Representation of Data/Objects

- Complement to emphasis on representation of activities
  - Foregoing representations all focused on activities
  - Weak capabilities for describing data and objects
  - Seen mostly as effects of activities
  - Numerous places where data descriptions were needed
    - e.g., Request List in elevator example
      - Supposed to be sorted (which way?)
      - Elements had fields (what types?)
- Problems in doing this well
  - What information needed/what questions need answers?
    - Hierarchical decomposition of data
    - Legal actions on data
    - Typing information
  - What forms of representation will be useful?
    - Natural language
    - Diagrammatic
    - Formal language

Primitive Data Representations

- Record Structures
- Array Structures
- Linked List Structures

Specification without Graphs and Diagrams

- The graphs are pictures of relations
- The semantics are provided by the relations
- What do we need the graphs for?
  - Clarity
  - Communication with non-CS people
- Do we need the graphs for communication with CS types?
- Might be more precise, concise
- Not everyone is “visual”

Use of Formalism Alone

- Main advantages are precision and rigor
- Semantic breadth of scope is possible.
  - Some things are hard to draw pictures of
- Main drawback is lack of clarity
  - At least for non-technical people
Formal Languages Approach

• Represent (model) specific aspects of software system using established formalisms (eg. set theory, logic, algebra) to provide the semantics
• Focus on some aspect(s), ignore the others
  – To provide clear understanding of that aspect
  – Avoid clutter
  – Provide rigor and detail about modeled aspect(s)
• We have already seen example(s) of this:
  – Use of finite mathematics, logic, graph theory to provide semantics for diagrams

Advantages of Formal Language

• Diagrams support clarity, good for customers, ??
  – Pictures support intuitive reasoning
  – Help identify gaps, shortcomings, weaknesses
  – Suggest truths, theorems, facts
  – But are generally based upon very weak semantics
    » Lack breadth of semantics
    » Often lack precision and detail
• Formal Languages, good for developers, ???
  – Strength is precision and rigor
  – Broad semantics are possible
  – Often feature considerable detail (that may interfere with clarity)

Defining Data Semantics with Formal Languages

• Focus on defining data types
  – Data type is a set of data instances all having some common characteristics and properties
  – Define the set, and characteristics
• User’s (client’s)-eye view of the data types to be used
• Describe the "accessing primitives" / "operators", "methods", functions providing the only mechanisms for manipulating instances of a given type
  – Dual notion to describing functions in terms of their data inputs and outputs
• Goal: Specify the types without specifying their implementation
• Being rigorous helps separate (even slightly) different notions of a data type from each other

Abstract Data Type Definition Approaches

• Natural language
• Diagrams
• Finite State Machines
• Axiomatic Set Theory
• Algebras


Algebraic Specification

• Draws upon the semantics of algebra to form the basis of the semantics of data types
• Define a type as being the elements of an algebra
• Define the type in terms of how its functions interact with each other
• Consists of two parts:
  – Function list
  – Function interrelations:
• Function list: function templates
• Function interrelations: how the functions interact with each other

Algebraic Specification of a Stack

FUNCTION LIST:

RELATION LIST:
Algebraic Specification of a Stack

FUNCTION LIST:
CREATE: \rightarrow \text{STACK}
PUSH: \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK}
TOP: \text{STACK} \rightarrow (\text{INTEGER} \cup \text{INTERR})
POP: \text{STACK} \rightarrow (\text{STACK} \cup \text{STACKERR})

RELATION LIST:
TOP (PUSH(s,i)) = i
TOP (CREATE) = INTERR
A Different Stack

FUNCTION LIST:
CREATE: \rightarrow TSTACK
PUSH: TSTACK \times INTEGER \rightarrow TSTACK
TOP: TSTACK \rightarrow \{ INTEGER \cup INTERR \}
POP: TSTACK \rightarrow \{ TSTACK \cup STACKERR \}
ISEMPTY: TSTACK \rightarrow \{ TRUE, FALSE \}

RELATION LIST:
TOP(PUSH(s,i)) = i
TOP(CREATE) = INTERR
POP(PUSH(s,i)) = s
POP(CREATE) = STACKERR
ISEMPTY(CREATE) = TRUE, ELSE = FALSE

Yet Another Stack

FUNCTION LIST:
CREATE: \rightarrow LSTACK
PUSH: LSTACK \times INTEGER \rightarrow LSTACK
TOP: LSTACK \rightarrow \{ INTEGER \cup INTERR \}
POP: LSTACK \rightarrow \{ LSTACK \cup STACKERR \}
LENGTH: LSTACK \rightarrow INTEGER

RELATION LIST:
TOP(PUSH(s,i)) = i
TOP(CREATE) = INTERR
POP(PUSH(s,i)) = s
POP(CREATE) = STACKERR
LENGTH(LSTACK) = ……….

What is this good for?

• Focus on the computational aspects of this type
• Provides simplification rules
  – E.g. defines inverses
  – Helps support reasoning about execution traces involving stacks

Example
Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Example
Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(pop(push(create, I)),I)),I)
Example

Create stack;
sum := 0;
I := read();
push(stack, I);
Do forever
  if (top(stack) = sentinel) then exit;
  sum := sum + top(stack);
pop(stack);
I := read();
push(stack, I);
end Do;
Print(sum);

Is stack empty when Print Statement is encountered?

Example: push(pop(push(create, I)), I)

Example

Create stack;
sum := 0;
I := read();
push(stack, I);
Do forever
  if (top(stack) = sentinel) then exit;
  sum := sum + top(stack);
pop(stack);
I := read();
push(stack, I);
end Do;
Print(sum);

Is stack empty when Print Statement is encountered?

Example: push(push(create, I), I)

Example

Create stack;
sum := 0;
I := read();
push(stack, I);
Do forever
  if (top(stack) = sentinel) then exit;
  sum := sum + top(stack);
pop(stack);
I := read();
push(stack, I);
end Do;
Print(sum);

Is stack empty when Print Statement is encountered?

Example: push(pop(push(create, I)), I)

Example

Create stack;
sum := 0;
I := read();
push(stack, I);
Do 10 times
  if (top(stack) = sentinel) then exit;
  sum := sum + top(stack);
I := read();
push(stack, I);
end Do;
Print(sum);

Length of stack now?

Example: push(pop(push(create, I)), I)

Example

Create stack;
sum := 0;
I := read();
push(stack, I);
Do forever
  if (top(stack) = sentinel) then exit;
  sum := sum + top(stack);
pop(stack);
I := read();
push(stack, I);
end Do;
Print(sum);

Is stack empty when Print Statement is encountered?

Example: push(create, I)

Other Types?

• Polynomial
• Set/bag
• Dequeue
• Tree
• Editor
Previous definition doesn’t address some characteristics

Other formalisms make it easier to specify other characteristics

Axiomatic Set Theory ADT Specification

- Semantics of the data type derived from semantics of axiomatic set theory
- Supports rigorous reasoning about design and code
- Describes data type in terms of its abstract behaviors
- Describes accessing functions in terms of relations to each other
- Hard to write correct specifications (and know it)
- Reading them is hard too

Axiomatic Specification of a Stack

STACK(CREATE)

It is true that you can create instances of objects of type “STACK”

Axiomatic Specification of a Stack

STACK(s) ∧ INTEGER(i) ⇒ PUSH(s,i) ≠ CREATE

Push makes a non-empty stack
Axiomatic Specification of Stack

STACK(s) ∧ STACK(s') ∧ INTEGER(i)
⇒ [ PUSH(s,i) = PUSH(s', i) ⇒ (s = s') ]

Pushing doesn't change what was already there
(not covered by algebraic specification)

Axiomatic Specification of Stack

STACK(s) ∧ INTEGER(i) ⇒ TOP(PUSH(s,i)) = i

Last in, first out

Pop and Push are inverses of each other
Axiomatic Specification of Stack

\[ \text{TOP}(\text{CREATE}) = \text{INTEGERERROR} \]
\[ \text{POP}(\text{CREATE}) = \text{STACKERROR} \]

Axiomatic Specification of a Stack

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \]
\[ \text{STACK}(\text{PUSH}(s,i)) \land \]
\[ [\text{POP}(s) \neq \text{STACKERROR} \Rightarrow \text{STACK}(\text{POP}(s))] \land \]
\[ [\text{TOP}(s) \neq \text{INTEGERERROR} \Rightarrow \text{INTEGER}(	ext{TOP}(s))] \]

Axiomatic Specification of a Stack

\[ \forall A \left[ A(\text{CREATE}) \land (\forall s)(\forall i) \right. \]
\[ \left. \left[ \text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \right. \right. \]
\[ \Rightarrow A(\text{PUSH}(s,i)) \land [s \neq \text{CREATE} \Rightarrow A(\text{POP}(s))] \left. \right] \land \]
\[ \Rightarrow \forall s [\text{STACK}(s) \Rightarrow A(s)] \right] \]
Full Axiomatic Specification of a Stack

\[ \begin{align*}
&\text{STACK(CREATE)} \land \text{STACK}(s) \land \text{INTEGER}(i) \implies \text{PUSH}(s,i) \neq \text{CREATE} \\
&\text{STACK}(s) \land \text{INTEGER}(i) \implies \text{TOP}(\text{PUSH}(s,i)) = s \\
&\text{TOP}(\text{CREATE}) = \text{INTEGERERROR} \\
&\text{POP}(\text{CREATE}) = \text{STACKERROR} \\
&\text{STACK}(s) \land \text{INTEGER}(i) \implies \text{STACK}(\text{PUSH}(s,i)) \\
&\text{POP}(s) \neq \text{STACKERROR} \implies \text{STACK}(\text{POP}(s)) \\
&\text{TOP}(s) \neq \text{INTEGERERROR} \implies \text{INTEGER}(\text{TOP}(s)) \\
&\forall A \left[ A(\text{CREATE}) \land (\forall s)(\forall i) \right] \\
&\left[ \text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \right] \\
&\implies A(\text{PUSH}(s,i)) \\
&\implies A(\text{POP}(s))
\end{align*} \]

What is this good for?

- Providing rigorous specifications about types to be built
- Providing rigorous specifications for modules to be built
- Supports logical reasoning about software using this type
- Proving rigorous theorems about software
- We will see more about this soon

Assignment #2

- Should be working on that now
- Adopt stakeholder perspective(s)
- Think about specific stake(s)
- Select specification approach(es) that help make clear to your selected stakeholder(s) what you have in mind for this system in the area of the selected stake
- Due on Monday, 4 March at 11:59PM

Review of Some Past Approaches

- Diagrammatic/Pictorial Activity specifications
  - DFG, CFG, FSA, MSC (Ladder chart)
  - And combinations of these (e.g. Statemate, UML)
- Data/artifact specifications
  - Algebraic and Axiomatic
- Something that combines elements of the above is next

Other Formal Approaches

- System structure: Its modules, their relations,
  - Z (pronounced “zed”)
  - Larch
  - VDL
- Concurrency structure
  - CSP (Cooperating Sequential Processes)
  - TSL (Task Sequence Language)

Z (Pronounced “Zed”)

- Developed at Oxford by Hoare, Spivey, etc.
- Represents WHAT software systems do without specifying HOW
- Uses set theory and function notation
- Describe systems as collections of SCHEMAS
  - inputs and outputs to functions
  - Invariants: statements whose truth is preserved by the functions
- Lots of idiosyncratic notation
- Intent is to make specifications brief, yet clear and precise
The “Birthday Book” Example

- Maintain a repository of information about birthdays
- Consists of (name, birthday) pairs
- Want to add pairs for people whose birthdays are to be remembered
- Want to know whose birthday falls on a given date
- Don’t care about how this is implemented

Example Schema

BirthdayBook

known : ° NAME

birthday : NAME → DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.

Example Schema

Name of the schema

BirthdayBook

known : ° NAME

birthday : NAME → DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.

Example Schema

Name of the schema

BirthdayBook

known : ° NAME

birthday : NAME → DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.

Example Schema

Name of the schema

BirthdayBook

known : ° NAME

birthday : NAME → DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.
Another Schema

This Delta symbol indicates that this schema will describe a state change

AddBirthday

Δ BirthdayBook

name? : NAME
date? : DATE

name? known
birthday = birthday' U (name? date?)

This Delta symbol indicates that this schema will describe a state change

AddBirthday

The ? after these symbols indicates that the element will be an input

name? known
birthday = birthday' U (name? date?)

When the schema describes a change of state then it is necessary to distinguish between the value of an element before the state change and the value after The denotes the value after the state change

Indicates there will be no change in schema state

FindBirthday

Ξ BirthdayBook

name? : NAME
date! : DATE

name? known
date! = birthday (name?)

Remind

Ξ BirthdayBook

today? : DATE
cards!: \{ n : known | birthday(n) = today? \}

cards! = \{ n : known | birthday(n) = today? \}

Indicates there will be no change in schema state

FindBirthday

Ξ BirthdayBook

name? : NAME
date! : DATE

name? known
date! = birthday (name?)

Remind

Ξ BirthdayBook

today? : DATE
cards!: \{ n : known | birthday(n) = today? \}

cards! = \{ n : known | birthday(n) = today? \}
Two more Schemas

BirthdayBook

| name? : NAME |
| date! : DATE |

FindBirthday

ε known date! = birthday (name?)

BirthdayBook

| today? : DATE |
| card! : NAME |

cards = { n : known | birthday(n) = today? }

This denotes the set of all elements n, drawn from the set known, such that birthday(n) = today?

What is Tokeneer?

- Software system for controlling access to a high-security area
- Developed so that security and other requirements could be formally verified
- Requirements specification is contained on a large document
  - Many different specification approaches are used
  - We saw some use cases
  - Now see some Z specifications

3 Domain Information

This section describes the existing Tokeneer system and the existing SSU. This allows the proposed system to parallel the existing system in terms of section 2.

The SSU is part of the larger Tokeneer system, an access system. Analysis of the Tokeneer system and the requirements on the system is necessary to answer the same questions on both. Understanding of the system can follow requirements to obtain better view of the system, and given that it continues to be implemented, this idea will not be altered.

6.4 Reading a Fingerprint

[Diagram of Tokeneer system with arrows and labels]

A finger will be read if the system is currently waiting for it and the user Token is in place.

- AddFingerIf
  - CommitUser
  - ValidateUser
  - ValidateFinger
  - CommandFrame
  - CommandFrame
  - GetCommand

- ReadFingerIf
  - ReadFinger
  - ReadFinger

- AddFingerIf
  - AddFinger
  - AddFinger
  - AddFinger
Z Features

- **Z** specifies the state of the domain being changed by the software system being described
  - And the ways in which it can be changed
- Like FSA in specifying state changes
- But with strong focus on data
  - Like ADT specifications in that way
- Schemas can be grouped and composed
  - Allows specifying large and complex systems

Z Stakeholders

- Emphasis on what a system is supposed to do
- Indication of how it looks externally
- Provides information of concern to
  - Users, Customers
    - Who want to know what the system does
    - But have to be mathematically sophisticated
  - Implementers
    - Who need to know what they have to build
  - Regulators
    - Who want to be able to reason to be sure that the system is well-conceived
- Less useful to
  - Technically unsophisticated