Developing Formal Specifications in Z

Software Requirements Engineering

Hossein Saiedian, Ph.D.

Electrical Engineering & Computer Science
University of Kansas
saiedian@eecs.ukans.edu

Fall 2006

Organization of the Tutorial

- Introduction to Formal Methods (10 Minutes)
- Introduction to Z with Case Studies (3 Hours)
- Formal Methods/Z Resources (10 Minutes)
- Discussion (10 Minutes)

Quality Software

- Software
  - an executable program
  - the structure of program components
  - functionality of an application program
  - the look and feel of an interface
  - *the program and all associated documents needed to operate and maintain it*
- Software pervades many aspects of our lives
- Absolutely essential to improve the software quality

What is Quality?

- A key issue in the field of information technology
- Multi-dimensional concept with different levels of abstraction
- Popular views:
  - *subjective, ambiguous, unclear meaning*
  - *luxury, class, taste*
  - *difficult to define and measure*
- Professional view
  - *quantifiable*
  - *manageable*
  - *improveable*

Quality: A Definition

- Crosby (1979):

  "Conformance to requirements"

Conformance to Requirements: Implications

- During the production process, measurements are continually taken to determine conformance to the those requirements:
  - *measurement model*
  - *project tracking and oversight*
  - *validation criteria*
  - *quality assurance system*
  - *plans, commitment to improvement*

  Managerial aspects

The use of process models is encouraged

Conformance to Requirements: Implications (continued)

- Requirements must be clearly stated such that they cannot be misunderstood:
  - *complete*
  - *unambiguous*
  - *verifiable*
  - *precise*
  - *concise*
  - *consistent*

  Technical aspects

The use of formal methods is encouraged

Formal Methods

Definitions

- A notation with a well defined syntax and semantics used to unambiguously specify the requirements of a software system.
- A formal method is expected to support the proof of correctness of the final implementation of the software with respect to its specification.
- The formal methods notation is used for *formal specification* of a software system:
  - As a process: translation of a non-mathematical description into a formal language
  - As a product: concise description of the properties of a system in a language with well defined semantics and formal deduction support

Why Formal Methods?

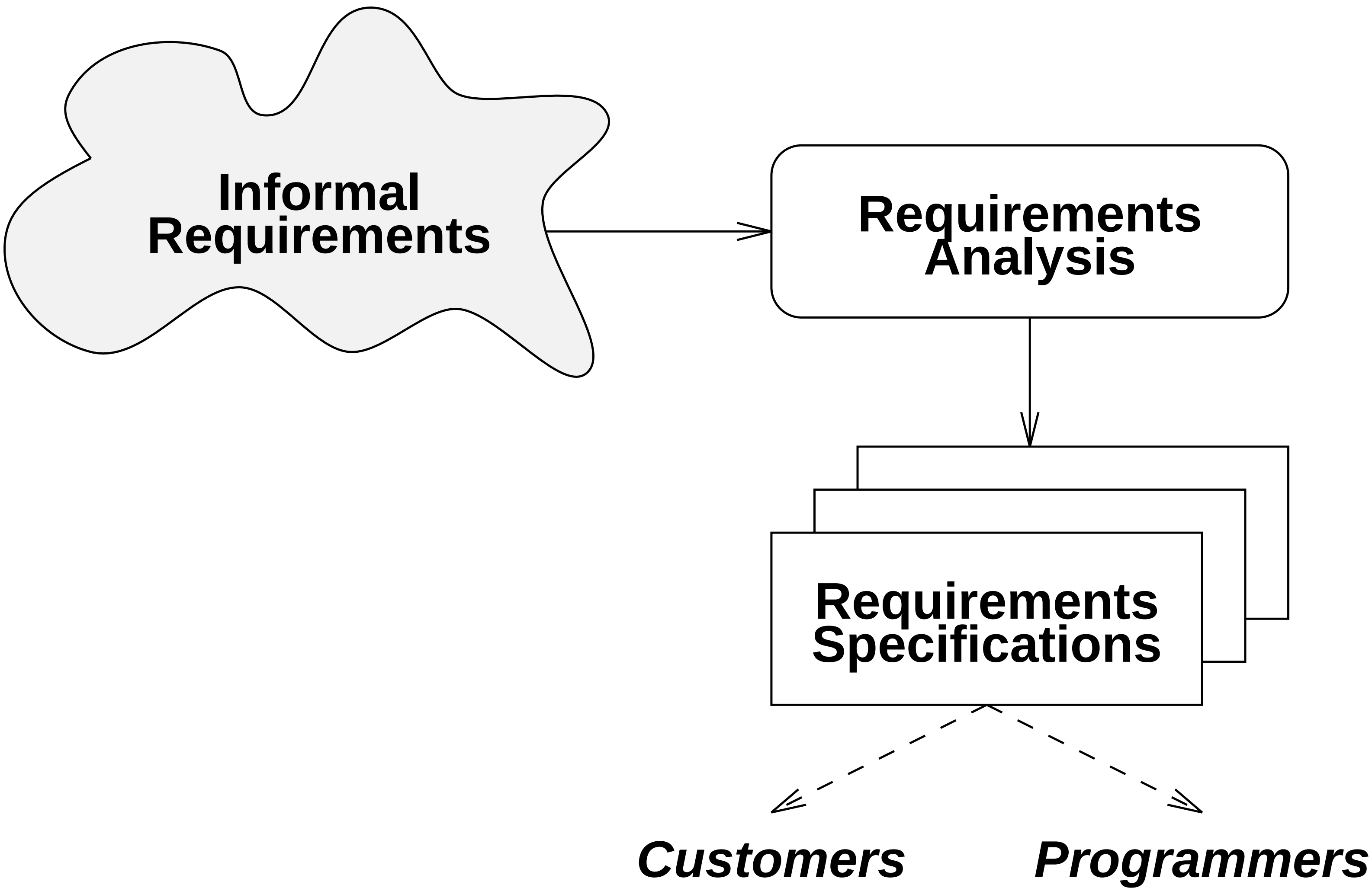
- To precisely and unambiguously describe the functionality of a software system at an abstract level that can be reasoned about formally using mathematics (or informally but rigorously)
- Precision and unambiguity are essential because for a large project many individuals have to agree on the interpretation of the specified functionality in order to produce a correct implementation.

When to Use Formal Methods

Software Development Life Cycle:

- Requirements analysis
- Requirements specification
  - As a process: when the functionality of the software is specified
  - As a product: where the expected functionality is recorded
- Architectural design
- Detailed design
- Implementation
- Testing

Requirements Specification as a Contract



Essential Properties of a Specification Document

- Correctness
- Completeness
- Unambiguous (one interpretation)
- Precision (unnecessary detail suppressed)
- Verifiable and traceable
- Independent from design
- Consistent (no conflicting features)
- Concise (lack of noise, irrelevant features)
- Annotated

Essential Properties of a Specification Document (continued)

For detailed explanations see [Davis 1993, Chapter 3]

Formal methods can assist in achieving most of the essential properties; more specifically:

- Unambiguity
- Verifiability
- Consistency
- Conciseness and Preciseness
- Annotation (especially true in case of Z notation)

Claimed Benefits, Drawbacks of Formal Methods

- Benefits:
  1. Unambiguous descriptions, i.e., there is one unique interpretation of a model given in a formal notation
  2. Analyzable products i.e., properties such as completeness and consistency can be checked by (automated) mechanisms
- Drawbacks:
  1. Mathematical skills, enthusiasm are required
  2. The development and verification process is costly and effort-intensive
  3. Complexity may explode with the dimension of the problem and can become unmanageable

Misconceptions about Formal Methods

Formal methods are for program verification only

- While formal methods can be applied during various stages of software development, their highest impact would be during the early stages, i.e., modeling and specification stages
- Program verification can be considered a secondary concern

Misconceptions about Formal Methods (continued)

Formal methods are too mathematical

- Based on mathematics, but the mathematics of formal methods is not very difficult. Most formal methods are based on the set theory and predicate logic
- Symbols can be learned through practice and training
- Not necessarily more complicated than implementation languages

Misconceptions about Formal Methods (continued)

An example in Z

LibSystem

$$members : \mathbb{P}\, Person$$
$$shelved : \mathbb{P}\, Book$$
$$checked : Book \twoheadrightarrow Person$$

$$shelved \cap \operatorname{dom} checked = \varnothing$$
$$\operatorname{ran} checked \subseteq members$$
$$\forall\, mem : Person \bullet \#(checked \rhd \{mem\}) \leq MaxLoan$$

Misconceptions about Formal Methods (continued)

Examples in C

- Production code: [Salus 1994]

```
if (rp -> p_flag & SSWAP) {
     rp -> p_flag =& ~SSWAP;
     aretu (u.u_ssav)
}
```

- A C declaration: [Kernighan & Ritchie, 1988, p.122]

```
int (*(*X[3])())[5];
```

- Copying string t to s: [Kernighan & Ritchie, 1988, p.105]

```
while (*s++=*t++);
```

Misconceptions about Formal Methods (continued)

Inapplicable to Real Projects

- A. Hall, Seven Myths of Formal Methods, *IEEE Software*, September 1990, pp. 11–19.
- J. Bowen and H. Hinchey (editors), *Applications of Formal Methods*, Prentice-Hall International, 1995.
  - Nuclear facility
  - Instrumentation systems (Z)
  - Voting system (VDM)
  - IBM's CICS (Z, B)
  - Railroad tracking, training, signaling system, (VDM)
  - Aerospace monitoring system (Z)

Misconceptions about Formal Methods (continued)

- Secure operating system (Larch)
- AT&T telephone switching system (Z)

- Of course more work is needed:
  - H. Saiedian, et al, An Invitation to Formal Methods, *IEEE Computer*, April 1996.
  - H. Saiedian (Guest Editor), *Journal of Systems and Software*, Special Issue on Formal Methods Technology Transfer, March 1998.

Need for Measurements for Formal Methods

- A large number of formal methods have been proposed
- A formal method notation comes with some common advise on how to be used; syntax and semantics are given but little indication is given on how to use it effectively
- A given FM is unlikely to be equally effective for all domains
- Required components of formal methods
  1. formal syntax and semantics
  2. conceptual model
  3. uniform notation of an interface
  4. sufficient expressive power to express relevant features
  5. hints and suggestion for refinement, implementation in a systematic way

Philosophical View of Formal Methods

The following is from John Rushby's Talk at LFMW97:

- In engineering, mathematical models of systems are built so that the properties of those systems can be *predicted* through the power of *calculation*.

  The power of mechanized calculation makes the construction of complex or optimized physical systems possible.

- Formal methods apply the same ideas to the construction of the complex logical design of computer systems:
  - Build a formal mathematical model of some aspect of a system
  - Calculate whether or not the system possess certain desired properties

Need for Measurements for Formal Methods (continued)

- Only qualitative and anecdotal evidence is available about the benefits and drawbacks
- For sound and objective cost analysis, a thorough study is needed to assess the degree to which the benefits and drawbacks are real
- Such a study or evidence will greatly help wider introduction of formal methods into industrial practice
- Currently the choice of one method over another is mostly a matter of personal taste not grounded on any objective evidence
- A strong need for an objective assessment of strength, weakness, application domain, and limitation of various formal methods to allow meaningful comparison and selection

Formal Methods Light: An alternative

- Do not promote full formalization; in most cases a less than completely formal approach is more helpful, convincing
- Integrate with existing, not necessarily formal models and approaches

  A formal method should contribute to and benefit from an existing tool or notation

Model-Based Formal Methods Notations

Model-based notations

- Support the development of an abstract model of the software product
- Support the behavioral description of the abstract model
- Examples: Z, Object-Z, VDM, Larch

Specification Language Z

- Jean-Raymond Abrial, late 1970s/early 1980s
- Under continuing development at the Programming Research Group, Oxford University
- A state-based modeling/specification language
- Set theory, predicate logic
- Functional specification of sequential systems
- Object-oriented variations
- Most popular formal methods notation

Z Schemas

- The building-block for structuring specifications
- Graphical notation

SchemaName

Signature

Predicates

- An alternative linear notation

$$SchemaName \mathrel{\widehat{=}} [\,Signature \mid Predicates\,]$$

Z Schemas (continued)

- Signature introduces variables and assigns them set theoretic types — similar to declarations in modern programming languages
- Predicates include expressions that relate the elements of signature:
  - Viewed in terms of invariants, pre- and post-conditions
  - ANDed by default; order is irrelevant

Identifiers in Z

- Identifiers may be composed of upper and lower case letters, digits, and the underscore character; must begin with a letter
- Identifiers may have suffixes:
  - $?$ means an input variable
  - $!$ means an output variable
  - $'$ means a new value (i.e., the after-operation value)
- Schema identifiers may have prefixes:
  - Δ means the state has changed (described later)
  - Ξ means no change in the state (described later)

An Example of a Z Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Reserve}} \\
\hline
passengers, passengers' : \mathbb{P}\, PERSON \\
p? : PERSON \\
\hline
\#passengers < CAPACITY \\
p? \notin passengers \\
passengers' = passengers \cup \{p?\} \\
\#passengers' \leq CAPACITY \\
\hline
\end{array}
$$

Alternative notation:

$$Reserve \mathrel{\widehat{=}} [p? : PERSON;\ \ldots \mid p? \notin passengers \wedge \ldots]$$

Notable Advantages of Z

- Procedural Abstraction: Schemas only describe what is to be done; issues relating to "how" are ignored
- Representational Abstraction: Schemas use high-level mathematical structures like arbitrary sets, functions, ..., without worrying about how these are to be implemented
- Allow annotation

  ... *informal descriptions* ...

  S

  $\ldots$

  $\ldots$

  ... *complement formal definitions* ...

Schema Calculus

A collection of notational conventions used to manipulate schemas and combine them into composite structures.

- Provides a framework to develop and present schemas in an incremental fashion, i.e., schemas can be constructed from existing schemas.
  - Analogous to "modular" program development.
  - Specifications become more manageable.
- Achieved primarily by means of "Schema Inclusion" and "Schema Linking" and Δ and Ξ conventions.

Schema Inclusion

- Suppose we have the following two schemas:

schemaA

$p, q : \mathbb{Z}$

$p \neq q$

schemaB

$r, s : \mathbb{N}$

$r \geq s$

- We can form a new schema by including two existing ones:

schemaAB

schemaA
schemaB

- If "expanded," *SchemaAB* will include:

schemaAB

$p, q : \mathbb{Z};\ r, s : \mathbb{N}$

$p \neq q \land r \geq s$

Schema Linking

- It is possible to use propositional connectives (e.g., $\wedge, \vee, \neg \; \Rightarrow, \Leftrightarrow$) and existing schemas to construct a new one.
- We can construct *schemaBA* as follows:

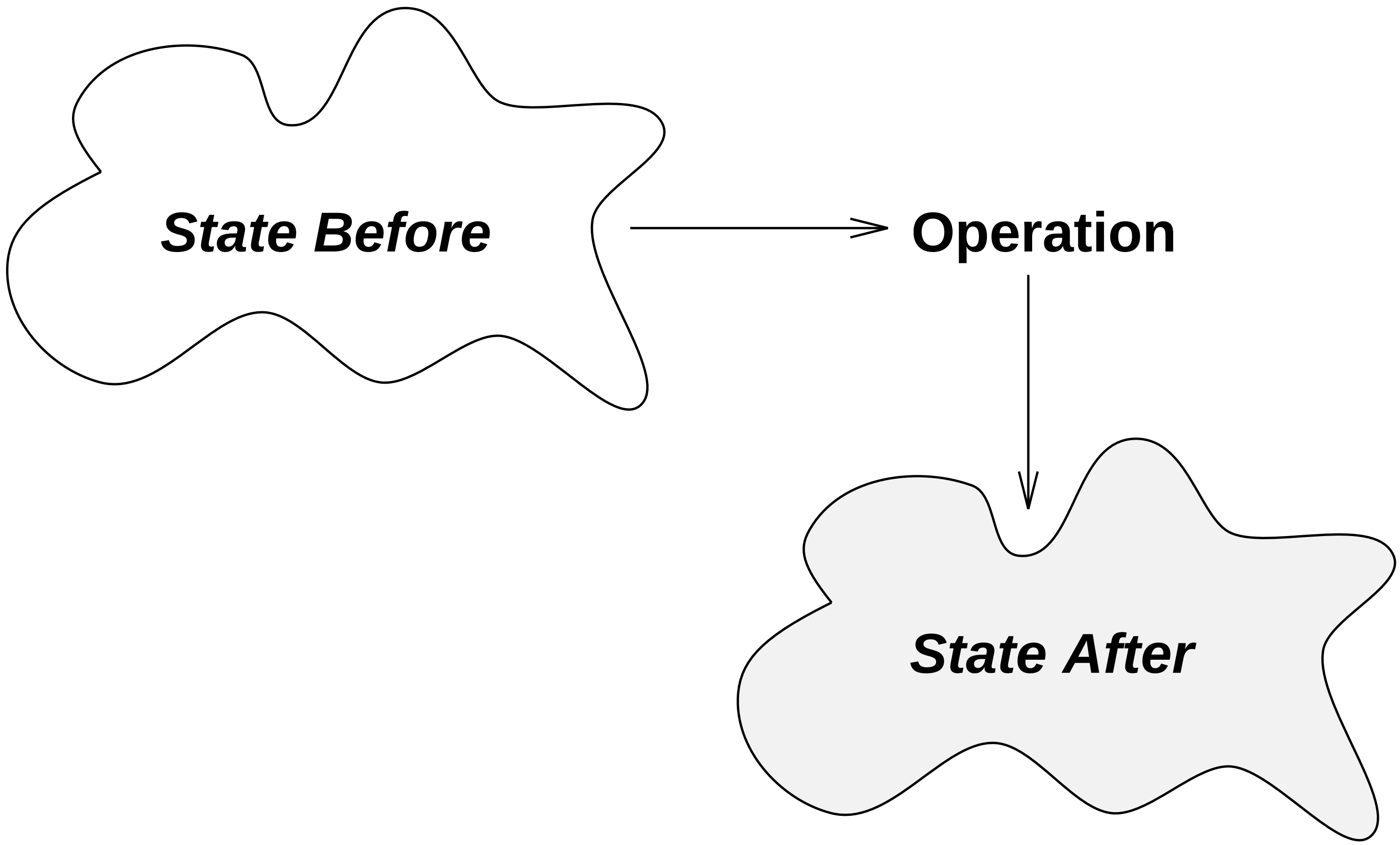
$$schemaBA \mathrel{\widehat{=}} schemaA \vee schemaB$$

- If "expanded," *schemaBA* will include:

$$\begin{array}{|l}
\hline
\textit{schemaBA} \\
p, q : \mathbb{Z} \\
r, s : \mathbb{N} \\
\hline
p \neq q \vee \\
r \geq s \\
\hline
\end{array}$$

Δ and Ξ Conventions

Operations and changes in the state data:



Δ and Ξ Conventions (continued)

Thus, to define an operation we must have:

Operation
StateData *StateData′*
Predicates describing *StateData* Predicates describing *StateData′*

Δ and Ξ Conventions (continued)

Consider operation *Reserve* again:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Reserve}} \\
\hline
passengers, passengers' : \mathbb{P}\, PERSON \\
p? : PERSON \\
\hline
\#passengers < CAPACITY \\
p? \notin passengers \\
passengers' = passengers \cup \{p?\} \\
\#passengers' \leq CAPACITY \\
\hline
\end{array}
$$

Δ and Ξ Conventions (continued)

- Conventionally, two schemas are given to define the state of data, one describing state before an operation, and one for describing state after the operation.

- For a simple reservation system we may have:

$$
\begin{array}{|l}
\hline
\textit{ResSyst} \\
\textit{passengers} : \mathbb{P}\,\textit{PERSON} \\
\hline
\#\textit{passengers} \leq \textit{CAPACITY} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{ResSyst}' \\
\textit{passengers}' : \mathbb{P}\,\textit{PERSON} \\
\hline
\#\textit{passengers}' \leq \textit{CAPACITY} \\
\hline
\end{array}
$$

Δ and Ξ Conventions (continued)

- *Reserve* operation can now be specified as:

$$
\begin{array}{|l}
\hline
\textbf{Reserve} \\
\Delta ResSyst \\
p? : PERSON \\
\hline
\#passengers < CAPACITY \\
p? \notin passengers \\
passengers' = passengers \cup \{p?\} \\
\hline
\end{array}
$$

Δ and Ξ Conventions (continued)

- While most operations bring about changes in the state data, some do not:

$$
\begin{array}{|l}
\hline
\textit{PassengerCount} \\
\hline
\textit{ResSyst} \\
\textit{ResSyst}' \\
\textit{count}! : \mathbb{N} \\
\hline
\textit{count}! = \#\textit{passengers} \\
\hline
\end{array}
$$

Δ and Ξ Conventions (continued)

- Δ convention is commonly used to represent change of state; it combines the definitions of two schemas, one describing the state before an operation and one describing the state after the operation:

$$
\begin{array}{|l}
\hline
\textit{\Delta ResSyst} \\
\hline
\textit{ResSyst} \\
\textit{ResSyst}' \\
\hline
\end{array}
$$

- Ξ convention is commonly used to represent no change of state; it can be defined in terms of Δ:

$$
\begin{array}{|l}
\hline
\textit{\Xi ResSyst} \\
\hline
\textit{\Delta ResSyst} \\
\hline
\textit{passengers} = \textit{passengers}' \\
\hline
\end{array}
$$

Δ and Ξ Conventions (continued)

- Operation *Reserve* can now be specified using Δ convention:

$$
\begin{array}{|l}
\hline
\textit{Reserve} \\
\Delta \textit{ResSyst} \\
p? : \textit{PERSON} \\
\hline
\#\textit{passengers} < \textit{CAPACITY} \\
p? \notin \textit{passengers} \\
\textit{passengers}' = \textit{passengers} \cup \{p?\} \\
\hline
\end{array}
$$

- Similarly, operation *PassengerCount* can now be specified using Ξ convention:

$$
\begin{array}{|l}
\hline
\textit{PassengerCount} \\
\Xi \textit{ResSyst} \\
\textit{count}! : \mathbb{N} \\
\hline
\textit{count}! = \#\textit{passengers} \\
\hline
\end{array}
$$

Δ and Ξ Conventions (Summary)

- The prefix Δ denotes a new schema built by combining the before and after specification of a state schema. The new schema (denoted by Δ) alarms about some changes made to the state; these changes must be clearly defined.
- The prefix Ξ denotes a new schema by combining the before and after specification of a state schema but with the rule that the before and after states are identical — no change is made to the state.

Presenting Specifications in Z

- Present given sets (types), user-defined sets, and global definitions
- Present abstract state of the system, followed by the $\Delta State$ and $\Xi State$ specifications:
  - $\Delta State \mathrel{\widehat{=}} [State; \; State']$
  - $\Xi State \mathrel{\widehat{=}} [\Delta State \mid State = State']$
- Present the initial state; shows that at least one state exists
- Present operations: Successful cases, followed by error cases, followed by a total definition

Always accompany the formal definitions with informal descriptions to explain their purposes.

Given and User-defined Sets/Types

- Given sets (types) are presented in upper case (or initial in upper case), enclosed in brackets:

  $[ACCT]$ $[Book, User]$

- User-defined sets or types – several ways of presenting them. Examples of an enumerated-like definition:

  $MESSAGE ::= \text{"}Full\text{"} \mid \text{"}Empty\text{"} \mid \text{"}OK\text{"}$

  $Colors ::= green \mid red \mid blue \mid white$

Global Definitions

- Introduced by means of axiomatic descriptions:

$$\begin{array}{|l} \textit{Description} \\ \hline \textit{Predicates} \end{array}$$

- If there is no constraining predicate:

$$\begin{array}{|l} \textit{Description} \end{array}$$

- Examples:

$$\begin{array}{|l} \mathit{Capacity} : \mathbb{N} \\ \hline \mathit{Capacity} = 200 \end{array}$$

$$\begin{array}{|l} \mathit{MaxQty} \end{array}$$

Abstract State of the System

- Every sequential system has an abstract state space which should be specified via a schema. For large systems, the abstract schema may be constructed of several other schemas using the schema calculus.

- An Example: A simple library system

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LibSystem}} \\ \hline
members : \mathbb{P}\, PERSON \\
shelved : \mathbb{P}\, BOOK \\
checked : BOOK \twoheadrightarrow PERSON \\ \hline
shelved \cap \operatorname{dom} checked = \varnothing \\
\operatorname{ran} checked \subseteq members \\
\forall\, mem : PERSON \bullet \#(checked \rhd \{mem\}) \leq MaxLoan \\ \hline
\end{array}$$

A Case Study in Z: A*PhoneDir* System

- Objective: Construct a telephone directory system (called *PhoneDir*), for a university to maintain a record of faculty and their telephone numbers.
- System Requirements:
  - A faculty may have one or more telephone numbers
  - Some faculty may not have a telephone number yet
  - A number may be shared by two or more faculty
  - Must be able to add new faculty and/or new entries
  - Must be able to remove faculty and/or existing entries
  - Must be able to query the system for a faculty or number
- Based on an example by Diller (1994).

Present Given, User-defined Types

- A type to represent individual persons

  $[PERSON]$

  We are not interested in more detail about persons.

- Can use natural numbers $\mathbb{N}$ to model telephone numbers, or as an alternative, can assume a given type:

  $[PHONE]$

- Note that we could have restricted the range, e.g.,

  $PHONE == 41000 \mathrel{..} 49999$

Present Given, User-defined Types (continued)

- Output messages:

$$
\begin{array}{lll}
\mathit{MESSAGE} & ::= & \text{`OK'} \\
 & | & \texttt{`Faculty already exists'} \\
 & | & \texttt{`No such faculty'} \\
 & | & \texttt{`Faculty has no number'} \\
 & | & \texttt{`Invalid number'} \\
 & | & \texttt{`Invalid entry'} \\
 & | & \texttt{`Entry already exists'}
\end{array}
$$

Abstract State of *PhoneDir* System

- A set of type *PERSON* representing the faculty:

  $faculty : \mathbb{P}\, PERSON$

- Abstract representation of an instance of *faculty*:

chen

stan

mary

hossein

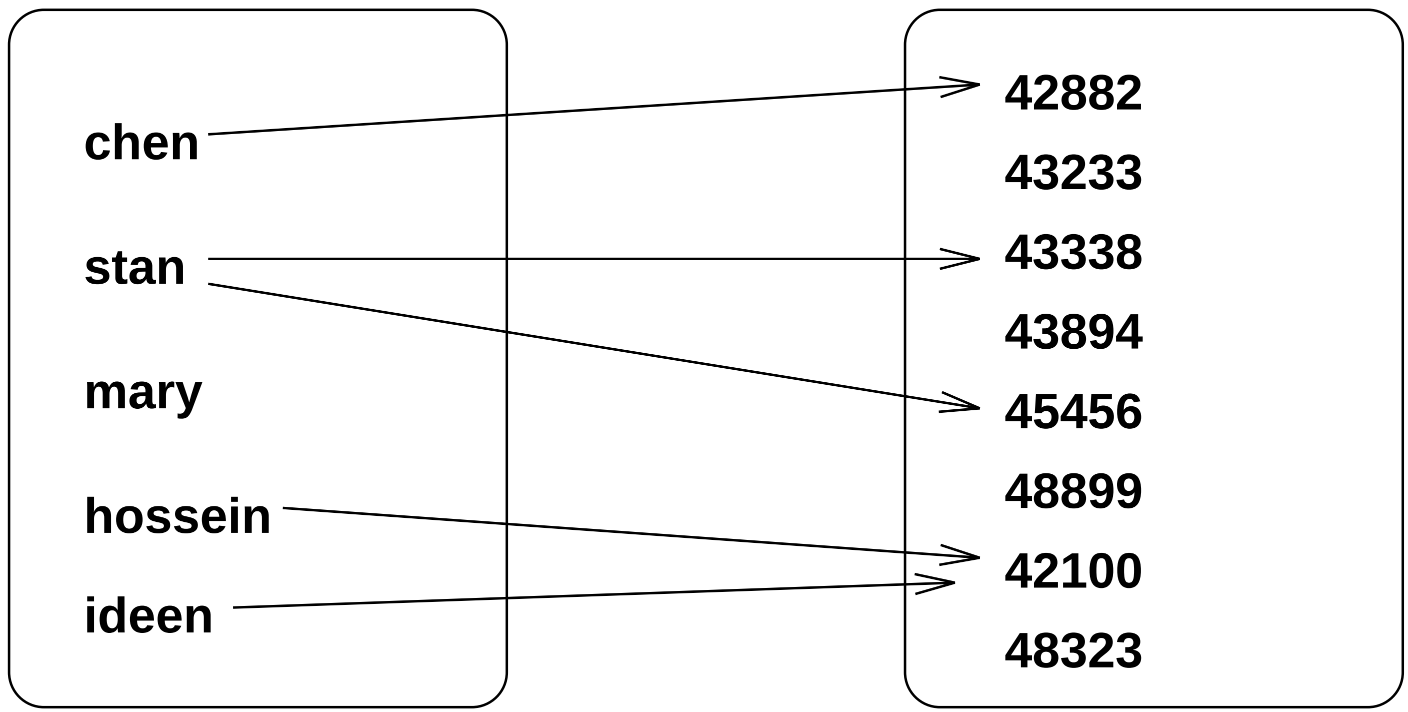
ideen

Abstract State of *PhoneDir* System (continued)

- We need another set representing a *directory* in the system:

  directory :????

- Abstract representation of an instance of *directory*:



- *directory* is a subset of Cartesian product of

  $PERSON \times PHONE$

  Pairs of values: one from $PERSON$, and one from $PHONE$.

Abstract State of *PhoneDir* System (continued)

- In Z, we represent the above as a "relation"
- A relation is a set of connections between two sets:

  $directory = \{(chen, 42882), (stan, 43338), (stan, 45456), \ldots\}$
- Alternatively, use $\mapsto$ symbol to show connections:

  $directory = \{chen \mapsto 42882, stan \mapsto 43338, stan \mapsto 45456, \ldots\}$
- Declaring a variable of type relation:

  $directory : PERSON \leftrightarrow PHONE$

Abstract State of *PhoneDir* System (continued)

Partial list of operations available on a relation:

- domain

  $\operatorname{dom} directory = \{chen, stan, hossein, ideen\}$

- range

  $\operatorname{ran} directory = \{42882, 43338, 45456, 42100\}$

- inverse

  $directory^{\sim} = \{42882 \mapsto chen, 43338 \mapsto stan, \ldots\}$

- Also domain restriction ($\lhd$), domain subtraction ($⩤$), range restriction ($\rhd$), range subtraction ($⩥$), relational image ($(\!|\ |\!)$), etc.

Abstract State of *PhoneDir* System (continued)

We can now define the before state and after state of *PhoneDir*:

- Before state

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PhoneDir}} \\
\hline
faculty : \mathbb{P}\, PERSON \\
directory : PERSON \leftrightarrow PHONE \\
\hline
\operatorname{dom} directory \subseteq faculty \\
\hline
\end{array}$$

- After state (decorate names with prime)

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PhoneDir}'} \\
\hline
faculty' : \mathbb{P}\, PERSON \\
directory' : PERSON \leftrightarrow PHONE \\
\hline
\operatorname{dom} directory' \subseteq faculty' \\
\hline
\end{array}$$

Defining $\Delta PhoneDir$ System

- $\Delta PhoneDir$:

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta PhoneDir} \\
\hline
faculty, faculty' : \mathbb{P}\, PERSON \\
directory, directory' : PERSON \leftrightarrow PHONE \\
\hline
\operatorname{dom} directory \subseteq faculty \\
\operatorname{dom} directory' \subseteq faculty' \\
\hline
\end{array}$$

- Or alternatively as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta PhoneDir} \\
\hline
PhoneDir \\
PhoneDir' \\
\hline
\end{array}$$

Defining $\Xi \mathit{PhoneDir}$ System

- $\Xi \mathit{PhoneDir}$:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\Xi \mathit{PhoneDir}} \\
\Delta \mathit{PhoneDir} \\
\hline
\mathit{faculty}' = \mathit{faculty} \\
\mathit{directory}' = \mathit{directory} \\
\hline
\end{array}
$$

Defining the Initial State of *PhoneDir* System

- Presenting the initial state
- Note that *InitPhoneDir* does not really exists

InitPhoneDir'

$PhoneDir'$

$faculty' = \{\}$
$directory' = \{\}$

Adding a New Faculty to *PhoneDir* System

- Input to the operation: $name? : PERSON$
- Output from the operation: $rep! : MESSAGE$
- Successful operation:

AddMemberOK

$$\Delta PhoneDir$$
$$name? : PERSON$$
$$rep! : MESSAGE$$

$$name? \notin faculty$$
$$faculty' = faculty \cup \{name?\}$$
$$directory' = directory$$
$$rep! = \text{‘OK’}$$

Adding a New Faculty to *PhoneDir* System (continued)

Error Cases:

- Faculty already exists in the *PhoneDir*, i.e., $name? \in faculty$

$$
\begin{array}{|l}
\hline
\textit{FacultyExists} \\
\Xi PhoneDir \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \in faculty \\
rep! = \text{‘}\texttt{Faculty already exists}\text{’} \\
\hline
\end{array}
$$

- A total definition:

$AddMember \mathrel{\widehat{=}} AddMemberOK \vee FacultyExists$

Adding an Entry to *PhoneDir* System

- Input to the operation:
  $name? : PERSON$ and $number? : PHONE$

- Output report from the operation: $rep! : MESSAGE$

- Successful operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddEntryOK}} \\
\Delta PhoneDir \\
name? : PERSON \\
number? : PHONE \\
rep! : MESSAGE \\
\hline
name? \in faculty \\
name? \mapsto number? \notin directory \\
directory' = directory \cup \{name? \mapsto number?\} \\
faculty' = faculty \\
rep! = \text{`OK'} \\
\hline
\end{array}
$$

Adding an Entry to *PhoneDir* System (continued)

Error Cases:

- Input not a legitimate faculty, i.e., $name? \notin faculty$

  Note that we have already defined a schema for this case (*NotFaculty*); we will reuse it.

- Entry already exists in *directory*, i.e., $name? \mapsto number? \in directory$

$$\begin{array}{|l}
\textit{EntryExists} \\
\hline
\Xi PhoneDir \\
number? : PHONE \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \mapsto number? \in directory \\
rep! = \text{`}\texttt{Entry already exists}\text{'} \\
\hline
\end{array}$$

Adding an Entry to *PhoneDir* System (continued)

- Constructing a total operation for adding an entry:

$$AddEntry \mathrel{\widehat{=}} AddEntryOK \lor NotFaculty \lor EntryExists$$

Removing a Faculty from *PhoneDir* System

- Input: $name? : PERSON$
- Output: $rep! : MESSAGE$
- Successful operation:

$$
\begin{array}{|l}
\textit{RemoveFacultyOK} \\
\hline
\Delta PhoneDir \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \in faculty \\
faculty' = faculty \setminus \{name?\} \\
directory' = \{name?\} \lhd directory \\
rep! = \text{'OK'} \\
\hline
\end{array}
$$

Side Notes: Operations on Relations

- Domain subtraction $\lhd\!\!\!-$: Given a set S and a relation R, $S \lhd\!\!\!- R$ constructs a new relation of pairs from R whose first elements (i.e., domain elements) are not in S. Example:

  $\{s_1, s_3\} \lhd\!\!\!- \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\} =$
  $\{s_2 \mapsto r_2, s_4 \mapsto r_4, s_5 \mapsto r_5\}$

- Domain restriction $\lhd$: Given a set S and a relation R, $S \lhd R$ constructs a new relation of pairs from R whose first elements (i.e., domain elements) are restricted to those in S. Example:

  $\{s_1, s_3\} \lhd \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\} =$
  $\{s_1 \mapsto r_1, s_3 \mapsto r_3\}$

- Similar operations for range of a relation, namely range subtraction ($\rhd\!\!\!-$) and range restriction ($\rhd$)

Removing a Faculty from *PhoneDir* System (continued)

- Error case: there is no such faculty to be removed, i.e., $name? \notin faculty$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad NotFaculty \quad}} \\
\Xi PhoneDir \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \notin faculty \\
rep! = \text{`}\texttt{No such faculty}\text{'} \\
\hline
\end{array}
$$

- A total definition:

$$RemoveFaculty \mathrel{\widehat{=}} RemoveFacultyOK \lor NotFaculty$$

Removing an Entry from *PhoneDir* System

- Input: $number? : PHONE$, $name? : PERSON$
- Output $rep! : MESSAGE$
- Successful operation:

$$
\begin{array}{|l}
\hline
\textit{RemoveEntryOK} \\
\Delta PhoneDir \\
number? : PHONE \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \mapsto number? \in directory \\
directory' = directory \setminus \{name? \mapsto number?\} \\
faculty' = faculty \\
rep! = \text{`OK'} \\
\hline
\end{array}
$$

Removing an Entry from *PhoneDir* System (continued)

- Error cases: entry does not exists, i.e.,

$name? \mapsto number? \notin directory$

$$
\begin{array}{|l}
\textit{InvalidEntry} \\
\hline
\Xi PhoneDir \\
number? : PHONE;\ name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \mapsto number? \notin directory \\
rep! = \text{`}\texttt{Invalid entry}\text{'} \\
\hline
\end{array}
$$

- A total definition:

$RemoveEntry \mathrel{\widehat{=}} RemoveEntryOK \lor InvalidEntry$

Querying the *PhoneDir* System

Query by person:

- Input: $name? : PERSON$
- Output: $numbers! : \mathbb{P}\,PHONE$
- Successful operation:

$$
\begin{array}{|l}
\hline
\textit{FindNumbersOK} \\
\Xi PhoneDir \\
name? : PERSON \\
numbers! : \mathbb{P}\,PHONE \\
rep! : MESSAGE \\
\hline
name? \in faculty \\
name? \in \operatorname{dom} directory \\
numbers! = directory(\!|\ \{name?\}\ |\!) \\
rep! = \text{`OK'} \\
\hline
\end{array}
$$

Side Notes: Operations on Relations

- Relational image $(\!|\ |\!)$: Given a set S and a relation R, applying relational image constructs a new set of elements from range of R which are related to by elements of S. Example

  $\{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\}(\!|\ \{s_1, s_3, s_9\}\ |\!) = \{r_1, r_3\}$

- Relational inverse: inverses the order of pairs. Example:

  $\{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\}^{\sim} = \{r_1 \mapsto s_1, r_2 \mapsto s_2, r_3 \mapsto s_3, r_4 \mapsto s_4, r_5 \mapsto s_5\}$

Querying the *PhoneDir* System (continued)

Error Cases:

- Input not a legitimate faculty: we already have a schema for this (*NotFaculty*); it will be reused

- Faculty does not have a number yet, i.e., $name? \notin directory$

$$
\begin{array}{|l}
\hline
\textbf{InvalidName} \\
\Xi PhoneDir \\
name? : PERSON \\
rep! : MESSAGE \\
\hline
name? \notin \operatorname{dom} directory \\
rep! = \text{`}\texttt{Faculty has no number}\text{'} \\
\hline
\end{array}
$$

- A total definition

  $FindNumbers \mathrel{\widehat{=}} FindNumbersOK \lor NotFaculty \lor InvalidName$

Querying the *PhoneDir* System (continued)

Querying by the number:

- Input: $number? : PHONE$
- Output: $names! : \mathbb{P}\, PERSON$, $rep! : MESSAGE$
- Successful operation

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindNamesOK}} \\
\hline
\Xi PhoneDir \\
number? : PHONE \\
names! : \mathbb{P}\, PERSON \\
rep! : MESSAGE \\
\hline
number? \in \mathrm{ran}\, directory \\
names! = directory^{-1} (\!|\, \{number?\} \,|\!) \\
rep! = \text{`OK'} \\
\hline
\end{array}
$$

Querying the *PhoneDir* System (continued)

Error Case:

- Invalid number, i.e., $number \notin \operatorname{ran} directory$,

$$
\begin{array}{|l}
\hline
\textit{InvalidNumber} \\
\Xi PhoneDir \\
number? : PHONE \\
rep! : MESSAGE \\
\hline
number? \notin \operatorname{ran} directory \\
rep! = \text{‘}\texttt{Invalid number}\text{’} \\
\hline
\end{array}
$$

- A total definition

$$FindNames \mathrel{\widehat{=}} FindNamesOK \lor InvalidNumber$$

Summary of Essential Operators on Relations

Operator	Synopsis	Meaning
$\leftrightarrow$	$R_1 \leftrightarrow R_2$	Binary relation between R_1 and R_2
$\mapsto$	$r_1 \mapsto r_2$	Maplet
dom	$\mathrm{dom}\, R$	Domain of R
ran	$\mathrm{ran}\, R$	Range of R
$\_(\!\mid \_ \mid\!)$	$R(\!\mid S \mid\!)$	Relational image
$\lhd$	$S \lhd R$	Domain restriction
$⩤$	$S ⩤ R$	Domain subtraction
$\rhd$	$R \rhd S$	Range restriction
$⩥$	$R ⩥ S$	Range subtraction
$\oplus$	$R_1 \oplus R_2$	Relational overriding

Functions in Z

- A special kind of relation: $S_1 \rightarrow S_2$
- Each element of S_1 is related to at most one element of S_2
- Several different kinds of functions:
  - total $\rightarrow$
  - partial $\pfun$
  - injective (one-to-one, i.e., no sharing of elements in S_2); can be partial ($\pinj$) or total ($\inj$)
  - surjective (onto, i.e., the range of function is the entire S_2); can be partial ($\psurj$) or total ($\surj$)
  - bijective function $\bij$: injective and surjective

An Example Using Functions in Z

- Consider a simple library program that manages available books that library users (borrowers) may want to barrow. The program maintains a table of which users have borrowed which book and allows borrowing or returning books.
- Based on the storage manager example given in [Woodcock and Loomes, 1989]; a more elaborate example is in [Diller 1994].
- Requirements
  - No book is borrowed simultaneously by more than one user
  - A library user may borrow more than one book
  - Some books may not be borrowed

- Some library users may not borrow any books

An Example Using Functions in Z (continued)

- Assume the following types for library users and library books:

  $[USER, BOOK]$

- Define $MESSAGE$ as follow:

$$
\begin{aligned}
MESSAGE ::= &\ \texttt{`OK'} \\
\mid &\ \texttt{`Book not available'} \\
\mid &\ \texttt{`Invalid return'}
\end{aligned}
$$

An Example Using Functions in Z (continued)

- Define the abstract state of the library system

$$
\begin{array}{l}
\textit{LibSys} \\
\hline
available : \mathbb{P}\, BOOK \\
borrowed : BOOK \pfun USER \\
\hline
available \cup \operatorname{dom} borrowed = BOOK \\
available \cap \operatorname{dom} borrowed = \emptyset
\end{array}
$$

Why using a partial function?

An Example Using Functions in Z (continued)

- Define $\Delta LibSys$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\Delta LibSys} \\
available, available' : \mathbb{P}\, BOOK \\
borrowed, borrowed' : BOOK \nrightarrow USER \\
\hline
available \cup \operatorname{dom} borrowed = BOOK \\
available \cap \operatorname{dom} borrowed = \varnothing \\
available' \cup \operatorname{dom} borrowed' = BOOK \\
available' \cap \operatorname{dom} borrowed' = \varnothing \\
\hline
\end{array}
$$

- Define $\Xi LibSys$

$\Xi LibSys \mathrel{\widehat=}$

$[\Delta LibSys \mid borrowed' = borrowed \land available' = available']$

An Example Using Functions in Z (continued)

- Define the initial abstract state

$$
\begin{array}{|l}
\hline
\textit{InitLibSys}' \\
\hline
\textit{LibSys}' \\
\hline
\textit{available}' = \textit{BOOK} \\
\textit{borrowed}' = \varnothing \\
\hline
\end{array}
$$

An Example Using Functions in Z (continued)

- Defining *CheckOut* operation (successful case)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CheckOutOK}} \\
\hline
\Delta LibSys \\
u? : USER \\
b? : BOOK \\
rep! : MESSAGE \\
\hline
b? \in available \\
available' = available \setminus \{b?\} \\
borrowed' = borrowed \cup \{b? \mapsto u?\} \\
rep! = \text{`OK'} \\
\hline
\end{array}
$$

An Example Using Functions in Z (continued)

- Error case: $b? \notin available$

$$
\begin{array}{|l}
\hline
\textit{NotAvailable} \\
\Xi LibSys \\
b? : BOOK \\
rep! : MESSAGE \\
\hline
b? \notin available \\
rep! = \text{`}\texttt{Book not available}\text{'} \\
\hline
\end{array}
$$

- A total definition of *CheckOut*

$CheckOut \mathrel{\widehat{=}} CheckOutOK \lor NotAvailable$

An Example Using Functions in Z (continued)

- Define *ReturnOK* for returning a book (successful case)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReturnOK}} \\
\hline
\Delta LibSys \\
u? : USER \\
b? : BOOK \\
rep! : MESSAGE \\
\hline
(b? \mapsto u?) \in borrowed \\
available' = available \cup \{b?\} \\
borrowed' = borrowed \setminus \{b? \mapsto u?\} \\
rep! = \text{‘OK’} \\
\hline
\end{array}
$$

An Example Using Functions in Z (continued)

- Error case: recording an incorrect return

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InvalidReturn}} \\
\hline
\Xi LibSys \\
u? : USER \\
b? : BLOCK \\
rep! : MESSAGE \\
\hline
(b? \mapsto u?) \notin borrowed \\
rep! = \text{`}\texttt{Invalid return}\text{'} \\
\hline
\end{array}
$$

- A total definition for *Return*

$$Return \mathrel{\widehat{=}} ReturnOK \lor InvalidReturn$$

Essential Operators on Functions

Operator	Meaning
⇸	Partial function
⇻	Finite partial function
→	Total function
⤀	Partial surjective function
↠	Total surjective function
↣	Partial injective function
⤔	Partial injective function
⤖	Total bijective function

Sequences in Z

- Another important typing mechanism: A sequence is a special kind of function (thus a kind of relation, or a kind of set). Operations available on functions can be applied to sequences

- Brackets are used to represent elements of a sequence: $colors = \langle green, white, red, blue \rangle$

- Representing $colors$ as a function:
  $\langle green, white, red, blue \rangle =$
  $\{(1, green), (2, white), (3, red), (4, blue)\}$

- In general, a sequence s of type T is a function from $\mathbb{N}$ to T, i.e., $\mathrm{dom}\, s = \{1, 2, 3, .., \#s\}$. Formally,
  $\mathrm{seq}\, T = \{f : \mathbb{N} \twoheadrightarrow T \mid \mathrm{dom}\, f = 1 \mathrel{..} \#f\}$

Sequences in Z (continued)

- Special operations available on a sequence:
  - $head(colors) = green$
  - $last(colors) = blue$
  - $front(colors) = \langle green, white, red \rangle$
  - $tail(colors) = \langle white, red, blue \rangle$
  - concatenation:
    $colors \frown \langle blue \rangle = \langle green, white, red, blue, blue \rangle$

An Example Using Sequences in Z

- Assume a *Queue* of non-repeating elements of type *ELEM*
- Operations: *Enqueue*, *Dequeue*, *IsEmpty*, and *QueueSize*
- Abstract state (generic construction)

$$
\begin{array}{|l}
\hline
\textit{Queue}\,[\textit{ELEM}\,] \\
q : \mathrm{seq}\,\textit{ELEM} \\
\hline
\#q = \#\,\mathrm{ran}\,q \\
\hline
\end{array}
$$

- Define $\textit{InitQueue}'$, $\Delta\textit{Queue}$, $\Xi\textit{Queue}$ Schemas

$$\Delta\textit{Queue} \mathrel{\widehat{=}} [\,\textit{Queue};\ \textit{Queue}' \mid \#q = \#\,\mathrm{ran}\,q \wedge \#q' = \#\,\mathrm{ran}\,q'\,]$$
$$\Xi\textit{Queue} \mathrel{\widehat{=}} [\,\Delta\textit{Queue} \mid q' = q\,]$$
$$\textit{InitQueue}' \mathrel{\widehat{=}} [\,\textit{Queue}' \mid q' = \langle\rangle\,]$$

An Example Using Sequences in Z (continued)

- Operation *Enqueue*

$$
\begin{array}{|l}
\hline
\textit{Enqueue}\,[\textit{ELEM}\,] \\
\Delta \textit{Queue} \\
e? : \textit{ELEM} \\
\textit{rep}! : \textit{MESSAGE} \\
\hline
(e? \notin \operatorname{ran} q \wedge q' = q \frown \langle e? \rangle \wedge \textit{rep}! = \text{‘}\mathtt{OK}\text{’}) \\
\vee \\
(e? \in \operatorname{ran} q \wedge q' = q \wedge \textit{rep}! = \text{‘}\mathtt{Duplicate\ entry}\text{’}) \\
\hline
\end{array}
$$

An Example Using Sequences in Z (continued)

- Operation *Dequeue*

$$
\begin{array}{|l}
\hline
\textit{Dequeue}[\textit{ELEM}] \\
\Delta \textit{Queue} \\
e! : \textit{ELEM} \\
\textit{rep}! : \textit{MESSAGE} \\
\hline
(q \neq \langle\rangle \land e! = \textit{head}(q) \land q' = \textit{tail}(q) \land \textit{rep}! = \text{`OK'}) \\
\lor \\
(q = \langle\rangle \land q' = q \land \textit{rep}! = \text{`Empty queue'}) \\
\hline
\end{array}
$$

An Example Using Sequences in Z (continued)

- Query operations

$$
\begin{array}{|l}
\hline
\textit{IsEmpty}\,[\textit{ELEM}\,] \\
\Xi \textit{Queue} \\
\textit{rep}! : \mathbb{B} \\
\hline
(q = \langle\rangle \land \textit{rep}!) \\
\lor \\
(q \neq \langle\rangle \land \neg\, \textit{rep}!) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{QueueSize}\,[\textit{ELEM}\,] \\
\Xi \textit{Queue} \\
\textit{size}! : \mathbb{N} \\
\hline
\textit{size}! = \# q \\
\hline
\end{array}
$$

An Example Using Sequences in Z (continued)

- Using the generic definitions: Construct a queue for 3-digit print jobs:

$$PrintQueue \mathrel{\widehat{=}} Queue[100 \mathrel{..} 999]$$

Other Notable Aspects of Z

- Bag data type
- More operators
- Generic constant definitions – an example:

$$
\begin{array}{|l}
[T] \\
\hline
\_\ \mathrm{unequal}\ \_ : T \leftrightarrow T \\
\hline
\forall r, s : T \bullet r\ \mathrm{unequal}\ s \Leftrightarrow \neg (r = s) \\
\hline
\end{array}
$$

- Calculation of pre-conditions

Calculating Pre-Conditions

- Objective: to demonstrate the validity of an operation, i.e., there exists at least one state in which the operation can be carried out.
- Alternatively, the objective can be stated as answering the following question: [Wordsworth 1992, p.123]

  For what combinations of inputs and starting states can we find outputs and ending states that satisfy the predicates of a given operation?

- Uses of existential quantifier operator

Calculating Pre-Conditions (continued)

- An informal approach to determining the pre-conditions: look for predicates not involving state-after variables (decorated with ′) or output variables (decorated with !)

- This approach works if

  1. pre-conditions are explicitly stated (preferably before any predicates involving state-after and output variables) and without implicitly relying on the state invariants
  2. pre-conditions are stated in terms of before-state (not after-state)

Calculating Pre-Conditions (continued)

- A more formal approach to calculate the $PreOp$ schema for some operation Op

  1. $PreOp \mathrel{\widehat{=}} Op$; expand $PreOp$

  2. Hide all after-state and output variables from the signature of $PreOp$; existentially quantify all such variables in the predicate part

  3. Simplify predicates in the $PreOp$ to eliminate variables decorated with $'$ or !

Calculating Pre-Conditions (continued)

- As an example consider operation *Reserve*:

$$
\begin{array}{|l}
\hline
\textit{Reserve} \\
\Delta ResSyst \\
p? : PERSON \\
\hline
p? \notin passengers \\
passengers' = passengers \cup \{p?\} \\
\hline
\end{array}
$$

Calculating Pre-Conditions (continued)

- Expand *Reserve* into *PreReserve*:

$$
\begin{array}{|l}
\hline
\textit{PreReserve} \\
passengers, passengers' : \mathbb{P}\, PERSON \\
p? : PERSON \\
\hline
\#passengers \leq CAPACITY \\
\#passengers' \leq CAPACITY \\
p? \notin passengers \\
passengers' = passengers \cup \{p?\} \\
\hline
\end{array}
$$

Calculating Pre-Conditions (continued)

- Hide after-state and output variables from signature; existentially quantify them in the predicate part:

$$
\begin{array}{|l}
\textit{PreReserve} \\
\hline
passengers : \mathbb{P}\, PERSON \\
p? : PERSON \\
\hline
\exists\, passengers' : \mathbb{P}\, PERSON \bullet \\
\quad (\#passengers \leq CAPACITY \land \\
\quad \#passengers' \leq CAPACITY \land \\
\quad p? \notin passengers \land \\
\quad passengers' = passengers \cup \{p?\}) \\
\hline
\end{array}
$$

Calculating Pre-Conditions (continued)

- Simplify the predicate by applying the *one-point* rule: substitute $passengers \cup \{p?\}$ where $passengers'$ occurs

PreReserve

$$passengers : \mathbb{P}\, PERSON$$
$$p? : PERSON$$

$$\#passengers \leq CAPACITY \land$$
$$\#(passengers \cup \{p?\}) \leq CAPACITY \land$$
$$p? \notin passengers$$

Calculating Pre-Conditions (continued)

- Further simplifications:

  1. Since $p? \notin passengers$ and $\#(passengers \cup \{p?\}) \leq CAPACITY$, conclude that $\#passengers + 1 \leq CAPACITY$

  2. Since $\#passengers \leq CAPACITY$ and $\#passengers + 1 \leq CAPACITY$, conclude that $\#passengers < CAPACITY$

Calculating Pre-Conditions (continued)

- We now have:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{PreReserve}\;}} \\
\mathit{passengers} : \mathbb{P}\,\mathit{PERSON} \\
p? : \mathit{PERSON} \\
\hline
\#\mathit{passengers} < \mathit{CAPACITY} \land \\
p? \notin \mathit{passengers} \\
\hline
\end{array}
$$

- The pre-conditions of *Reserve* operation are:
  1. $\#passengers < CAPACITY$ and
  2. $p? \notin passengers$

Side Notes for Calculating Pre-Conditions

- One-point rule is defined as follows: [Wordsworth 1992]

  $(\exists x : T \bullet (x = E \land P(x))) \Leftrightarrow (E \in T \land P(E))$

  where x is a variable, $P(x)$ is a predicate in which x is free, T is a set expression, and E is an expression of appropriate type.

- Z supports a schema operation called pre

- For a schema S, $\mathrm{pre}\, S$ is the result of hiding all variables decorated with $'$ and !

According to Spivey [Spivey 1992, p.77], $\mathrm{pre}\, S$ contains only the components of S corresponding to the state before the operation and its input.

Formal Methods Resources

- Articles
- Books
- Conference Proceedings/Journals
- WWW/FTP Archives
- Electronic Forums
- Postal Mailing List

Introductory and Informative Articles

Introduction to Z:

- J. M. Spivey, An Introduction to Z and Formal Specifications, *Software Engineering Journal*, January 1989, pp. 40–50.

- H. Saiedian, Formal Methods in Information Systems Engineering, in *Software Requirements Engineering*, 2e, Thayer and Dorfman (editors), pp. 336-349, IEEE-CS, Los Alamitos, CA, 1997.

Articles (continued)

Introduction to Formal Methods

- A. Hall, Seven Myths of Formal Methods, *IEEE Software*, pp. 11–19, September 1990.

- H. Saiedian, et al, An Invitation to Formal Methods, *IEEE Computer*, Vol. 29, No. 4, April 1996.

- Luqi and J. Goguen, Formal Methods: Promises and Problems, *IEEE Software*, Vol. 14, No. 1, pp. 73–85, January 1997.

Articles (continued)

Industrial Issues:

- H. Saiedian, Guest Editor, Research Issues in Formal Methods Technology Transfer, *Journal of Systems and Software*, March 1998 (special issue on technology transfer).
- H. Saiedian and M. Hinchey, Challenges in the Successful Transfer of Formal Methods Technology into Industrial Applications, *Information and Software Technology*, 38(5):313–322, May 1996.
- D. Craigen, S. Gerhart and R. Ralston, Formal Methods Reality Check: Industrial Usage, *IEEE Trans. on Software Engineering*, pp. 90–98, February 1995.

Articles (continued)

Educational Issues:

- H. Saiedian, Mathematics of Computing, *Computer Science Education*, Vol. 3(3), pp. 203–221, 1992.

- D. Garlan, Making Formal Methods Education Effective for Professionals, *Information and Software Technology*, pp. 261–268, May/June 1995.

- J. Wing, Hints to Specifiers, in *Teaching and Learning Formal Methods*, Dean and Hinchey (editors), Springer-Verlag, 1996.

Z Books

More than a dozen textbooks on Z; some popular titles include:

- V. Alagar and K. Periyasamy, *Specification of Software Systems*, Springer-Verlag, 1998.
- R. Barden, S. Stepney and D. Cooper, *Z in Practice*, Prentice-Hall, 1994.
- A. Diller, *Z: An Introduction to Formal Methods*, John Wiley, 2nd Edition, 1994.
- N. Dean and M. Hinchey, *Teaching and Learning Formal Methods*, Academic Press, 1996.
- I. Hayes (editor), *Specification Case Studies*, Prentice-Hall, 2nd Edition, 1993.
- J. Jacky, *The Way of Z*, Cambridge, 1997.

Z Books (continued)

- B. Potter and J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*, 2e, Prentice-Hall, 1995.
- J. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 2nd Edition, 1992.
- J. Woodcock and J. Davies, *Using Z: Specification, Proof and Refinement*, Prentice-Hall, 1995.
- J. Wordsworth, *Software Development with Z*, Addison-Wesley, 1992.

Conferences Proceedings

- *Formal Methods Europe*. Proceedings published in the Springer-Verlag *Lecture Notes in Computer Science* series
- *Z User Meeting* (ZUM). Proceedings published in the Springer-Verlag *Workshops in Computing* series since the 4th meeting in 1989. Proceedings of ZUM 1995–98 published in Springer-Verlag's LNCS series. Last two years' proceedings:
  - J. Bowen M. Hinchey, and D. Till (editors), *ZUM'97: The Z Formal Specification Notation*, Proceedings of the 10th International Conference of Z Users, Readings, UK, LNCS 1212, Springer-Verlag, 1997.
  - J. Bowen, A. Fett, M. Hinchey (editors) , *ZUM'98: The Z Formal Specification Notation*, Proceedings of the 11th International Conference of Z Users, Berlin, Germany,

LNCS 1493, Springer-Verlag, 1998.

Journals

Journals that regularly publish articles on or related to formal methods and Z or run special issues include:

- *Formal Aspects of Computing*, Springer-Verlag
- *Formal Methods in System Design*, Kluwer Academic
- *The Computer Journal*, BCS/IEE
- *Information and Software Technology*, Elsevier Science
- *IEEE Computer*, IEEE-CS
- *IEEE Software*, IEEE-CS
- *IEEE Trans. Software Engineering*, IEEE-CS

Formal Methods/Z Archives

Global WWW Home Pages — Fairly Comprehensive

Include advance programs, articles, bibliographies, courses, meetings, personalities, standards, technical-reports, tools, ...

- www.afm.sbu.ac.uk/fm/
- www.afm.sbu.ac.uk/z/
- shemesh.larc.nasa.gov/fm/
- www.cs.cmu.edu/Groups/formal-methods/formal-methods.html

Electronic Forums

USENET Newsgroups:

- comp.specification.misc, a discussion group interested in issues related to formal methods
- comp.specification.z, a discussion group interested in discussing or addressing questions related to Z

References (cited in the presentation)

- A. Davis, *Software Requirements: Objects, Functions, and States*, Prentice-Hall, Revision Edition, 1993.
- A. Diller, *Z: An Introduction to Formal Methods*, John Wiley, 2nd Edition, 1994.
- J. Spivey, *The Z Notation*, Second Edition, Prentice-Hall, 1992.
- P. Salus, *A Quarter Century of UNIX*, Addison-Wesley, 1994.
- B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd Ed., Prentice-Hall, 1988.
- J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, 1989.
- J. Wordsworth, *Software Development with Z*, Addison-Wesley, 1992.