Why Formal Specification?

Most people accept bugs as something unavoidable. The reasons for this are:

- **Complexity** of the task.
- Insufficiency of the tests.
- Deficiencies in the Environment.
- **Economic** constraints.
- Lack of foundations.

We look at these reasons more closely.
Popular Fallacies: Complexity

Assumption

“Programmers can never imagine the countless ways in which a program can fail, or the many different ways a program is used.”

— Leonard Lee, Journalist

Alternative

Software shouldn’t be too complicated. We can produce a compact description that explains how a program is supposed to behave.

Popular Fallacies: Tests

Assumption

“There are always particular combinations of circumstances that have somehow eluded the test plan.”

— Mitch Kapor, Lotus
Popular Fallacies: Tests

Assumption

“There are always particular combinations of circumstances that have somehow eluded the test plan”

— Mitch Kapor, Lotus

Alternative

Software must be made correct by construction – not testing. The proper role of testing is to confirm our understanding of the requirements and the environment.

Popular Fallacies: Environment

Assumption

No matter what we do, our programs will sometimes fail anyway. There are bugs in our compilers, operating systems, window managers, and all the other system software on which we depend.
Assumption

For most applications, the market does not demand high quality. After all, „it is only software“.

— Anonymous Programmer

Alternative

Coping with mediocre software is expensive. Mediocre software can cause severe financial losses or even cost lives. Less spectacular is the daily effort spend to identify and prevent those problems.
Popular Fallacies: Foundations

Assumption

„The number of times civil engineers make mistakes is very small. And at first you might think, what’s wrong with us? It’s because it’s like we’re building the first skyscraper every time.“
— Bill Gates, Microsoft

Alternative

Computing is a mature science. We can build on a legacy of logical ingenuity that reaches back to antiquity. We can – yes, we have to become better.

The specification notation Z...

• is a set of conventions for presenting mathematical text
• describes computing systems (hardware as well as software).
• was developed 1977–1990 at the University of Oxford with industrial partners (IBM, Inmos)
• is standardized (ANSI, BSI, ISO)
• has its name from Ernst Zermelo (axiomatic set theory of Zermelo-Fraenkel)
• is the most widespread specification language today

A First Example in Z

We look at the following C function:

```c
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;
}
```

What does this code do?
A First Example in Z

Again, but with documentation:

```c
int iroot(int a) // Integer square root
{
    int i, term, sum;
    term = 1; sum = 1;
    for (i = 0; sum <= a; i++) {
        term = term + 2;
        sum = sum + term;
    }
    return i;
}
```

How does this code work?

A First Example in Z

The name and the comment for `iroot` are not as helpful as they might seem.

- Some numbers don’t have integer square roots. What happens if you call `iroot` with a set to 3?

A First Example in Z

The name and the comment for `iroot` are not as helpful as they might seem.

- Some numbers don’t have integer square roots. What happens if you call `iroot` with a set to 3?

- For negative numbers integer square roots are not defined (in $\mathbb{R}$). What happens if you call `iroot` with a set to -4?
A First Example in Z

The name and the comment for iroot are not as helpful as they might seem.

- Some numbers don’t have integer square roots. What happens if you call iroot with a set to 3?
- For negative numbers integer square roots are not defined (in R). What happens if you call iroot with a set to -4?

Conclusion: Names and comments are not enough to describe the behavior completely.

A First Example in Z

Here is a specification for iroot – in a Z-paragraph:

\[
iroot : \mathbb{N} \rightarrow \mathbb{N} \\
\forall a : \mathbb{N} \cdot iroot(a) \ast iroot(a) \leq a < (iroot(a) + 1) \ast (iroot(a) + 1)
\]

This axiomatic definition is as a paragraph indented and (typically) the part of an bigger text.

Let’s look at the different parts of the definition.

A First Example in Z

The declaration of iroot

\[
iroot : \mathbb{N} \rightarrow \mathbb{N}
\]

corresponds to the C-declaration

\[
\text{int iroot(int a)}
\]

Recognize: iroot doesn’t receive negative numbers and also returns none.
A First Example in Z

The Predicate

\( \forall a : \mathbb{N} \ni \text{iroot}(a) \ast \text{iroot}(a) \leq a < (\text{iroot}(a) + 1) \ast (\text{iroot}(a) + 1) \)

shows that \( \text{iroot} \) returns the biggest integer square root:

\[
\begin{align*}
\text{iroot}(3) &= 1 \\
\text{iroot}(4) &= 2 \\
\text{iroot}(8) &= 2 \\
\text{iroot}(9) &= 3
\end{align*}
\]

The predicate corresponds to the C function definition – it describes only what the function does without explaining how to do it.

Specifying a Text Editor

Let’s look at a simple text editor.

We can

- insert text
- move the cursor right and left
- delete the character in front of the cursor

Basic Types

We declare a basic type [...] as the set of all characters:

CHARR

We make no further statement about CHAR – because this is a specification!
Basic Types

We declare a \textit{basic type} [...] as the set of all characters:

\[ [\text{CHAR}] \]

We make no further statement about \textit{CHAR} – because this is a specification!

We introduce \textit{TEXT} as an \textit{abbreviation definition} for a sequence of characters:

\[ \text{TEXT} = \text{seq CHAR} \]

Abbreviations are like \textit{macros} in traditional programming languages.

Axiomatic Descriptions

An \textit{axiomatic description} defines a constant for the whole specification – such as the size of the text:

\[
\begin{align*}
\text{maxsize} & : \mathbb{N} \\
\text{maxsize} & \leq 65535
\end{align*}
\]

\textit{maxsize} is a constant, however we haven’t specified its value.

In Z, functions such as \textit{iroot} are a kind of a constant.

A State Schema for the Editor

We model the document of the text editor as two texts: \textit{left} is the text \textit{before} the cursor, and \textit{right} is text following it.

Here is the state schema for the editor:

\[
\begin{array}{l}
\text{Editor} \\
\hline
\text{left, right} : \text{TEXT} \\
\#(\text{left} \text{^ right}) \leq \text{maxsize}
\end{array}
\]

\text{^}: concatenation operator
\# : count operator
Schemas

A schema describes an aspect of the specified system. 
A schema consists of:

Name. Identifies the schema (often type name!).

Declaration part. Introduces local state variables.

Predicate part. Describes
- State invariants as well as
- Relations
  - between the state variables themselves or
  - between state variables and constants

Initialization Schemas

Every system has a special state in which it starts up. In Z this state is described by a schema conventionally named Init.

Init = left = right = ⟨⟩

⟨⟩: empty sequence
The Init schema includes the Editor schema.
⇒ All definitions from Editor apply to Init as well.

The inclusion allows an incremental specification.

Printing Characters

We want to model the insertion of a character. 
Therefore, we define printing as the set of printing characters. 
(as axiomatic definition without predicates)

| printing : P CHAR

P: Power set (= set of all subsets)
Insert Operation

The insertion is modeled by an operation schema. Operation schemas define the effect of functions.

```
Insert
\[ \Delta_{Editor} \]
ch?: CHAR
\[ ch? \in \text{printing} \]
left' = left \cap (ch?)
right' = right
```

\( \Delta_{Editor} \): Operation schema on Editor
ch?: Input variable
ch? \in \text{printing}: Precondition
left', right': State after the operation

Moving the Cursor

We define a control character…

```
right\_arrow: CHAR
right\_arrow \notin \text{printing}
```

…and the corresponding operation:

```
Forward
\[ \Delta_{Editor} \]
ch?: CHAR
\[ ch? = right\_arrow \]
left' = left \cap head(right)
right' = tail(right)
```

Why does this definition not always work?
Moving the Cursor

We extend the definition with an additional explicit precondition.

\[
\begin{align*}
\text{Forward} & \\
\Delta \text{Editor} & \\
\text{ch?} : \text{CHAR} & \\
\text{ch?} = \text{right\_arrow} & \\
\text{right} \neq \langle \rangle & \\
\text{left'} = \text{left} \uparrow \text{head(right)} & \\
\text{right'} = \text{tail(right)} &
\end{align*}
\]

Goal: Forward should work in all situations.

We define a special „end of file“ condition ...

\[
\begin{align*}
\text{EOF} & \\
\text{Editor} & \\
\text{right} = \langle \rangle &
\end{align*}
\]

...as well as „character is right arrow“.

\[
\begin{align*}
\text{RightArrow} & \\
\text{ch?} : \text{CHAR} & \\
\text{ch?} = \text{right\_arrow} &
\end{align*}
\]
Moving the Cursor

Now we define the total Forward operation:

\[ T_{\text{forward}} \equiv \text{Forward} \lor (\text{EOF} \land \text{RightArrow} \land \Xi) \]

\(\Xi\): defines a schema consisting of schemas
\(\land\): combines states an operations
\(\lor\): distinct alternatives
\(\Xi\): schema remains unchanged

Exercise: Complete the specification of the text editor!

---

The Elements of Z

As every model based specification language, Z has numerous type constructors and operations.

- Sets and Declarations
- Tuples and Binary Relations
- Enumerations and Sequences
- Predicates

---

Sets and Declarations

**Sets** \{red, yellow, green\}
Sets and Declarations

Sets \{red, yellow, green\}

Declarations \( i : \mathbb{Z} \quad \text{signal} : \text{LAMP} \)

Tuple \( \text{EMPLOYEE} = \text{ID} \times \text{NAME} \times \text{DEPARTMENT} \)

Andreas, Rahul : \text{EMPLOYEE}

\[
\begin{align*}
\text{Andreas} & = (0019, \text{andreas}, \text{computer}\_\text{science}) \\
\text{Rahul} & = (0020, \text{rahul}, \text{computer}\_\text{science})
\end{align*}
\]

Pairs and Binary Relations

Pairs \((0019, \text{andreas})\)

Binary Relations alternative notation for pairs:

\[
\begin{align*}
0019 & \leadsto \text{andreas} \\
\text{phone} & : \text{NAME} \leadsto \text{PHONE} \\
\text{phone} & = \{ \\
\text{naomi} & \leadsto 64011, \\
\text{andreas} & \leadsto 64011, \\
\text{andreas} & \leadsto 64012, \\
\text{rahul} & \leadsto 64013,
\}
\end{align*}
\]
Pairs and Binary Relations

Pairs (0019, andreas)

Binary Relations alternative notation for pairs:

\[
\text{phone : NAME} \rightarrow \text{PHONE} \\
\text{phone} = \\
\quad \{ \\
\quad \quad \text{naomi} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64012, \\
\quad \quad \text{rahul} \rightarrow 64013, \\
\quad \} \\
\]

dom phone

43

Pairs and Binary Relations

Pairs (0019, andreas)

Binary Relations alternative notation for pairs:

\[
\text{phone : NAME} \rightarrow \text{PHONE} \\
\text{phone} = \\
\quad \{ \\
\quad \quad \text{naomi} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64012, \\
\quad \quad \text{rahul} \rightarrow 64013, \\
\quad \} \\
\]

dom phone = \{...andreas, rahul,...\}

44

Pairs and Binary Relations

Pairs (0019, andreas)

Binary Relations alternative notation for pairs:

\[
\text{phone : NAME} \rightarrow \text{PHONE} \\
\text{phone} = \\
\quad \{ \\
\quad \quad \text{naomi} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64011, \\
\quad \quad \text{andreas} \rightarrow 64012, \\
\quad \quad \text{rahul} \rightarrow 64013, \\
\quad \} \\
\]

dom phone = \{...andreas, rahul,...\}

ran phone

45
Pairs and Binary Relations

Pairs (0019, andreas)

Binary Relations alternative notation for pairs:

0019 $\rightarrow$ andreas

<table>
<thead>
<tr>
<th>phone : NAME $\rightarrow$ PHONE</th>
</tr>
</thead>
</table>
| phone = {{
|     naomi $\rightarrow$ 64011,
|     andreas $\rightarrow$ 64011,
|     andreas $\rightarrow$ 64012,
|     rahul $\rightarrow$ 64013,
|     |
| }}

\[\text{dom } phone = \{ \ldots \text{andreas, rahul,} \ldots \}\]

\[\text{ran } phone = \{ \ldots 64011,64012,64013 \ldots \}\]

---

Operators for Relations

Lookup phone(\{rahul, naomi\})

\[\text{Lookup } phone(\{rahul, naomi\}) = \{64011,64013\}\]
Operators for Relations

**Lookup** \( \text{phone}(\{ \text{rahul, naomi} \}) = \{64011, 64013\} \)

**Domain Restriction**
\( \{ \text{andreas, naomi} \} \prec \text{phone} \)

**Range Restriction**
\( \text{phone} \succ \{64011\} \)
Operators for Relations

**Lookup** \( \text{phone}\{(rahul, naomi)\} = \{64011, 64013\} \)

**Domain Restriction**
\( \{\text{andreas, naomi}\} \bowtie \text{phone} = \{\text{andreas} \rightarrow 64011, \text{andreas} \rightarrow 64012, \text{naomi} \rightarrow 64011\} \)

**Range Restriction**
\( \text{phone} \triangleright \{64011\} = \{\text{andreas} \rightarrow 64011, \text{naomi} \rightarrow 64011\} \)

**Update**
\( \text{phone} \oplus \{\text{rahul} \rightarrow 64014\} = \{\text{andreas} \rightarrow 64011, \text{andreas} \rightarrow 64012, \text{rahul} \rightarrow 64014\} \)
Enumerations and Sequences

Enumerations as type definition (no order)
\[ DAYS ::= \text{fri} | \text{mon} | \text{sat} | \text{sun} | \text{thu} | \text{tue} | \text{wed} \]

Sequence axiomatic definition
\[
\begin{align*}
\text{weekday} &: \text{seq} DAYS \\
\text{weekday} &= \langle \text{mon}, \text{tue}, \text{wed}, \text{thu}, \text{fri} \rangle
\end{align*}
\]

Operators
\[ \text{head}(\text{weekday}) = \text{mon} \]
Enumerations and Sequences

Enumerations as type definition (no order)
\[ \text{DAYS ::= fri | mon | sat | sun | thu | tue | wed} \]

Sequence axiomatic definition
\[
\text{weekday : seq DAYS} \\
\text{weekday = (mon, tue, wed, thu, fri)}
\]

Operators
\[ \text{head(weekday) = mon} \]
\[ \text{week == sun} \uparrow \text{weekday} \uparrow \text{sat} \]

Sequences are just functions whose domains are consecutive numbers, starting with one.
\[ \text{weekday(3) = wed} \]

Logic

Predicates restrict the set of possible states.
\[
\begin{align*}
\text{d}_1, \text{d}_2 & : 1 \ldots 6 \\
\text{d}_1 + \text{d}_2 = 7
\end{align*}
\]

Quantifiers
\[
\text{divides : } \mathbb{Z} \rightarrow \mathbb{Z} \\
\forall \text{d, n} : \mathbb{Z} \cdot \text{d divides } n \iff n \text{ mod } d = 0
\]

Binary relations (e.g. divides) can be written in infix syntax.
Further quantifiers: \( \exists \) and \( \exists_1 \)
Boolean Types

Z has no built-in Boolean type.
Reason – Boolean types often lead to unreadable specifications.

\[
BOOLEAN ::= \text{true} \mid \text{false}
\]

beam, door : BOOLEAN
beam ⇒ door ???

Better:

BEAM ::= \text{off} \mid \text{on}
DOOR ::= \text{closed} \mid \text{open}

so we can write beam = on ⇒ door = closed
Case Study: Revision Control

Many revision control systems use locks. This means that at any time only one person can edit the file.

We model such a system in Z.

First, we define the permissions for the documents:

\[
\text{permission} : \text{DOCUMENT} \rightarrow \text{PERSON}
\]

Example:

\[
doug, aki, phil : \text{PERSON} \\
\text{spec, design, code : DOCUMENT}
\]

\[
\text{permission} = \{ (\text{spec, doug}), (\text{design, doug}), (\text{design, aki}), \ldots \}
\]

We model who has checked out a document:

\[
\text{Documents} \\
\text{checked\_out} : \text{DOCUMENT} \rightarrow \text{PERSON} \\
\text{checked\_out} \subseteq \text{permission} \\
\rightarrow : \text{partial function}
\]

A schema for the check-out of a document:

\[
\text{CheckOut} \\
\Delta \text{Documents} \\
p? : \text{PERSON} \\
d? : \text{DOCUMENT} \\
d? \notin \text{dom checked\_out} \\
(d?, p?) \in \text{permission} \\
\text{checked\_out}' = \text{checked\_out} \cup \{(d?, p?)\}
\]
Case Study: Revision Control

CheckOut needs to become a total operation:

\[
\text{CheckedOut} 
\begin{align*}
\exists \text{Documents} \\
d? &: \text{DOCUMENT} \\
d? &\in \text{dom checked\_out}
\end{align*}
\]

\[
\text{Unauthorized} 
\begin{align*}
\exists \text{Documents} \\
p? &: \text{PERSON} \\
d? &: \text{DOCUMENT} \\
(d?, p?) &\not\in \text{permission}
\end{align*}
\]

\[T_{\text{CheckOut}} \doteq \text{CheckOut} \lor \text{CheckedOut} \lor \text{Unauthorized}\]

Summary

- Z describes the behavior of a system by schemas.
- A schema describes an aspect of the specified system.
- Schemas can be composed to bigger schemas.
  - allows incremental definition
  - and incremental presentation
- Rich set of built-in types and operators.
- Basis for program proofs.
Literature

The Way of Z (Jonathan Jacky)
- all described examples and tutorials

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

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

Fuzz Type Checker
(http://spivey.oriel.ox.ac.uk/mike/fuzz/)
- Type checker and \LaTeX\ macros for Linux