Lecture 9
Introduction to Z

David Garlan
Carnegie Mellon University

Models of Software Systems
15-671
Fall 1995

State Machine Models: The Story So Far

- Ways of describing state machines
  - > graphically
  - > as a 4-tuple
    - (1) explicit enumeration
    - (2) using predicates
  - > as a special notation
    - (1) pre- & post-conditions
    - (2) input, output, exceptions
- Ways of reasoning about state machines

how?

  - > satisfaction of an invariant
  - > reachability

The Next Five Lectures

- New ways of describing state machines
  - > Z global declarations
  - > Z schemas
- New ways of reasoning
  - > inferring derived properties of a state machine
  - > calculating preconditions
  - > adequacy of initial state
- New ways of relating state machines
  - > refinement/ implementation relationship

    does what we specified

The Mathematics of Z

- Typed set theory and logic
  - sets, relations, functions
  - first-order predicate logic
- Schemas
  - “named” tuples -- like records
- Schema calculus
  - mathematical operators for building bigger schemas out of smaller ones
- Syntactic conventions
  - to make descriptions tractable
  - to relate to programming

Schemas

- The primary means of structuring formal specifications in Z is the use of *schemas*
- Schemas describe
  - > what states a system can occupy
  - > what operations can happen
  - > relationships between parts of a complex system

The Method of Z

- Combine mathematics with prose
  - > to explain the meaning of the symbols
  - > to motivate the design decisions
- Idioms for structuring large specifications
  - > to make each part of the mathematics understandable
  - > to reuse patterns of exposition and pieces of specification
- Focus on abstract properties of a system
  - > rather than algorithms, data structures, etc.

State Machines in Z
Global Variables

STATE ::= patients | field | setup | beam-on

EVENT::= select_patient | select_field
 | enter | start | stop | ok | intlk

FSM == (STATE x EVENT) £ STATE

enumerated types

type abbreviation

| no_change, transitions, control: FSM
|---
| control = no_change – transitions
| no_change = {s: STATE; e: EVENT • (s,e) ¢ e}
| transitions = { (patients, enter) ¢ fields,
| (fields, select_patient) ¢ patients,
| (fields, enter) ¢ setup.
| (setup, select_patient) ¢ patients, ...}

global declarations

Example (continued)

Init: P State

Init = {patients}

Problems with this Approach

- Ok for simple state machines, but ...
- Global variables don't scale
- What about infinite state spaces?
- What about parameters?
- What about exceptions?

State Machines in Z Schemas

A Simple Example[1]

A small database for recording people's names and birthdays.

The system should allow us to:

- add new people to the database
- lookup the birthday of a person
- find the names of people with birthdays on a given day

1. From: J.M. Spivey, *The Z Notation,* 1989

The State Space

The state of a system is described by a schema.

SchemaName

declarations

---

state invariant

BirthdayBook

[NAME, DATE] ← Given Sets

BirthdayBook

known: P NAME

birthday: NAME £ DATE

---

known = *dom* birthday

Example

known = { David, Judy, Robert }

birthday = { David ¢ 24-June,
Judy ¢ 26-August,
Robert ¢ 8-July}

The invariant is satisfied:

known= *dom* birthday

Observation

- There is:
  - > No limit on the number of entries
  - > No implied order of entries
  - > No restriction on format
- But there is a precise statement that:
  - > Each person has only one birthday
  - > Two people may share a birthday
  - > Some people may not be in the database

Initial State

$$
\begin{array}{|l}
\hline
\textbf{InitBirthdayBook} \\
\textbf{BirthdayBook} \\
\hline
\textbf{birthday} = \varnothing \\
\textbf{known} = \varnothing \\
\hline
\end{array}
$$

Operations

```
┌── AddBirthday ─────────────────
│ Δ BirthdayBook
│ name?: NAME
│ date?: DATE
├────────────────────
│
```

Observations:

- state before the operation
- state after the operation
- inputs and outputs

Operations (2)

AddBirthday

Δ **BirthdayBook**

name?: NAME

date?: DATE

known, known': P NAME

birthday, birthday': NAME£ DATE

---

known = ***dom*** **birthday**

known' = ***dom*** **birthday'**

Add Birthday

AddBirthday

Δ **BirthdayBook**

name?: NAME

date?: DATE

---

name? $\notin$ known

birthday' = birthday $\cup$ { name? ¢ date? }

pre-condition

post-condition

Derived Components

The invariant

known = *dom* birthday

allows us to calculate <u>known</u> from <u>birthday</u>.

It is a <u>derived component.</u>

Some useful laws about *dom*:

***dom* (f g) = *dom* f *dom* g**

***dom* { a ¢ b } = { a }**

Reasoning About the Specification

known'

= (after-state invariant)

***dom* birthday'**

= (post-condition)

***dom* (birthday {name? ¢ date?})**

= (law about *dom*)

***dom* birthday *dom* {name? ¢ date?}**

= (law about *dom*)

***dom* birthday {name?}**

= (before-state invariant)

known {name?}

Find Birthday

FindBirthday

BirthdayBook
name?: NAME
date!: DATE

---

name? known
date! = birthday (name?)

Δ BirthdayBook

known = known'
birthday = birthday'

Remind

Remind

BirthdayBook

date?: DATE

names!: $\mathbb{P}$ NAME

---

names! =

{n: known | birthday(n) = date?}

Summary

State space

+

Fragile operations

If the pre-condition of any operation is violated the system may:

- **ignore the operation**
- **crash**
- **break down later**

Error Handling

Modify each operation to return a result.

REPORT ::= ok | already-known | not-known

```
┌─Success ─────────────────
│ result!: REPORT
├──────────────────
│ result! = ok
└──────────────────────────
```

Successful Operations

```
┌── Success ──────────────
│  result!: REPORT
├──────────────────
│  result! = ok
└─────────────────────
```

Add Birthday- revised

AddBirthday   Success =

Δ BirthdayBook
name?: NAME
date?: DATE
result! : REPORT
name?   known
birthday' = ...
$\wedge$
result! = ok

merge declarations

conjoin predicates

Detecting Errors

AlreadyKnown

BirthdayBook

name?: NAME

result! : REPORT

name?   known

result! = already-known

If name? is already known don't change anything and report already-known

Combining the Parts

RAddBirthday $\widehat{=}$

(AddBirthday   Success)

AlreadyKnown

The Other Operations

Similarly:

RFindBirthday $\widehat{=}$

(FindBirthday   Success)

NotKnown

RRemind $\widehat{=}$

Remind   Success

already robust

Advantages of Approach

Separation of concerns:
- **consider each idea separately**

Focus of understanding:
- **mind-sized chunks**

Modularity:
- **reuse pieces**

Observation

It is possible to combine specifications using $\wedge$ and $\vee$ using the Schema Calculus.....

....<u>even though you can't combine programs!</u>

Other Uses of the Schema Calculus

Separate:

- **single entity (e.g., process, file, record) and its place in the larger system**
- **different views of the same system**
- **system functions and access control**

Summary

- Z is a simple mathematical framework in which to:
  - > describe systems abstractly yet precisely
  - > compose a system out of small pieces
  - > use old specifications to build new specifications
  - > reason about properties of a system
  - > relate views of a system

Observations About the Z View of State Machines

- State invariant is *assumed* to hold across operations
  - > so don't have to *prove* it is maintained
- Operations applied outside of precondition lead to "chaos"
  - > so always want final precondition to be "true"
- Pre- and post-conditions are not distinguished syntactically
  - > but may be derived
- Focus on state, not traces
- Inputs, outputs, exceptions are simply notational conventions