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 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

Informal Requirements → Requirements Analysis → Requirements Specifications

Customers, Programmers
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 $\mathbb{Z}$

\[
\begin{align*}
\text{LibSystem} & \quad \text{members} : \mathbb{P} \text{Person} \\
& \quad \text{shelved} : \mathbb{P} \text{Book} \\
& \quad \text{checked} : \text{Book} \to \text{Person} \\
\text{shelved} \cap \text{dom checked} &= \emptyset \\
\text{ran checked} &\subseteq \text{members} \\
\forall \text{mem} : \text{Person} \cdot \#(\text{checked} \triangleright \{\text{mem}\}) &\leq \text{MaxLoan}
\end{align*}
\]
Misconceptions about Formal Methods (continued)

Examples in C

• Production code: [Salus 1994]

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

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

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

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

```c
    while (*s++=*t++);
```
Misconceptions about Formal Methods (continued)

Inapplicable to Real Projects


  - 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:
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

Copyright Saiedian © 2006 (For exclusive use of EECS Software Engineering Students)
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

<table>
<thead>
<tr>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
</tr>
</tbody>
</table>
```

- An alternative linear notation

```
SchemaName \= [Signature | 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:
  - `∆` means the state has changed (described later)
  - `Ξ` means no change in the state (described later)
An Example of a Z Schema

<table>
<thead>
<tr>
<th>Reserve</th>
</tr>
</thead>
<tbody>
<tr>
<td>passengers, passengers' : ( \mathbb{P} ) PERSON</td>
</tr>
<tr>
<td>( p? : ) PERSON</td>
</tr>
<tr>
<td>#passengers &lt; CAPACITY</td>
</tr>
<tr>
<td>( p? \notin \text{passengers} )</td>
</tr>
<tr>
<td>passengers' = passengers ( \cup ) {( p? })</td>
</tr>
<tr>
<td>#passengers' \leq ) CAPACITY</td>
</tr>
</tbody>
</table>

Alternative notation:

\[
\text{Reserve} \overset{=}\{ p? : \text{PERSON}; \ldots | p? \notin \text{passengers} \wedge \ldots \}
\]
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 ...

  \[
  S \underbrace{\ldots}_{\ldots} \underbrace{\ldots}_{\ldots}
  \]

  ... 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 $\Delta$ and $\Xi$ conventions.
Schema Inclusion

- Suppose we have the following two schemas:

  \[
  \text{schemaA} : \begin{cases}
  p, q : \mathbb{Z} \\
  p \neq q
  \end{cases}
  \]

  \[
  \text{schemaB} : \begin{cases}
  r, s : \mathbb{N} \\
  r \geq s
  \end{cases}
  \]

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

  \[
  \text{schemaAB} = \text{schemaA} \cup \text{schemaB}
  \]

- If “expanded,” \text{SchemaAB} will include:

  \[
  \text{schemaAB} : \begin{cases}
  p, q : \mathbb{Z}; r, s : \mathbb{N} \\
  p \neq q \land r \geq s
  \end{cases}
  \]
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:

  \[ schemaBA \hat{=} schemaA \lor schemaB \]

- If “expanded,” \(schemaBA\) will include:

\[
\begin{align*}
\text{schemaBA} \\
p, q : \mathbb{Z} \\
r, s : \mathbb{N} \\
p \neq q \lor r \geq s
\end{align*}
\]
Operations and changes in the state data:
Thus, to define an operation we must have:

\[
\begin{align*}
\text{Operation} & \\
\text{StateData} & \\
\text{StateData}' & \\
\text{Predicates describing StateData} & \\
\text{Predicates describing StateData}' & 
\end{align*}
\]
Consider operation *Reserve* again:

<table>
<thead>
<tr>
<th>Reserve</th>
</tr>
</thead>
<tbody>
<tr>
<td>passengers, passengers': PERSON</td>
</tr>
<tr>
<td>p? : PERSON</td>
</tr>
</tbody>
</table>

#passengers < CAPACITY
p? \notin passengers
passengers' = passengers \cup \{p?\}
#passengers' \leq CAPACITY
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:

\[
\begin{align*}
\text{ResSyst} & \quad \text{passengers} : \mathbb{P} \text{PERSON} \\
\#\text{passengers} & \leq \text{CAPACITY} \\
\text{ResSyst}' & \quad \text{passengers}' : \mathbb{P} \text{PERSON} \\
\#\text{passengers}' & \leq \text{CAPACITY}
\end{align*}
\]
• Reserve operation can now be specified as:

\[
\text{Reserve} \\
\text{ResSyst} \\
\text{ResSyst}' \\
p? : \text{PERSON} \\
\#\text{passengers} < \text{CAPACITY} \\
p? \notin \text{passengers} \\
\text{passengers}' = \text{passengers} \cup \{p?\}
\]
While most operations bring about changes in the state data, some do not:

\[ \text{PassengerCount} \]
\[ \text{ResSyst} \]
\[ \text{ResSyst}' \]
\[ \text{count!} : \mathbb{N} \]
\[ \text{count!} = \#\text{passengers} \]
△ and Ξ Conventions (continued)

- △ 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:

\[
\begin{array}{c}
\Delta \text{ResSyst} \\
\text{ResSyst} \\
\text{ResSyst}' \\
\end{array}
\]

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

\[
\begin{array}{c}
\Xi \text{ResSyst} \\
\Delta \text{ResSyst} \\
passengers = passengers' \\
\end{array}
\]
\( \Delta \) and \( \Xi \) Conventions (continued)

- Operation \textit{Reserve} can now be specified using \( \Delta \) convention:

\[
\begin{align*}
\text{Reserve} \\
\Delta \text{ResSyst} \\
p? : \text{PERSON} \\
\#\text{passengers} < \text{CAPACITY} \\
p? \notin \text{passengers} \\
\text{passengers}' = \text{passengers} \cup \{p?\}
\end{align*}
\]

- Similarly, operation \textit{PassengerCount} can now be specified using \( \Xi \) convention:

\[
\begin{align*}
\text{PassengerCount} \\
\Xi \text{ResSyst} \\
count! : \mathbb{N} \\
count! = \#\text{passengers}
\end{align*}
\]
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:
  - $\Delta State \doteq [State; State']$
  - $\Xi State \doteq [\Delta State \mid State = State']$

• 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 ::= "Full" | "Empty" | "OK"
  Colors ::= green | red | blue | white
  ```
Global Definitions

- Introduced by means of axiomatic descriptions:

  \[
  \begin{array}{l}
  \text{Description} \\
  \text{Predicates}
  \end{array}
  \]

- If there is no constraining predicate:

  \[
  \begin{array}{l}
  \text{Description}
  \end{array}
  \]

- Examples:

  \[
  \begin{array}{l}
  \text{Capacity} : \mathbb{N} \\
  \text{Capacity} = 200 \\
  \text{MaxQty}
  \end{array}
  \]
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

\[
\text{LibSystem} \\
\text{members : } \mathbb{P} \text{ PERSON} \\
\text{shelved : } \mathbb{P} \text{ BOOK} \\
\text{checked : BOOK } \rightarrow \text{ PERSON} \\
\text{shelved } \cap \text{ dom checked } = \emptyset \\
\text{ran checked } \subseteq \text{ members} \\
\forall \text{ mem : PERSON } \bullet \#(\text{checked } \triangleright \{\text{mem}\}) \leq \text{MaxLoan}
\]
A Case Study in Z: APhoneDir 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.,
  \(PHONE == 41000 \ldots 49999\)
Present Given, User-defined Types (continued)

- Output messages:

\[
\text{MESSAGE ::= 'OK'
| 'Faculty already exists'
| 'No such faculty'
| 'Faculty has no number'
| 'Invalid number'
| 'Invalid entry'
| 'Entry already exists'}
\]
Abstract State of PhoneDir System

- A set of type \textit{PERSON} representing the faculty:
  \[ \text{faculty} : \mathbb{P} \text{PERSON} \]

- Abstract representation of an instance of \textit{faculty}:

\begin{center}
\begin{tabular}{c}
chen \\
stan \\
mary \\
hossein \\
ideen
\end{tabular}
\end{center}
Abstract State of PhoneDir System (continued)

- We need another set representing a directory in the system: 
  \textit{directory :????}

- Abstract representation of an instance of directory:
• directory is a subset of Cartesian product of 

\[ PERSON \times PHONE \]

Pairs of values: one from \textit{PERSON}, and one from \textit{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:
  \[ \text{directory} = \{ (chen, 42882), (stan, 43338), (stan, 45456), \ldots \} \]

- Alternatively, use ↦ symbol to show connections:
  \[ \text{directory} = \{ \text{chen} \mapsto 42882, \text{stan} \mapsto 43338, \text{stan} \mapsto 45456, \ldots \} \]

- Declaring a variable of type relation:
  \[ \text{directory} : \text{PERSON} \leftrightarrow \text{PHONE} \]
Abstract State of *PhoneDir* System (continued)

Partial list of operations available on a relation:

- **domain**
  \[ \text{dom directory} = \{ chen, stan, hossein, ideen \} \]

- **range**
  \[ \text{ran directory} = \{ 42882, 43338, 45456, 42100 \} \]

- **inverse**
  \[ directory^\sim = \{ 42882 \mapsto chen, 43338 \mapsto stan, \ldots \} \]

- Also domain restriction (\( \triangleleft \)), domain subtraction (\( \triangleleft \)), range restriction (\( \triangleright \)), range subtraction (\( \triangleright\)), relational image (\( \|\| \)), etc.
Abstract State of *PhoneDir* System (continued)

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

- **Before state**

  \[
  \begin{array}{l}
  \text{PhoneDir} \\
  \text{faculty} : \mathbb{P} \text{PERSON} \\
  \text{directory} : \text{PERSON} \leftrightarrow \text{PHONE} \\
  \text{dom directory} \subseteq \text{faculty}
  \end{array}
  \]

- **After state (decorate names with prime)**

  \[
  \begin{array}{l}
  \text{PhoneDir'} \\
  \text{faculty'} : \mathbb{P} \text{PERSON} \\
  \text{directory'} : \text{PERSON} \leftrightarrow \text{PHONE} \\
  \text{dom directory'} \subseteq \text{faculty'}
  \end{array}
  \]
Defining \( \Delta \text{PhoneDir System} \)

- \( \Delta \text{PhoneDir} \):

  \[
  \Delta \text{PhoneDir} \\
  \text{faculty, faculty' : } \mathbb{P} \text{ PERSON} \\
  \text{directory, directory' : } \text{PERSON} \leftrightarrow \text{PHONE} \\
  \text{dom directory } \subseteq \text{faculty} \\
  \text{dom directory'} \subseteq \text{faculty'}
  \]

- Or alternatively as:

  \[
  \Delta \text{PhoneDir} \\
  \text{PhoneDir} \\
  \text{PhoneDir'}
  \]
Defining \( \Xi \text{PhoneDir System} \)

- \( \Xi \text{PhoneDir}: \)
  
  \[
  \begin{array}{c}
  \Xi \text{PhoneDir} \\
  \Delta \text{PhoneDir} \\
  \text{faculty}' = \text{faculty} \\
  \text{directory}' = \text{directory}
  \end{array}
  \]
Defining the Initial State of *PhoneDir System*

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

\[
\begin{align*}
\text{InitPhoneDir'} &= \text{null} \\
\text{PhoneDir'} &= \text{null} \\
\text{faculty'} &= \emptyset \\
\text{directory'} &= \emptyset
\end{align*}
\]
Adding a New Faculty to \textit{PhoneDir} System

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

\[
\begin{align*}
\text{AddMemberOK} \\
\Delta \text{PhoneDir} \\
name?: \text{PERSON} \\
rep!: \text{MESSAGE}
\end{align*}
\]

\[
\begin{align*}
\text{name?} \notin \text{faculty} \\
\text{faculty'} = \text{faculty} \cup \{\text{name?}\} \\
\text{directory'} = \text{directory} \\
\text{rep!} = 'OK'
\end{align*}
\]
Adding a New Faculty to PhoneDir System (continued)

Error Cases:

- Faculty already exists in the PhoneDir, i.e., name? ∈ faculty

\[
\begin{align*}
\text{FacultyExists} & \quad \text{[Faculty already exists]} \\
\exists \text{PhoneDir} & \\
\text{name? : PERSON} & \\
\text{rep! : MESSAGE} & \\
\text{name? ∈ faculty} & \\
\text{rep! = ‘Faculty already exists’}
\end{align*}
\]

- A total definition:

\[
\text{AddMember} \overset{\sim}{=} \text{AddMemberOK} \vee \text{FacultyExists}
\]
Adding an Entry to *PhoneDir* System

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

- **Output report from the operation:** $rep\,!\, : MESSAGE$

- **Successful operation:**

  $AddEntryOK$

  $\Delta PhoneDir$

  $name\,?\, : PERSON$  
  $number\,?\, : PHONE$  
  $rep\,!\, : MESSAGE$

  $name\,?\, \in \text{faculty}$  
  $name\,?\, \rightarrow number\,?\, \notin \text{directory}$  
  $directory' = directory \cup \{name\,?\, \rightarrow number\,?\}$  
  $faculty' = faculty$  
  $rep\,! = 'OK'"
Adding an Entry to *PhoneDir* System (continued)

Error Cases:

- Input not a legitimate faculty, i.e., $name \notin \text{faculty}$

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

- Entry already exists in *directory*, i.e.,

  $name \rightarrow number \in \text{directory}$

\[
\text{EntryExists} \\
\exists \text{PhoneDir} \\
\text{number?} : \text{PHONE} \\
\text{name?} : \text{PERSON} \\
\text{rep!} : \text{MESSAGE} \\
\text{name?} \rightarrow \text{number?} \in \text{directory} \\
\text{rep!} = \text{‘Entry already exists’}
\]
Adding an Entry to *PhoneDir* System (continued)

- Constructing a total operation for adding an entry:

\[
AddEntry \hat{=} AddEntryOK \lor NotFaculty \lor EntryExists
\]
Removing a Faculty from *PhoneDir* System

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

```
RemoveFacultyOK
ΔPhoneDir
name? : PERSON
rep! : MESSAGE

name? ∈ faculty
faculty' = faculty \ {name?}
directory' = \{name?\} ⊲ directory
rep! = ‘OK’
```
Side Notes: Operations on Relations

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

\[
\{s_1, s_3\} \triangleleft \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\} = \\
\{s_2 \mapsto r_2, s_4 \mapsto r_4, s_5 \mapsto r_5\}
\]

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

\[
\{s_1, s_3\} \triangleleft \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3, s_4 \mapsto r_4, s_5 \mapsto r_5\} = \\
\{s_1 \mapsto r_1, s_3 \mapsto r_3\}
\]

- Similar operations for range of a relation, namely range subtraction ($\triangleright$) and range restriction ($\triangleright\triangleright$)
Removing a Faculty from PhoneDir System (continued)

• Error case: there is no such faculty to be removed, i.e.,
  \[ \text{name?} \not\in \text{faculty} \]

  \[
  \exists \text{PhoneDir} \\
  \text{name?} : \text{PERSON} \\
  \text{rep!} : \text{MESSAGE} \\
  \text{name?} \not\in \text{faculty} \\
  \text{rep!} = '\text{No such faculty}'
  \]

• A total definition:

  \[
  \text{RemoveFaculty} \overset{\Delta}{=} \text{RemoveFacultyOK} \lor \text{NotFaculty}
  \]
Removing an Entry from PhoneDir System

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

\[
\text{RemoveEntryOK} \\
\Delta \text{PhoneDir} \\
\text{number?} : \text{PHONE} \\
\text{name?} : \text{PERSON} \\
\text{rep!} : \text{MESSAGE} \\
\text{name?} \leftrightarrow \text{number?} \in \text{directory} \\
\text{directory'} = \text{directory} \setminus \{\text{name?} \leftrightarrow \text{number?}\} \\
\text{faculty'} = \text{faculty} \\
\text{rep!} = 'OK'
\]
Removing an Entry from *PhoneDir* System (continued)

- Error cases: entry does not exists, i.e.,
  
  \[
  \text{name}\? \mapsto \text{number}\? \notin \text{directory}
  \]

  \[
  \text{InvalidEntry} = \exists \text{PhoneDir} \\
  \text{number}\? : \text{PHONE}; \text{name}\? : \text{PERSON} \\
  \text{rep!} : \text{MESSAGE} \\
  \text{name}\? \mapsto \text{number}\? \notin \text{directory} \\
  \text{rep!} = \text{‘Invalid entry’}
  \]

- A total definition:
  
  \[
  \text{RemoveEntry} \triangleq \text{RemoveEntryOK} \lor \text{InvalidEntry}
  \]
Querying the PhoneDir System

Query by person:

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

\[
\begin{align*}
\text{FindNumbersOK} & \quad \text{FindNumbersOK} \\
\exists \text{PhoneDir} & \quad \exists \text{PhoneDir} \\
\text{name?} : \text{PERSON} & \quad \text{name?} : \text{PERSON} \\
\text{numbers!} : \mathbb{P} \text{ PHONE} & \quad \text{numbers!} : \mathbb{P} \text{ PHONE} \\
\text{rep!} : \text{MESSAGE} & \quad \text{rep!} : \text{MESSAGE} \\
\text{name?} \in \text{faculty} & \quad \text{name?} \in \text{dom directory} \\
\text{numbers!} = \text{directory}(|\{\text{name?}\}|) & \quad \text{numbers!} = \text{directory}(|\{\text{name?}\}|) \\
\text{rep!} = 'OK' & \quad \text{rep!} = 'OK'
\end{align*}
\]
Side Notes: Operations on Relations

- Relational image $\left( \left| \right| \right)$: 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

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

- 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 \textit{PhoneDir} System (continued)

Error Cases:

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

- Faculty does not have a number yet, i.e., name\( \not\in \text{directory} \)

\[
\text{InvalidName} \\
\exists \text{PhoneDir} \\
\text{name? : PERSON} \\
\text{rep! : MESSAGE} \\
\text{name? \not\in \text{dom directory}} \\
\text{rep! = ‘Faculty has no number’}
\]

- A total definition

\[
\text{FindNumbers} \overset{\sim}{=} \text{FindNumbersOK} \lor \text{NotFaculty} \lor \text{InvalidName}
\]
Querying the *PhoneDir* System (continued)

Querying by the number:

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

```
FindNamesOK
\exists PhoneDir
\exists PhoneDir
number? : PHONE
names! : \( \mathbb{P} \) PERSON
rep! : MESSAGE
number? \in \text{ran directory}
names! = \text{directory}^{-1}(\{number?\})
rep! = \text{‘OK’}
```
Querying the *PhoneDir* System (continued)

Error Case:

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

\[
\begin{align*}
\text{InvalidNumber} & \quad \text{=} \quad \text{InvalidNumber} \\
\exists \text{PhoneDir} & \\
number? & \colon \text{PHONE} \\
rep! & \colon \text{MESSAGE} \\
\hline
\text{number?} & \notin \text{ran directory} \\
rep! & = \text{‘Invalid number’}
\end{align*}
\]

- A total definition

\[
\text{FindNames} \overset{\hat{}}{=} \text{FindNamesOK} \lor \text{InvalidNumber}
\]
## Summary of Essential Operators on Relations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Synopsis</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>↔</td>
<td>$R_1 \leftrightarrow R_2$</td>
<td>Binary relation between $R_1$ and $R_2$</td>
</tr>
<tr>
<td>→</td>
<td>$r_1 \rightarrow r_2$</td>
<td>Maplet</td>
</tr>
<tr>
<td>dom</td>
<td>dom $R$</td>
<td>Domain of $R$</td>
</tr>
<tr>
<td>ran</td>
<td>ran $R$</td>
<td>Range of $R$</td>
</tr>
<tr>
<td>$-(</td>
<td>-</td>
<td>)$</td>
</tr>
<tr>
<td>△</td>
<td>$S \triangle R$</td>
<td>Domain restriction</td>
</tr>
<tr>
<td>⊲</td>
<td>$S \triangleleft R$</td>
<td>Domain subtraction</td>
</tr>
<tr>
<td>△</td>
<td>$R \triangleright S$</td>
<td>Range restriction</td>
</tr>
<tr>
<td>△</td>
<td>$R \triangleright S$</td>
<td>Range subtraction</td>
</tr>
<tr>
<td>⊕</td>
<td>$R_1 \oplus R_2$</td>
<td>Relational overriding</td>
</tr>
</tbody>
</table>
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 $\rightarrow$
  - injective (one-to-one, i.e., no sharing of elements in $S_2$); can be partial ($\leftrightarrow$) or total ($\rightarrow$)
  - surjective (onto, i.e., the range of function is the entire $S_2$); can be partial ($\rightarrow$) or total ($\rightarrow$)
  - bijective function $\leftrightarrow$: 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 borrow. 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:

  \[ \text{USER, BOOK} \]

- Define \text{MESSAGE} as follow:

  \[
  \text{MESSAGE} ::= \text{‘OK’} \\
  | \text{‘Book not available’} \\
  | \text{‘Invalid return’}
  \]
An Example Using Functions in Z (continued)

- Define the abstract state of the library system

```
LibSys

available : \text{\(\mathbb{P}\) \text{BOOK}}
borrowed : \text{BOOK} \rightarrow \text{USER}

available \cup \text{dom} \text{borrowed} = \text{BOOK}
available \cap \text{dom} \text{borrowed} = \emptyset
```

Why using a partial function?
An Example Using Functions in Z (continued)

- Define $\Delta LibSys$

  $\Delta LibSys$
  
  $\begin{align*}
  \text{available, available'} : \mathbb{P} \text{BOOK} \\
  \text{borrowed, borrowed'} : \text{BOOK} \rightarrow \text{USER}
  \end{align*}$

  $\begin{align*}
  \text{available} \cup \text{dom borrowed} = \text{BOOK} \\
  \text{available} \cap \text{dom borrowed} = \emptyset \\
  \text{available'} \cup \text{dom borrowed'} = \text{BOOK} \\
  \text{available'} \cap \text{dom borrowed'} = \emptyset
  \end{align*}$

- Define $\Xi LibSys$

  $\Xi LibSys \doteq$

  $[\Delta LibSys \mid \text{borrowed'} = \text{borrowed} \land \text{available'} = \text{available'}]$
An Example Using Functions in Z (continued)

- Define the initial abstract state

\[
\begin{align*}
\text{InitLibSys}' & \quad \text{LibSys}' \\
\text{available}' & = \text{BOOK} \\
\text{borrowed}' & = \emptyset
\end{align*}
\]
An Example Using Functions in Z (continued)

- Defining CheckOut operation (successful case)

\[
\text{CheckOutOK} \quad \Delta \text{LibSys} \\
\ u? : \text{USER} \\
\ b? : \text{BOOK} \\
\ rep! : \text{MESSAGE} \\
\]

\[
b? \in \text{available} \\
\text{available}' = \text{available} \setminus \{b?\} \\
\text{borrowed}' = \text{borrowed} \cup \{b? \rightarrow u?\} \\
\ rep! = 'OK' \\
\]
An Example Using Functions in Z (continued)

- Error case: \( b? \notin \text{available} \)

\[
\begin{align*}
\text{NotAvailable} \\
\exists \text{LibSys} \\
\text{\( b? : \text{BOOK} \)} \\
\text{\( rep! : \text{MESSAGE} \)} \\
\text{\( b? \notin \text{available} \)} \\
\text{\( rep! = \text{‘Book not available’} \)}
\end{align*}
\]

- A total definition of \textit{CheckOut}

\[
\text{CheckOut} \doteq \text{CheckOutOK} \lor \text{NotAvailable}
\]
An Example Using Functions in Z (continued)

- Define $\text{ReturnOK}$ for returning a book (successful case)

\[
\text{ReturnOK} \\
\Delta \text{LibSys} \\
u? : \text{USER} \\
b? : \text{BOOK} \\
\text{rep!} : \text{MESSAGE} \\

(b? \mapsto u?) \in \text{borrowed} \\
\text{available'} = \text{available} \cup \{b?\} \\
\text{borrowed'} = \text{borrowed} \setminus \{b? \mapsto u?\} \\
\text{rep!} = '\text{OK}'
\]
An Example Using Functions in Z (continued)

- Error case: recording an incorrect return

\[
\begin{align*}
\text{InvalidReturn} & \equiv \exists LibSys \quad \forall u? : \text{USER} \\
& \quad \forall b? : \text{BLOCK} \\
& \quad \forall rep! : \text{MESSAGE} \\
& \quad (b? \rightarrow u?) \notin \text{borrowed} \\
& \quad rep! = \text{‘Invalid return’}
\end{align*}
\]

- A total definition for \textit{Return}

\[\text{Return} \triangleq \text{ReturnOK} \lor \text{InvalidReturn}\]
### Essential Operators on Functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>Partial function</td>
</tr>
<tr>
<td>↦</td>
<td>Finite partial function</td>
</tr>
<tr>
<td>→</td>
<td>Total function</td>
</tr>
<tr>
<td>→</td>
<td>Partial surjective function</td>
</tr>
<tr>
<td>→</td>
<td>Total surjective function</td>
</tr>
<tr>
<td>↦</td>
<td>Partial injective function</td>
</tr>
<tr>
<td>↦</td>
<td>Partial injective function</td>
</tr>
<tr>
<td>→</td>
<td>Total bijective function</td>
</tr>
</tbody>
</table>
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:
  \[\text{colors} = \langle \text{green, white, red, blue} \rangle\]

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

- In general, a sequence \(s\) of type \(T\) is a function from \(\mathbb{N}\) to \(T\), i.e., \(\text{dom } s = \{1, 2, 3, \ldots, \#s\}\). Formally, \(\text{seq } T = \{f : \mathbb{N} \rightarrow T \mid \text{dom } f = 1 \ldots \#f\}\).
Sequences in Z (continued)

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

$$\begin{align*}
\text{Queue}\ [ELEM] & \quad q : \text{seq}\ ELEM \\
\#q & = \#\text{ran}\ q
\end{align*}$$

- Define $InitQueue'$, $\Delta Queue$, $\Xi Queue$ Schemas

$$\begin{align*}
\Delta Queue & \doteq [\text{Queue};\ \text{Queue}' \mid \#q = \#\text{ran}\ q \land \#q' = \#\text{ran}\ q'] \\
\Xi Queue & \doteq [\Delta Queue \mid q' = q] \\
InitQueue' & \doteq [\text{Queue}' \mid q' = \langle \rangle]
\end{align*}$$
An Example Using Sequences in Z (continued)

- Operation *Enqueue*

\[
\text{Enqueue}[\text{ELEM}] \\
\Delta \text{Queue} \\
e? : \text{ELEM} \\
\text{rep!} : \text{MESSAGE} \\
\]

\[
(e? \notin \text{ran } q \land q' = q \land \langle e? \rangle \land \text{rep!} = 'OK') \\
\lor \\
(e? \in \text{ran } q \land q' = q \land \text{rep!} = 'Duplicate entry')
\]
An Example Using Sequences in Z (continued)

- Operation *Dequeue*

\[
\text{Dequeue } [ELEM] \quad \Delta \text{Queue} \quad e! : ELEM \quad \text{rep!} : \text{MESSAGE}
\]

\[
\begin{align*}
(q \neq \langle \rangle & \land e! = \text{head}(q) \land q' = \text{tail}(q) \land \text{rep!} = \text{‘OK’}) \\
\lor \\
(q = \langle \rangle & \land q' = q \land \text{rep!} = \text{‘Empty queue’})
\end{align*}
\]
An Example Using Sequences in Z (continued)

- Query operations

\[
\text{IsEmpty} \quad [ELEM]
\]
\[\exists Queue\]
\[rep! : \mathbb{B}\]
\[(q = \langle \rangle \land rep!)
\]
\[\lor\]
\[(q \neq \langle \rangle \land \neg rep!)
\]

\[
\text{QueueSize} \quad [ELEM]
\]
\[\exists Queue\]
\[size! : \mathbb{N}\]
\[size! = \#q\]
An Example Using Sequences in Z (continued)

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

\[ \text{PrintQueue} \triangleq \text{Queue}[100..999] \]
Other Notable Aspects of Z

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

\[
\exists [T] \quad _\text{unequal}_ : T \leftrightarrow T \\
\forall r, s : T \cdot r \text{ unequal } s \iff \neg (r = s)
\]

- 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. $PreOp \triangleq Op$; 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 \textit{Reserve}:

\[\text{Reserve} \quad \Delta \text{ResSyst} \quad p? : \text{PERSON} \]

\[p? \notin \text{passengers} \quad \text{passengers}' = \text{passengers} \cup \{p?\}\]
Calculating Pre-Conditions (continued)

- Expand \( \text{Reserve} \) into \( \text{PreReserve} \):

\[
\begin{align*}
\text{PreReserve} & : \mathbb{P} \text{PERSON} \\
p? & : \text{PERSON} \\
\#\text{passengers} & \leq \text{CAPACITY} \\
\#\text{passengers'} & \leq \text{CAPACITY} \\
p? & \notin \text{passengers} \\
\text{passengers'} & = \text{passengers} \cup \{p?\}
\end{align*}
\]
Calculating Pre-Conditions (continued)

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

\[
\begin{align*}
\text{PreReserve} \\
passengers : \mathbb{P} \text{PERSON} \\
p? : \text{PERSON} \\
\exists \text{passengers'} : \mathbb{P} \text{PERSON} \cdot \\
& \left( \#\text{passengers} \leq \text{CAPACITY} \land \\
& \#\text{passengers'} \leq \text{CAPACITY} \land \\
& p? \notin \text{passengers} \land \\
& \text{passengers'} = \text{passengers} \cup \{p?\} \right)
\end{align*}
\]
Calculating Pre-Conditions (continued)

- Simplify the predicate by applying the one-point rule:
  substitute passengers ∪ {p?} where passengers’ occurs

```
PreReserve

  passengers : PERSON
  p? : PERSON

  #passengers ≤ CAPACITY ∧
  #(passengers ∪ {p?}) ≤ CAPACITY ∧
  p? ∉ passengers
```
Calculating Pre-Conditions (continued)

- Further simplifications:

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

  2. Since \( \#\text{passengers} \leq \text{CAPACITY} \) and
     \[ \#\text{passengers} + 1 \leq \text{CAPACITY}, \]
     conclude that \( \#\text{passengers} < \text{CAPACITY} \)
Calculating Pre-Conditions (continued)

- We now have:

  \[
  \text{PreReserve} \quad \begin{align*}
  \text{passengers} &: \mathbb{P} \text{PERSON} \\
  p?: &\text{PERSON} \\
  \#\text{passengers} &< \text{CAPACITY} \land \\
  p? &\notin \text{passengers}
  \end{align*}
  \]

- The pre-conditions of \textit{Reserve} operation are:
  1. \( \#\text{passengers} < \text{CAPACITY} \) and
  2. \( p? \notin \text{passengers} \)
Side Notes for Calculating Pre-Conditions

- One-point rule is defined as follows: [Wordsworth 1992]
  \[
  (\exists x : T \bullet (x = E \land P(x))) \iff (E \in T \land P(E))
  \]
  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 \texttt{pre}\nn
- For a schema \(S\), \texttt{pre} \(S\) is the result of hiding all variables decorated with \(\,'
\) and \(!\).

According to Spivey [Spivey 1992, p.77], \texttt{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:


Introduction to Formal Methods


Industrial Issues:


Articles (continued)

Educational Issues:


• 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:


Z Books (continued)


Conferences Proceedings

- **Formal Methods Europe.** Proceedings published in the Springer-Verlag *Lecture Notes in Computer Science* series

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
- *The Computer Journal*, BCS/IEE
- *Information and Software Technology*, Elsevier Science
- *IEEE Computer*, IEEE-CS
- *IEEE Software*, 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)