Software Models and Representations
Part 4

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 breath 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 (e.g., 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:

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:
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: → STACK

PUSH: STACK X INTEGER → STACK
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \rightarrow STACK

PUSH: STACK \times INTEGER \rightarrow STACK

TOP: STACK \rightarrow \{ INTEGER \cup INTERR \}

POP: STACK \rightarrow \{ STACK \cup STACKERR \}

RELATION LIST:

TOP (PUSH(s,i)) = i

TOP (CREATE) = INTERR
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) = \text{INTERR}

POP (PUSH(s,i)) = s

POP (CREATE) = \text{STACKERR}

A Different 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} \}

ISEMPTY: \text{STACK} \rightarrow \{ \text{TRUE, FALSE} \}

RELATION LIST:

TOP (PUSH(s,i)) = i

TOP (CREATE) = \text{INTERR}

POP (PUSH(s,i)) = s

POP (CREATE) = \text{STACKERR}

ISEMPTY(CREATE) = \text{TRUE, ELSE = FALSE}
Yet Another Stack

FUNCTION LIST:

CREATE: \( \rightarrow \) STACK

PUSH: STACK \( \times \) INTEGER \( \rightarrow \) STACK

TOP: STACK \( \rightarrow \) \{ INTEGER \cup \) INTERR \}

POP: STACK \( \rightarrow \) \{ STACK \cup \) STACKERR \}

LENGTH: STACK \( \rightarrow \) INTEGER

RELATION LIST:

TOP (PUSH(s,i)) = i

TOP (CREATE) = INTERR

POP (PUSH(s,i)) = s

POP (CREATE) = STACKERR

LENGTH (STACK) = ..........

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);

Is stack empty when Print Statement is encountered?
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);

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);

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);

Example: push(create, 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?
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) \implies \text{PUSH}(s, i) \neq \text{CREATE}
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

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{TOP(PUSH}(s,i)) = i \]
Axiomatic Specification of Stack

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

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

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

Expected behavior of empty stack
Axiomatic Specification of a Stack

STACK(s) ∧ INTEGER(i) ⇒
  STACK(PUSH(s,i)) ∧
  [POP(s) ≠ STACKERROR ⇒ STACK(POP(s))] ∧
  [TOP(s) ≠ INTEGERERROR ⇒ INTEGER(TOP(s))]
Axiomatic Specification of a Stack

∀ A [ A(CREATE) ∧ (∀ s)(∀ i)

    [STACK(s) ∧ INTEGER(i) ∧ A(s)

    ⇒ A(PUSH(s,i)) ∧ [s ≠ CREATE ⇒ A(POP(s)) ] ]

⇒ ∀ s [STACK(s) ⇒ A(s ) ]
Full Axiomatic Specification of a Stack

\[
\begin{align*}
\text{STACK}(\text{CREATE}) & \\
\land \text{STACK}(s) \land \text{INTEGER}(i) & \implies \text{PUSH}(s, i) \neq \text{CREATE} \\
\land \text{STACK}(s) \land \text{STACK}(s') \land \text{INTEGER}(i) & \implies \{ \text{PUSH}(s, i) = \text{PUSH}(s', i) \implies (s = s') \} \\
\land \text{STACK}(s) \land \text{INTEGER}(i) & \implies \text{TOP}(\text{PUSH}(s, i)) = 1 \\
\land \text{STACK}(s) \land \text{INTEGER}(i) & \implies \text{POP}(\text{PUSH}(s, i)) = s \\
\land \text{TOP}(\text{CREATE}) & = \text{INTEGERERROR} \\
\land \text{POP}(\text{CREATE}) & = \text{STACKERROR} \\
\land \text{STACK}(s) \land \text{INTEGER}(i) & \implies \\
& \quad \text{STACK}(\text{PUSH}(s, i)) \land \\
& \quad \{ \text{POP}(s) \neq \text{STACKERROR} \implies \text{STACK}(\text{POP}(s)) \} \land \\
& \quad \{ \text{TOP}(s) \neq \text{INTEGERERROR} \implies \text{INTEGER}(\text{TOP}(s)) \} \\
\land \forall A \quad [A(\text{CREATE}) \land (\forall s)(\forall i) \land \\
& \quad [\text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \\
& \quad \implies A(\text{PUSH}(s, i)) \land [s \neq \text{CREATE} \implies A(\text{POP}(s))] \} \\
& \quad \implies \forall s [\text{STACK}(s) \implies A(s)] \\&
\end{align*}
\]

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
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

```
known : P 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 : P 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 : P 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

**BirthdayBook**

- **known**: \( \mathcal{P} \text{NAME} \)
- **birthday**: \( \text{NAME} \rightarrow \text{DATE} \)

**Known** is a set of elements of type `NAME`.

- **known = dom birthday**

**BirthdayBook** is a function defined on a set of `NAME` and mapping into a set `DATE`.

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

---

### Example Schema

**BirthdayBook**

- **known**: \( \mathcal{P} \text{NAME} \)
- **birthday**: \( \text{NAME} \rightarrow \text{DATE} \)

**Known** is a set of elements of type `NAME`.

- **known = dom birthday**

**BirthdayBook** is a function defined on a set of `NAME` and mapping into a set `DATE`.

**The invariant part of the schema:**

- The set `known` is the domain of definition of the function `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

\[ \Delta \text{BirthdayBook} \]

name? : NAME
date? : DATE

name? #known
birthday' = birthday \cup \{name? \rightarrow date? \}
```

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

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

AddBirthday

BirthdayBook

ame? : NAME
date? : DATE

name? \# known
birthday' = birthday \cup \{name? \rightarrow date? \}

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

Two more Schemas

FindBirthday

BirthdayBook

ame? : NAME
date! : DATE

name? \# known
date! = birthday (name?)

Remind

BirthdayBook

today? : DATE
cards! : \{NAME

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

FindBirthday

\[ \Xi \text{BirthdayBook} \]

- name? : NAME
- date! : DATE

name? \in known

date! = \text{birthday}(\text{name?})

Remind

\[ \Xi \text{BirthdayBook} \]

today? : DATE

cards! : \{ NAME \}

\[ \text{cards!} = \{ n : \text{known} \mid \text{birthday}(n) = \text{today}? \} \]
Two more Schemas

FindBirthday

BirthdayBook

name? : NAME

date! : DATE

name? \in known

date! = birthday (name?)

Remind

BirthdayBook

today? : DATE

cards! : \{NAME

\}
cards! = \{ n : known \mid \text{birthday}(n) = \text{today}\}

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

Z Summary

- Schemas can be grouped and composed
- More notation: aimed at facilitating terse, precise communication
- Emphasis on what a system is supposed to do
- Indication of how it looks externally
- (Like Abstract Data Type specifications) basis for going on to think about HOW to implement