SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2018

09 Module Interface Specification (H&S Ch. 7, Ghezzi Ch. 4)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

January 23, 2018

09 Module Interface Specification (H&S Ch. 7, Ghezzi Ch. 4)

- Administrative details
- pdfnup
- Overview of MIS
- MIS Template
  - Syntax
  - Semantics
- Sequence example (abstract object)

Administrative Details

- Assignment 1
  - Partner Files: January 28, 2018
  - Part 2: January 31, 2018
- Questions on assignment?

pdfnup

- If you like to print the lecture and/or tutorial slides consider printing 4 (or more) slides per page
- Consider using pdfjam
- pdfjam provides pdfnup for "n-upping" pages
- `alias pdfnup='pdfnup --nup 2x2 --frame true --paper letterpaper --scale 0.9'`

Sequences

- A sequence is an ordered collection of elements of the same type
  - Elements can occur more than once
  - Sometimes referred to as a list
  - Similar to an array
- Declare a sequence of type T by *sequence of* T
- $< x_0, x_1, ..., x_n >$ for $n \geq 0$ for a sequence with elements $x_0, x_1, ..., x_n$
- $<>$ is the empty sequence
- Position in a sequence is zero relative

Overview of MIS

- The MIS precisely specifies the modules observable behaviour - what the module does
- The MIS does not specify the internal design
- The idea of an MIS is inspired by the principles of software engineering
- Advantages
  - Improves many software qualities
  - Programmers can work in parallel
  - Assumptions about how the code will be used are recorded
  - Test cases can be decided on early, and they benefit from a clear specification of the behaviour
  - A well designed and documented MIS is easier to read and understand than complex code
  - Can use the interface without understanding details

Overview of MIS

- Options for specifying an MIS
  - Trace specification
  - Pre and post conditions specification
  - Input/output specification
  - Before/after specification - module state machine
  - Algebraic specification
- Best to follow a template

MIS Template

- Uses
  - Imported constants, data types and access programs
- Syntax
  - Exported constants and types
  - Exported functions (access routine interface syntax)
- Semantics
  - State variables
  - State invariants
  - Assumptions
  - Access routine semantics
  - Local functions
  - Local types
  - Local constants
  - Considerations

MIS Uses Section

- Specify imported constants
- Specify imported types
- Specify imported access programs
- The specification of one module will often depend on using the interface specified by another module
- When there are many modules the uses information is very useful for navigation of the documentation
- Documents the use relation between modules

MIS Syntax Section

- Specify exported constants
- Specify exported types
- Specify access routine names, the input and output parameter types and exceptions
- Show access routines in tabular form
  - Important design decisions are made at this point
  - Later we will discuss qualities of a good interface, like minimal, essential, etc.
  - The goal is to have the syntax match many implementation languages
  - The mapping to a programming language will not always be the same; it depends on the syntax of the programming language

Syntax of a Sequence Module (Abstract Object)

Exported Constants

MAX_SIZE $= 100$

Syntax of a Sequence Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
Seq_init			
Seq_add	integer, integer		FULL, POS
Seq_del	integer		POS
Seq_setval	integer, integer		POS
Seq_getval	integer	integer	POS
Seq_size		integer	

MIS Semantics Section

- State variables
  - Give state variable(s) name and type
  - State variables define the state space
  - If a module has state then it will have "memory"
- State invariant
  - A predicate on the state space that restricts the "legal" states of the module
  - After every access routine call, the state should satisfy the invariant
  - Cannot have a state invariant without state variables
  - Just stating the invariant does not "enforce" it, the access routine semantics need to maintain it
  - Useful for understandabilty, testing and for proof

Semantics Section Continued

- Local functions, local types and local constants
  - Declared for specification purposes only
  - Not available at run time
  - Helpful to make complex specifications easier to read
- Considerations
  - For information that does not fit elsewhere
  - Useful to tell the user if the module violates a quality criteria

Sequence MIS Semantics

State Variables

What type should the state variable have?

State Invariant

What state invariant should we have?

Assumptions

Seq_init() is called before any other access program

Sequence MIS Semantics

State Variables

s: sequence of integer

State Invariant

What state invariant should we have?

Assumptions

Seq_init() is called before any other access program

Sequence MIS Semantics

State Variables

s: sequence of integer

State Invariant

$|s| \leq \text{MAX\_SIZE}$

Assumptions

Seq_init() is called before any other access program

Sequence MIS Semantics Continued

Access Routine Semantics

Seq_init():

- transition: What should the state transition be?
- exception: none

Seq_add(i, p):

- transition: $s := s[0..i-1] || < p > || s[i..?]$
- exception: $exc := (|s| = ?)$ What exceptions? How characterized?

Sequence MIS Semantics Continued

Access Routine Semantics

Seq_init():

- transition: $s :=<>$
- exception: none

Seq_add(i, p):

- transition: $s := s[0..i-1]|| < p > ||s[i..|s|-1]$
- exception: $exc := (|s| =?)$ What exceptions? How characterized?

Sequence MIS Semantics Continued

Access Routine Semantics

Seq_init():

- transition: $s :=<>$
- exception: none

Seq_add(i, p):

- transition: $s := s[0..i-1]||<p>||s[i..|s|-1]$
- exception:
  $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL} \mid i \notin [0..|s|] \Rightarrow \text{POS})$

Access Routine Semantics Continued

Seq_del(i):

- transition: $s :=?$
- exception: $exc :=?$

Seq_setval(i, p):

- transition: ?
- exception: ?

Seq_getval(i):

- output: ?
- exception: ?

Access Routine Semantics Continued

Seq_del(i):

- transition: $s := s[0..i-1]||s[i+1..|s|-1]$
- exception: $exc :=?$

Seq_setval(i, p):

- transition: ?
- exception: ?

Seq_getval(i):

- output: ?
- exception: ?

Access Routine Semantics Continued

Seq_del(i):

- transition: $s := s[0..i-1] || s[i+1..|s|-1]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_setval(i, p):

- transition: ?
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_getval(i):

- output: ?
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Access Routine Semantics Continued

Seq_del(i):

- transition: $s := s[0..i-1]||s[i+1..|s|-1]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_setval(i, p):

- transition: $s[i] := p$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_getval(i):

- output: ?
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Access Routine Semantics Continued

Seq_del(i):

- transition: $s := s[0..i-1] || s[i+1..|s|-1]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_setval(i, p):

- transition: $s[i] := p$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Seq_getval(i):

- output: $out := s[i]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{POS})$

Access Routine Semantics Continued

Seq_size():

- output: ?
- exception: ?

Access Routine Semantics Continued

Seq_size():

- output: $out := |s|$
- exception: none

Homework

How would you implement Seq in Python? Remember Seq is an abstract object.