Formale Spezifikation mit Z

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

2005-11-24

Übersicht

- Warum formale Spezifikation?
- Die Spezifikationssprache Z
- Die Bausteine von Z
- Fallstudie: Versionskontrolle

Warum formale Spezifikation?

Die meisten Menschen nehmen *Programmfehler* als etwas *unvermeidliches* hin.[1]

Zu den aufgeführten *Gründen* gehören:

- Die *Komplexität* der Aufgabe
- Die Unzulänglichkeit von *Tests*
- Die Mängel der *Umgebungen*
- *Ökonomische* Zwänge
- Fehlende *Grundlagen*

Wir schauen uns diese Gründe einmal näher an.

[1] Dieses und folgende Beispiele entstammen Jonathan Jacky, „The way of Z“, Cambridge University Press, 1997

Verbreitete Irrtümer: Komplexität

Annahme

„Programmierer können niemals alle verschiedenen Möglichkeiten eines Programms betrachten.“

— Chris Peters, Microsoft

Alternative

Software sollte nicht zu komplex sein. Wir brauchen *kompakte und exakte Beschreibungen*, was ein Programm tun sollte.

Verbreitete Irrtümer: Tests

Annahme

„Programme haben immer Fehler, da wir nicht alles testen können.“

— Leonard Lee, Journalist

Alternative

Korrektheit erreicht man durch die Konstruktion – nicht durch Testen. Der Sinn des Testens ist zu prüfen, ob unsere Annahmen über die Programmumgebung stimmen.

Verbreitete Irrtümer: Umgebungen

Annahme

„Was immer wir tun – unsere Programme werden doch Fehler produzieren. Unsere Entwicklungsumgebungen, Betriebssysteme, ja selbst die Hardware kann Fehler haben.“

— Mitch Kapor, Lotus

Alternative

Wenn wir unser Programm verstehen, können wir unsere eigenen Fehler beheben und um die anderen herumarbeiten – bis sie repariert werden oder wir einen besseren Hersteller finden.

Verbreitete Irrtümer: Ökonomie

Annahme

„Für die meisten Anwendungen verlangen Kunden keine hohe Qualität. Schließlich ist es nur Software.“

— *Anonymer Programmierer*

Alternative

Umgang mit schlechter Software ist außerordentlich teuer. Schlechte Software kann große Vermögensschäden oder gar Menschenleben kosten. Weniger spektakulär ist der alltägliche Aufwand für das Erkennen und Vermeiden von Problemen.

Verbreitete Irrtümer: Grundlagen

Annahme

„Architekten machen wenig Fehler. Warum wir? Weil wir jedes mal wieder den ersten Wolkenkratzer bauen.“

— *Bill Gates, Microsoft*

Alternative

Informatik ist eine gereifte Wissenschaft. Wir können auf Jahrhunderte mathematischer und algorithmischer Grundlagen zurückblicken. Wir können – ja, wir *müssen* viel besser sein.

Die Spezifikationssprache Z…

- ist eine Sprache zur Beschreibung *mathematischer Sachverhalte*
- dient vor allem zur Beschreibung von *Rechnersystemen* (Software wie Hardware).
- entwickelt 1977–1990 von der Universität Oxford und industriellen Partnern (IBM, Inmos)
- standardisiert (ANSI, BSI, ISO)
- hat ihren Namen von *Ernst Zermelo* (aus der axiomatischen Mengentheorie von Zermelo-Fraenkel)
- ist heute die in der Praxis verbreitetste formale Spezifikationssprache

Ein erstes Beispiel in Z

Wir betrachten die C-Funktion f:

```
int f(int a)
{
    int i, term, sum;

    term = 1; sum = 1;
    for (i = 0; sum <= a; i++) {
        term = term + 2;
        sum = sum + term;
    }
    return i;
}
```

Was tut dieser Code?

Ein erstes Beispiel in Z (2)

Hier noch einmal, aber mit etwas Dokumentation:

```
int iroot(int a)  // Ganzzahlige Wurzel
{
    int i, term, sum;

    term = 1; sum = 1;
    for (i = 0; sum <= a; i++) {
        term = term + 2;
        sum = sum + term;
    }
    return i;
}
```

Funktioniert dieser Code?

Ein erstes Beispiel in Z (3)

Name und Kommentar für `iroot` sind nicht so hilfreich, wie man annehmen könnte:

- Manche Zahlen haben keine ganzzahligen Wurzeln. Was passiert, wenn wir `iroot(3)` aufrufen?
- Für negative Zahlen sind Wurzeln (in $\mathbb{R}$) nicht definiert. Was passiert, wenn wir `iroot(-4)` aufrufen?

Fazit: Name und Kommentar genügen nicht, um das Verhalten vollständig zu beschreiben.

Ein erstes Beispiel in Z (4)

Hier ist eine Spezifikation für `iroot` – in einem Z-*Absatz:*

$$
\begin{array}{|l}
iroot : \mathbb{N} \to \mathbb{N} \\
\hline
\forall\, a : \mathbb{N} \bullet iroot(a) * iroot(a) \leq a < (iroot(a) + 1) * (iroot(a) + 1)
\end{array}
$$

Diese *axiomatische Definition* ist als Absatz eingerückt und (typischerweise) Teil eines größeren Textes.

Wir betrachten die einzelnen Teile der Definition.

Ein erstes Beispiel in Z (5)

Die *Deklaration* von *iroot*

$$iroot : \mathbb{N} \to \mathbb{N}$$

entspricht der C-Deklaration

```
int iroot(int a)
```

Man bemerke: `iroot` erhält keine negativen Zahlen und gibt auch keine zurück.

Ein erstes Beispiel in Z (6)

Das *Prädikat*

$$\forall\, a : \mathbb{N} \bullet iroot(a) * iroot(a) \leq a < (iroot(a) + 1) * (iroot(a) + 1)$$

zeigt, dass *iroot* die *größte* ganzzahlige Wurzel zurückliefert:

$$\begin{aligned} iroot(3) &= 1 \\ iroot(4) &= 2 \\ iroot(8) &= 2 \\ iroot(9) &= 3 \end{aligned}$$

Das Prädikat entspricht der C-Funktionsdefinition – sagt aber lediglich, *was iroot* tut, nicht jedoch, *wie iroot* dies tut.

Spezifikation eines Texteditors

Wir betrachten einen einfachen Texteditor.

Wir können

- Text eingeben
- den Cursor nach links und rechts bewegen
- das Zeichen rechts vom Cursor löschen.

Grundtypen

Wir definieren einen Zeichentyp als *Grundtyp* in [...]:

$$[CHAR]$$

Wir machen keine weiteren Aussagen über $CHAR$ – schließlich ist dies eine Spezifikation!

Wir führen $TEXT$ als *Abkürzung* für eine Zeichenfolge ein:

$$TEXT == \mathrm{seq}\, CHAR$$

Abkürzungen sind wie *Makros* in herkömmlichen Programmiersprachen.

Axiomatische Definition

Eine *axiomatische Definition* definiert Konstanten für die gesamte Spezifikation – hier die Größe des Textes:

$$
\begin{array}{|l}
maxsize : \mathbb{N} \\
\hline
maxsize \leq 65535
\end{array}
$$

maxsize ist zwar eine Konstante mit definierten Eigenschaften, ihr exakter Wert ist jedoch nicht festgelegt.

Auch die Funktion *iroot* wurde in Z als Konstante definiert.

Ein Editor-Schema

Wir modellieren den Texteditor mit zwei Dokumenten: *left* steht *vor* der aktuellen Cursor-Position, *right* ist *danach*.

Der Zustand wird durch ein *Schema* beschrieben:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Editor}} \\
\hline
left, right : TEXT \\
\hline
\#(left \frown right) \leq maxsize \\
\hline
\end{array}
$$

$\frown$: Konkatenation zweier Folgen
#: Anzahl der Elemente

Schemata

Ein *Schema* beschreibt einen *Aspekt* des spezifizierten Systems.

Ein Schema besteht aus

Name. Identifiziert das Schema (oft auch Typname!).

Deklarationsteil. Führt lokale *Zustandsvariablen* ein

Prädikatsteil. Beschreibt

- *Zustandsinvarianten* sowie
- *Beziehungen*
  - zwischen Zustandsvariablen selbst oder
  - zwischen Zustandsvariablen und Konstanten

Initialisierung

Jedes System kennt einen besonderen *Startzustand*, der in Z traditionell *Init* genannt wird:

$$\begin{array}{|l}\hline \textit{Init} \\ \hline \textit{Editor} \\ \hline \textit{left} = \textit{right} = \langle\,\rangle \\ \hline \end{array}$$

$\langle\,\rangle$: leere Sequenz

Das *Init*-Schema *schließt* das *Editor*-Schema *ein*
$\Rightarrow$ Alle Definitionen aus *Editor* sind in *Init* verfügbar.

Das Einschließen ermöglicht *inkrementelles* Spezifizieren.

Druckbare Zeichen

Wir möchten das *Einfügen* eines Zeichens modellieren.

Hierfür definieren wir *printing* als eine Menge druckbarer Zeichen (als axiomatische Definition ohne Prädikate):

$$\begin{array}{|l} \textit{printing} : \mathbb{P}\,\textit{CHAR} \end{array}$$

$\mathbb{P}$: Potenzmenge (= Menge der Untermengen)

Einfüge-Operation

Das Einfügen wird modelliert durch ein *Operations-Schema*.

Operations-Schemata definieren die Wirkung von *Funktionen:*

$$
\begin{array}{|l}
\hline
\textit{Insert} \\
\Delta Editor \\
ch? : CHAR \\
\hline
ch? \in printing \\
left' = left \frown \langle ch? \rangle \\
right' = right \\
\hline
\end{array}
$$

$\Delta Editor$: Operations-Schema auf *Editor*
ch?: Eingabevariable
$ch? \in printing$: Vorbedingung
$left', right'$: Zustand *nach* der Operation

Cursor bewegen

Wir definieren ein *Steuerzeichen*…

$$
\begin{array}{|l}
right\_arrow : CHAR \\
\hline
right\_arrow \notin printing
\end{array}
$$

…und die entsprechende Operation:

$$
\begin{array}{|l}
\hline
\textit{Forward} \\
\Delta Editor \\
ch? : CHAR \\
\hline
ch? = right\_arrow \\
left' = left \frown head(right) \\
right' = tail(right) \\
\hline
\end{array}
$$

Warum funktioniert diese Definition nicht immer?

Cursor bewegen (2)

Wir erweitern die ursprüngliche Definition um eine weitere *explizite Vorbedingung*:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Forward}} \\
\Delta \mathit{Editor} \\
ch? : \mathit{CHAR} \\
\hline
ch? = \mathit{right\_arrow} \\
\mathit{right} \neq \langle\,\rangle \\
\mathit{left}' = \mathit{left} \frown \mathit{head}(\mathit{right}) \\
\mathit{right}' = \mathit{tail}(\mathit{right}) \\
\hline
\end{array}
$$

Forward bleibt aber *partiell:* es funktioniert nur unter bestimmten (Vor-)Bedingungen.

Cursor bewegen (3)

Ziel: *Forward* soll unter allen Bedingungen funktionieren.

Wir definieren die spezielle Bedingung „Cursor am Ende des Textes“...

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{EOF}} \\
\mathit{Editor} \\
\hline
\mathit{right} = \langle\,\rangle \\
\hline
\end{array}
$$

...sowie die Bedingung „Das Zeichen ist Cursor-nach-rechts“:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{RightArrow}} \\
ch? : \mathit{CHAR} \\
\hline
ch? = \mathit{right\_arrow} \\
\hline
\end{array}
$$

Cursor bewegen (4)

Nun definieren wir eine *totale Forward*-Operation:

$$T\_forward \mathrel{\widehat{=}} \mathit{Forward} \vee (\mathit{EOF} \wedge \mathit{RightArrow} \wedge \Xi \mathit{Editor})$$

$\widehat{=}$: definiert ein Schema aus bestehenden Schemata
$\wedge$: kombiniert Zustände und Operationen
$\vee$: trennt Alternativen
Ξ: Schema bleibt unverändert

Übung: Ergänzen Sie den Editor!

Die Bausteine von Z

Wie jede modellbasierte Spezifikationssprache kennt Z zahlreiche Typkonstruktoren und Operatoren:

- Mengen und Deklarationen
- Tupel und Abbildungen
- Aufzählungen und Folgen
- Logische Prädikate

Mengen und Deklarationen

Mengen $\{red, yellow, green\}$

Deklarationen $i : \mathbb{Z} \quad signal : LAMP$

Tupel $EMPLOYEE == ID \times NAME \times DEPARTMENT$

$$
\begin{array}{|l}
Andreas, Valentin : EMPLOYEE \\
\hline
Andreas = (0019, andreas, informatik) \\
Valentin = (0020, valentin, informatik)
\end{array}
$$

Paare und Abbildungen

Paare $(0019, andreas)$

Abbildungen alternative Schreibweise für Paare: $0019 \mapsto andreas$

$$
\begin{array}{|l}
phone : NAME \leftrightarrow PHONE \\
\hline
phone = \{ \\
\quad naomi \mapsto 64011, \\
\quad andreas \mapsto 64011, \\
\quad andreas \mapsto 64012, \\
\quad valentin \mapsto 64013, \\
\quad \vdots \\
\}
\end{array}
$$

$$
\begin{array}{l}
\operatorname{dom} phone = \{\ldots andreas, valentin, \ldots\} \\
\operatorname{ran} phone = \{\ldots 64011, 64012, 64013 \ldots\}
\end{array}
$$

Mehr über Abbildungen

Nachschlagen $phone(\!| \{valentin, naomi\} |\!) = \{64011, 64013\}$

Einschränkung des Definitionsbereichs
$\{andreas, naomi\} \lhd phone = \{andreas \mapsto 64011, andreas \mapsto 64012, naomi \mapsto 64011\}$

Einschränkung des Wertebereichs
$phone \rhd \{64011\} = \{andreas \mapsto 64011, naomi \mapsto 64011\}$

Aktualisieren
$phone \oplus \{valentin \mapsto 64014\} = \{andreas \mapsto 64011, andreas \mapsto 64012, valentin \mapsto 64014\}$

Aufzählungen und Folgen

Aufzählungen als *Typdefinition* (ohne Ordnung)
$DAYS ::= fri \mid mon \mid sat \mid sun \mid thu \mid tue \mid wed$

Folge per axiomatischer Definition:

$$\begin{array}{|l} weekday : \mathrm{seq}\, DAYS \\ \hline weekday = \langle mon, tue, wed, thu, fri \rangle \end{array}$$

Zugriff
$head(weekday) = mon$
$week == sun \frown weekday \frown sat$

Folgen sind Abbildungen (Funktionen) von ganzen Zahlen, beginnend mit 1:
$weekday(3) = wed$

Logik

Prädikate grenzen die Menge der möglichen Zustände ein:

$$\begin{array}{|l} d_1, d_2 : 1 \mathinner{..} 6 \\ \hline d_1 + d_2 = 7 \end{array}$$

Quantifizierer

$$\begin{array}{|l} divides : \mathbb{Z} \leftrightarrow \mathbb{Z} \\ \hline \forall\, d, n : \mathbb{Z} \bullet d \;\underline{\mathrm{divides}}\; n \Leftrightarrow n \bmod d = 0 \end{array}$$

Binärrelationen (wie hier *divides*) können stets infix geschrieben werden.

Weitere Quantifizierer: $\exists$ und $\exists_1$

Boolesche Werte

In Z gibt es *keinen Datentyp für Boolesche Werte.*

Grund – Boolesche Werte führen häufig zu unleserlichen Spezifikationen

$$BOOLEAN ::= true \mid false$$

$$\begin{array}{|l} beam, door : BOOLEAN \\ \hline beam \Rightarrow door \qquad ??? \end{array}$$

Besser:

$$\begin{aligned} BEAM &::= off \mid on \\ DOOR &::= closed \mid open \end{aligned}$$

und wir können $beam = on \Rightarrow door = closed$ schreiben.

Fallstudie: Versionskontrolle

Viele Werkzeuge zur Versionskontrolle arbeiten mit *Sperren:* Zu jeder Zeit darf nur eine Person das Dokument bearbeiten.

Wir modellieren ein solches System in Z. Zunächst definieren wir Zugriffsrechte für Dokumente:

$$[PERSON, DOCUMENT]$$

$$\begin{array}{|l} permission : DOCUMENT \leftrightarrow PERSON \end{array}$$

Versionskontrolle (2)

Beispiel:

$$\begin{array}{|l} doug, aki, phil : PERSON \\ spec, design, code : DOCUMENT \end{array}$$

$$permission = \{(spec, doug), (design, doug), (design, aki), \ldots\}$$

Wir modellieren, wer welches Dokument in Bearbeitung hat:

$$\begin{array}{|l} \hline Documents \\ checked\_out : DOCUMENT \nrightarrow PERSON \\ \hline checked\_out \subseteq permission \\ \hline \end{array}$$

$\nrightarrow$: partielle Funktion

Versionskontrolle (3)

Ein Schema für den *check-out* eines Dokumentes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CheckOut}} \\
\Delta Documents \\
p? : PERSON \\
d? : DOCUMENT \\
\hline
d? \notin \operatorname{dom} checked\_out \\
(d?, p?) \in permission \\
checked\_out' = checked\_out \cup \{(d?, p?)\} \\
\hline
\end{array}
$$

Versionskontrolle (4)

CheckOut muss wieder eine totale Operation werden:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CheckedOut}} \\
\Xi Documents \\
d? : DOCUMENT \\
\hline
d? \in \operatorname{dom} checked\_out \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Unauthorized}} \\
\Xi Documents \\
p? : PERSON \\
d? : DOCUMENT \\
\hline
(d?, p?) \notin permission \\
\hline
\end{array}
$$

$$T\_CheckOut \mathrel{\widehat{=}} CheckOut \lor CheckedOut \lor Unauthorized$$

Zusammenfassung

- In Z wird das Verhalten eines Systems durch *Schemata* beschrieben
- Ein Schema beschreibt einen Aspekt des spezifizierten Systems
- Schemata lassen sich zu größeren Schemata zusammensetzen:
  - ermöglicht inkrementelles Arbeiten
  - und inkrementelle Darstellung
- Reiche Menge an Grundtypen und Operationen verfügbar
- Grundlage für Programmbeweise

Literatur

The Way of Z (Jonathan Jacky) – alle hier beschriebenen Beispiele und Tutorials

The Z Notation (http://www.comlab.ox.ac.uk/archive/z.html) – Die Z-Notation

The Z Glossary (ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zglossary.ps.Z) – Z-Glossar

Fuzz Type Checker (http://spivey.oriel.ox.ac.uk/mike/fuzz/) – Typprüfer und LaTeX-Makros für Linux