegen und dividiert den erhaltenen Wert so lange durch 10, bis das Ergebnis innerhalb des Intervalls $[-0,1; 0,1]$ liegt. Unabhängig von der übergebenen Zahl verweigert das Programm in der abgedruckten Form beharrlich seinen Dienst und verfängt sich stets auf's Neue in einer Endlosschleife.

Genau wie im Fall des oben geschilderten FORTRAN-Bugs reduziert sich die Ursache auf die Vertauschung von Punkt und Komma – anstelle eines Punkts wurde in der Intervallgrenze zur Trennung der Ziffern ein Komma eingefügt. Da die Programmiersprache C das Komma als eigenständigen Operator kennt, führt die Vertauschung zu keinem Fehler.

Mit Hilfe des Komma-Operators werden in C mehrere Anweisungen verkettet, so dass der Wert von **x** zunächst mit 0 verglichen und anschließend der Ausdruck 1 ausgewertet wird. Da die Konstante 1 in C dem Wahrheitswert True entspricht und der Wert des letzten ausgewerteten Teilausdrucks gleichzeitig dem Wert des Gesamtausdrucks, ist die Schleifenbedingung permanent erfüllt. Kurzum: Eine Endlosschleife entsteht.

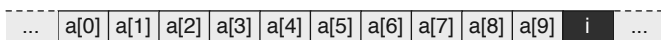
Das zweite Beispielprogramm definiert die Funktion **abs**, die den Absolutwert des vorzeichenbehafteten Übergabeparameters berechnet. Hierzu wird der Wert des **int**-Arguments **x** mit 0 verglichen und gegebenenfalls mit Hilfe des Negationsoperators in eine positive Zahl gewandelt. Anschließend wird der Inhalt von **x** an die aufrufende Funktion zurückgegeben. Ein genauerer Blick auf die Zuweisung innerhalb des If-Körpers zeigt, dass an dieser Stelle ein Komma anstelle des nötigen Semikolons eingetippt wurde. Hierdurch müsste die Return-Instruktion ungewollt dem Körper der If-Anweisung zugeordnet werden und für $x \geq 0$ ein undefinierter Rückgabewert entstehen. Anders als im ersten Beispiel wird dieser Fehler aber durch den Compiler erkannt. Der Grund hierfür ist einfach: Um Probleme dieser Art zu vermeiden, verbietet die C-Grammatik, die Return-Instruktion in Kombination mit dem Komma-Operator zu verwenden.

Das dritte Beispielprogramm definiert die parameterlose Prozedur **fill**, die das zehnelementige Array **a** und die Laufvariable **i** lokal deklariert. Innerhalb der Funktion werden alle Elemente des Arrays mit Hilfe einer For-Schleife mit 0 initialisiert. Das Programm enthält einen gravierenden Fehler, der sich in Abhängigkeit des verwendeten Compilers unterschiedlich auswirkt und eine Endlosschleife verursachen kann. Die Ursache für das Fehlverhalten geht auf die Schleifenendbedingung **i<=10** zurück. Da das erste Array-Element in C immer den Index 0 besitzt, wird die Schleife nicht zehn, sondern elfmal ausgeführt. Die elfte Iteration führt dazu, dass die zugewiesene 0 in den Speicher außerhalb des Arrays geschrieben wird.

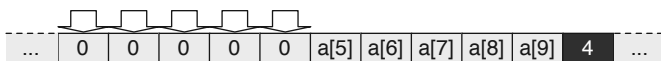
Soweit so gut. Aber warum kann das Programm zu einer Endlosschleife führen? Der Grund hierfür liegt in der Speicheranordnung lokaler Variablen. Die meisten älteren C-Compiler ordnen die Variablen direkt hintereinander in absteigender Reihenfolge an und erzeugen die in Abb. 2.5 dargestellte Anordnung.

Das Speicherabbild deckt auf, warum in diesem Fall eine Endlosschleife entsteht. Die letzte Zuweisung (**a[10]=0**) schreibt eine Null an diejenige Speicherstelle, in der normalerweise die Variable **i** gespeichert wird. Konsequenterweise wird **i** vor

■ Speicherabbild vor Schleifeneintritt



■ Speicherabbild innerhalb der Schleife



■ Speicherabbild nach der letzten Iteration



Abb. 2.5 Speicherabbild auf dem Stack. Die meisten Compiler ordnen lokale Variablen in absteigender Speicherrichtung an, so dass die Variablen **a[10]** und **i** in diesem Beispiel dieselbe Speicheradresse referenzieren

dem Erreichen der Abbruchbedingung stets auf 0 zurückgesetzt und die komplette Schleife hierdurch neu gestartet.

Tatsächlich ist die Chance, mit dem abgebildeten Programm unter realen Bedingungen eine Endlosschleife zu erzeugen, mittlerweile gering. Dies liegt daran, dass fast alle modernen Compiler die Stack-Elemente nicht mehr lückenlos anordnen, sondern zusätzliche Platzhalter einfügen, um die Vorhersage des Speicherabbaus zu verhindern. Auf diese Weise wird es schwerer, eine Software durch gezielt herbeigeführte Pufferüberläufe anzugreifen; gleichsam führt der Mechanismus dazu, dass Fehler wie der geschilderte, heute in vielen Fällen unbemerkt bleiben.

2.2 Semantische Fehlerquellen

Während die *Syntax* einer Programmiersprache den textuellen Aufbau der Quelltexte beschreibt, beschäftigt sich die *Semantik* mit der Bedeutung der einzelnen Konstrukte und damit mit der Interpretation der Sprachbausteine durch den Compiler. Im Gegensatz zu Syntaxfehlern, deren Ursachen in den meisten Fällen eher einfacher Natur sind, erweisen sich semantische Fehler häufig als deutlich tiefgründiger. Entsprechend schwierig gestaltet sich in der Praxis deren Behebung. Auf der positiven Seite lassen sich viele Semantikfehler durch ein gewisses Maß an Programmierdisziplin im Ansatz vermeiden bzw. durch geeignete Analysetechniken nachträglich erkennen.

Wie weitreichend die Folgen solcher Fehler sein können, erfuhr die Bevölkerung der Vereinigten Staaten von Amerika am Montag, den 15. Januar 1990 am eigenen Leib. Das Unheil begann um 14:30 mit einem Ausfall der Schaltzentrale des AT&T-Telefonnetzes in Manhattan. Das AT&T-Telefonsystem wird durch 114 regionale Schaltzentralen gebildet, die untereinander vernetzt sind und von der Zentralstelle in New Jersey koordiniert werden. Das System ist so konzipiert, dass im Falle eines Ausfalls eines Vermittlungsknotens mehrere *Out-of-Service-Nachrichten* an

```
at_and_t.c
...
switch (line) {
...
    case THING1:
        doit1();
        break;

    case THING2:
        if (x == FOO) {
            do_first_stuff();

            if (y == BAR)
                /* Skip "do_later_stuff" function call
                 by dropping out of the If-Statement */
                break;

            do_later_stuff();
        }
        initialize();
        break;

    default:
        processing();
}
...
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

Abb. 2.6 Code-Struktur der Vermittlungssoftware des AT&T-Telefonnetzes

die benachbarten Netzknoten gesendet werden. Der Empfänger dieser Nachricht – so die Theorie – vermerkt den Ausfall in seiner internen Routing-Tabelle und leitet die entgegengenommenen Telefongespräche über andere, noch intakte Knoten weiter. In der Realität verursachte der Empfang einer Out-of-Service-Meldung den Zusammenbruch des Vermittlungsknotens, so dass dieser seinerseits mit dem Senden von Out-of-Service-Meldungen reagierte. In einer Kettenreaktion wurde eine Flut von Meldungen erzeugt, die einen Vermittlungsknoten selbst nach einem Reset aufgrund des gleichen Software-Fehlers erneut zum Absturz brachte. Nach kurzer Zeit lag ein Drittel des gesamten AT&T-Telefonnetzes brach.

Erst nach neun Stunden gelang es AT&T, das Netz zu stabilisieren und zum Normalbetrieb zurückzukehren. Von den 138 Millionen Anrufen, die in dieser Zeit getätigt wurden, konnten mehr als die Hälfte nicht vermittelt werden. Neben einem erheblichen Image-Schaden und einem Umsatzverlust, den AT&T mit geschätzten 60 Millionen US-Dollar bezifferte, erzeugte der Vorfall ein lautes Echo in der Presse. Dieser wuchs sich in der öffentlichen Wahrnehmung rasch zu einem Vertrauensverlust in die gesamte Computertechnik aus – wenn auch nur zu einem temporären [82, 37].

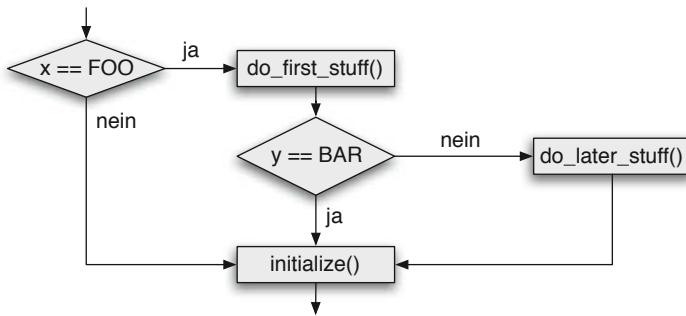


Abb. 2.7 Korrekte Initialisierungsreihenfolge des AT&T-Switch

Die gravierenden Symptome des Systemausfalls ließen zunächst auf einen gezielten Hacker-Angriff schließen, der sich im Laufe der Untersuchungen jedoch nicht bestätigte. Stattdessen wurde ein fehlerhafter Code-Abschnitt in der Software isoliert, die in allen 114 Vermittlungsknoten für die Weiterleitung der Anrufe zum Einsatz kam.

Der fehlerhafte Programmcode stammt aus einer Initialisierungsroutine und ist in abstrahierter Form in Abb. 2.6 dargestellt (vgl. [164]). Innerhalb eines umschließenden Switch-Case-Konstrukts führt das C-Programm zunächst eine Fallunterscheidung bezüglich der Variablen `line` durch. Für unsere Betrachtung ist der zweite Case-Zweig von Bedeutung, in dem verschiedene Initialisierungsroutinen in Abhängigkeit der Variableninhalte von `x` und `y` aufgerufen werden.

Der korrekte Kontrollfluss ist durch das Ablaufdiagramm in Abb. 2.7 vorgegeben. Um eine einwandfreie Initialisierung zu gewährleisten, wird die Variable `x` zunächst mit einem gewissen Wert verglichen und gegebenenfalls die Funktion `do_first_stuff` aufgerufen. Ist zusätzlich die Variable `y` auf einen bestimmten Wert gesetzt, soll mit Hilfe des C-Schlüsselworts `break` aus der aktuellen Klammerungsebene herausgesprungen und der Aufruf von `do_later_stuff` verhindert werden. Unabhängig von den Werten von `x` und `y` wird am Ende die Funktion `initialize` aufgerufen.

Sehen Sie bereits den Fehler? Die Ursache der Vermittlungsausfälle geht auf die falsche Verwendung des Schlüsselworts `break` zurück, dessen genaue Semantik erfahrungsgemäß von vielen Programmierern nicht hinreichend genau verstanden wird. Anders als der Urheber der Software unterstellt, unterbricht `break` nicht die aktuelle Klammerungsebene, sondern lediglich die innerste Do-, While- oder Switch-Umgebung. Folgerichtig beendet der so platzierte `break`-Befehl nicht nur den aktuellen If-Befehl, sondern gleich das gesamte Switch-Konstrukt. Dies wiederum hat zur Folge, dass zwar, wie gewünscht, der Aufruf von `do_later_stuff` unterbunden wird, der Aufruf von `initialize` aber ebenfalls unterbleibt. Exakt dieser ausgelassene Funktionsaufruf führte zu dem Fehler, der in Form einer unglücklichen Kettenreaktion für den bedeutendsten Ausfall des Telefonnetzes in der Geschichte von AT&T führte.

Der AT&T-Bug bringt zwei Aspekte typischer Software-Fehler klar zum Vorschein. Zum einen demonstriert der Vorfall, dass die Eigenschaft der nahezu beliebigen Replizierbarkeit in Hinsicht auf die resultierenden Fehlerszenarien zum Bumerang wird. Anders als physikalisch bedingte Hardware-Fehler, die in der Regel zum singulären Ausfall einer einzigen Baugruppe führen, betreffen Software-Fehler alle homogen installierten Komponenten in gleichem Maße. Hätte der Fehler in der AT&T-Vermittlungssoftware nicht alle Vermittlungsknoten simultan betroffen, wäre eine Kettenreaktion unmöglich und das Ausmaß des Ausfalls ungleich geringer gewesen.

Zum anderen ist es keine Überraschung, dass der AT&T-Bug auf einen Fehler innerhalb des Codes zur Fehlerbehandlung zurückgeht. Im Gegensatz zur Hauptfunktionalität wird der Fehlerbehandlungscode in der Praxis nur vergleichsweise wenig und unter Umständen gar nicht getestet. Die Gründe hierfür sind vielfältig und gehen unter anderem auf die Schwierigkeit zurück, ein bestimmtes Fehlerszenario unter Laborbedingungen gezielt herbeizuführen. Erschwert wird die Situation durch die schier gigantische Anzahl an erdenklichen Fehlerszenarien, die Computersysteme dieser Größenordnung aufweisen.

War der AT&T-Fehler einfach die Konsequenz aus der nicht mehr kontrollierbaren Komplexität heutiger Computersysteme und damit im Vorfeld unvermeidbar? Die Antwort ist ein klares Nein. Sowohl die Wahl eines restriktiven Sprachstandards, wie z. B. MISRA-C, als auch die Durchführung eines C_0 -Überdeckungstests hätten den Fehler im Vorfeld vermieden. Auf die Details dieser Techniken werden wir in den Kapiteln 3.1.2 bzw. 4.4.2 zurückkommen.

An dieser Stelle wenden wir uns erneut der NASA zu. Ein Blick in die vergleichsweise kurze Geschichte der US-amerikanischen Raumfahrt zeigt, dass nicht nur die Mariner-Mission unter einem schlechten Stern stand. Neben ihren unbestreitbaren Erfolgen musste die NASA im Laufe der Raumfahrtgeschichte weitere Rückschläge hinnehmen, von denen viele auf das Versagen der eingesetzten Software zurückgeführt werden konnten. So ging auch das Mars-Surveyor-'98-Programm unter dem Auge einer breiten Öffentlichkeit als spektakulärer Rückschlag in die Geschichte der US-amerikanischen Raumfahrt ein. Im Rahmen der Mission wurden zwei Raumsonden zum Mars geschickt, um dessen atmosphärische Bedingungen detailliert zu erkunden. Während eine der Sonden, der *Polar Lander*, auf der Marsoberfläche aufsetzen sollte, war es die Aufgabe des *Climate Orbiters*, den roten Planeten auf einer kreisförmigen Umlaufbahn zu umrunden und aus sicherer Entfernung zu analysieren.

Der *Climate Orbiter* wurde am 11. Dezember 1998 erfolgreich in Cape Canaveral gestartet und begann seine neun Monate lange Reise zum Mars. Pünktlich am 23. September 1999 wurde die Anflugphase eingeleitet und wie geplant mit der gezielten Annäherung an die vorberechnete Umlaufbahn begonnen. Die Konzeption sah vor, die letzte Phase der Annäherung mit Hilfe eines als *Aerobraking* bezeichneten Prinzips durchzuführen, das schon zuvor im Rahmen anderer Missionen erfolgreich eingesetzt wurde. Hierzu tritt die Sonde zunächst, wie in Abb. 2.8 skizziert, in einen elliptischen Orbit ein, dessen nächster Punkt nur ca. 150 km von der Marsoberfläche entfernt ist und damit bereits die obersten Schichten der Atmosphäre streift. Durch

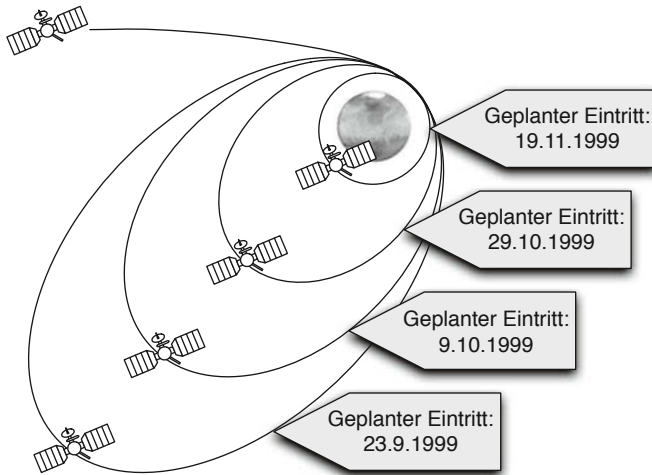


Abb. 2.8 Eintritt des Mars Climate Orbiters der NASA in die Aerobraking-Phase

die atmosphärische Reibung wird die Raumsonde leicht abgebremst, so dass sich die elliptische Umlaufbahn mit jedem Umlauf langsam an den später einzunehmenden kreisförmigen Orbit annähert. Nach 57 Tagen ist die Aerobraking-Phase beendet und die finale Kreisbahn erreicht.

Um 9:06 Uhr trat der Mars Climate Orbiter, wie erwartet, in den Funkschatten des roten Planeten ein, den er um 9:27 verlassen und den Kontakt wiederherstellen sollte. Die Kontrollstation wartete jedoch vergeblich auf die Wiederaufnahme des Funkkontakts und nach kurzer Zeit war klar, dass die Sonde unwiderruflich verloren war. Später stellte sich heraus, dass sich die Sonde bis auf 57 km der Marsoberfläche genähert hatte – rund 100 km weniger als vorberechnet. Die hohe atmosphärische Reibung in dieser niedrigen Höhe ließ den Orbiter in kürzester Zeit in einem hellen Feuerball verglühen.

Die Analyse der Telemetriedaten brachte den Fehler noch am selben Tag zum Vorschein. Zur Kurskorrektur beim Landeanflug griff der Orbiter auf eine von Lockheed Martin bereitgestellte Lookup-Tabelle zurück, das sogenannte *Small forces file*. Lockheed Martin legte die Daten in imperialen Einheiten $\text{lbs} \times \text{s}$ (*pound force seconds*) ab, von der NASA wurden die Werte aber, wie international üblich, nach dem metrischen System als $\text{N} \times \text{s}$ (*Newton-Sekunden*) interpretiert. Die verwendeten Werte waren dadurch um den Faktor 4.45 zu groß, und die Überkompensation der zu korrigierenden Bahnabweichung drängte den Mars-Orbiter schließlich auf die fatale Umlaufbahn in nur noch 57 km Höhe.

Der Software-Fehler, der die Mars-Mission in ein Desaster verwandelte, fällt genau wie der weiter oben beschriebene AT&T-Bug in die große Klasse der Semantikfehler. Im direkten Vergleich zeigt sich, dass die Natur beider Fehler trotzdem eine

andere ist. Der AT&T-Bug geht auf die falsche Interpretation eines C-Konstrukts zurück und hat seinen Ursprung damit in der *Sprachsemantik*. Im Falle des Mars Climate Orbiters hingegen wurde die Programmiersprache vollständig korrekt verwendet. Die Fehlerursache geht auf einen Typkonversionsfehler zurück und ist damit eine klassische Fehlinterpretation der *Programmsemantik*. In Abschnitt 3.2 werden wir auf die Problematik der Datentypkonversion genauer eingehen und Ansätze aufzeigen, wie sich Konversionsfehler dieser und ähnlicher Art erfolgreich vermeiden lassen.

Der Verlust des Mars Climate Orbiters war ein herber Rückschlag für die NASA. Zum vollständigen Desaster wurde das Projekt schließlich am 3. Dezember 1999, als auch noch der Mars Polar Lander verloren ging. Gegen 21:10 mitteleuropäischer Zeit, just im Moment des Eintritts in die Marsatmosphäre, brach der Funkkontakt für immer ab. Der Grund des Kommunikationsverlusts ist bis heute unbekannt, genauso wie das Schicksal der Raumsonde. Alle späteren Versuche, den Kontakt mit dem Polar Lander wiederherzustellen, blieben erfolglos. Die mutmaßliche Ursache für das Versagen wird, wie im Falle des Climate Orbiters, ebenfalls der Software zugeschrieben. Vermutlich wurden durch das Ausfahren der Landebeine Vibrationen erzeugt, die von der Software als das Aufsetzen der Sonde auf den Mars interpretiert wurden. Die fatale Fehleinschätzung führte zum sofortigen Abschalten der Bremsdüse, so dass die Sonde mit unverringelter Geschwindigkeit auf der Marsoberfläche aufschlug und zerschellte.

2.3 Parallelität als Fehlerquelle

Die spektakulären Verluste gleich beider Raumsonden des Mars-Surveyor-'98-Programms täuschen gelegentlich darüber hinweg, dass die NASA bereits in früheren Mars-Missionen mit Software-Problemen zu kämpfen hatte. So auch im Rahmen der Mars-Pathfinder-Mission, der wir heute neben wichtigen Informationen über die Gesteins- und Bodenbeschaffenheit auch zahlreiche spektakuläre Bilder der Marsoberfläche zu verdanken haben. Der *Pathfinder* wurde am 4. Dezember 1996 gestartet und landete exakt 7 Monate später, am 4. Juli 1997 auf der Marsoberfläche. Die Sonde selbst bestand aus einer Landeeinheit und dem *Sojourner*, einem speziell für die Marsoberfläche konstruierten Roboterfahrzeug zur Erkundung des angrenzenden Terrains.

Die Landung des Pathfinders verlief nach Plan und auch der Sojourner konnte erfolgreich von der Landeeinheit abgekoppelt werden. Die Bilder, die der Marsroboter kurz darauf der Erde übermittelte, beeindruckten die NASA und die Öffentlichkeit gleichermaßen. Entsprechend getrübt wurde die Freude, als der Sojourner kurze Zeit später begann, seine Systeme in unregelmäßigen Abständen neu zu starten und mit jedem Reset die aktuell gesammelten Daten zu verlieren. Nach 18 Stunden Dauerbetrieb gelang es der NASA, den Fehler in den Entwicklungslaboren zu reproduzieren. Die Auswertung der Protokolldaten machte deutlich, dass die Fehlerursache auf das Phänomen der *Prioritäteninversion* und damit auf ein lange bekanntes Problem der parallelen Programmierung zurückging.

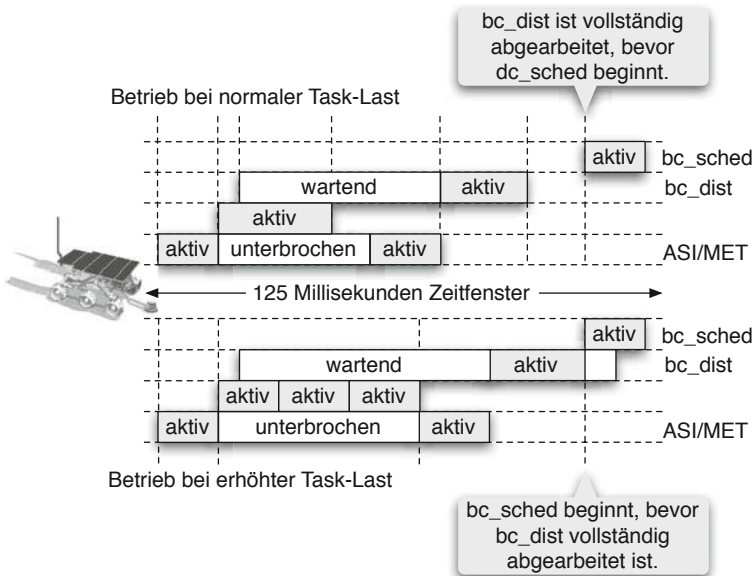


Abb. 2.9 Prioritäteninversion als Ursache unregelmäßiger Resets des Mars-Rovers Sojourner

Um das Phänomen der Prioritäteninversion zu verstehen, wagen wir einen tieferen Blick in die Software-Struktur der Raumsonde. Intern arbeitet der Sojourner mit einer Variante des Echtzeitbetriebssystems VxWorks. Das Verhalten der Sonde wird durch mehrere nebenläufige Tasks gesteuert, die mit unterschiedlichen Prioritäten versehen sind. Der eingesetzte *Scheduler* arbeitet preemtiv, d. h., ein Task niedriger Priorität wird durch den Betriebssystemkern unterbrochen, sobald ein Task höherer Priorität lauffähig ist. Die höchste Priorität besitzt der Task **bc_sched** (*scheduler task*). Dieser besitzt die Aufgabe, den Datenverkehr auf dem internen 1553-Bus zu koordinieren, über den ein Großteil der prozessübergreifenden Kommunikation abgewickelt wird. Ein zweiter Task, **bc_dist** (*distributor task*), übernimmt die eigentliche Verteilung der Daten und wird mit der dritthöchsten Priorität betrieben. Beide Tasks werden mit Hilfe eines Hardware-Timers mit einer Frequenz von 8 Hz periodisch aktiviert. In jedem 125-Millisekunden-Fenster wird zunächst der **bc_dist**-Task und anschließend der **bc_sched**-Task gestartet. Die Spezifikation sieht dabei zwingend vor, dass der **bc_dist**-Task vollständig abgearbeitet sein muss, bevor der **bc_sched**-Task aktiviert wird. Neben diesen zwei hochprioren Tasks laufen im System weitere Tasks mit niedriger Priorität. Zu diesen gehört auch der **ASI/MET**-Task zur Behandlung eingehender meteorologischer Daten.

Abb. 2.9 zeigt zwei verschiedene Task-Aktivierungsprofile innerhalb eines einzelnen 125-Millisekunden-Fensters. Das obere Profil beschreibt den vorgesehenen Betriebsablauf des Sojourners. Zu Beginn des Zeitfensters wird der niedripriore Task **ASI/MET** aktiviert und mit der Übertragung meteorologischer Daten begon-

nen. Da um den Kommunikationsbus mehrere Tasks konkurrieren, wird dieser, wie in solchen Fällen üblich, durch ein Semaphor geschützt. Da sich der Bus zur Zeit der Aktivierung von **ASI/MET** im Idle-Zustand befand, erhält der Task den exklusiven Zugriff. Kurze Zeit später wird der Distributor-Task **bc_dist** aktiviert. Da er ebenfalls auf den Bus zugreifen möchte, wird er durch das Semaphor-Prinzip so lange in den Wartezustand versetzt, bis **ASI/MET** beendet ist. Dieser wird jedoch durch die relativ niedrige Priorität häufiger selbst unterbrochen, so dass sich dessen Terminierung aufgrund der übrigen Busaktivität verzögert. Unter normaler Last wird **ASI/MET** früh genug beendet, so dass auch **bc_dist** rechtzeitig vor der Aktivierung von **bc_sched** terminiert.

Während des Betriebs des Sojourners war die Buslast jedoch deutlich erhöht, so dass sich die Abarbeitung des **ASI/MET**-Tasks zu Peak-Zeiten so weit verzögerte, dass **bc_sched** vor der vollständigen Abarbeitung von **bs_dist** gestartet wurde (vgl. Abb. 2.9 unten). Folgerichtig reagierte die Software des Sojourners mit einem Hardware-Reset, um die Sonde wieder in einen konsistenten Zustand zu bringen.

Bei genauerer Betrachtung des Szenarios zeigte sich, dass der **bc_dist**-Task zwar eine hohe Priorität besitzt, ihm diese jedoch nichts nützt. Aufgrund des von **ASI/MET** gehaltenen Semaphors verlängert sich die Wartezeit von **bc_dist** mit jedem weiteren Task, der eine höhere Priorität als **ASI/MET** besitzt. Kurzum: Ein niederpriorer Task verzögert die Ausführung eines höherprioreren Tasks. Dieses Phänomen ist heute wohlverstanden und wird in der Literatur als *Prioritäteninversion* bezeichnet.

So kompliziert das Phänomen der Prioritäteninversion auf den ersten Blick erscheinen mag, so einfach lässt es sich in der Praxis beheben. Wird der Task **ASI/MET** anstelle seiner niedrigen nativen Priorität so lange mit der hohen Priorität von **bc_dist** ausgeführt, bis er den blockierten Semaphor wieder freigibt, ist die Unterbrechung durch andere Tasks nicht mehr möglich. In der Konsequenz wird dadurch auch die für den Reset verantwortliche lange Wartezeit von **bc_dist** deutlich verringert. Dieses Prinzip der *Prioritätenvererbung* wird betriebssystemseitig von Vx-Works vollständig unterstützt. Damit war es der NASA möglich, den Fehler durch das Einspielen eines Software-Patches zu beseitigen und die Pathfinder-Mission schließlich doch noch zu einem vollen Erfolg der US-amerikanischen Raumfahrtgeschichte werden zu lassen.

Nichtsdestotrotz drängt sich auch hier die Frage auf, ob der Fehler im Vorfeld hätte vermieden werden können und wenn ja, mit welchen Mitteln? Aufgrund ihrer unregelmäßigen Auftrittcharakteristik sind Fehler, die durch die parallele Ausführung konkurrierender Ablaufströme verursacht werden, nur schwierig zurückzufolgern. Häufig liegt kein einziger Testfall vor, der das Problem zuverlässig reproduziert. Schlimmer noch: Die teilweise extremen Randbedingungen, die zum Auslösen des Fehlers nötig sind, stellen sich in vielen Fällen erst während des Betriebs ein. Damit fallen alle Black-Box-basierten Techniken und damit auch der klassische Software-Test als zuverlässige Methode im Vorfeld aus, da sie die inneren Strukturen der Software vollständig außer Acht lassen. Fehler dieser Art lassen sich nur durch die sorgfältige Konstruktion und Analyse der internen Programmstrukturen und damit mit Methoden der konstruktiven Qualitätssicherung und der White-Box-

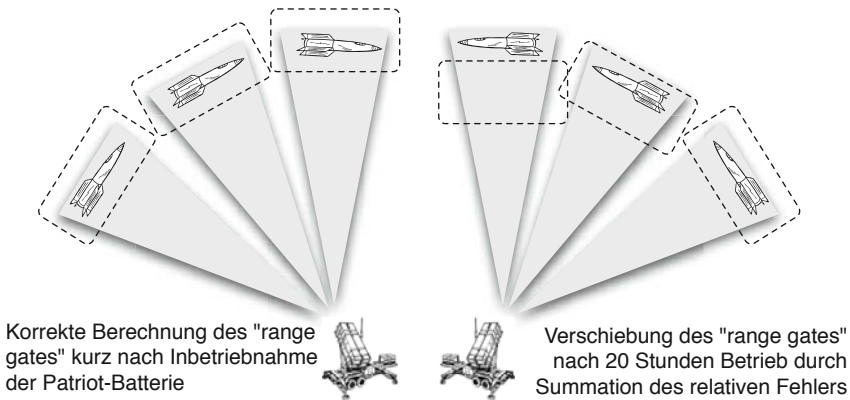


Abb. 2.10 Betriebsdauerbedingte Verschiebung des Range-gates eines Patriot-Raketenabwehrsystems

Analyse vermeiden. Beide werden wir in den Kapiteln 3 und 5 einer genaueren Betrachtung unterziehen.

2.4 Numerische Fehlerquellen

Wir versetzen uns für einen Moment zurück in das Jahr 1991. Während des ersten Golfkriegs feuert die irakische Armee am 25. Februar 1991 mehrere Scud-Raketen in Richtung Saudi-Arabien ab. Die von der US-Armee zur Flugabwehr installierte Patriot-Batterie reagiert zu spät und die gestartete Abfangrakete kann gegen die mit rasanter Geschwindigkeit herannahende Scud nichts mehr ausrichten. Die fatalen Folgen sind uns noch heute gegenwärtig: In der Nacht schlägt die Scud-Rakete zielgenau in eine amerikanische Kaserne im saudi-arabischen Dharaan ein und fordert 28 Todesopfer. Über 90 Menschen werden durch den Raketeneinschlag teilweise schwer verletzt [169]. Tragischerweise war der Software-Fehler, der für das Fehlverhalten verantwortlich war, zum Zeitpunkt des Ereignisses bereits bekannt, die korrigierte Software jedoch noch nicht auf den betreffenden Patriot-Batterien installiert.

Die Ursache des Fehlers geht auf die fundamentale Beschränkung aller digitalen Rechnerarchitekturen zurück, Werte ausschließlich diskret darzustellen. Spezifikationsseitig basieren die meisten Berechnungsmodelle jedoch auf kontinuierlichen Werten – so auch im Falle der Patriot-Flugabwehr. Der Fehler, der durch die rechnerseitige Diskretisierung entsteht, ist in den meisten Fällen klein genug, um sich nicht negativ auf das Systemverhalten auszuwirken. Der Patriot-Fehler demonstriert jedoch eindringlich, dass sich auch kleine Ungenauigkeiten durch unachtsame Programmierung zu großen Fehlern aufsummieren können.

Sobald eine Patriot-Batterie ein Zielobjekt erfasst hat, beginnt sie mit der Verfolgung der *Flugtrajektorie*. Hierzu berechnet die Software, wie in Abb. 2.10 darge-

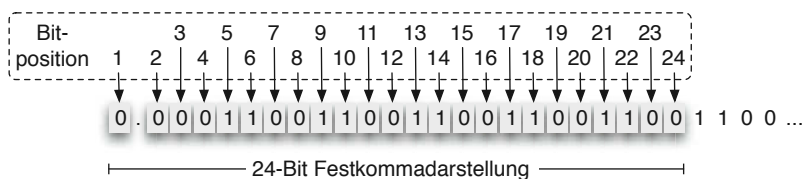


Abb. 2.11 Die Zahl 10^{-1} besitzt im Binärsystem keine endliche Repräsentation

stellt, permanent das sogenannte *Range gate*. Dieses beschreibt den Luftkorridor, innerhalb dessen das Zielobjekt als nächstes erscheint. Die Berechnungsvorschrift ist eine Funktion, die zum einen die Geschwindigkeit des Flugobjekts und zum anderen die Zeit der letzten Radardetektion als Parameter verarbeitet. Der letztgenannte Parameter wird in Form der *Systemzeit* entgegengenommen.

Intern wird die Systemzeit als Integer-Wert gespeichert und entspricht der Anzahl der verstrichenen Zehntelsekunden seit Inbetriebnahme der Abwehrbatterie. Die Systemzeit wird ständig erhöht, so dass im Laufe des Betriebs immer größere Absolutwerte gespeichert werden müssen. Vorsorglich wurde der darstellbare Wertebereich so gewählt, dass ein numerischer Überlauf im realen Betrieb faktisch ausgeschlossen ist.

Da der entsprechende Programmcode zur Berechnung des Range gates die verstrichene Zeit in Sekunden erwartet, die Systemzeit aber in Zehntelsekunden vorliegt, wird sie vor jeder Berechnung durch die Multiplikation mit dem Faktor 10^{-1} in das Zielformat konvertiert. Genau an dieser Stelle nahm das Verhängnis seinen Lauf. Intern verwendet die Patriot-Software eine 24-Bit-Festkommadarstellung. Da 10^{-1} keine endliche Repräsentation im Binärsystem besitzt, kann der Wert nur angenähert werden (vgl. Abb. 2.11). Der hierdurch verursachte Fehler ist nur minimal und beläuft sich, ausgedrückt im Dezimalsystem, auf ca. 9.5×10^{-7} . Die Patriot-Software ist jedoch so konzipiert, dass sich der relative Fehler in vollem Maße auf den Absolutwert der Systemzeit durchschlägt, d. h., der verursachte Fehler nimmt mit zunehmender Betriebsdauer kontinuierlich zu.

Die Einträge in Tabelle 2.1 fassen die verursachte Verschiebung des Range gates in Abhängigkeit von der Betriebsdauer zusammen [27]. Ein kritischer Wert wird bei einem Dauerbetrieb von 20 Stunden erreicht. Ab diesem Zeitpunkt ist das Range gate so weit verschoben, dass sich das Zielobjekt vollständig außerhalb befindet – ein erfolgreiches Abfangmanöver kann in keinem Fall mehr gelingen. Am 25. Februar 1991 war die betroffene Patriot-Batterie bereits seit 100 Stunden im Dauereinsatz und damit bereits mehr als 80 Stunden außer Gefecht.

Obwohl es sich bei dem Patriot-Bug um einen klassischen Numerikfehler handelt, der in vielen anderen Applikationen in ähnlicher Form vorhanden ist, wurde er dadurch begünstigt, dass das Patriot-System ursprünglich nicht für die Raketenabwehr gebaut wurde. In erster Linie sind die Patriot-Batterien als Boden-Luft-System zur Abwehr feindlicher Kampffjets konzipiert. In einem solchen Einsatzszenario war

Tabelle 2.1 Auswirkung des Konversionsfehlers auf die Patriot-Funktionstüchtigkeit

Betriebs- stunden [h]	Betriebs- sekunden [s]	Umgerechnete Zeit [float]	Berechnungs- fehler [s]	Range-gate- Verschiebung [m]
0	0	0	0	0
1	3600	3599,9966	0,0034	7
8	28800	28799,9725	0,0274	55
20	72000	71999,9313	0,0687	137
48	172800	172799,8352	0,1648	330
72	259200	259199,7528	0,2472	494
100	360000	359999,6667	0,3433	687

kein Patriot-System jemals zuvor so lange in Betrieb, als die im ersten Golfkrieg eingesetzten Systeme.

2.5 Portabilitätsfehler

Die Übertragung einer Applikation auf eine andere Hardware-Plattform oder ein anderes Betriebssystem gehört für gewöhnlich nicht zu den dankbarsten Aufgaben eines Software-Entwicklers. Zum einen ist die Portierung der Software mit teilweise erheblichen Anpassungsarbeiten verbunden, deren Auswirkungen bis auf die Architekturebene herunterreichen können. Zum anderen lässt sich das Verhalten portierter Programme in der Praxis nur sehr eingeschränkt vorhersagen. In einigen Fällen läuft die Software auf dem Zielsystem wie gehabt, in anderen fällt die Applikation durch diffiziles Fehlverhalten auf oder verweigert vollständig seinen Dienst.

Die Gründe hierfür sind vielfältig. Häufig sind es zeitliche Abhängigkeiten, die über die korrekte Ausführung eines Programms entscheiden. Beispielsweise werden viele Verklemmungsprobleme konkurrierender Prozesse im laufenden Betrieb nur dadurch vermieden, dass die Prozesse in einer bestimmten zeitlichen Reihenfolge bedient werden. Eine minimale Änderung des Prozess-Schedulers oder auch nur die Erhöhung der Prozessortaktrate kann dazu führen, dass die Applikation augenscheinlich einfriert oder einen Systemabsturz verursacht. Dieses Beispiel zeigt ein typisches Charakteristikum vieler Portabilitätsfehler: Das Fehlverhalten der Software ist oft bereits in der originalen Programmversion angelegt, wirkt sich auf der Ursprungsplattform aufgrund ihrer spezifischen Gegebenheiten aber nicht aus. Der Fehler verharrt in diesen Fällen so lange im Verborgenen, bis ihn der Umgebungswechsel sein volles Unheilpotenzial entfalten lässt.

Ein historisches Beispiel für einen Software-Fehler dieser Kategorie liefert uns der Jungferflug der Ariane V, der vielen noch in lebendiger Erinnerung sein mag. Die Ariane V ist die fünfte Generation von Trägerraketen, die unter der Leitung der *European Space Agency* (ESA) entwickelt wurde und ursprünglich als Trägersystem für die Europäische Raumfähre Hermes Verwendung finden sollte (vgl. Abb. 2.12). Hermes wurde nie gebaut, so dass die Ariane V, wie bereits ihre Vorgängerin Ariane IV, heute als reines Trägersystem für unbemannte Nutzlasten eingesetzt wird.

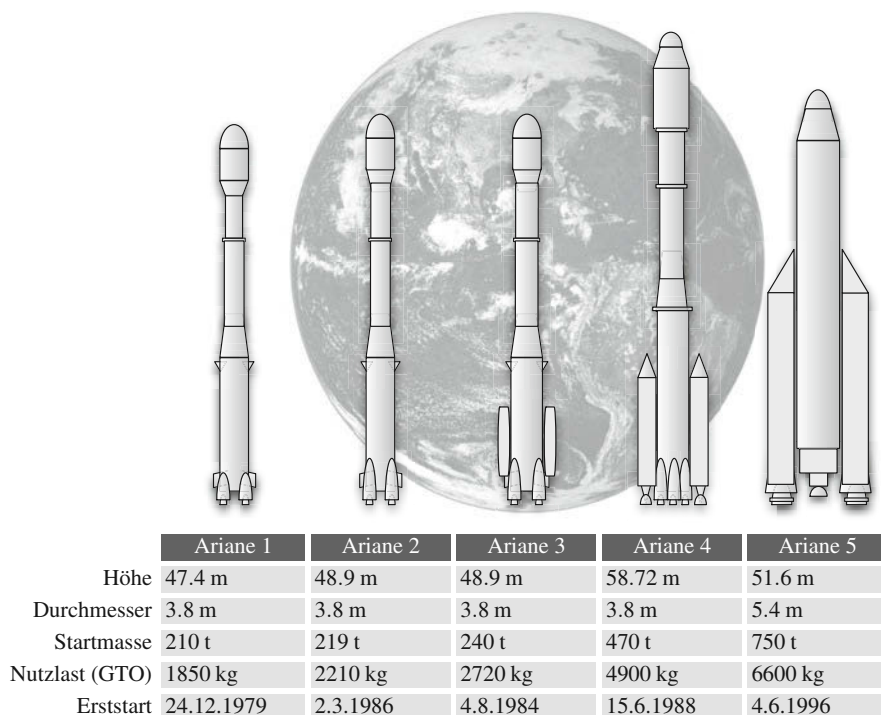


Abb. 2.12 Die Ariane-Flotte der ESA

So beeindruckend sich die Erfolgsbilanz der in vielen Starts erprobten Ariane IV liest, so desaströs begann die Geschichte ihres hoch gehandelten Nachfolgers. Nach zehnjähriger Entwicklungsdauer und einem kumulierten Budget von ca. sieben Milliarden Euro, startete die erste Ariane V am 4. Juni 1996 um 14:34:06 vom Weltraumbahnhof Kourou in Französisch-Guayana. Der Flug verlief alles andere als nach Plan. Als die Rakete in einer Höhe von ca. 3700 Metern eine Horizontalgeschwindigkeit von umgerechnet 32768.0 Einheiten erreicht hatte, begann die Steuerungslogik eine Flut von Steuerbefehlen zur Korrektur einer angeblichen Kursabweichung von 20 Grad zu erzeugen. Durch die Belastung der extremen Kurswechsel drohte die Rakete auseinanderzubrechen, und das Sicherheitssystem leitete nach 39 Sekunden die Selbstzerstörung ein. Kaum begonnen, endete der Jungferflug der Ariane V abrupt in einem gigantischen Feuerwerk.

Was war passiert? – Die letzte gemessene Horizontalgeschwindigkeit gibt uns einen ersten Hinweis auf die Fehlerursache. 32768 ist der kleinste Wert, den eine vorzeichenbehaftete 16-Bit-Integer-Zahl nicht mehr aufnehmen kann. Damit deutet alles auf einen Fehler hin, der durch einen numerischen Überlauf verursacht wurde. Dass wir uns mit dieser Vermutung auf der richtigen Fährte befinden, zeigt der in Abb. 2.13 skizzierte Code-Abschnitt aus dem mehrere Millionen Zeilen umfassen-

```
ariane.adb
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get(vertical_veloc_sensor);
    sensor_get(horizontal_veloc_sensor);
    vertical_veloc_bias := integer(vertical_veloc_sensor);
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```

Abb. 2.13 Ausschnitt aus der ADA-Implementierung der Ariane-V-Software

den Quelltext der Ariane V. Wie schon im Falle der Ariane IV ist die Software in der Programmiersprache ADA implementiert. ADA wurde in den Siebzigerjahren entwickelt und ähnelt optisch der Programmiersprache Pascal, verfolgt darüber hinaus jedoch zahlreiche fortschrittliche Konzepte wie die dynamische Laufzeitüberprüfung oder die integrierte Ausnahmebehandlung.

In dem besagten Code-Abschnitt werden zunächst die zwei Float-Variablen `vertical_veloc_sensor` und `horizontal_veloc_sensor`, sowie die zwei Integer-Variablen `vertical_veloc_bias` und `horizontal_veloc_bias` deklariert. Durch den Aufruf der Prozedur `sensor_get` werden die beiden Float-Variablen mit den von der Sensorik gemessenen Vertikal- und Horizontalgeschwindigkeiten beschrieben. Im nächsten Schritt werden die Float-Werte in vorzeichenbehaftete Integer-Zahlen konvertiert und den beiden Bias-Variablen zugewiesen. Beide Integer-Variablen sind 16 Bit breit, so dass im Gegensatz zum Float-Datentyp, der über einen wesentlich größeren Zahlenbereich verfügt, keine Zahlen größer als 32768 dargestellt werden können. Der Absturz der Ariane V hat seine Ursache damit in einem klassischen Typkonversionsfehler.

Zwangsläufig drängt sich an dieser Stelle die Frage auf, wie ein derart klassischer Programmierfehler die Testphase unbemerkt überstehen konnte. Auch hier ist die Antwort schnell gefunden und geht auf die historischen Wurzeln des Programmabschnitts zurück. Für die Ariane V wurden nicht alle Teile der Software von Grund auf neu entwickelt. Viele Code-Fragmente, zu denen auch der Programmausschnitt in Abb. 2.13 gehört, wurden aus der Code-Basis der äußerst zuverlässigen Ariane IV

übernommen. Insbesondere der für den Fehler verantwortliche Code wurde innerhalb des Ariane-IV-Programms ausgiebig getestet und konnte seine vermeintliche Korrektheit in vielen erfolgreich absolvierten Flügen unter Beweis stellen.

Gänzlich unberücksichtigt blieb bei der Code-Übernahme allerdings die Tatsache, dass die Horizontalbeschleunigung der wesentlich größer dimensionierten Ariane V die maximal erreichbare Geschwindigkeit ihres Vorgängermodells um das Vierfache übertrifft. Mit anderen Worten: Das Risiko eines potenziellen Numerikfehlers aufgrund der Inkompatibilität des Float- und Integer-Zahlenbereichs war bereits in der Ariane IV vorhanden, führte aber dank der vergleichsweise geringen Horizontal- und Vertikalgeschwindigkeit zu keiner realen Fehlfunktion. Das Beispiel zeigt wie kaum ein anderes, wie sehr die Korrektheit eines Programms mit seiner Einbettung in das Gesamtsystem verbunden ist. Diese Erkenntnis hat weiten Einfluss auf das gesamte Gebiet des Software-Tests, mit dem wir uns in Kapitel 4 im Detail beschäftigen.

2.6 Optimierungsfehler

Die Optimierung von Programmcode dient in den meisten Fällen zur Verbesserung der Laufzeit- und Speicherplatzeffizienz kritischer Programmabschnitte. Da die Optimierung innerhalb eines Software-Projekts in der Regel erst sehr spät durchgeführt werden kann, muss sie mit entsprechender Vorsicht vorangetrieben werden. Neben den klassischen Optimierungszielen *Laufzeit* und *Speicherplatz* sind viele Programmierer versucht, durch die Verwendung geeigneter Programmierkniffe auch die Code-Größe zu minimieren. Hierzu wird durch die geschickte Ausnutzung von Randbedingungen die vormals allgemein gehaltene Programmstruktur durch eine maßgeschneiderte und wesentlich kompaktere Variante ersetzt. Damit folgt die Optimierung der Code-Größe exakt den Vorgehensweisen, die auch zur Laufzeit- und Speicherplatzoptimierung zum Einsatz kommen. Dass solche Optimierungen mit besonderer Vorsicht zu genießen sind, zeigt eine ältere Version des Unix-Werkzeugs Mail, mit dessen Hilfe E-Mails direkt von der Unix-Kommandozeile entweder gesendet oder empfangen werden können. Entsprechend zweigeteilt präsentiert sich die Syntax des Werkzeugs:

■ Empfangen

```
mail [-e] [-h] [-p] [-P] [-q] [-r] [ -f file ]
```

■ Senden

```
mail [-t] [-w] [ -m ... ] <recipient 1> <recipient 2> ...
```

Die genaue Bedeutung der einzelnen Optionen ist für unsere Betrachtung sekundär. An dieser Stelle wollen wir uns vor allem mit der Frage beschäftigen, wie die Applikation intern feststellt, ob sie zum Senden oder zum Empfangen von Mails aufgerufen wurde. Ein Blick hinter die Kulissen zeigt, dass die Funktionsweise durch eine sehr spezifische Analyse der Übergabeparameter bestimmt wird. Anstatt alle Parameter der Reihe nach zu analysieren und detailliert auszuwerten, bedient sich der Autor des Programms eines Tricks. Wie ein gezielter Blick auf die Aufruf-Syntax

zeigt, wird das Programm genau dann zum Empfangen von Mail gestartet, wenn der letzte Übergabeparameter entweder selbst mit einem Bindestrich eingeleitet wird oder das zusätzliche Argument der Option `-f` ist. Der Programmcode der besagten Mail-Version macht sich dieses Analyseergebnis zunutze und überprüft die Übergabeparameter wie folgt:

```
if (argv[argc-1][0] == '-' || (argv[argc-2][1] == 'f')) {
    readmail(argc, argv);
} else {
    sendmail(argc, argv);
}
```

Obwohl die If-Abfrage den Sachverhalt augenscheinlich exakt abbildet, ist dem Autor einen diffizilen Denkfehler unterlaufen. Der linke Teil der If-Bedingung ist korrekt und prüft zuverlässig ab, ob es sich bei dem letzten Übergabeparameter um eine Option handelt oder nicht. Mit dem rechten Teil der If-Bedingung soll überprüft werden, ob der vorletzten Übergabeparameter der Option `-f` entspricht. Genau hier hat der Autor das Problem überoptimiert, da ausschließlich der zweite Buchstabe des vorletzten Parameters verglichen wird. Das Programm müsste sich demnach überlisten lassen, wenn es mit einer zweielementigen Recipient-Liste aufgerufen wird, deren erster Empfängername ein „f“ als zweiten Buchstaben besitzt. Mit den folgenden Argumenten aufgerufen wird die besagte Applikation in der Tat in den Empfangs- und nicht in den Sendemodus versetzt:

mail Effie Dirk

Der Fehler zeigt eine für viele Optimierungsfehler typische Auftrettscharakteristik. Damit sich der Programmierfehler überhaupt auswirkt, müssen sehr spezielle Randbedingungen eintreten. In unserem Fall muss die Applikation mit mindestens zwei Empfängeradressen aufgerufen werden, von denen die vorletzte Adresse ein „f“ als zweiten Buchstaben besitzt. Die Chance, dass der Fehler während der Entwicklung erkannt wird, ist hierdurch äußerst gering.

So schwer der Fehler mit den klassischen Methoden des Software-Tests erkannt werden kann, so einfach ist dessen Korrektur. Der fehlerhafte If-Zweig wurde durch eine zusätzliche Abfrage ergänzt, so dass neben der Überprüfung des Buchstabens „f“ auch das Vorhandensein des Optionsstrichs „-“ explizit getestet wird.

```
if (argv[argc-1][0] == '-' ||
    ((argv[argc-2][0] == '-') && argv[argc-2][1] == 'f')) {
    readmail(argc, argv);
} else {
    sendmail(argc, argv);
}
```

Hätte dieser schwer zu erkennende Software-Fehler bereits im Vorfeld vermieden werden können? Die Antwort ist auch hier ein klares Ja. Die Optimierung, mit dem der Autor des Programms die Analyse der Übergabeparameter auf einen trickreichen Einzeiler reduziert, ist hier schlicht fehl am Platz. Die Analyse der Übergabeoptionen ist eine Aufgabe, die von jedem Kommandozeilenwerkzeug gleichermaßen

ßen durchgeführt werden muss und geradezu nach einer Standardlösung schreit. In diesem Sinne ist auch die durchgeführte Fehlerkorrektur reine Symptombekämpfung und weit von einer sauberen Lösung entfernt. In Abschnitt 5.2.1.3 werden wir erneut auf das Mail-Beispiel zurückkommen und zwei solide Programmalternativen kennen lernen.

2.7 Von tickenden Zeitbomben

Vielleicht gehören Sie auch zu den Menschen, die der Milleniumsnacht am 31.12.1999 mit gemischten Gefühlen entgegenfieberten. Neben der seltenen Ehre, den (rechnerischen) Beginn eines neuen Jahrtausends miterleben zu dürfen, hing der *Y2K-Bug* (*Millenium-Bug*) wie ein Damoklesschwert über dem so sehnlich erwarteten Ereignis. Potenziell betroffen waren alle Software-Systeme, die zur Speicherung einer Jahreszahl nur zwei anstelle von vier Dezimalziffern verwendeten. In diesem Fall führt der Datumswechsel vom 31.12.1999 auf den 1.1.2000 zu einem numerischen Überlauf, d. h., die gespeicherte Zahl 99 springt schlagartig auf 00 zurück. Dieses Verhalten bedeutet mitnichten, dass ein solches Software-System mit einem Fehler reagiert. Trotzdem bestand für zahlreiche Applikationen die Gefahr, dass z. B. die Berechnung der Differenz zweier Jahreszahlen zu schwerwiegenden Inkonsistenzen in den Datenbeständen führen könnte. Bezüglich der potenziellen Auswirkungen, die ein solcher Fehler nach sich ziehen würde, waren der Phantasie kaum Grenzen gesetzt. Szenarien, in denen das Land mit automatisch versendeten Mahnschreiben überfluten wird, gehörten zu den harmloseren. Die Befürchtungen gingen so weit, dass in der besagten Neujahrsnacht viele Geldautomaten abgeschaltet wurden und zahlreiche Flugzeuge am Boden blieben.

Wie vielen von uns noch in guter Erinnerung ist, waren die verursachten Schäden weit geringer und unspektakulärer als erwartet. Einigen Experten zufolge wurde das Schadenspotenzial des Millenium-Bugs im Vorfeld schlicht überschätzt, andere Experten sahen in den geringen Folgen die Früchte jahrelanger Vorbereitung. In der Tat wurden Ende der Neunzigerjahre gigantische Investitionen getätigt, um die im Einsatz befindliche Software auf den Jahrtausendwechsel vorzubereiten. Viele der anfälligen Systeme stammten noch aus den frühen Tagen der Computertechnik und wurden in antiquierten Programmiersprachen wie COBOL oder FORTRAN verfasst. Der Markt für COBOL-Programmierer erlebte schlagartig eine Renaissance, die vorher kaum jemand für möglich gehalten hätte. Neben den Investitionen in die Software selbst, bereiteten sich viele Firmen durch zahlreiche Infrastrukturmaßnahmen auf das drohende Desaster vor. Insbesondere Systeme zur (geografisch) verteilten Datensicherung erlebten einen vorher nicht gekannten Boom. So gesehen hatte der Millenium-Bug auch seine gute Seite. Nahezu die gesamte IT-Industrie wurde von einem Investitionsschub gepackt, von dem wir heute noch in verschiedenster Form profitieren.

Ist der Millenium-Bug ein Einzelfall? Leider nein, denn im Jahr 2038 werden unsere Rechnersysteme erneut auf eine harte Probe gestellt. Potenziell betroffen sind alle Computer, die zur Zeitdarstellung intern das POSIX-Format verwenden.

```
2038_bug.c
#include <stdio.h>
#include <time.h>
#include <limits.h>

int main (int argc, char **argv)
{
    time_t t = 0x7FFFFFFF - 5;
    int i;

    for (i=0; i<10; i++) {
        time_t s = t;
        printf("%s", asctime (gmtime (&s)));
        t++;
    }
    return 0;
}
```

Abb. 2.14 Ist Ihr Betriebssystem fit für das Jahr 2038?

Darunter fallen neben fast allen Unix-basierten Systemen wie Linux und Mac OS X auch Windows und viele Betriebssysteme aus dem Embedded-Bereich.

Zur Darstellung der Zeit definieren diese Betriebssysteme einen Datentyp `time_t`, der die seit dem 1. Januar 1970 verstrichene Zeit in Sekunden repräsentiert. `time_t` ist in vielen Betriebssystemen als vorzeichenbehaftete 32-Bit-Integer-Zahl definiert, so dass pünktlich am 19. Januar 2038 um 3:14:08 UTC – exakt 2147483647 Sekunden nach Beginn der Zählung am 1. Januar 1970 – ein numerischer Überlauf eintreten wird.

Die Folgen sind ähnlich unvorhersehbar wie schon im Falle des Millenium-Bugs, jedoch besteht die berechtigte Hoffnung, dass im Jahr 2038 viele der heute als kritisch einzustufenden Systeme immun gegen den Fehler sein werden. Der Grund hierfür ist die sich bereits heute vollziehende 64-Bit-Migration der gängigen Betriebssysteme. Da der Datentyp `time_t` dort in der Regel bereits als 64-Bit-Integer-Zahl dargestellt wird, verschwindet die Überlaufproblematik, sobald die entsprechende Applikation auf einem 64-Bit-Betriebssystem neu übersetzt wird. In einigen Fällen bleibt das Problem jedoch weiterhin bestehen, insbesondere dann, wenn eine Applikation explizit nur die unteren 32 Bit der entsprechenden `time_t`-Variablen ausliest. Folgerichtig kommen wir auch im Falle des 2038-Bugs nicht umhin, alle relevanten Applikationen im Vorfeld auf dessen Kompatibilität zu untersuchen.

Auf die Frage, ob Ihr Betriebssystem anfällig gegenüber dem 2038-Problem ist oder nicht, gibt das C-Programm in Abb. 2.14 Auskunft. Zu Beginn wird die Variable `t` vom Typ `time_t` deklariert und mit einem Wert knapp unter der 32-Bit-Überlaufschwelle initialisiert. Innerhalb der sich anschließenden For-Schleife wird der Wert von `t` sukzessive um eins erhöht und mit Hilfe der Funktionen `gmtime` und `asctime` zunächst in eine Datums- und Zeitangabe und danach in eine druckba-

re Zeichenkette konvertiert. Unter den 32-Bit-Versionen der Betriebssysteme Linux und Mac OS X produziert das Programm beispielsweise die folgende Ausgabe:

```
Tue Jan 19 03:14:02 2038
Tue Jan 19 03:14:03 2038
Tue Jan 19 03:14:04 2038
Tue Jan 19 03:14:05 2038
Tue Jan 19 03:14:06 2038
Tue Jan 19 03:14:07 2038
Fri Dec 13 20:45:52 1901
Fri Dec 13 20:45:53 1901
Fri Dec 13 20:45:54 1901
Fri Dec 13 20:45:55 1901
```

Numerische Überlaufprobleme wie die oben geschilderten kommen in der Praxis weit häufiger vor als landläufig vermutet. Viele Probleme dieser Art lassen sich jedoch mit ein wenig Programmiererfahrung von vorne herein vermeiden – die nötige Sensibilität des Programmierers vorausgesetzt. Als Beispiel betrachten wir die in Abb. 2.15 dargestellte Beispielimplementierung des C-Makros `time_after` zum Vergleich zweier Zeitangaben. Das Makro nimmt die beiden Zeitpunkte als Parameter entgegen und evaluiert genau dann zu `True`, wenn der erste Übergabeparameter einen späteren Zeitpunkt als der zweite beschreibt. Das Makro wurde für den Fall konzipiert, dass Zeitpunkte in Form ganzzahliger Werte des Typs `long` repräsentiert werden.

Ein Makro mit der beschriebenen Funktionalität ist unter anderem fester Bestandteil des Linux-Kernels. Dort wird es häufig zusammen mit der globalen Variablen `jiffies` verwendet, in der die Anzahl der *Clock-Ticks* gespeichert ist, die seit dem Start des Kernels verstrichen sind. Innerhalb der Service-Routine des Linux-Timer-Interrupts wird der Inhalt der Variablen sukzessive um eins erhöht und damit auf den aktuellen Stand gebracht.

Die in Abb. 2.15 dargestellte Funktion `time_sensitive` demonstriert eine Beispielanwendung des `time_after`-Makros. Innerhalb der Funktion wird zunächst der aktuelle Wert der Variablen `jiffies` ausgelesen, die Konstante `HZ` addiert und anschließend der Variablen `deadline` zugewiesen. `HZ` gibt an, wie viele Timer-Interrupts pro Sekunde ausgelöst werden, so dass der berechnete Zeitpunkt genau eine Sekunde in der Zukunft liegt. Danach wird ein zeitkritischer, jedoch nicht weiter spezifizierter Code-Abschnitt ausgeführt und anschließend die Einhaltung der Zeitvorgabe durch den Vergleich der Variablen `deadline` mit dem aktuellen Inhalt der Variablen `jiffies` überprüft.

Die Variable `jiffies` ist als vorzeichenlose 32-Bit-Integer-Zahl deklariert, so dass der größte speicherbare Wert $2^{32} - 1$ beträgt. Nach 4294967295 Clock-Ticks verursacht die Erhöhung einen numerischen Überlauf, der den Wert auf 0 zurücksetzt. Bezogen auf eine Auslösefrequenz des Timer-Interrupts von 1000 Hz, wie sie viele Kernel der 2.6er-Reihe zugrunde legen, findet der erste Überlauf nach ca. 50 Tagen statt – eine Zeitspanne, die insbesondere während der internen Testphase selten erreicht wird. Legen wir eine für 2.4er-Kernel übliche Auslösefrequenz von 100 Hz zugrunde, entsteht der erste Überlauf gar erst nach 497 Tagen.

```

time_after.c
#define time_after(unknown, known) ((unknown) > (known))
1
2
void time_sensitive() {
3
4     unsigned long deadline = jiffies + HZ;
5     /* deadline = aktuelle Zeit plus 1 Sekunde */
6
7     ...
8
9     if (time_after(jiffies, deadline)) {
10        /* Deadline wurde verletzt */
11        ...
12    } else {
13        /* Deadline wurde eingehalten */
14        ...
15    }
16
17 }

```

Abb. 2.15 Viele Programme sind anfällig für numerische Überläufe, die im Rahmen des Software-Tests schwer zu entdecken sind. In diesem Fall kommt der Fehler erst nach mehreren Tagen zum ersten Mal zum Vorschein

Bei jedem Überlauf evaluiert das oben definierte Makro `time_after`, wie das folgende Beispiel zeigt, zu einem falschen Ergebnis. Dabei spielt es keine Rolle, wie nahe beide Zeitpunkte nebeneinander liegen:

```

time_after(UINT_MAX+1, UINT_MAX)
= time_after(4294967295+1, 4294967295)
= time_after(0, 4294967295)
= (0 > 4294967295)
= FALSE

```

Die Ursache geht auf die zwar einsichtige, aber zugleich naive Implementierung des `time_after`-Makros zurück. Durch die folgende Umformulierung lässt sich die Überlaufproblematik auf trickreiche Weise lösen, ohne die Laufzeit des Makros zu verschlechtern:

```

#define time_after(unknown, known)
    ((long)(known) - (long)(unknown) < 0)

```

Angewendet auf das obige Beispiel liefert das Makro trotz des numerischen Überlaufs jetzt das korrekte Ergebnis:

```

time_after(UINT_MAX+1, UINT_MAX)
= time_after(4294967295+1, 4294967295)
= time_after(0, 4294967295)
= ((long)4294967295 - (long)0 < 0)
= ((-1) - 0 < 0)
= (-1 < 0)
= TRUE

```

```
jiffies.h
#define time_after(unknown, known)      1
    ((long)(known) - (long)(unknown) < 0)      2
#define time_before(unknown, known)      3
    ((long)(unknown) - (long)(known) < 0)      4
#define time_after_eq(unknown, known)      5
    ((long)(unknown) - (long)(known) >= 0)      6
#define time_before_eq(unknown, known)      7
    ((long)(known) - (long)(unknown) >= 0)      8
```

Abb. 2.16 Auszug aus der Datei `linux/jiffies.h`

In der Tat implementiert der Linux-Kernel `time_after` und eine Reihe ähnlicher Makros exakt auf diese Weise, so dass ein Überlauf der Variablen `jiffies` beim Vergleich zweier nahe beieinander liegender Zeitpunkte keine Probleme verursacht. Die Makros des Linux-Kernels befinden sich in der Datei `linux/jiffies.h`, die auszugsweise in Abb. 2.16 dargestellt ist.

Einen Wermutstropfen gibt es allerdings auch hier zu beklagen. Viele Software-Entwickler bedienen sich der Makros erst gar nicht und programmieren den Vergleich zweier Zeitpunkte manuell aus. Die Frage, welche Implementierungsvariante die meisten Programmierer hier bevorzugen, möge sich der Leser selbst beantworten. Im Zweifel hilft ein Reboot – zumindest alle 49 Tage.

2.8 Spezifikationsfehler

Alle bisher vorgestellten Software-Fehler waren Implementierungsfehler, d. h. ihre Ursachen gingen auf eine fehlerhafte Umsetzung der Spezifikation zurück. In der Tat lassen sich die meisten in der Praxis beobachtbaren Defekte auf Fehler in der Implementierung zurückführen – nicht zuletzt aus dem Grund, dass die Spezifikation oft nur vage formuliert oder gar nicht vorhanden ist. In anderen Fällen ist bereits die Spezifikation fehlerhaft. Spezifikationsfehler sind schon deshalb als kritisch einzustufen, da fast alle Methoden der Software-Qualitätssicherung auf die korrekte Umsetzung der Spezifikation in die Implementierung fokussieren. Für die Erkennung von Spezifikationsfehlern bleiben diese Methoden außen vor. Insbesondere ist von dieser Limitierung auch die *formale Verifikation* betroffen, die als einzige der hier vorgestellten Technik im mathematischen Sinne vollständig ist. Die Vollständigkeit bedeutet an dieser Stelle nichts anderes, als dass für ausnahmslos alle möglichen Eingaben sichergestellt wird, dass die Implementierung der Spezifikation genügt. Fehler in der Spezifikation liegen damit auch hier konstruktionsbedingt außerhalb des Leistungsspektrums dieser Verfahren.

Ein klassischer Spezifikationsfehler trat am 14.9.1993 schlagartig in das Licht der Öffentlichkeit, als der Lufthansa-Flug 2904 bei der Landung auf dem Flughafen Warschau-Okecie verunglückte. Nach dem Aufsetzen schießt der Airbus A320 über die Landebahn hinaus und kommt erst durch die Kollision mit einem künstlichen

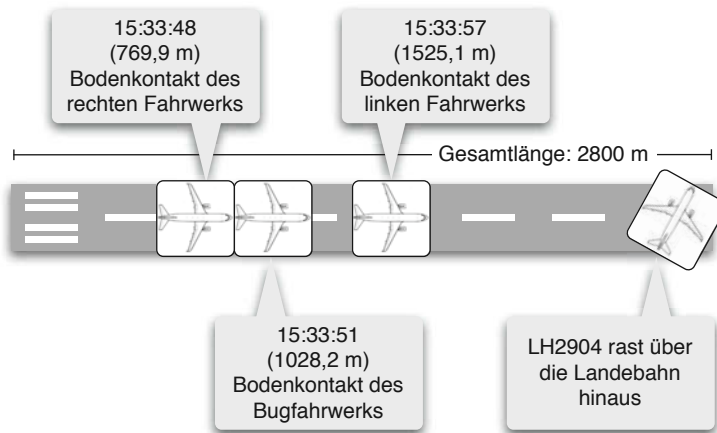


Abb. 2.17 Crash der Lufthansa-Maschine 2904 in Warschau

Erdwall zum Stillstand. Durch den seitlichen Aufprall werden die Treibstofftanks des Flügels aufgerissen und die Maschine geht kurze Zeit später in Flammen auf. Trotz der enormen Zerstörung überleben das Unglück 68 Passagiere – für den Kopiloten und einen Passagier kommt allerdings jede Hilfe zu spät.

Was war passiert? Um 15:29 befindet sich der Lufthansa-Airbus kurz vor Warschau und erhält von der Anflugkontrolle die Daten für den Endanflug. Die meteorologischen Bedingungen sind nicht optimal an diesem Tag. Im Bereich des Flughafens setzt starker Regen ein und die Piloten der vorher gelandeten Maschinen berichten von teilweise erheblichen Scherwinden während des Anflugs. Als Reaktion auf die Scherwindmeldung erhöht der Pilot zur Stabilisierung von LH 2904 die Anfluggeschwindigkeit um 20 Knoten. In Bodennähe wird das Flugzeug durch unerwartet hohe Rückenwinde weiter vorangetrieben, so dass sich die Maschine der Landebahn mit deutlich erhöhter Geschwindigkeit als normal nähert und erst nach 770 Metern um 15:33 und 48 Sekunden mit dem rechten Fahrwerk aufsetzt. Damit hat die Maschine bereits ein Viertel der nur rund 2800 Meter langen Landebahn überflogen. 3 Sekunden später setzt das Bugrad auf, den Piloten gelingt es jedoch nicht, die Schubumkehr zu aktivieren. Erst weitere 6 Sekunden später registriert der Bordcomputer das Aufsetzen des linken Fahrwerks und damit den Bodenkontakt der gesamten Maschine. Zu diesem Zeitpunkt bleiben den Piloten nur noch 1275 Meter bis zum Ende der Runway – zu wenig, um den Airbus rechtzeitig zum stehen zu bringen. Kurz vor dem Ende der Landebahn dreht der Pilot die Maschine quer, um den Aufprall abzumildern. Abb. 2.17 fasst den zeitlichen Ablauf der Ereignisse in einer Übersichtsgrafik zusammen.

Doch warum gelang es den Piloten erst so spät, die Schubumkehr zu aktivieren? Die Ursache geht auf einen Schutzmechanismus des Airbus A320 zurück, der die Aktivierung der Schubumkehr während des Flugs verhindern soll. Ob sich das Flugzeug noch in der Luft befindet oder bereits Bodenkontakt besteht, wird durch das

boolesche Signal *A/G* repräsentiert und von der Airbus-Software durch die Auswertung der folgenden Parameter berechnet:

p_l : Kompressionsdruck des linken Schockabsorbers

p_r : Kompressionsdruck des rechten Schockabsorbers

v_l : Drehgeschwindigkeit des linken Fahrwerks

v_r : Drehgeschwindigkeit des rechten Fahrwerks

Der verunglückte Airbus verwendete zur Berechnung des *A/G*-Signals die folgende Formel:

$$A/G = (\min(p_l, p_r) > 12.000 \text{ kg}) \vee (\min(v_l, v_r) > 72 \text{ kts}) \quad (2.3)$$

Der Bodenkontakt ist demnach hergestellt, sobald der Kompressionsdruck beider Schockabsorber mindestens 12 Tonnen beträgt oder sich die Räder des linken und des rechten Fahrwerks beide mit mehr als 72 Knoten drehen. Durch die disjunktive Verknüpfung wird der Bodenkontakt bereits dann erkannt, wenn nur eine der beiden Bedingungen erfüllt ist.

Im Falle von LH 2904 führte die Kombination von starken Seitenwinden und großen Wassermengen auf der Landebahn dazu, dass die Airbus-Steuerung erst 9 Sekunden nach dem Aufsetzen des rechten Fahrwerks den Bodenkontakt des Flugzeugs registrierte. Zum einen wurde aufgrund der Seitenwinde der geforderte Kompressionsdruck von 12.000 kg erst sehr spät erreicht, zum anderen führte starkes Aquaplaning dazu, dass das Rad des linken Fahrwerks nicht schnell genug die benötigte Geschwindigkeit aufbauen konnte.

Die gewonnenen Erkenntnisse ließen den Schluss zu, dass die in Gleichung (2.3) dargestellte Spezifikation ungeeignet ist, um den Bodenkontakt auch bei widrigen Wetterbedingungen zuverlässig zu erkennen. Die Spezifikation wurde geändert und heute reicht bei allen Airbus-Flugzeugen vom Typ A320 der Lufthansa-Flotte bereits ein Anpressdruck von 2.500 kg aus, um den Bodenkontakt zu erkennen. Der Crash von LH 2904 würde sich unter den damals herrschenden meteorologischen Bedingungen mit den geänderten Parametern wahrscheinlich nicht wiederholen.

Ob die neue Parametrisierung ausreicht, um auch bei noch höheren Seitenwinden den Bodenkontakt rechtzeitig zu erkennen, steht in den Sternen. Entsprechenden Forderungen, das manuelle Setzen des *A/G*-Signals durch die Piloten zuzulassen, stehen Sicherheitsbedenken entgegen. In Falle von LH 2904 hätte das Unglück durch die manuelle Aktivierung der Schubumkehr vermieden werden können, allerdings ist diese Möglichkeit im Zeitalter terroristischer Bedrohungen mit weiteren unkalkulierbaren Risiken verbunden. Wird die Schubumkehr während des Flugs aktiviert, bedeutet dies den sicheren Absturz.

An die realen Folgen einer solchen Aktivierung erinnert das Schicksal des Lauda-Air-Flugs 004 von Bangkok nach Wien. Als am 26.5.1991 während des Steigflugs die Schubumkehr des linken Triebwerks aufgrund eines technischen Defekts auslöst, wird das Flugzeug über Thailand mit 223 Passagieren an Bord regelrecht zerrissen.

2.9 Nicht immer ist die Software schuld

Alle bisher betrachteten Fehler haben eines gemeinsam: Sie begründen sich auf Fehler in der zugrunde liegenden Software und nicht der Hardware. Statistisch gesehen ist diese Tatsache keine Überraschung, schließlich lässt sich das Versagen von Hardware-Komponenten in den meisten Fällen auf physikalische Einwirkungen oder altersbedingte Verschleißerscheinungen zurückführen. Systemausfälle aufgrund von Entwurfsfehlern sind dagegen vergleichsweise selten.

Umso erstaunter reagierte die Fachwelt und die Öffentlichkeit, als 1993, just ein paar Monate nachdem Intel frenetisch die Markteinführung des Pentium-I-Prozessors feierte, die ersten Berichte über falsche Berechnungen der Divisionseinheit bekannt wurden. Als erster bemerkte Professor Thomas R. Nicely vom Lynchburg College in Virginia, USA, dass die Divisionseinheit des Pentium I für ausgewählte Zahlen marginal falsche Ergebnisse lieferte. Aufgefallen waren die Diskrepanzen, als Nicely sein altes System, basierend auf einem Intel-80486-Prozessor, durch ein neueres und wesentlich schnelleres Pentium-System ersetzte. Obwohl die meisten arithmetischen Operationen auf beiden Systemen keine Unterschiede zeigten, gab es vereinzelte Berechnungsfolgen, die in den letzten Nachkommastellen voneinander abwichen. Nicely dachte keineswegs an einen Hardwarefehler und vermutete die Fehlerquelle zunächst in der Software. Aber selbst unter Abschaltung aller Compiler-Optimierungen zeigte das Pentium-System die gleichen fehlerhaften Berechnungen wieder und wieder. Schließlich gelang es Nicely, den Divisionsfehler erfolgreich auf die Fließkommaeinheit des Pentium-Prozessors zurückzuführen.

Erwartungsgemäß wurde die Nachricht von Intel mit mäßiger Begeisterung aufgenommen. In der Tat wusste das Unternehmen über den Fehler bereits Bescheid, als Nicely den Pentium-Bug publik machte. Intel hoffte schlicht, dass der Fehler aufgrund seiner geringen Auftrittsscharakteristik unbemerkt bleiben würde. Die Chancen standen gut, schließlich berechneten die fehlerhaften Pentium-Prozessoren in den allermeisten Fällen das korrekte Ergebnis. Zudem sind die arithmetischen Fehler der Zahlenkombinationen, auf die sich der Pentium-Bug überhaupt auswirkt, äußerst gering. Wird beispielsweise die Zahl 824633 702 441 auf einem fehlerhaften Pentium-Prozessor durch sich selbst dividiert, ist das Ergebnis gleich 0,999999996274709702 und nicht wie erwartet gleich 1,0. Mit etwas Aufwand lassen sich auch dramatischere Fälle konstruieren. So führt die Berechnung

$$z = x - \frac{x}{y} \times y \quad (2.4)$$

für die Werte $x = 4195835$ und $y = 3145727$ auf einem 80486-Prozessor zu dem korrekten Ergebnis 0, auf einem fehlerhaften Pentium I jedoch zu 256.

Nachdem die ersten kompromittierenden Zahlenpaare gefunden waren, ließen weitere nicht lange auf sich warten. Langsam aber sicher begann sich die öffentliche Diskussion zu einem ernsthaften Problem für den Chip-Hersteller zu entwickeln. Intel begegnete der Öffentlichkeit zunächst mit einer Reihe von Studien, die eine genauere Untersuchung sowie eine Abschätzung der Fehlerauftrittswahrscheinlichkeiten für verschiedene Anwendungsdomänen zum Inhalt hatten (vgl. Tabelle 2.2).

Tabelle 2.2 Risikobewertung des Pentium-FDIV-Bugs durch Intel (Auszug aus [59])

Class	Applications	MTBF	Impact of failure in div/rem/tran
Word processing	Microsoft Word, Wordperfect, etc.	Never	None
Spreadsheets (basic user)	123, Excel, QuattroPro (basic user runs fewer than 1000 div/day)	27,000 years	Unnoticeable
Publishing, Graphics	Print Shop, Adobe Acrobat viewers	270 years	Impact only on Viewing
Personal Money Management	Quicken, Money, Managing Your Money, Simply Money, TurboTax (fewer than 14,000 divides per day)	2,000 years	Unnoticeable
Games	X-Wing, Falcon (flight simulator), Strategy Games	270 years	Impact is benign, (since game)

Diese und andere Untersuchungen kamen allesamt zu dem Schluss, dass der Fehler als unbedenklich einzustufen sei. Trotzdem wurde der öffentliche Druck so stark, dass sich Intel schließlich gezwungen sah, eine der größten Rückrufaktionen in der IT-Geschichte zu starten. Zu diesem Zeitpunkt waren bereits ca. 2 Millionen Pentium-Prozessoren im Alltagseinsatz, von denen jedoch lediglich 10 % an Intel zurückgesandt wurden. Darüber hinaus mussten schätzungsweise 500 000 Prozessoren aus den eigenen Lagerbeständen und 1 500 000 aus Händlerbeständen vernichtet werden. Rückblickend wird der finanzielle Schaden des Pentium-FDIV-Bugs auf ca. 475 Millionen US-Dollar geschätzt.

Obwohl der Pentium-Bug kurzfristig auch das Image von Intel ins Wanken brachte, war dieser Effekt nicht nachhaltig. Intel hat es verstanden, nach den anfänglich eher unglücklichen Beschwichtigungsversuchen am Ende geschickt auf den öffentlichen Druck zu reagieren. Das Image der Firma Intel ist heute ungebrochen hoch.

Doch was war die Ursache für die falschen Berechnungen des Pentium-Prozessors und warum traten Rechenfehler nur derart vereinzelt auf? Der Grund liegt in der Architektur der Fließkommaeinheit, die zur Division zweier Zahlen die *Radix-4-SRT-Division* verwendet. Im Gegensatz zu klassischen Divisionseinheiten, die das Ergebnis in einem iterativen Prozess Nachkommastelle für Nachkommastelle erzeugen, erlaubt die Radix-4-SRT-Division, in jedem Iterationsschritt zwei Nachkommastellen auf einen Schlag zu berechnen. Die Idee der SRT-Division ist nicht neu und geht auf die Arbeiten von Sweeney [49], Robertson [223] und Tocher [256] zurück, die den Divisionsalgorithmus unabhängig voneinander in den späten Fünfzigerjahren entwickelten.

Eine der Grundideen der SRT-Division besteht darin, das Divisionsergebnis intern nicht im Binärformat darzustellen. An die Stelle der Binärziffern 0 und 1 treten Koeffizienten mit den Werten -2 , -1 , 0 , 1 oder 2 . Der im i -ten Iterationsschritt zu wählende Wert des Koeffizienten q_i wird in Abhängigkeit des Dividenden p_i und des Divisors d aus einer Tabelle ausgelesen. Hierzu verwaltet der Pentium-I-Prozessor

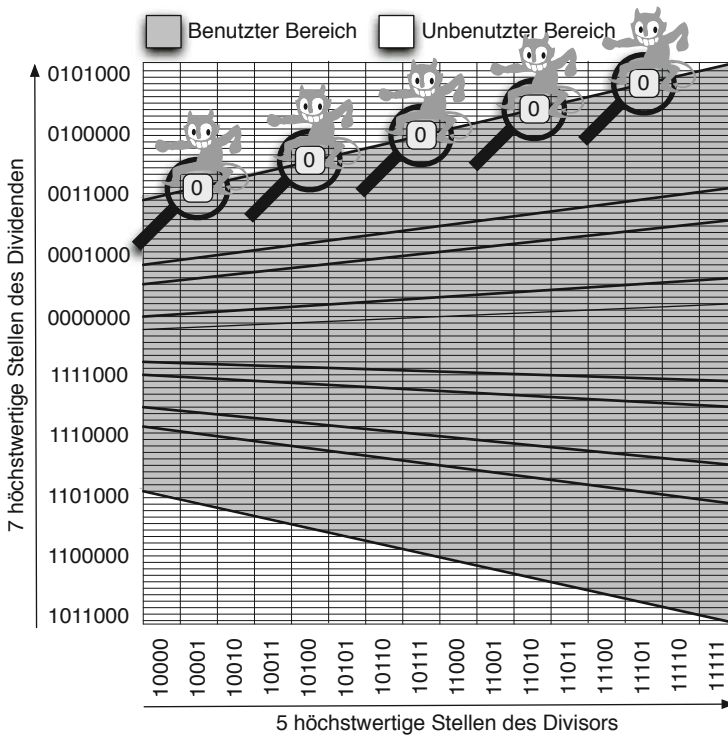


Abb. 2.18 Die Radix-4-Koeffizientenmatrix des Pentium-I-Prozessors

intern eine zweidimensionale Matrix mit 2048 Einträgen, von denen aber nur insgesamt 1066 verwendet werden (vgl. Abb. 2.18). Zur Adressierung werden die ersten 7 Stellen des Dividenten p_i als Zeilenindex und die ersten 5 Stellen des Divisors d als Spaltenindex verwendet. Entsprechend dem Wertebereich von q_i enthält jede Zelle einen der Werte $-2, -1, 0, 1$ oder 2 .

Der Pentium-FDIV-Bug hat seine Ursache schlicht in fünf falschen Tabelleneinträgen an der oberen Grenze zwischen dem benutzten und dem unbenutzten Bereich. Anstelle des (korrekten) Werts 2 enthalten die Zellen in fehlerhaften Pentium-Prozessoren den Wert 0 . Die Tatsache, dass der Pentium-Bug in der Praxis nur bei sehr wenigen Zahlenkombinationen auftritt, liegt an der unregelmäßigen Wahrscheinlichkeitsverteilung, mit der die einzelnen Zellen ausgelesen werden. Wären die Zugriffe auf alle Zellen der Matrix statistisch gleichverteilt, wäre der Pentium-Bug in der Praxis wesentlich häufiger zu beobachten und damit mit hoher Wahrscheinlichkeit vor der Auslieferung erkannt und behoben worden.

Aus mathematischer Sicht ist der Pentium-FDIV-Bug heute gut untersucht. Einige der wichtigsten Erkenntnisse gehen auf die Arbeiten von Alan Edelman zurück, der die Natur des Pentium FDIV-Bugs in [80] treffend zusammenfasst:

“The bug in the Pentium was an easy mistake to make, and a difficult one to catch.”

Alan Edelman [80]

Mit dem Erscheinen der Nachfolgemodelle, dem Pentium Pro und dem Pentium MMX, gehörte der FDIV-Bug vollends der Vergangenheit an. Sollte dem Divisionsdebakel überhaupt etwas Positives abzugewinnen sein, so ist es wahrscheinlich die Erkenntnis, dass es keine fehlerfreie Hardware gibt und wahrscheinlich nie geben wird. Galt die Software damals in der öffentlichen Meinung als die einzige ernstzunehmende Quelle für Computerfehler, so hat sich diese Einstellung mit dem Bekanntwerden des FDIV-Bugs nachhaltig geändert. Seit die Komplexität von Hardware der Komplexität von Software kaum noch nachsteht, gehört die lange vorherrschende Ansicht der fehlerfreien Hardware ins Reich der Visionen.

Auswirkungen hat diese Erkenntnis insbesondere für den Bereich eingebetteter Systeme. Zur Gewährleistung der mitunter sehr hohen Zuverlässigkeitsanforderungen muss neben der funktionalen Korrektheit der Software-Komponenten und der physikalischen Beständigkeit der Bauelemente auch die *funktionale Korrektheit* der Hardware als Risikofaktor berücksichtigt werden.

2.10 Fehlerbewertung

In den vorangegangenen Abschnitten haben wir einige der prominentesten Software-Fehler der Computergeschichte kennen gelernt und ihr Wesen genauer beleuchtet. Deren Auswirkungen waren dabei genauso vielfältig wie deren Ursachen, so dass sich mit dem Begriff des Software-Fehlers nicht nur ein quantitativer Aspekt, sondern immer auch ein qualitativer verbindet. Zur Verdeutlichung dieses qualitativen Aspekts betrachten wir die drei C-Fragmente in Abb. 2.19. Jedes Beispiel beschreibt ein typisches Fehlersymptom zusammen mit der beobachteten Auftretenswahrscheinlichkeit.

Das erste Programm initialisiert die Variable **x** zunächst mit dem größten darstellbaren Integer-Wert **INT_MAX** und beschreibt anschließend die Variable **y** mit dem Wert **x+1**. Aufgrund der Bereichsüberschreitung verursacht die Addition auf der Mehrzahl der Hardware-Architekturen einen numerischen Überlauf. Legen wir, wie auf den meisten Architekturen üblich, die Zweierkomplementdarstellung zugrunde, so entspricht der Wert von **y** nach der Zuweisung der kleinsten darstellbaren Zahl **INT_MIN**. Kurzum: Der entstandene arithmetische Fehler ist maximal. Weiter wollen wir für dieses Beispiel annehmen, dass der Fehler in jährlichen Abständen auftritt.

Das zweite Programm initialisiert die Variable **x** ebenfalls mit dem Wert **INT_MAX**, führt die Addition im Gegensatz zum ersten Beispiel jedoch gesättigt aus. Hierzu wird der drohende Überlauf mit Hilfe eines zusätzlich eingefügten If-Befehls abgefangen und die Addition nur dann ausgeführt, wenn der Ergebniswert innerhalb des Integer-Zahlenbereichs liegt. Im Falle eines drohenden Überlaufs wird der Wert von **x** unverändert in die Variable **y** übernommen. Obwohl das Programm immer noch ein falsches Ergebnis berechnet, fällt der arithmetische Fehler im Gegensatz




Beispiel 1	Beispiel 2	Beispiel 3
<pre> ... int x = INT_MAX; int y = x+1; ... </pre>	<pre> ... int y; int x = MAX_INT; if (x < MAX_INT) y = x+1; else y = x; ... </pre>	<pre> ... if (x == 47) y = x / 0; ... </pre>
 <p>Symptom: Überlauf Auftritt: Jährlich</p>	 <p>Symptom: Sättigung Auftritt: Jährlich</p>	 <p>Symptom: Absturz Auftritt: Stündlich</p>

Abb. 2.19 Welcher Fehler ist Ihr größter Feind?

zur ersten Lösung um ein Vielfaches geringer aus. Auch hier wollen wir annehmen, dass der Fehler jährlich auftritt.

Im dritten Beispiel wird die Variable x innerhalb des If-Zweigs durch 0 dividiert. Ohne weitere Programmierkniffe löst die Division einen Ausnahme-Interrupt aus, der das Betriebssystem dazu veranlasst, den laufenden Prozess umgehend zu beenden. Weiter wollen wir annehmen, dass der Programmabsturz stündlich auftritt.

Die Frage, die wir an dieser Stelle zu beantworten versuchen, lässt sich plakativ wie folgt formulieren:

„Welcher Fehler ist Ihr größter Feind?“

Haben Sie die Frage für sich bereits beantwortet? Vielleicht gehören auch Sie zu der großen Gruppe von Befragten, die nicht lange zögert und sich nach kurzer Zeit eindeutig auf einen Fehler festlegt. Obwohl die Antworten in der Regel schnell gegeben werden, fallen sie für gewöhnlich denkbar unterschiedlich aus. Bei genauerem Hinsehen ist dieses Ergebnis wenig überraschend, da die korrekte Beantwortung der gestellten Frage ohne weitere Information im Grunde genommen unmöglich ist. Der fehlende Parameter ist die Sichtweise, die wir zur Beurteilung der drei vorgestellten Fehler einnehmen. Je nachdem, ob wir in der Rolle des *Software-Entwicklers* oder in der Rolle des *Anwenders* argumentieren, kommen wir zu völlig unterschiedlichen Ergebnissen.

Die Kriterien, die wir aus Anwendersicht anlegen, sind offensichtlich. Die Schwere eines Fehlers steigt proportional mit seiner Auftretswahrscheinlichkeit und dem Grad seiner Auswirkung. In der Rolle des Anwenders werden wir deshalb den im dritten Beispiel beschriebenen Fehler als unseren größten Feind erachten. Sowohl die Auswirkung in Form des vollständigen Programmabsturzes, als auch die Auftretshäufigkeit sprechen eine eindeutige Sprache.

Aus der Sicht des Software-Entwicklers stellt sich die Situation gänzlich anders dar. Die Auftretswahrscheinlichkeit und der Grad seiner Auswirkung sind

auch hier die bestimmenden Parameter eines Software-Fehlers, allerdings in konträrer Weise. Während für den Anwender die Abwesenheit von Defekten im Vordergrund steht, liegt der Fokus des Entwicklers auf deren Aufdeckung. Die Wahrscheinlichkeit, einen Software-Fehler bereits während der Entwicklung zu identifizieren, steigt proportional mit dessen Auftrittswahrscheinlichkeit. Aufgrund seiner Auftritts- und Auswirkungscharakteristik hätte der dritte hier beschriebene Fehler kaum eine Chance, bis zur Produktauslieferung zu überleben. Die gerade einmal jährliche Auftrittsscharakteristik der Fehler 1 und 2 erweisen sich an dieser Stelle als deutlich kritischer. Das Risiko ist groß, dass beide Fehler unbemerkt die Testphase überstehen.

Wenden wir uns nun der Symptomatik der ersten beiden Fehler zu. Während das Sättigungsprinzip die Überlaufproblematik des zweiten Beispiels effektiv abfedert, zieht der numerische Fehler des ersten Beispiels weitere Kreise. Hier wechselt der Wert von ∞ schlagartig vom größten auf den kleinsten darstellbaren Wert. Als die Schubkraft des Flugzeugs interpretiert, in dem Sie sich gerade befinden mögen, löst die Vorstellung mit Recht ein gewisses Unbehagen aus. Wahrscheinlich würden Sie in diesem Fall die Lösung des zweiten Beispiels vorziehen. Der numerische Fehler wirkt sich durch die gesättigte Addition kaum auf das Endergebnis aus und bleibt für Mensch und Technik mit hoher Wahrscheinlichkeit folgenlos.

So sehr die Abschwächung der Fehlersymptomatik aus Anwendersicht zu begrüßen ist, so sehr erschwert sie die Arbeit des Software-Entwicklers. Treten die Symptome eines Fehlers nur marginal in Erscheinung, so stehen seine Chancen gut, bis in die Produktversion hinein zu überleben. In der Rolle des Software-Entwicklers werden wir daher den zweiten Fehler als unseren größten Feind erachten.

Desaströse Fehler, wie der Totalabsturz in Beispiel 3, sind damit Segen und Fluch zugleich. Dieser duale Charakter, den wir bei der Betrachtung eines Defekts niemals aus dem Auge verlieren dürfen, hat weitreichende Auswirkungen auf das gesamte Gebiet der Software-Qualitätssicherung. Im Grunde genommen müssten Entwicklungs- und Produktversion unterschiedlichen Paradigmen folgen. Programme müssten so formuliert sein, dass vorhandene Software-Fehler während der Entwicklung mit fatalen Auswirkungen in Erscheinung treten, sich in der Produktversion dagegen in Schadensbegrenzung üben. Den Gedanken fortgesetzt bedeutet dieses Vorgehen jedoch nichts anderes, als dass die getestete Software eine andere ist, als die im Feld eingesetzte. Damit wäre eines der wichtigsten Prinzipien der Software-Entwicklung auf eklatante Weise verletzt.

Das Gedankenexperiment offenbart uns ein Grundproblem der Software-Entwicklung und lässt uns erahnen, dass die Software-Qualitätssicherung sowohl zu den schwierigsten als auch zu den spannendsten Herausforderungen gehört, vor die uns das Computerzeitalter heute stellt. Eines zeigt der historische Streifzug dieses Kapitels ganz deutlich: Software-Fehler sind allgegenwärtig und selbst kleinste Ursachen können zu dramatischen Folgen führen – eine Erkenntnis, die im Angesicht heutiger Code-Größen von mehreren Millionen Zeilen Quellcode zu sensibilisieren vermag. Insbesondere vor dem Hintergrund der beharrlich fortschreitenden Computerisierung in Bereichen, in denen Leib und Leben von der Korrektheit der

eingesetzten Hard- und Software-Komponenten unmittelbar abhängen, wird die zukünftige Bedeutung der Software-Qualitätssicherung besonders deutlich.

Trotzdem haben wir keinen Grund zu verzagen: Unsere Analyse hat gezeigt, dass die hier vorgestellten Fehler fast ausnahmslos durch die konsequente Anwendung von einer oder mehreren Methoden der Software-Qualitätssicherung im Vorfeld zu vermeiden gewesen wären. Verlieren wir also keine Zeit und tauchen ein in die Theorie und Praxis der uns heute zur Verfügung stehenden Technologien, die uns im Kampf gegen Software-Fehler zur Seite stehen.



<http://www.springer.com/978-3-642-35699-5>

Software-Qualität

Hoffmann, D.W.

2013, XIV, 568 S. 306 Abb., Softcover

ISBN: 978-3-642-35699-5