Formal Modeling with Z: An Introduction

Luca Viganò

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002
Overview

• Today:
  – Modeling with formal languages: why and how?
  – A standard introductory example.
  – The Z language — first definitions.

• Next classes: Z in detail, the mathematical toolkit, and applications.
• **Goal:** specify the requirements as far as possible, but abstract as possible.

• **Definition:** A model is a construction or mathematical object that describes a system or its properties.
Which Modeling Language?

- There are hundreds! Differences include:

  **System view:** static, dynamic, functional, object-oriented,…
  **Degree of abstraction:** e.g. requirements versus system architecture.
  **Formality:** informal, semi-formal, formal.
  **Religion:** OO-school (OOA/OOD, OMT, Fusion, UML), algebraic specification, Oxford Z/CSP-Sect, Church of HOL,…

- Examples:

  Function trees, data-flow diagrams, E/R diagrams, syntax diagrams, data dictionaries, pseudo-code, rules, decision tables, (variants of) automata, Petri-nets, class diagrams, CRC-cards, message sequence charts,…

- Why are UML or other semi-formal languages not enough?
Disadvantages of Semi-Formal Languages

- Modeling is detailed and intuitive (and "simple", i.e. also for managers and laymen).

- Semantics of models/diagrams is often imprecise.

- Often only syntax.
Example: Problems with E/R-Diagrams

- Are the relations "directed"?
- Several properties cannot be specified graphically (e.g. constraints).
- etc.

We will employ Z to formalize semi-formal diagrams and models.
Formal Languages

• A language is formal if its syntax and semantics are defined formally (mathematically).

• Formal languages allow for the design, the development, the verification, the testing and the maintenance of a system:
  – remove ambiguities and introduce precision,
  – structure information at an appropriate abstraction level,
  – support the verification of design properties,
  – are supported by tools and systems.

• Using mathematics (and formal methods) may appear to be expensive, but in the long run it pays off (and how!).
Z ("zed")

- Is a very expressive formal language.
- Based on first-order logic with equality (PL1=) and typed set-theory.
- Has a mathematical toolkit: a library of mathematical definitions and abstract data-types (sets, lists, bags, ...).
- Supports the structured modeling of a system, both static and dynamic:
  - modeling/specification of data of the system,
  - functional description of the system (state transitions).
- Is supported by several tools and systems.
A number of successfully employed Formal Methods are based on PL1= with type-theory, e.g.

- VDM ("Vienna Development Method", 80's),
- B (applied extensively in France).

Other formal languages:

- Equational logic or Horn logic (in algebraic specifications),
- PL1=,
- Higher-order logic (HOL).

Z:

- Applied successfully since 1989 (Oxford University Computing Laboratory), e.g. British government requires Z-specifications for security-critical systems.
- Is (will soon be) an ISO standard.
A mathematical model that describes the intended behavior of a system is a **formal specification**.

Q: Why do we need such a specification if we can simply write a program? Why not directly implement the program?

A: A program can be quite cryptic, and therefore we need a specification that describes formally the intended behavior at the appropriate abstraction level.
Example: What does the following simple SML-program do?

fun int_root a =
(* integer square root *)
  let val i = ref(0);
    val k = ref(1);
    val sum = ref(1);
  in
    while (!sum <= a) do
      (k := !k+2;
       sum := !sum + !k;
       i := !i+1);
    !i
  end;
An Example (3)

- The program is efficient, short and well-structured.

- The program name and the comment suggest that int_root simply computes the "integer square root" of the input, but is it really the case?

- Moreover: What happens in special input cases, e.g. when the input is 0 or -3?

- Such questions can be answered by code-review (or reverse-engineering), but this requires time and can be problematic for longer programs.

- The key is abstraction: understanding the code must be separated from understanding its “function”.

For example, consider a VCR whose only documentation is the blue-print of its electronic.
A Example (4)

• Solution: we can specify the program in Z.

Formalize what the system must do without specifying/prescribing how.

• We specify int_root in Z by means of a so-called axiomatic definition (or axiomatic description):

\[
\text{int_root} : \mathbb{Z} \rightarrow \mathbb{N}
\]

\[
\forall a : \mathbb{N} \bullet \text{let } y = \text{int_root}(a) \bullet \\
\quad y \times y \leq a < (y + 1) \times (y + 1) \\
\quad \forall a : \mathbb{N} \setminus \{0\} \bullet \text{int_root}(-a) = 0
\]

More about Z \rightarrow implementation in a few weeks.
An Introductory Example: The Birthdaybook

- The Birthdaybook is a small database containing peoples’ names and their birthdays.

- A simple event-model:

![Event model diagram]

- A structured Z-specification in 3 steps:
  1. Define the (auxiliary) functions and types of the system.
  2. Define the state-space of the system.
  3. Define the operations of the system (based on the relations of the state-space).
Step 1. Define the (auxiliary) functions and types of the system:

- Basic types

\[ [\text{NAME}, \text{DATE}] \]

The precise form of names and dates is not important (e.g. strings, 06/03, 03/06, 6.3, 06.03, March 6, 6.Mar, or ...).
The Birthdaybook: Z-Specification (2)

Step 2. Define the state-space of the system using a Z-schema:

\[
\begin{array}{l}
\text{Birthdaybook} \\
\text{known : } \mathbb{P} \text{ NAME} \\
birthday : \text{NAME } \rightarrow \text{DATE} \\
\text{known = dom birthday}
\end{array}
\]

Name of schema declaration of typed variables (represent observations of the state)
relationships between values of vars (are true in all states of the system and are maintained by every operation on it)

Notation and remarks:

- *known* is the set (symbol \(\mathbb{P}\)) of names with stored birthdays,
- *birthday* is a partial function (symbol \(\rightarrow\)), which maps some names to the corresponding birthdays,
- The relation between *known* and *birthday* is the invariant of the system: the set *known* corresponds to the domain (dom) of the function *birthday*.
The Birthdaybook: Z-Specification (3)

- Example of a possible state of the system:

  \[
  \begin{align*}
  \text{known} & = \{\text{Susy, Mike, John}\} \\
  \text{birthday} & = \{\text{John} \mapsto 25.\text{Mar}, \\
  & \quad \text{Susy} \mapsto 20.\text{Dec}, \\
  & \quad \text{Mike} \mapsto 20.\text{Dec}\}
  \end{align*}
  \]

- Invariant \( \text{known} = \text{dom birthday} \) is satisfied:
  - \( \text{birthday} \) stores a date for exactly the three names in \( \text{known} \).

- N.B.:
  - no limit on stored birthdays,
  - no particular (prescribed) order of the entries,
  - each person has only one birthday (\( \text{birthday} \) is a function),
  - two persons can have the same birthday.
Step 3. Define the operations of the system (based on the relations of the state-space).

- Some operations modify the state and some leave it unchanged.

- Some operations have input and/or output:

  ![System diagram]

  - Examples of operations: AddBirthday, FindBirthday, Remind (and Init).
The Birthdaybook: Z-Specification (5)

Add the birthday of a person, who is not yet known to the system:

\[
\begin{align*}
\text{AddBirthday} & \quad \Delta \text{BirthdayBook} \\
\text{name} & \in \text{NAME} \\
\text{date} & \in \text{DATE} \\
\text{name} \notin \text{known} \\
\text{birthday}' & = \\
\text{birthday} \cup \{\text{name} \mapsto \text{date}\}
\end{align*}
\]

- Name of operation (schema) ———
  structured import (symbol $\Delta$)
  input of operation (symbol `?`)
  input of operation (symbol `?`)
  precondition for success of operation
  extend the birthday function
  (if precondition is satisfied)

- This schema modifies the state:
  - it describes the state before (variables without ')
  - and that after the operation (variables with ').

- Note that we do not specify what happens when the precondition is not satisfied.

- It is possible to extend (refine) the specification so that an error message is generated.
The Birthdaybook: Z-Specification (6)

- We expect that \textit{AddBirthday} extends the set of known names with the new name:
  
  $\text{known}' = \text{known} \cup \{\text{name}?>\}$

- We can use the specification of \textit{AddBirthday} to prove this, by exploiting the invariant of the state before and after the operation:

  $\text{known}' = \text{dom} \text{birthday}'$
  $= \text{dom}(\text{birthday} \cup \{\text{name}?> \rightarrow \text{date}?>\})$ \hspace{1cm} [invariant after]
  $= (\text{dom} \text{birthday}) \cup (\text{dom}\{\text{name}?> \rightarrow \text{date}?>\})$ \hspace{1cm} [spec of \textit{Addbirthday}]
  $= (\text{dom} \text{birthday}) \cup \{\text{name}?>\}$ \hspace{1cm} [fact about \text{dom}]
  $= \text{known} \cup \{\text{name}?>\}$ \hspace{1cm} [fact about \text{dom}]  

- Proving such properties ensures that the specification is correct:

  We can analyze the behavior of the system without having to implement it!
Find the birthday of a person known to the system:

\[
\text{FindBirthday} \quad \Xi \text{BirthdayBook} \\
\text{name? : NAME} \\
\text{date! : DATE} \\
\text{name?} \in \text{known} \\
\text{date!} = \text{birthday(name?)}
\]

This schema leaves the state unchanged and is equivalent to:

\[
\text{FindBirthday} \quad \Delta \text{BirthdayBook} \\
\text{name? : NAME} \\
\text{date! : DATE} \\
\text{known}' = \text{known} \\
\text{birthday}' = \text{birthday} \\
\text{name?} \in \text{known} \\
\text{date!} = \text{birthday(name?)}
\]
The Birthdaybook: Z-Specification (8)

• Find out who has his birthday at some particular date:

\[
\text{Remind } \exists \text{BirthdayBook} \text{ today? : DATE cards! : } \mathbb{P} \text{ NAME}
\]

\[
cards! = \{ n \in \text{known} \mid \text{birthday}(n) = \text{today?} \}
\]

cards! is a set of names, to whom "birthday-cards" should be sent.

• Initial state of the system:

\[
\text{InitBirthdayBook } \text{BirthdayBook}
\]

\[
\text{known} = \emptyset
\]

known = \emptyset implies birthday is also empty.
The Birthdaybook: Z-Specification (9)

- What does the Z-specification tell us about the implementation?
- It describes what the system does without specifying/prescribing how.
- For example, the Z-specification identifies **legal** and **illegal** data and operations. Illegal operations are for instance:
  - simultaneous addition of the birthdays of two persons,
  - addition of the birthday of a person who is already known to the system \( \text{name?} \in \text{known} \).
    An operation \( \text{ChangeBirthday} \) is not specified and could be added, or only realized in the implementation.
- More in the next classes.
Summary

• Z is an expressive language (PL1= and typed set-theory).

• Z supports structured, static and dynamic, modeling.

• More about Z:

  http://archive.comlab.ox.ac.uk/z.html

• Tools and systems: see the course webpage.
  – ZETA (an open environment, including a type-checker; emacs Zeta-Mode)
    /usr/local/zeta
  – HOL-Z tool (an embedding of Z in the theorem prover Isabelle).
  – Object-Z (an object-oriented extension of Z).
  – Books about Z and LaTeX style-file.