AN ECLECTIC COMPREHENSIVE APPROACH TO THE V&V OF SOFTWARE MODULES

Ali Mili, NJIT
NII, Tokyo, Japan
January 13, 2012
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
A DISCIPLINE OF MODULAR PROGRAMMING

What is Modular Programming? What is Modularity? When do we say that a Software Product is Modular?
A DISCIPLINE OF MODULAR PROGRAMMING

What is Modular Programming? What is Modularity? When do we say that a Software Product is Modular?

- Key to Modularity: *Information Hiding.*
  - Each module captures an important design decision that is hidden from other modules.
  - Separation between specification and implementation. Other modules need to know the specification, not the implementation.
A DISCIPLINE OF MODULAR PROGRAMMING

Why is Modularity Important?

- Productivity. Each module is developed from its specification; less need for communication.
- Maintainability. Because modules are designed independently, no change propagation.
- Testability. Each module can be tested against its own specification; no need for global insights.
- Reliability. Modularity leads to low coupling, hence lower error propagation.
- Reusability. Modularity is a tenet of bottom-up software design, which supports software reuse.
A DISCIPLINE OF MODULAR PROGRAMMING

How to support Modularity:
 Object Oriented Programming Languages support modularity by providing means to separate specification from implementation, but OO is neither necessary nor sufficient:
  • Possible to write modular software in non-OO languages.
  • Possible to write non-modular software in OO languages.
 In addition, OO languages represent specifications by signatures, which barely qualify.
 Modularity: A discipline, more than anything.
A DISCIPLINE OF MODULAR PROGRAMMING

Teaching Impacts

- Many students write mediocre quality software,
  - Know little about software specifications,
  - Know little about program correctness, correctness verification.

- Many SE/Programming textbooks, even when they acknowledge SE principles, violate them egregiously. Example Data Structure textbooks:
  - Make no clear distinction specification vs implementation,
  - Let minor implementation details affect important design decisions,
  - Little consideration for verification and validation.
PLAN

- A Discipline of Modular Programming
- Teaching Practice
  - Undergraduate: Data Structures
  - Graduate: Verification and Validation
- A Discipline of Specification
  - Specification Generation
  - Specification Validation
- Verification
- Testing
  - Test Data Selection
  - Oracle Selection and Implementation
- Reliability Estimation
- Conclusion
- References
TEACHING PRACTICE: A DATA STRUCTURES COURSE

Data Structures presented in three steps:

- **Specification.** What is this data type? Not how to implement it.
- **Usage.** What can we do with a data type such as this?
- **Implementation.** What are all the combinations of data structures/programs that provide the specified functionality.
TEACHING PRACTICE: A DATA STRUCTURES COURSE

Data Structures presented in three steps:

- **Specification.**
  - Describe behavior/observable properties.

- **Usage.**
  - Write applications that use the data type.
  - Data type is provided in executable form.
  - Students only know its specification.

- **Implementation.**
  - Modeling: From data types to data structures.
  - Considering candidate implementations.
  - Ensuring that applications written above behave the same way when one implementation is replaced by another.
Teaching Practice: A Verification and Validation Course

Graduate Level V&V Course: A One Term Project to Specify, Develop and Validate an ADT (300-600 LOC).

- A Specification Phase. Defining the ADT.
- An Implementation Phase. Implementing the ADT.
- A Verification Phase. Ensuring that the ADT is correct with respect to (part of the) specification.
- A Testing Phase. Testing the ADT against (the other part of) the specification.
- A Reliability Estimation Phase. Estimating the MTTF of the ADT at delivery.
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation

Verification

Testing
  • Test Data Selection
  • Oracle Selection and Implementation

• Reliability Estimation
• Conclusion
• References
A DISCIPLINE OF SPECIFICATION

Principles of Good Specification:

- Properties of the Product:
  - *Formality*. Well-defined semantics

- Properties of the Process:
  - *Completeness*. Capturing all the requirements.
  - *Minimality*. Capturing nothing but the requirements.
A Discipline of Specification

A Specification Lifecycle

<table>
<thead>
<tr>
<th>Team Step</th>
<th>Specification team (three members)</th>
<th>Validation team (one member)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification step (three weeks)</td>
<td>Generating the Axiomatic Specification From the ADT’s informal description</td>
<td>Generating Validation Data from the ADT’s informal description</td>
</tr>
<tr>
<td>Validation step (one week)</td>
<td>Updating the specification/ providing rationale/ Reviewing interpretation</td>
<td>Validating the Specification against generated validation data/ Negotiating with the specification team</td>
</tr>
</tbody>
</table>
A DISCIPLINE OF SPECIFICATION

A Specification Model/ Notation:
- Model Based Specifications,
- Behavioral Specifications.

Which do you like best?
- A Hint: most popular specification languages (Z, B, VDM, etc) are model based.
- We like....
A DISCIPLINE OF SPECIFICATION

A Specification Model/ Notation:
- Model Based Specifications,
- Behavioral Specifications.

Which do you like best?
- A Hint: most popular specification languages (Z, B, VDM, etc) are model based.
- We like... Behavioral specifications.
A DISCIPLINE OF SPECIFICATION

Specifying a Stack

- Model based languages:
  - List of items,
  - Operations on the list.

- What vs How:
  - List is a mathematical concept.
  - We do not specify how the list is implemented.
  - Latitude left to the designer/ implemener.

Several stacks do not involve any list.
A DISCIPLINE OF SPECIFICATION

Stack with identical elements:

- **Data:** int n;
- **Operations:**
  - `init`: n=0;
  - `size`: return n;
  - `empty`: return (n==0);
  - `push(a)`: n++;
  - `pop`: if n>0 {n--};
  - `top`: if n>0 {return a;} else {return error;}
A Discipline of Specification

Stack with two elements (e.g. { and }):

- **Data:** `int n;`
  - `n=14: 11110 \Rightarrow \{\};`  
  - `N=21: 110101 \Rightarrow {}{}{}`

- **Operations:**
  - `init`: `{n=1;}`
  - `size`: `{return int(log2(n));}`
  - `empty`: `{return (n==1);}`
  - `push(a)`: `{n=2*n; if (a=='\{') {n++;}}`
  - `pop`: `{if n>1 {n=n div 2;}}`
  - `top`: `{if n=1 {return error;} else if (n mod 2==0) {return ' \};} else {return '{';}}`
A DISCIPLINE OF SPECIFICATION

Stack with four elements (e.g. +, -, * and /):

- Data: int n;
  
n=448: 13010 ⇒ /+-+.

- Operations:
  
  - *init*: {n=1;}
  - size: {return int(log4(n));}
  - empty: {return (n==1);}
  - push(a): {n=4*n+cod(a);} // cod: (+,0),(-,1),(*,2),(/,3)
  - pop: {if n>1 {n=n div 4;}}
  - top: {if n=1 {return error;} else {decod(n mod 4)}}
    // decod: (0,+),(1,-),(2,*),(3,/)
A Discipline of Specification

Who says a stack is a list? By modeling a stack as a list, model-based specifications are guilty of three sins:

- They may be taking the initiative away from the designer (the designer may want to decide what model is best suited).
- Worse, they may be misleading the designer down the wrong path (it is not necessarily a list).
- Also, they specify individually each method of the ADT, further reducing the latitude of the designer.
A DISCIPLINE OF SPECIFICATION

How, then, to specify a stack?

- Inputs,
- Outputs,
- Externally observable input/ output behavior.

Stack:

- Input space:
  \[ \{\text{init, pop, top, size, empty}\} \cup \{\text{push}\} \times \text{itemtype}. \]
- Output space:
  \[ \text{itemtype} \cup \text{integer} \cup \text{boolean} \cup \{\text{error}\}. \]
A Discipline of Specification

Stack:

- **Input space, X:**
  \[ \{\text{init, pop, top, size, empty}\} \cup \{\text{push}\} \times \text{itemtype}. \]
  Distinction between:
  - **O-operations:** Change the state of the stack.
    - \( O-\text{ops} = \{\text{init, push, pop}\}. \)
  - **V-operations:** report on the state of the stack but do not alter it.
    - \( V-\text{ops} = \{\text{top, size, empty}\}. \)

- **Output space, Y:**
  \[ \text{itemtype} \cup \text{integer} \cup \text{boolean} \cup \{\text{error}\}. \]
A Discipline of Specification

Stack:
- Input space, X:
  \{\text{init}, \text{pop}, \text{top}, \text{size}, \text{empty}\} \cup \{\text{push}\} \times \text{itemtype}.
- Output space, Y:
  \text{itemtype} \cup \text{integer} \cup \text{boolean} \cup \{\text{error}\}.

Externally observable input/ output behavior. A function from X to Y?
A Discipline of Specification

Stack:
- Input space, $X$: 
  \[ \{\text{init, pop, top, size, empty}\} \cup \{\text{push}\} \times \text{itemtype}. \]
- Output space, $Y$: 
  \[ \text{itemtype} \cup \text{integer} \cup \text{boolean} \cup \{\text{error}\}. \]

Externally observable input/output behavior. A function from $X$ to $Y$?

A function from $X^*$ to $Y$!
- \[ \text{stack}(\text{init.push}(a).\text{top}) = a. \]
- \[ \text{stack}(\text{init.push}(a).\text{push}(b).\text{push}(c).\text{pop.size.top}) = b. \]
- \[ \text{stack}(\text{init.pop.size.top.pop.push}(a).\text{pop.size}) = 0. \]
- \[ \text{stack}(\text{pop.init.size.push}(a).\text{pop.push}(a).\text{empty}) = \text{false}. \]
A DISCIPLINE OF SPECIFICATION

We cannot go on like this... We find a closed form representation.

- **Axioms**: Behavior of the stack for trivial input sequences.
- **Rules**: Behavior of the stack for complex input sequences as a function of its behavior on simpler input sequences.
A Discipline of Specification

Axioms.

- **Top Axioms.**
  - stack(init.top)=error.
  - stack(init.h.push(a).top)=a.

- **Size Axiom.**
  - stack(init.size)=0.

- **Empty Axiom.**
  - stack(init.empty)=true.
  - stack(init.push(a).empty)=false.
A DISCIPLINE OF SPECIFICATION

Rules. Where h, h’, h+: X*. h+ ≠ ε.

- **Init Rule.**
  - stack(h’.init.h) = stack(init.h).

- **Init Pop Rule.**
  - stack(init.pop.h)=stack(init.h).

- **Push Pop Rule.**
  - stack(init.h.push(a).pop.h+)= stack(init.h.h+).

- **Size Rule.**
  - stack(init.h.push(a).size) = 1 + stack(init.h.size).

- **Empty Rules.**
  - stack(init.h.push(a).h’.empty) ⇒ stack(init.h.h’.empty).
  - stack(init.h.empty) ⇒ stack(init.h.pop.empty).
A DISCIPLINE OF SPECIFICATION

Rules. Where $h, h', h^+: X^*$. $h^+ \neq \varepsilon$.

- **V-Operation Rules.**
  - $\text{stack(init.h.top.h^+)} = \text{stack(init.h.h^+)}$.
  - $\text{stack(init.h.size.h^+)} = \text{stack(init.h.h^+)}$.
  - $\text{stack(init.h.empty.h^+)} = \text{stack(init.h.h^+)}$.

Not sure the specification is valid? That’s what specification validation is here for.
A Discipline of Specification

Specification Validation

System Requirements, Operational
Constraints, Customer Needs,
Applicable Regulations,
Stakeholder Demands

Validation

FORMAL SPECIFICATION

Verification
## A Discipline of Specification

**Validation by Redundancy**

<table>
<thead>
<tr>
<th>Team Step</th>
<th>Specification Team (three members)</th>
<th>Validation Team (one member)</th>
</tr>
</thead>
</table>
| Specification step (three weeks) | - Generating the Axiomatic Specification  
                              | - From the ADT’s informal description                                | - Generating Validation Data                                    |
|                               |                                                                          | - from the ADT’s informal description                             |
| Validation step (one week)    | - Updating the specification/ providing rationale/ Reviewing interpretation | - Validating the Specification against generated validation data  |
|                               |                                                                          | - Negotiating with the specification team                         |
A DISCIPLINE OF SPECIFICATION

V&V Team produced the following input/output clauses:


Is the specification valid with respect to this V&V data?
- Are these theorems of the axiomatic system?
A Discipline of Specification

stack(pop.init.top.pop.push(a).size.push(b).top.pop.push(d).top.pop.size.top)
= \{\text{init rule: } \text{stack}(h'.init.h) = \text{stack}(\text{init}.h)\}

stack(init.top.pop.push(a).size.push(b).top.pop.push(d).top.pop.size.top)
= \{\text{V-op rule: } \text{stack}(\text{init}.h.vop.h+) = \text{stack}(\text{init}.h.h+)\}

stack(init.pop.push(a).push(b).pop.push(d).pop.top)
= \{\text{rule: } \text{stack}(\text{init}.h.push(a).pop.h+) = \text{stack}(\text{init}.h.h+)\}

stack(init.pop.push(a).top)
= \{\text{second top axiom: } \text{stack}(\text{init}.h.push(a).top) = a\}

a.
A DISCIPLINE OF SPECIFICATION

\[
\text{stack(push(c).init.pop.init.pop.push(d).top.size.push(d).push(c).top.pop.push(d).size)} = \{\text{init rule: stack(h',init.h)=stack(init.h)}\}
\]

\[
\text{stack(init.pop.push(d).top.size.push(d).push(c).top.pop.push(d).size)} = \{\text{v-op rule: stack(init.vop.h+)=stack(init.h+)}\}
\]

\[
\text{stack(init.pop.push(d).push(d).push(c).pop.push(d).size)} = \{\text{rule: stack(init.h.push(a).pop.h+)=stack(init.h.h+)}\}
\]

\[
\text{stack(init.pop.push(d).push(d).push(d).size)}
\]
A Discipline of Specification

\[
\text{stack}(\text{init}.\quad \text{pop}.\quad \text{push}(d).\quad \text{push}(d).\quad \text{push}(d).\quad \text{size})
\]
\[
= \{\text{init pop rule: stack}(\text{init}.\text{pop}.\text{h})=\text{stack}(\text{init}.\text{h})\}
\]
\[
\text{stack}(\text{init}.\text{push}(d).\text{push}(d).\text{push}(d).\text{push}(d).\text{size})
\]
\[
= \{\text{size rule: stack}(\text{init}.\text{h}.\text{push}(a).\text{size})=1+\text{stack}(\text{init}.\text{h}.\text{size})\}
\]
\[
1+\text{stack}(\text{init}.\text{push}(d).\text{push}(d).\text{push}(d).\text{size})
\]
\[
= \{\text{size rule: stack}(\text{init}.\text{h}.\text{push}(a).\text{size})=1+\text{stack}(\text{init}.\text{h}.\text{size})\}
\]
\[
1+1+\text{stack}(\text{init}.\text{push}(d).\text{size})
\]
\[
= \{\text{size rule: stack}(\text{init}.\text{h}.\text{push}(a).\text{size})=1+\text{stack}(\text{init}.\text{h}.\text{size})\}
\]
\[
1+1+1+\text{stack}(\text{init}.\text{size})
\]
\[
= \{\text{size axiom}\}
\]
\[
1+1+1+0 = 3.
\]
A Discipline of Specification

stack(pop.init.pop.pop.push(a).size.empty.push(a).pop.top.push(b).push(b).empty)
= \{init rule: stack(h'.init.h) = stack(init.h)\}

stack(init.pop.pop.push(a).size.empty.push(a).pop.top.push(b).push(b).empty)
= \{init pop rule: stack(init.pop.h) = stack(init.h)\}

stack(init.push(a).size.empty.push(a).pop.top.push(b).push(b).empty)
= \{vop rule: stack(init.h.vop.h+) = stack(init.h.h+)\}

stack(init.push(a).push(a).pop.push(b).push(b).empty)
A Discipline of Specification

stack(init.push(a).push(a).pop.push(b).push(b).empty)
= \{\text{rule: stack(init.push(a).pop.h+)=} stack(init.h.h+)}
stack(init.push(a).push(b).push(b).empty)
⇒ \{\text{rule: stack(init.h.push(a).h’.empty)⇒ stack(init.h.h’.empty)}\}
stack(init.push(a).push(b).empty)
⇒ \{\text{rule: stack(init.h.push(a).h’.empty)⇒ stack(init.h.h’.empty)}\}
stack(init.push(a). empty)
⇒ \{\text{empty axiom: stack(init.push(a).empty)=false}\}
false.

Anything that implies false equals false. QED.
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
**Verification**

In ASE 1998: To verify a program with respect to a complex specification, one can apply:
- Static analysis wrt some parts of the specification.
- Dynamic testing with respect some parts.
- Fault tolerance with respect to other parts.

Rationale:
- Law of Diminishing Returns.
- Also,
  - Static analysis is very easy with respect to some type of specifications,
  - Testing is very reliable with respect to some type of specifications,
  - Fault tolerance is very efficient with respect to some type of specifications.
- Complementary characteristics.
**Verification**

In the case of ADT specifications: Very easy to verify implementation against axioms.  

- stack(init.top)=error.  
  - \{true\} s.init(); y=s.top() \{y=error\}

- stack(init.h.push(a).top)=a.  
  - a, y: itemtype;  
    - \{true\} s.init(); ..h..; s.push(a); y=s.top() \{y=a\}

- stack(init.size)=0.  
  - y: int;  
    - \{true\} s.init(); y=s.size() \{y=0\}

- stack(init.empty)=true.  
  - y: bool;  
    - \{true\} s.init(); y=s.empty() \{y=true\}

- stack(init.push(a).empty)=false.  
  - y: bool;  
    - \{true\} s.init(); s.push(a); y=s.empty() \{y=false\}
**Verification**

Observations:
- Has none of the difficulties associated with static analysis (no loop invariants, simple code, straightforward initial conditions).
- Detects faults pertaining to wrong initializations,
- Ensures that base cases are handled properly.

Verify code against the rules? No way.
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
TESTING: TEST DATA SELECTION

Four issues to address:

- How to generate test data.
- How to generate test oracle.
- How to analyze test results.
- What is the test standard (when do we know we are done testing?).
Testing: Test data selection

Four issues to address:

- How to generate test data.
  - Test datum: input sequence.
  - Random test generation.
  - Reflects a pre-specified usage pattern.
TESTING: TEST DATA SELECTION

Usage Pattern: Probability Distribution over Input Space.
Testing: Test data Selection

Testing according to usage pattern
Arbitrary testing
TESTING: TEST DATA SELECTION

Test Driver:

```c
{int nbf=0;  // number of unsuccessful tests
 for (int i=0; i<testsize; i++)
{
    switch (i%9)
    {
      case 0: initrule();  case 1: initpoprule();
      case 2: pushpoprule(); case 3: sizerule();
      case 4: emptyrulea(); case 5: emptyruleb();
      case 6: voprulenetop(); case 7: voprulesize();
      case 8: vopruleempty();
    }
    cout << "failure rate: " << nbf << " out of " << testsize << endl;
}
```
Testing: Test data Selection

Function PushPopRule();
{
    // stack(init.h.push(a).pop.h+)=stack(init.h.h+)
    randomGenerate(h,hplus,a);
    s.init(); h; s.push(a); s.pop(); hplus; // left-hand side of the rule
    itemtype storetop=s.top(); int storesize=s.size(); bool storeempty=s.empty();
    // we save the state of the stack (without knowing how it is represented)
    s.init(); h; hplus; // right hand side of rule. Are we in the same state?
    return (storetop==s.top()) && (storesize==s.size()) && (storeempty==s.empty())
}
Plan

- A Discipline of Modular Programming
- Teaching Practice
  - Undergraduate: Data Structures
  - Graduate: Verification and Validation
- A Discipline of Specification
  - Specification Generation
  - Specification Validation
- Verification
- Testing
  - Test Data Selection
  - Oracle Selection and Implementation
- Reliability Estimation
- Conclusion
- References
RELIABILITY ESTIMATION

Cleanroom Reliability Testing:

- No Unit Testing,
  - Static Analysis as a substitute for unit testing.
- All executions of the program are carried out under public scrutiny, and recorded.
- Precise record of all faults removed from the first execution on.
Reliability Estimation

Reliability Model:

\[ MTTF_N = MTTF_0 \times R^N, \]

where

- \( MTTF_N \): Estimated reliability after \( N \) faults are removed,
- \( MTTF_0 \): Estimated reliability at the start of testing,
- \( R \): Reliability growth factor (by how much MTTF grows every time we remove a fault),
## Reliability Estimation

<table>
<thead>
<tr>
<th>Run #</th>
<th>Number of Failure Free Executions</th>
<th>Fault Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>879</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2104</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>30000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>150000</td>
<td></td>
</tr>
</tbody>
</table>
## Reliability Estimation

<table>
<thead>
<tr>
<th>N</th>
<th>$MTTF_N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
</tr>
<tr>
<td>4</td>
<td>879</td>
</tr>
<tr>
<td>5</td>
<td>2104</td>
</tr>
<tr>
<td>6</td>
<td>30000</td>
</tr>
<tr>
<td>7</td>
<td>150000</td>
</tr>
</tbody>
</table>

Use Regression to determine:
- $MTTF_0$,
- $R$,

From which we compute $MTTF_8$ and return that as the estimated reliability of the software at delivery.
RELIABILITY ESTIMATION

How do we know we are done testing?
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
CONCLUSION

Disciplined Approach, Using sound SE Principles, encompasses many lifecycle phases:

- Requirements Specification (Behavioral),
- Specification Validation,
- Static Analysis against simple specification clauses (involving simple code),
- Dynamic Testing,
  - Random Test Data Generation, per usage pattern,
  - Using specification rules as test oracle,
- Reliability Estimation
  - Cleanroom Discipline, no Unit Testing.
  - Concludes when reliability estimate exceeds required reliability, or with current estimate of reliability.
CONCLUSION

Some Lessons Learned,

- It is possible to achieve good Return on Investment in software V&V by dispatching different methods to different specifications.
  - VSI project funded by Qatar.
- Give Students a solid engineering background for reasoning about software
  - First day: we tried it and it works.
  - Last day: MTTF with respect to this formal axiomatic specification is 173 million executions.
- Non Mainstream Languages?
  - Academia: preparing students for lifelong learning,
  - Best language for SW Engineers: Japanese.
  - Seeking Qatar funding for SE languages for education.
PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
REFERENCES


PLAN

• A Discipline of Modular Programming
• Teaching Practice
  • Undergraduate: Data Structures
  • Graduate: Verification and Validation
• A Discipline of Specification
  • Specification Generation
  • Specification Validation
• Verification
• Testing
  • Test Data Selection
  • Oracle Selection and Implementation
• Reliability Estimation
• Conclusion
• References
Thank You

• Professor Zhen Jiang Hu,
• Professor Hu’s research team,
• National Informatics Institute,
• Japan,

For being such gracious hosts.