CAS 741, CES 741 (Development of Scientific Computing Software)

Fall 2017

14 Module Interface Specification (MIS)

Dr. Spencer Smith

Faculty of Engineering, McMaster University

October 27, 2017

Module Interface Specification (MIS)

- Administrative details
- Questions?
- Module guide example
- Integration testing
- Mathematical review ([4] and separate slides)
  - Multiple assignment statement
  - Conditional rules
  - etc.
- MIS overview
- Modules with external interaction
- Abstract objects
- Abstract data types
- Generic MIS
- Inheritance

Administrative Details

- GitHub issues for colleagues
  - Assigned 1 colleague (see `Repos.xlsx` in repo)
  - Provide at least 5 issues on their VnV plan
  - Grading as before
  - Due by Tuesday, Oct 31, 11:59 pm
- For MG presentation, we'll try to use my laptop only
- Template for MG in repo

Administrative Details: Deadlines

MG Present	Week 08	Week of Oct 30
MG	Week 09	Nov 8
MIS Present	Week 10	Week of Nov 13
MIS	Week 11	Nov 22
Impl. Present	Week 12	Week of Nov 27
Final Documentation	Week 13	Dec 6

Administrative Details: Presentation Schedule

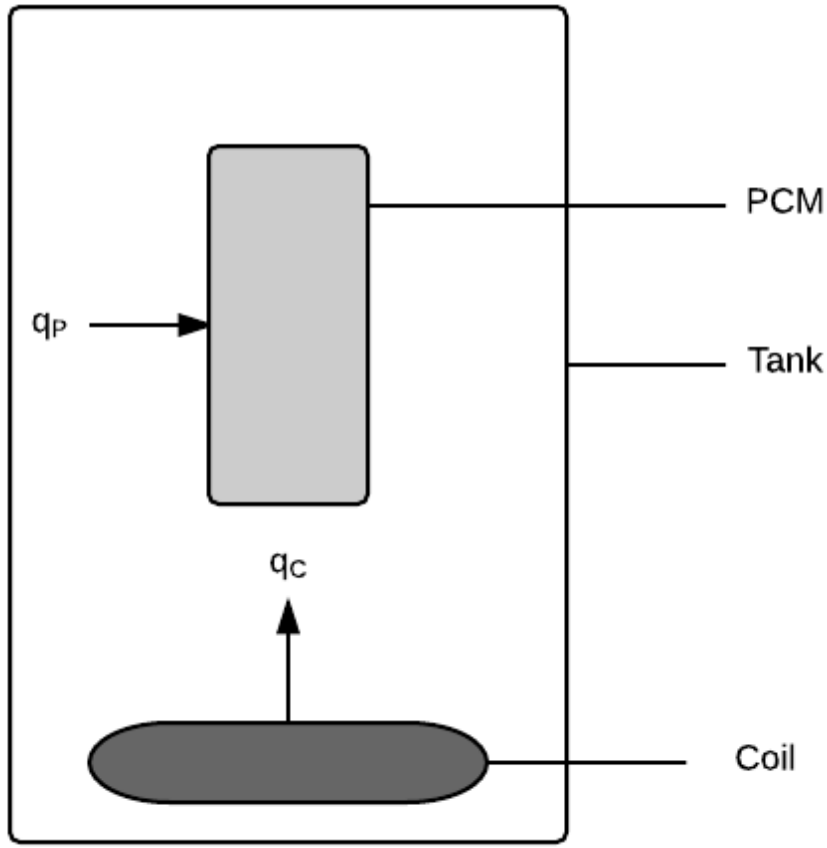
- MG Present
  - **Tuesday: Xiaoye, Yuzhi, Devi, Keshav, Alex P, Paul**
  - **Friday: Shusheng, Jason, Geneva, Alex S, Isobel, Steven**
- MIS Present
  - Tuesday: Isobel, Keshav, Paul
  - Friday: Shusheng, Xiaoye, Devi
- Impl. Present
  - Tuesday: Alexander S., Steven, Alexandre P.
  - Friday: Jason, Geneva, Yuzhi

Questions?

- Questions about Module Guide and the presentation?

Solar Water Heating System Example

- https://github.com/smiths/swhs
- Solve ODEs for temperature of water and PCM
- Solve for energy in water and PCM
- Generate plots



Anticipated Changes?

What are some anticipated changes?

Hint: the software follows the Input $\rightarrow$ Calculate $\rightarrow$ Output design pattern

Anticipated Changes

- The specific hardware on which the software is to run
- The format of the initial input data
- The format of the input parameters
- The constraints on the input parameters
- The format of the output data
- The constraints on the output results
- How the governing ODEs are defined using the input parameters
- How the energy equations are defined using the input parameters
- How the overall control of the calculations is orchestrated
- The implementation of the sequence data structure
- The algorithm used for the ODE solver
- The implementation of plotting data

Module Hierarchy by Secrets

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	Input Parameters Module Output Format Module Temperature ODEs Module Energy Equations Module Control Module Specification Parameters
Software Decision Module	Sequence Data Structure Module ODE Solver Module Plotting Module

Table: Module Hierarchy

Example Modules from SWHS

Hardware Hiding Modules

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

Example Modules from SWHS

Input Parameters Module

Secrets: The data structure for input parameters, how the values are input and how the values are verified. The load and verify secrets are isolated to their own access programs (like submodules).

Services: Gets input from user (including material properties, processing conditions, and numerical parameters), stores input and verifies that the input parameters comply with physical and software constraints. Throws an error if a parameter violates a physical constraint. Throws a warning if a parameter violates a software constraint.

Implemented By: SWHS

Example Modules from SWHS

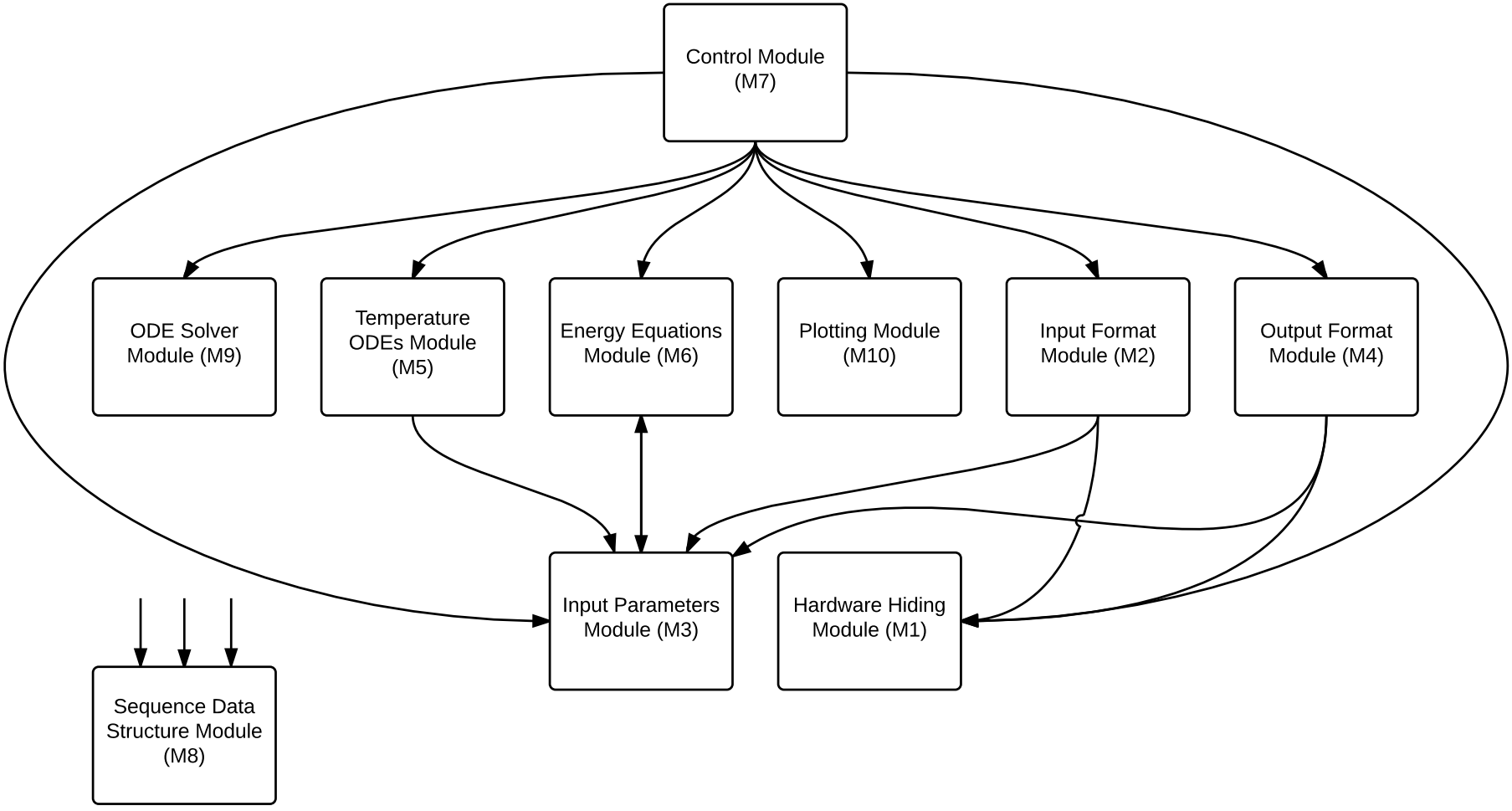
ODE Solver Module

Secrets: The algorithm to solve a system of first order ODEs initial value problem from a given starting time until the given event function shows termination.

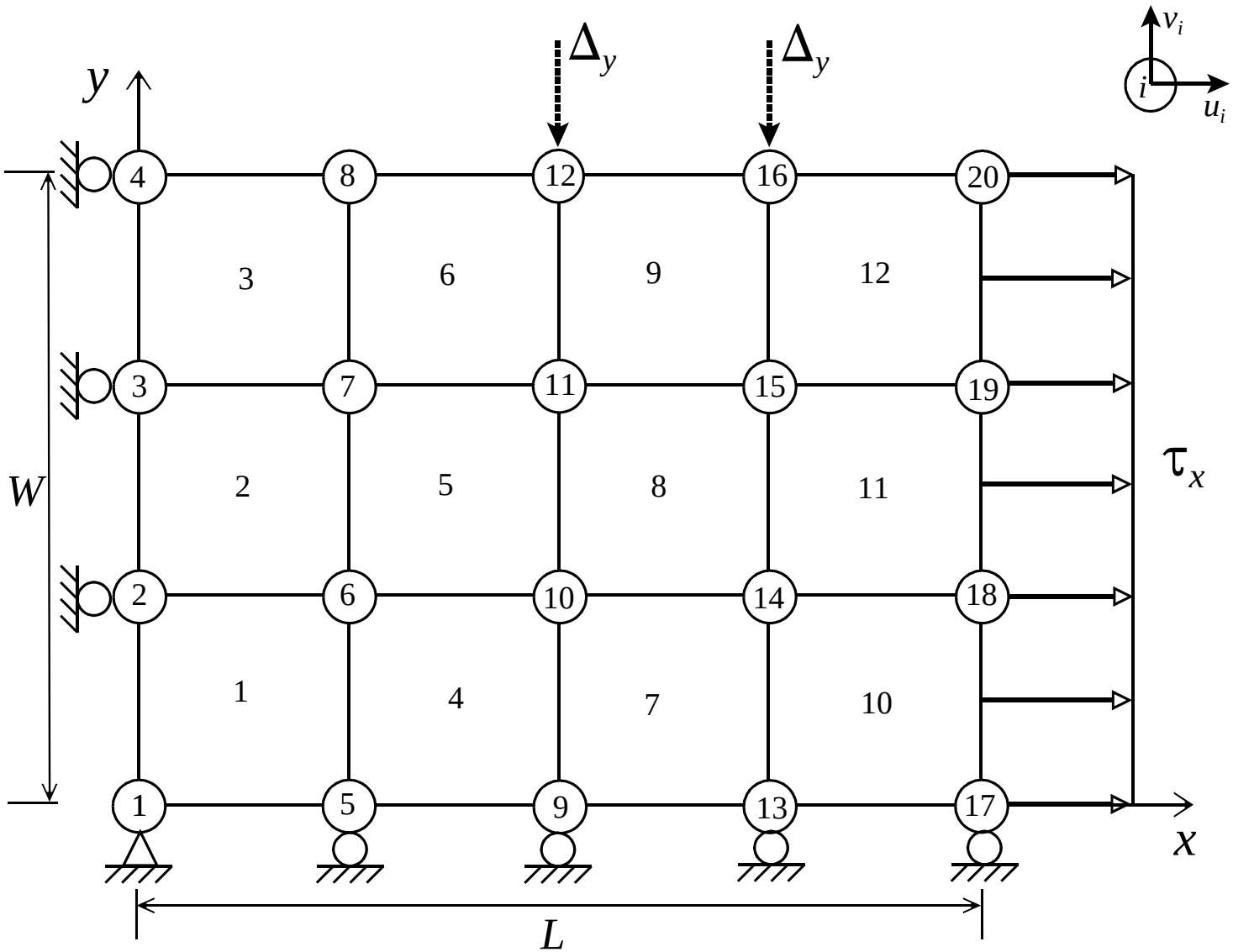
Services: Solves an ODE using the governing equation, initial conditions, event function and numerical parameters.

Implemented By: Matlab

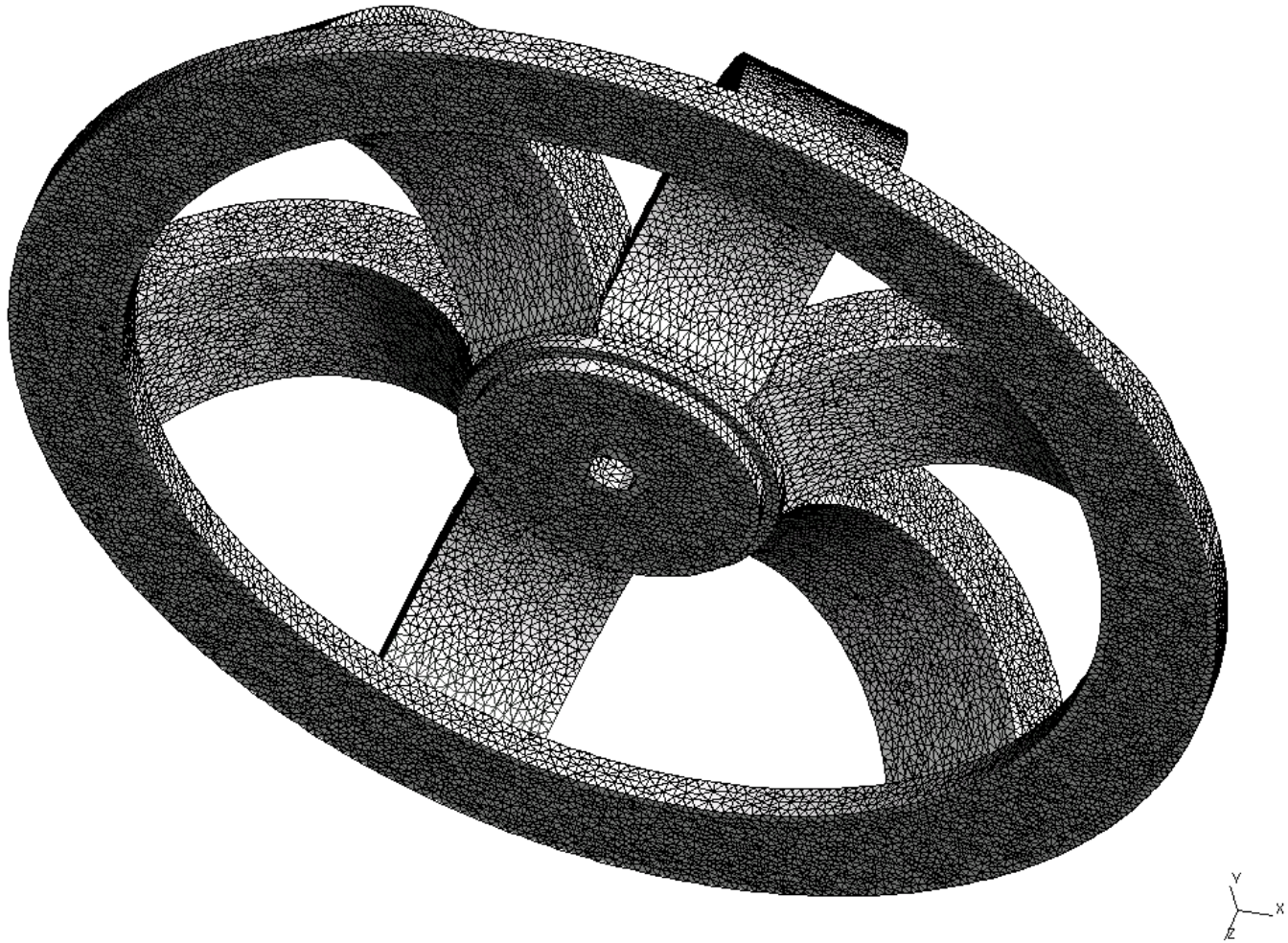
SWHS Uses Hierarchy (approximately)



Mesh Generator Example



Mesh Generator Complex Circular Geometry



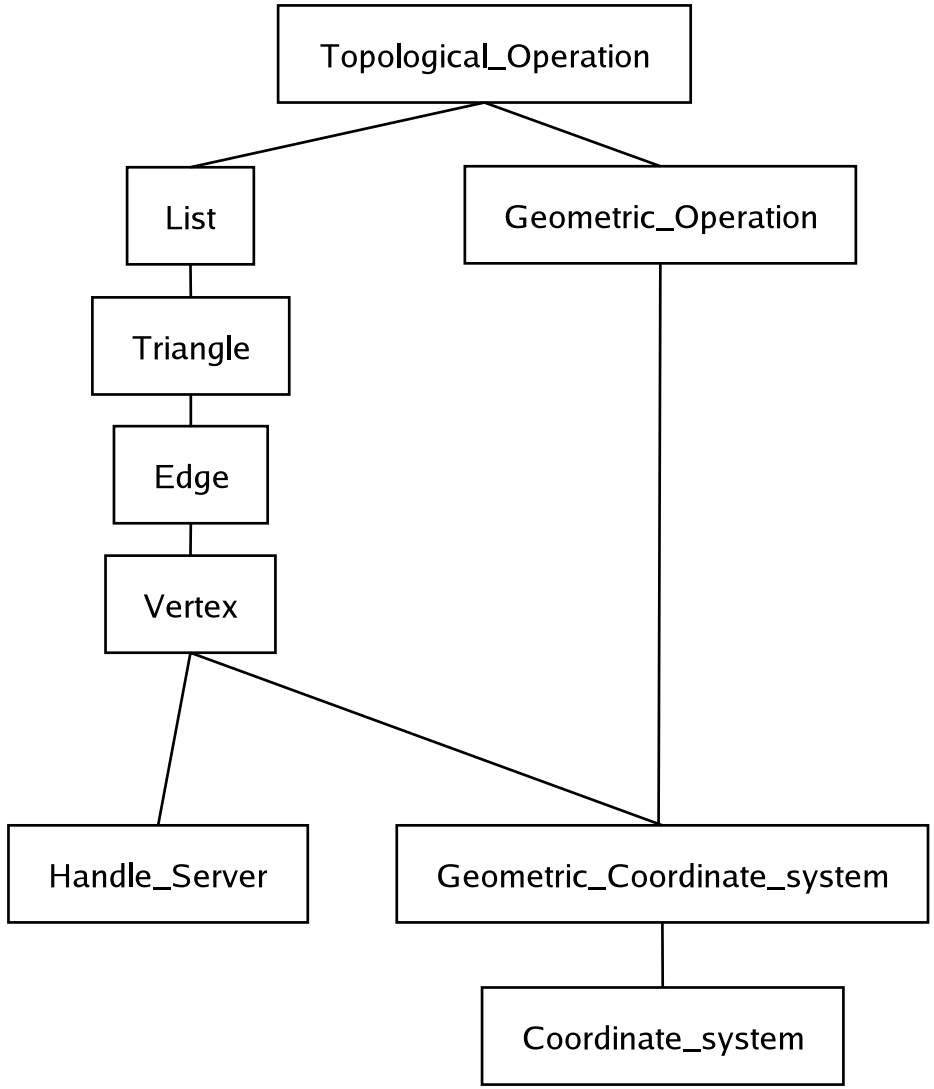
Mesh Generator Example: Design Goals

- Independent and flexible representation for each mesh entity
- Complete separation of geometric data from the topology
- The mesh generator should work with different coordinate systems
- A flexible data structure to store sets of vertices, edges and triangles
- Different mesh generation algorithms with a minimal amount of local changes

Example Mesh Gen Modular Decomposition

Link

Another Mesh Generator Uses Hierarchy [2]



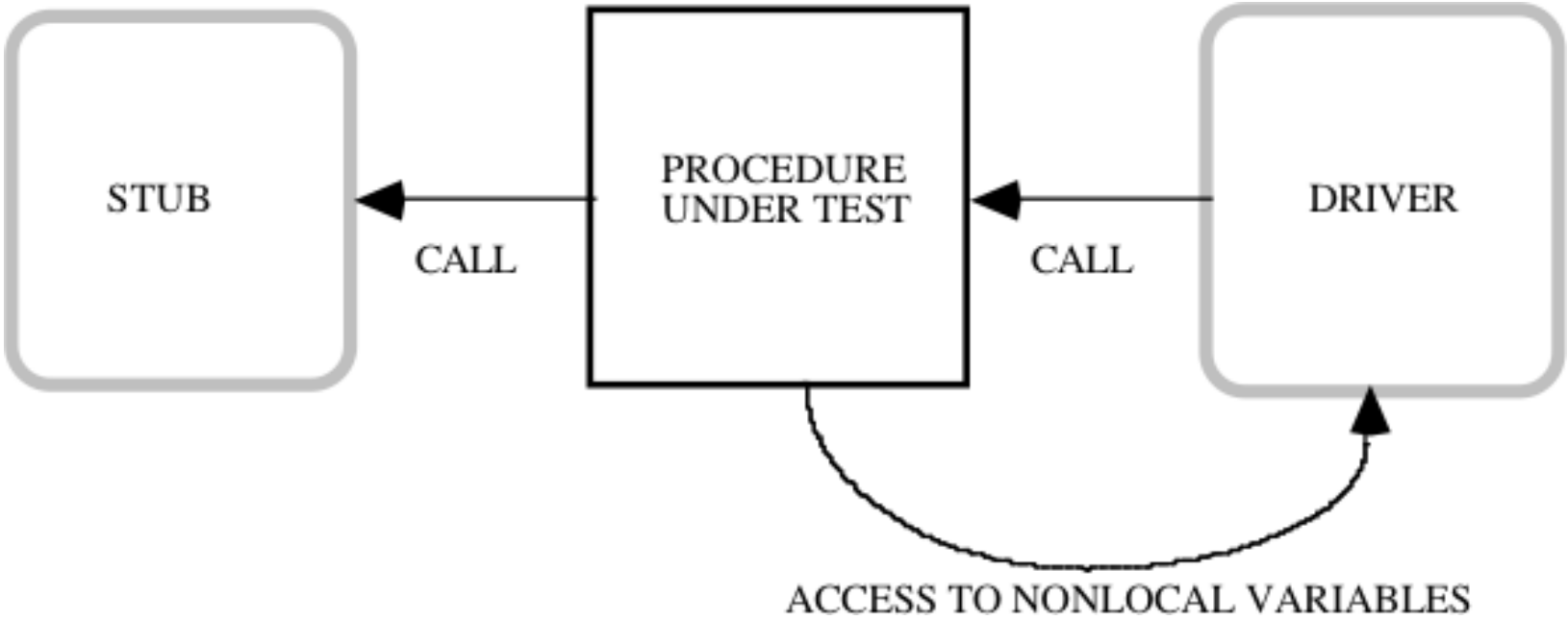
Module Testing

Is it possible to begin testing before all of the modules have been implemented when there is a use relation between modules?

Module Testing [3]

- Scaffolding needed to create the environment in which the module should be tested
- Stubs - a module used by the module under test
- Driver - module activating the module under test

Testing a Functional Module [3]



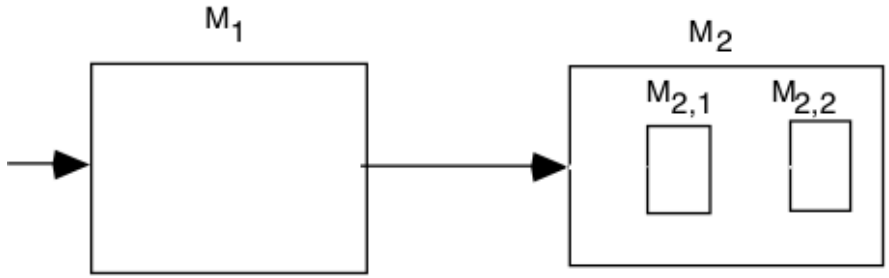
Integration Testing

- Big-bang approach
  - First test individual modules in isolation
  - Then test integrated system
- Incremental approach
  - Modules are progressively integrated and tested
  - Can proceed both top-down and bottom-up according to the USES relation

Integration Testing and USES relation

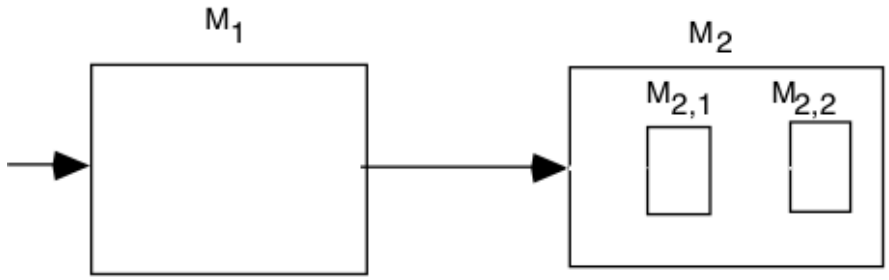
- If integration and test proceed bottom-up only need drivers
- Otherwise if we proceed top-down only stubs are needed

Example [3]



- M_1 USES M_2 and M_2 IS_COMPOSED_OF $\{M_{2,1}, M_{2,2}\}$
- In what order would you test these modules?

Example [3]



- M_1 USES M_2 and M_2 IS_COMPOSED_OF $\{M_{2,1}, M_{2,2}\}$
- Case 1
  - Test M_1 providing a stub for M_2 and a driver for M_1
  - Then provide an implementation for $M_{2,1}$ and a stub for $M_{2,2}$
- Case 2
  - Implement $M_{2,2}$ and test it by using a driver
  - Implement $M_{2,1}$ and test the combination of $M_{2,1}$ and $M_{2,2}$ (i.e. M_2) by using a driver
  - Finally implement M_1 and test it with M_2 using a driver for M_1

Overview of MIS

- See Hoffman and Strooper [4]
- 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
  - The goal is to have the syntax match many implementation languages

Syntax of a Sequence Module

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

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: $s := <>$
- 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: $s := s[0..i-1]||s[i+1..|s|-1]$
- 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: $out := |s|$
- exception: none

Exception Signaling

- Useful to think about exceptions in the design process
- Will need to decide how exception signalling will be done
  - A special return value, a special status parameter, a global variable
  - Invoking an exception procedure
  - Using built-in language constructs
- Caused by errors made by programmers, not by users
- Write code so that it avoid exceptions
- Exceptions will be particularly useful during testing

Assumptions versus Exceptions

- The assumptions section lists assumptions the module developer is permitted to make about the programmer's behaviour
- Assumptions are expressed in prose
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation
- Interface design should provide the programmer with a means to check so that they can avoid exceptions
- When an exceptions occurs no state transitions should take place, any output is *don't care*

Quality Criteria

- Consistent
  - Name conventions
  - Ordering of parameters in argument lists
  - Exception handling, etc.
- Essential - omit unnecessary features
- General - cannot always predict how the module will be used
- As implementation independent as possible
- Minimal - avoid access routines with two potentially independent services
- High cohesion - components are closely related
- Low coupling - not strongly dependent on other modules
- Opaque - information hiding

SWHS Example

Look at SWHS repo

Modules with External Interaction

- In general, some modules may interact with the environment or other modules
- Environment might include the keyboard, the screen, the file system, motors, sensors, etc.
- Sometimes the interaction is informally specified using prose (natural language)
- Can introduce an environment variable
  - Name, type
  - Interpretation
- Environment variables include the screen, the state of a motor (on, direction of rotation, power level, etc.), the position of a robot

External Interaction Continued

- Some external interactions are hidden
  - Present in the implementation, but not in the MIS
  - An example might be OS memory allocation calls
- External interaction described in the MIS
  - Naming access programs of the other modules
  - Specifying how the other module's state variables are changed
  - The MIS should identify what external modules are used

MIS for GUI Modules

- Could introduce an environment variable
- window: sequence [RES_H][RES_V] of pixelT
  - Where window[r][c] is the pixel located at row r and column c, with numbering zero-relative and beginning at the upper left corner
  - Would still need to define pixelT
- Could formally specify the environment variable transitions
- More often it is reasonable to specify the transition in prose
- In some cases the proposed GUI might be shown by rough sketches

Display Point Masses Module Syntax

Exported Access Programs

Routine name	In	Out	Exc
DisplayPointMassesApplet		DisplayPointMassesApplet	
paint			

Display Point Masses Module Semantics

Environment Variables

win : 2D sequence of pixels displayed within a web-browser

DisplayPointMassesApplet():

- transition: The state of the abstract object ListPointMasses is modified as follows:
  ListPointMasses.init()
  ListPointMasses.add(0, PointMassT(20, 20, 10))
  ListPointMasses.add(1, PointMassT(120, 200, 20))
  ...

paint():

- transition *win* := Modify window so that the point masses in ListPointMasses are plotted as circles. The centre of each circles should be the corresponding x and y coordinates and the radius should be the mass of the point mass.

Specification of ADTs

- Similar template to abstract objects
- "Template Module" as opposed to "Module"
- "Exported Types" that are abstract use a ?
  - pointT = ?
  - pointMassT = ?
- Access routines know which abstract object called them
- Use "self" to refer to the current abstract object
- Use a dot "." to reference methods of an abstract object
  - *p*.xcoord()
  - *self*.*pt*.dist(*p*.point())
- Similar notation to Java
- The syntax of the interface in C is different

Syntax Line ADT Module

Template Module

lineADT

Uses

pointADT

Exported Types

lineT = ?

Syntax Line ADT Module Continued

Routine name	**In**	**Out**	**Exceptions**
new lineT	pointT, pointT	lineT	
start		pointT	
end		pointT	
length		real	
midpoint		pointT	
rotate	real		

Semantics Line ADT Module

State Variables

s: pointT
e: pointT

State Invariant

None

Assumptions

None

Access Routine Semantics Line ADT Module

new lineT (p_1, p_2):

- transition: $s, e := p_1, p_2$
- output: $out := self$
- exception: none

start:

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

end:

- output: $out := e$
- exception: none

Access Routine Semantics Continued

length:

- output: $out := s.\text{dist}(e)$
- exception: none

midpoint:

- output: $out :=$

$$\text{new pointT}(\text{avg}(s.\text{xcoord}, e.\text{xcoord}), \text{avg}(s.\text{ycoord}, e.\text{ycoord}))$$

- exception: none

rotate (φ):
φ is in radians

- transition: $s.\text{rotate}(\varphi), e.\text{rotate}(\varphi)$
- exception: none

Line ADT Local Functions

Local Functions

avg: real $\times$ real $\rightarrow$ real

avg$(x_1, x_2) \equiv \frac{x_1 + x_2}{2}$

Generic Modules

- What if we have a sequence of integers, instead of a sequence of point masses?
- What if we want a stack of integers, or characters, or pointT, or pointMassT?
- Do we need a new specification for each new abstract object?
- No, we can have a single abstract specification implementing a family of abstract objects that are distinguished only by a few variabilities
- Rather than duplicate nearly identical modules, we parameterize one generic module with respect to type(s)
- Advantages
  - Eliminate chance of inconsistencies between modules
  - Localize effects of possible modifications
  - Reuse

Generic Stack Module Syntax

Generic Module

Stack(T)

Exported Constants

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
...	...	...	...

Stack Module Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
s_init			
s_push	T		FULL
s_pop			EMPTY
s_top		T	EMPTY
s_depth		integer	

Semantics

State Variables

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

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

Assumptions

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq$ MAX_SIZE

Assumptions

s_init() is called before any other access routine

Access Routine Semantics

s_init():

- transition:
- exception:

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

- transition: $s := <>$
- exception:

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

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

s_push(x):

- transition:
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

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

s_push(x):

- transition: $s := s || < x >$
- exception:

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

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

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition:
- exception:

Access Routine Semantics

s_init():

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

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition: $s := s[0..|s| - 2]$
- exception:

Access Routine Semantics

s_init():

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

s_push(x):

- transition: $s := s || < x >$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FULL})$

s_pop():

- transition: $s := s[0..|s| - 2]$
- exception: $exc := (|s| = 0 \Rightarrow \text{EMPTY})$

Access Routine Semantics Continued

s_top():

- output:
- exception:

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

- output: $out := s[|s| - 1]$
- exception:

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

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

s_depth():

- output:
- exception:

Access Routine Semantics Continued

s_top():

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

s_depth():

- output: $out := |s|$
- exception:

Access Routine Semantics Continued

s_top():

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

s_depth():

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

Stack Module Properties

$\{true\}$

s_init()

$\{|s'| = 0\}$

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

s_push(x)

$\{|s'| = |s| + 1 \wedge s'[|s'| - 1] = x \wedge s'[0..|s| - 1] = s[0..|s| - 1]\}$

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

s_push(x)

s_pop()

$s' = s$

Object Oriented Design

- One kind of module, ADT, called class
- A class exports operations (procedures) to manipulate instance objects (often called methods)
- Instance objects accessible via references
- Can have multiple instances of the class (class can be thought of as roughly corresponding to the notion of a type)

Inheritance

- Another relation between modules (in addition to USES and IS_COMPONENT_OF)
- ADTs may be organized in a hierarchy
- Class B may specialize class A
  - B inherits from A
  - Conversely, A generalizes B
- A is a superclass of B
- B is a subclass of A

Template Module Employee

Routine name	**In**	**Out**	**Except**
Employee	string, string, moneyT	Employee	
first_Name		string	
last_Name		string	
where		siteT	
salary		moneyT	
fire			
assign	siteT		

Inheritance Examples

Template Module Administrative_Staff **inherits** Employee

Routine name	In	Out	Exception
do_this	folderT		

Template Module Technical_Staff **inherits** Employee

Routine name	In	Out	Exception
get_skill		skillT	
def_skill	skillT		

Inheritance Continued

- A way of building software incrementally
- Useful for long lived applications because new features can be added without breaking the old applications
- A subclass defines a subtype
- A subtype is substitutable for the parent type
- Polymorphism - a variable referring to type A can refer to an object of type B if B is a subclass of A
- Dynamic binding - the method invoked through a reference depends on the type of the object associated with the reference at runtime
- All instances of the sub-class are instances of the super-class, so the type of the sub-class is a subtype
- All instances of Administrative_Staff and Technical_Staff are instances of Employee

Dynamic Binding

- Many languages, like C, use static type checking
- OO languages use dynamic type checking as the default
- There is a difference between a type and a class once we know this
  - Types are known at compile time
  - The class of an object may be known only at run time

Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Point ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointT	real, real	PointT	
xcoord		real	
ycoord		real	
dist	PointT	real	

Semantics

State Variables

xc: real
yc: real

Point Mass ADT Module

Template Module

PointMassT **inherits** PointT

Uses

PointT

Syntax

Exported Types

PointMassT = ?

Point Mass ADT Module Continued

Exported Access Programs

Routine name	In	Out	Exceptions
new PointMassT	real, real, real	PointMassT	NegMassExcep
mval		real	
force	PointMassT	real	
fx	PointMassT	real	

Semantics

State Variables

ms: real

Point Mass ADT Module Semantics

new PointMassT(x, y, m):

- transition: $xc, yc, ms := x, y, m$
- output: $out := self$
- exception: $exc := (m < 0 \Rightarrow \text{NegativeMassException})$

force(p):

- output:

$$out := \text{UNIVERAL\_G} \frac{self.ms \times p.ms}{self.\text{dist}(p)^2}$$

- exception: none

Examples

- Example Point Line and Circle
- Example Robot Path
- Example Vector Space
- Example Othello Program
- Example Maze Formal Specification (Dr. v. Mohrenschildt)
- Mustafa ElSheikh Mesh Generator [1]
- Wen Yu Mesh Generator [5]
- Sven Barendt Filtered Backprojection
- Sanchez sDFT

References I

Jacques Carette, Mustafa ElSheikh, and W. Spencer Smith.
A generative geometric kernel.
In *ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 53–62, January 2011.

Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac.
Semi-formal design of reliable mesh generation systems.
Advances in Engineering Software, 35(12):827–841, 2004.

References II

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli.
Fundamentals of Software Engineering.
Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper.
Software Design, Automated Testing, and Maintenance: A Practical Approach.
International Thomson Computer Press, New York, NY, USA, 1995.

References III


W. Spencer Smith and Wen Yu.
A document driven methodology for improving the quality of a parallel mesh generation toolbox.
Advances in Engineering Software, 40(11):1155–1167, November 2009.