Combining a Formal Method and PSP for Improving Software Process: An Initial Report
- (Work-In-Progress Report) -

Shigeru KUSAKABE, Yoichi OMORI, Keijiro ARAKI

Kyushu University, Japan

Outline

- Background
- Personal Software Process
- Formal Method
- Process Improvement with Formal Methods
  - Case Report
- Concluding Remarks

Background

Using Formal methods is a promising approach to

- high quality products within shorter period
- reliable and dependable systems
    - recommended in standards: ISO/IEC15408, IEC61508

Many engineers and managers are interested,

... but actually introduced in very limited cases (in Japan)

- formal methods seem too research oriented (esoteric)?
    - hard to apply to real projects by usual developer?
- many developers seem less aware of process data
    - estimate cost-performance without baseline performance

When, where, how to introduce FM in a feasible manner?

Seven Myths of Formal Methods

Anthony Hall, IEEE Software, 1990

Seven Myths

1. Formal methods can guarantee that software is perfect.
2. Formal methods are all about program proving.
3. Formal methods are only useful for safety-critical systems.
4. Formal methods require highly trained mathematicians.
5. Formal methods increase the cost of development.
6. Formal methods are unacceptable to users.
7. Formal methods are not used on real, large scale software.

Seven Facts of Formal Methods

1. Formal methods are fallible.
2. Formal methods are all about specifications.
3. Formal specifications helps with any system.
4. The mathematics for specification is easy.
5. Writing a formal specification decreases the cost of development.
6. Formal specifications help users understand what they are getting.
7. Formal methods are used daily on industrial projects.

Goal & Approach

Self-managed individuals ready for effective and efficient software development with formal methods if necessary.

- Assumtion: we can effectively introduce formal methods into a disciplined and analyzable software process.
  - After establishing discipline, improve software process with formal methods from an engineering point of view.
- Our first trial: process improvement with developer friendly FM for well-defined & customizable process.
  - well-defined & w/ data collection support -> easy to analyze
  - customizable -> easy to extend with formal methods

– Case study, rather than strictly controlled experiment

Outline

- Background
- Personal Software Process
- Formal Method
- Process Improvement with Formal Methods
  - Case Report
- Concluding Remarks

PSP: Personal Software Process*

Providing a framework that helps us to analyze where to improve our personal process:

- **Phases**: plan, detailed design, detailed design review, code, code review, compile, unit test, and post mortem, with a set of associated **scripts, forms, and templates**.
- **Data**: time and defects injected and removed for each phase, size, size and time estimating error, cost-performance index, defects injected and removed per hour, personal yield, appraisal and failure cost of quality, and the appraisal to failure ratio.

* Service Mark of Carnegie Mellon University, Software Engineering Institute

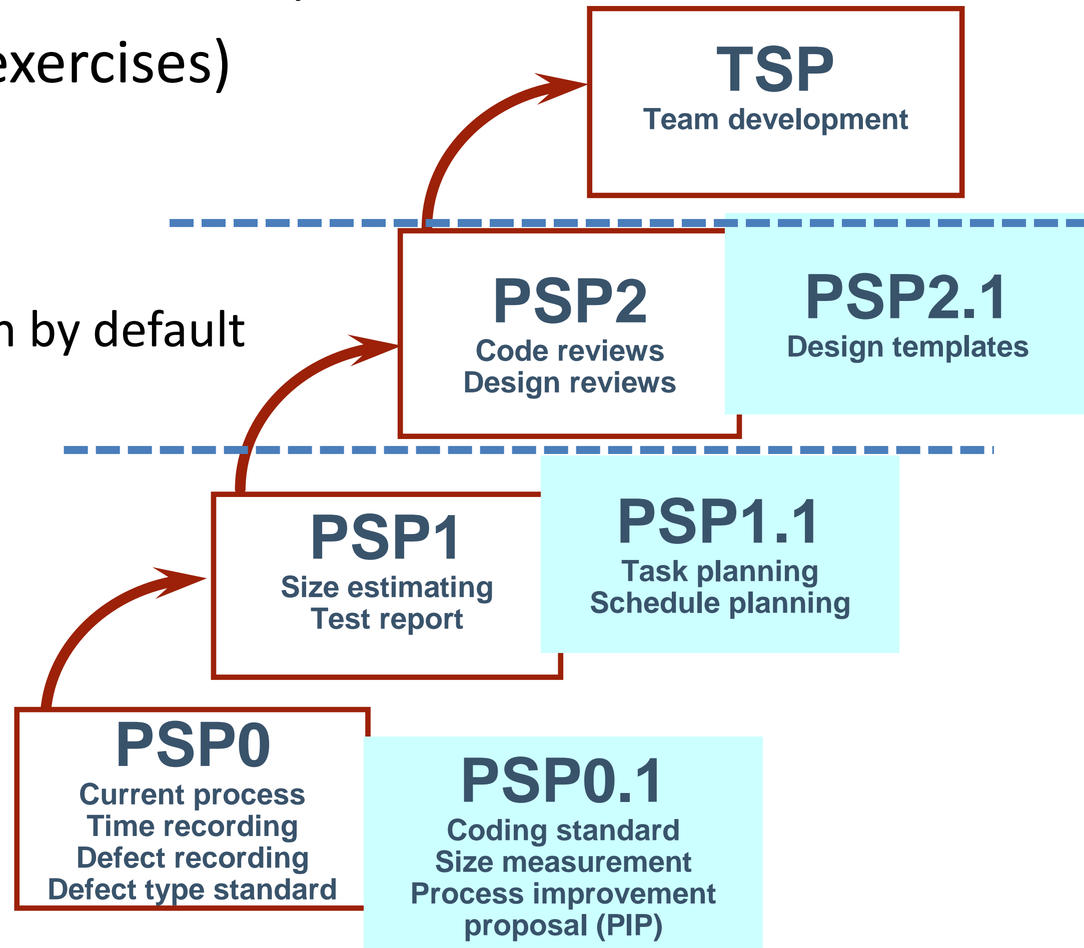
Introduce Formal Methods in PSP

PSP course structure (8-program version)

- PSP0*: measurement (2 exercises)
- PSP1*: estimate (2)
- PSP2*: quality (4)
  - very simple formal notation by default

Process extension (variation)

1. Collect process data to PSP X as baseline data
   - Time, defect (type, fix time, ..)
2. Analyze baseline data and consider how to improve
3. Start using FM after PSP X



Outline

- Background
- Personal Software Process
- Formal Method
- Process Improvement with Formal Methods
  - Case Report
- Concluding Remarks

Formal Methods

Useful in reducing defects injected into software

- Mathematically describing the system enables efficient and effective (automatic) reasoning
- Elimination of ambiguity leads to improving quality of software development as well as software itself

Various methods

– more than 100 methods, ...

Different levels of formality

– for example, level 0, level 1, and level 2

Introducing Formal Methods to Project

(In Japan,) Projects using formal methods are very limited[*0], while engineers & managers seem interested in formal methods.

- In order to break this situation, several organizations / groups, such as SEC[*1] of IPA[*2], are trying to establish guidelines to introduce formal methods in real projects.

*0 One of the most famous succeeded projects is Felica IC chip firmware

*1 SEC: Software Engineering Center, *2 IPA: Information-technology Promotion Agency

Our challenge: Establish reference models of software development process with formal methods, starting from a developer-friendly one. (level 0)

Different levels of formality

- **Level 0**: In this light-weight level, we **develop a formal specification and then a program from the specification informally**. This may be the most cost-effective approach in many cases.
- **Level 1**: We may adopt formal development and formal verification in a more formal manner to produce software. For example, **proofs of properties or refinement from the specification to an implementation** may be conducted. This may be most appropriate in high-integrity systems involving safety or security.
- **Level 2**: Theorem provers may be used to perform **fully formal machine-checked proofs**. This kind of activity may be very expensive and is only practically worthwhile if the cost of defects is extremely expensive.

Hoare logic

{N ≧ 0}

```
i := 1;
f := 1;
While i ≦ N
Do
  f := f * i;
  i := i + 1
End
```

{f = N!}

Proof

$1 \le i \le N+1 \wedge f=(i-1)! \wedge i \le N$
$\Rightarrow 1 \le i+1 \le N+1 \wedge f*i=i!$

$\{ 1 \le i+1 \le N+1 \wedge f*i=i! \}$
$f := f * i$
$\{ 1 \le i+1 \le N+1 \wedge f=i! \}$

$N \ge 0$
$\Rightarrow 1 \le 1 \le N+1$
$\wedge 1=(1-1)!$

$\{ 1 \le 1 \le N+1$
$\wedge 1=(1-1)! \}$
$i := 1$
$\{ 1 \le i \le N+1$
$\wedge 1=(i-1)! \}$

$\{ 1 \le i+1 \le N+1 \wedge f=(i-1)! \wedge i \le N \}$
$f := f * i$
$\{ 1 \le i+1 \le N+1 \wedge f=i! \}$

$\{ 1 \le i+1 \le N+1 \wedge f=i! \}$
$i := i + 1$
$\{ 1 \le i \le N+1 \wedge f=(i-1)! \}$

$\{N \ge 0\}$
$i := 1$
$\{ 1 \le i \le N+1$
$\wedge 1=(i-1)! \}$

$\{ 1 \le i \le N+1$
$\wedge 1=(i-1)! \}$
$f := 1$
$\{ 1 \le i \le N+1$
$\wedge f=(i-1)! \}$

$\{ 1 \le i \le N+1 \wedge f=(i-1)! \wedge i \le N \}$
$f := f * i; i := i + 1$
$\{ 1 \le i \le N+1 \wedge f=(i-1)! \}$

$\{ 1 \le i \le N+1 \wedge f=(i-1)! \}$
While $i \le N$ Do $f := f * i; i := i + 1$ End
$\{ 1 \le i \le N+1 \wedge f=(i-1)! \wedge i > N \}$

$1 \le i \le N+1 \wedge f=(i-1)!$
$\wedge i > N$
$\Rightarrow f = N!$

$\{N \ge 0\}$
$i := 1; f := 1$
$\{ 1 \le i \le N+1 \wedge f=(i-1)! \}$

$\{ 1 \le i \le N+1 \wedge f=(i-1)! \}$
While $i \le N$ Do $f := f * i; i := i + 1$ End
$\{f = N! \}$

$\{N \ge 0\}$ $i := 1; f := 1;$ While $i \le N$ Do $f := f * i; i := i + 1$ End $\{f = N! \}$

Different levels of formality

- **Level 0**: In this light-weight level, we **develop a formal specification and then a program from the specification informally**. This may be the **most cost-effective** approach in many cases.
- **Level 1**: We may adopt formal development and formal verification in a more formal manner to produce software. For example, **proofs of properties or refinement from the specification to an implementation** may be conducted. This may be most appropriate in high-integrity systems involving safety or security.
- **Level 2**: Theorem provers may be used to perform **fully formal machine-checked proofs**. This kind of activity may be very expensive and is only practically worthwhile if the cost of defects is extremely expensive.

VDM

VDM (Vienna Development Method) (1970s, IBM Vienna):

- a collection of techniques for developing computer systems from formally expressed models (specifications)
- VDM-SL (ISO/IEC 13817-1)
  - Well-defined (c.f. UML), executable (c.f. Z)
  - (DVM++ is its object-oriented extension version.)
- support for different abstraction:
  - Implicit/Explicit, functional/state-based
- tool (interpreter for executable specification)
- support for Japanese: useful for other stakeholders

=> Developer friendly !?

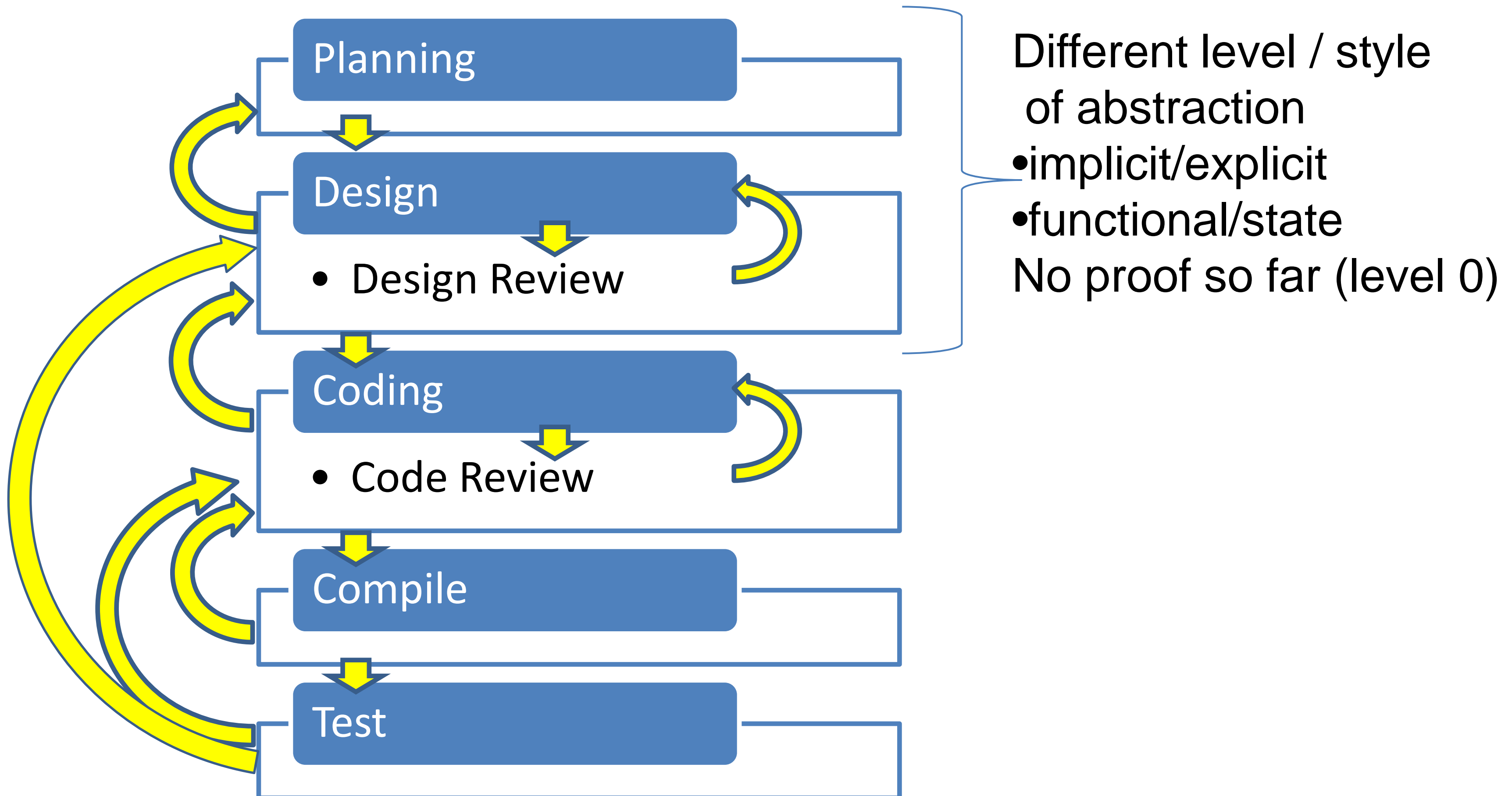
Example

```
-- explicit specification example
functions
 fact : nat -> nat
 fact(n) ==
  cases n :
   0 -> 1,
   others -> n * fact(n-1)
  end
```

```
-- implicit specification example
functions
 IAddAddress(name: Name, address: Address, book: AddressBook)  r: AddressBook
 IAddAddress(name, address, book) == is not yet specified
 pre name not in set dom(book)
 post r = book munion {name |-> address}
```

Why VDM?

Reduction of ambiguity in a phase may reduce the defects in the following phases, and may help finding the defects in the preceding phases.



Outline

- Background
- Personal Software Process
- Formal Method
- Process Improvement with Formal Methods
  - Case Report
- Concluding Remarks

Process Improvement Case Report

Concern

- Can we (non-expert) figure out how to use a formal method, VDM, in a guided manner?

Setting

- A graduate student : not familiar with formal methods
  - Basic experience of programming and software development project such as PBL (Project-Based Learning)
- Defect prevention based on personal historical data

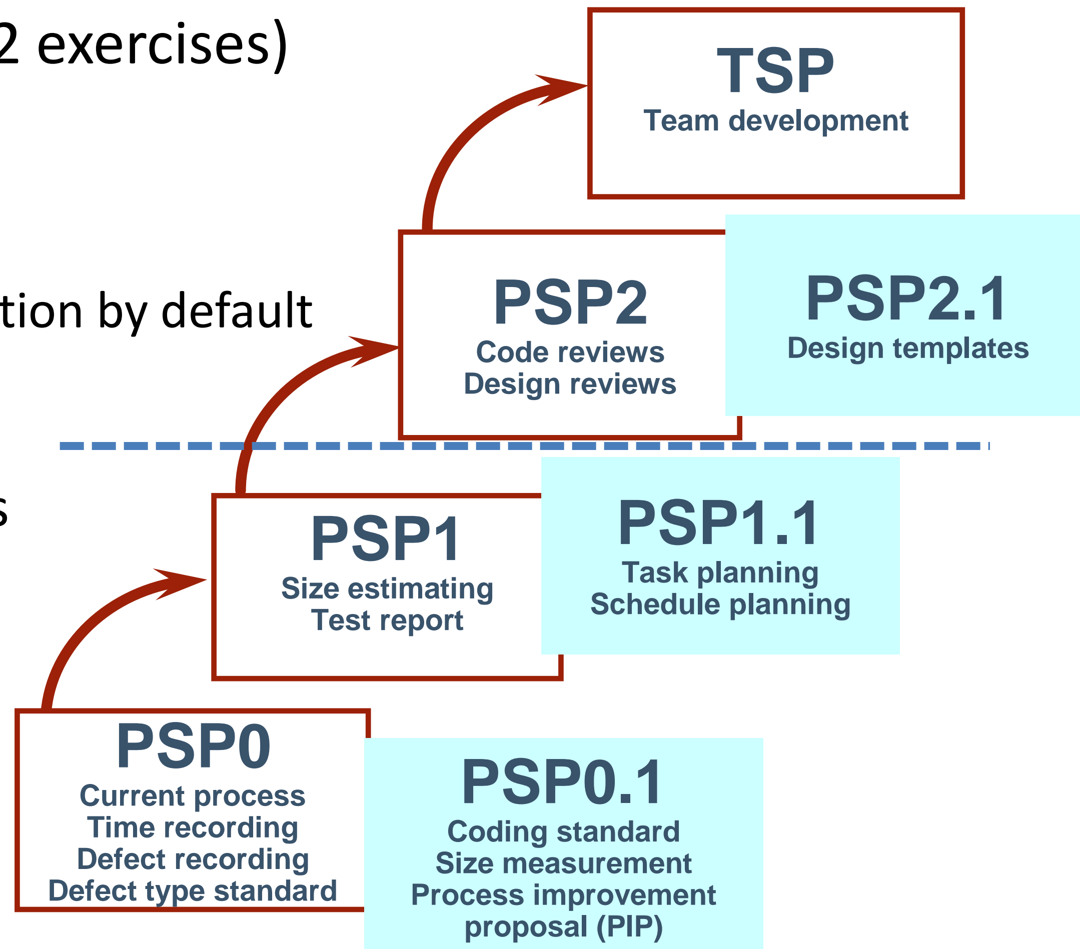
Introduce VDM in PSP

PSP course structure

- PSP0*: measurement (2 exercises)
- PSP1*: estimate (2)
- PSP2*: quality (2/4)
  - very simple formal notation by default

Process extension

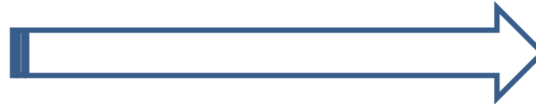
1. Collect baseline data: process data until PSP1*
   - Time, defect (type, fix time, ..)
2. Analyze baseline data and consider how to improve
3. Start using FM from PSP2



Defect Data

Personal process improvement with formal methods based on the defect data:

- defect type
- time to fix defect
- defect injection phase
- defect removal phase
- brief explanation

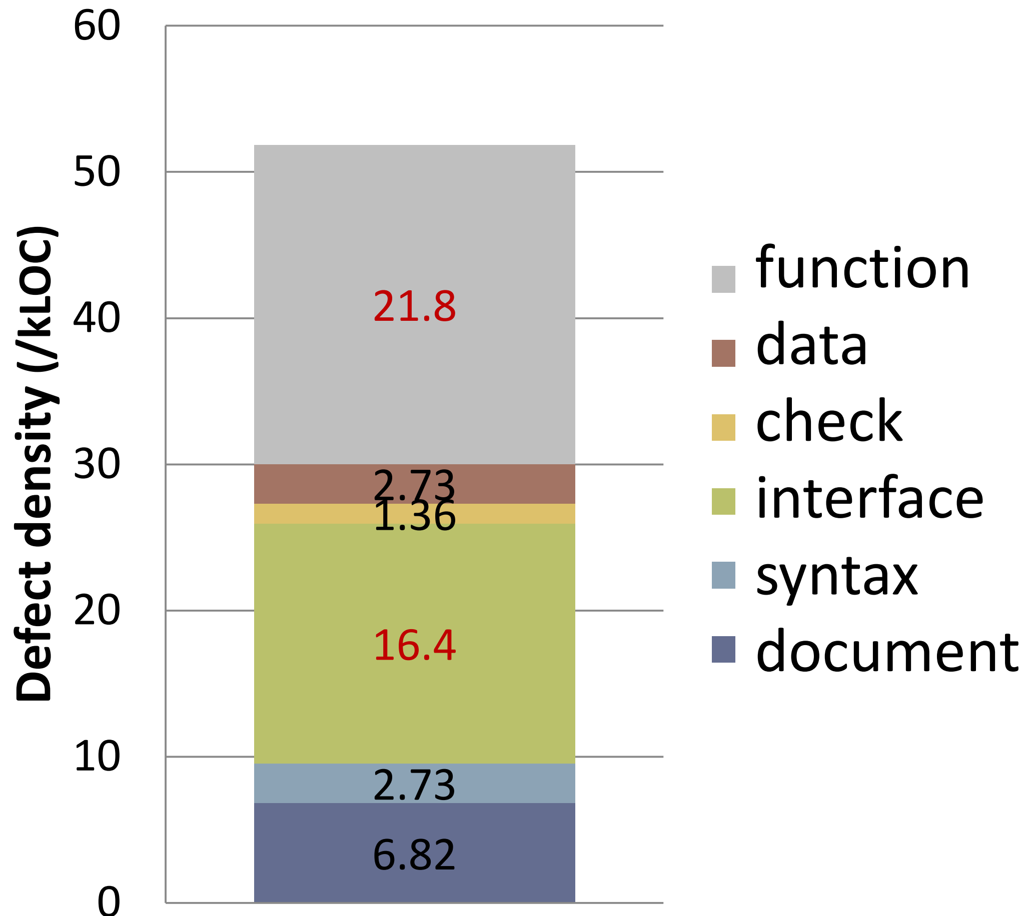


Defect type:

- Documentation
- Syntax
- Build, Package
- Assignment
- Interface
- Checking
- Data
- Function
- System
- Environment

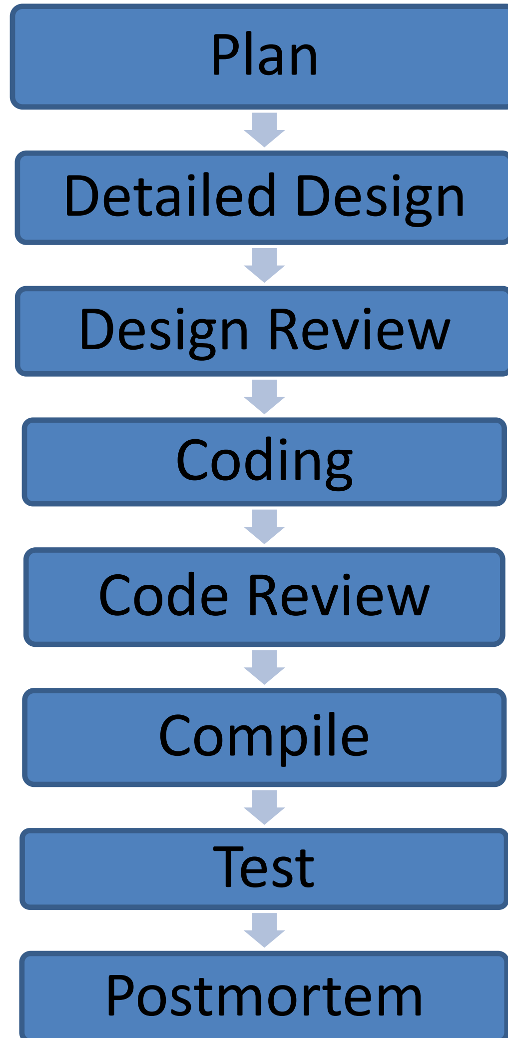
Baseline Process Data (defects)

(PSP 0, PSP 0.1, PSP 1, PSP 1.1)



	target type		ave. fix time (mi)
I-1	Interface	insufficient breakdown	15.8
F-1	Function	loop control	10.3
F-2	Function	logic	6.8

Process Modification



Base process: PSP 2

- design
  - UML

  - VDM++

- design review
  - with customized check list (manual check), and tool support

Added Steps

[Step 1:] Prevention of I-1 type defects

- Write signature of methods in VDM++ in detailed design
- Use VDMTools for syntax and type check

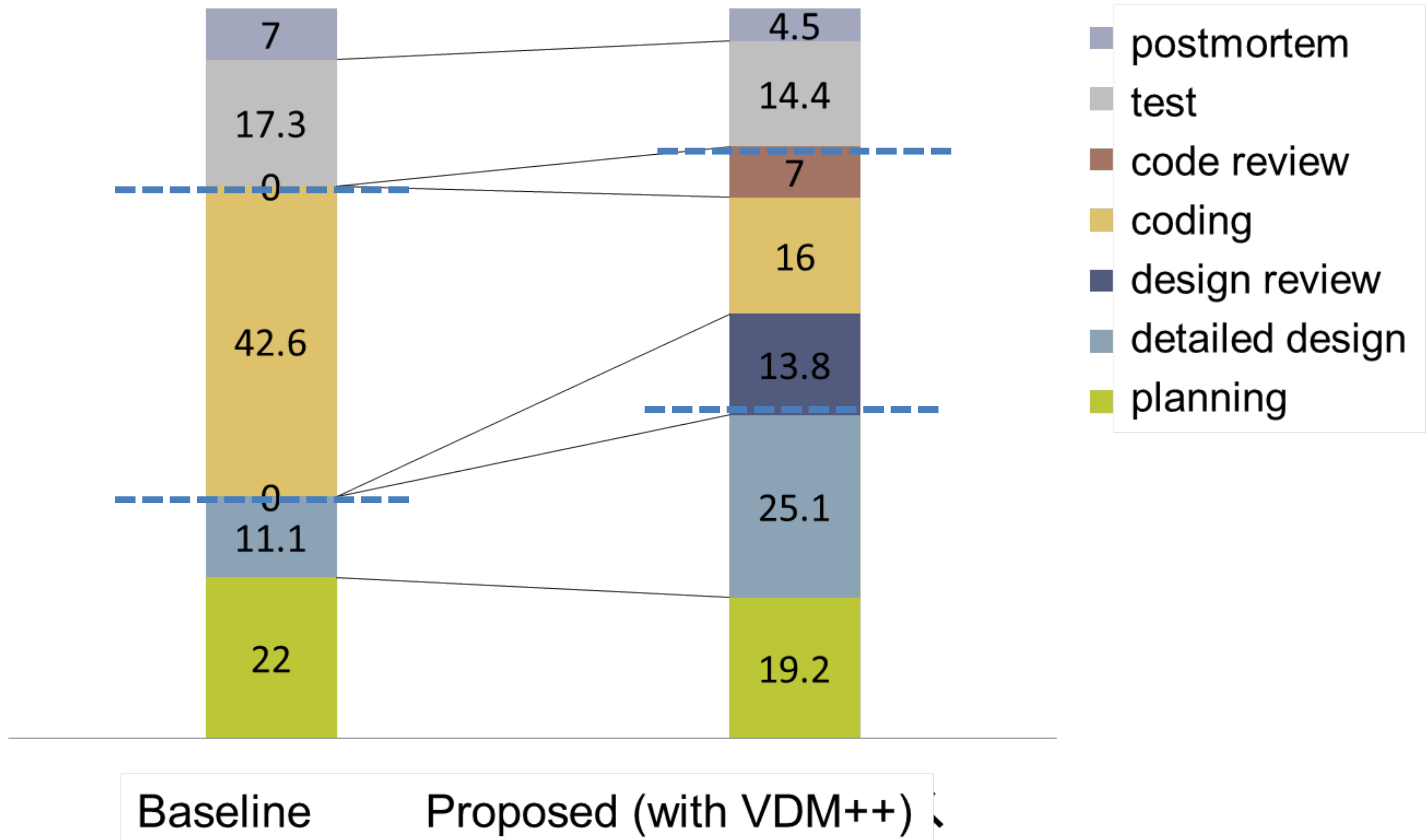
[Step 2:] Prevention of F-1 type defects

- Describe sequence handling part in VDM++

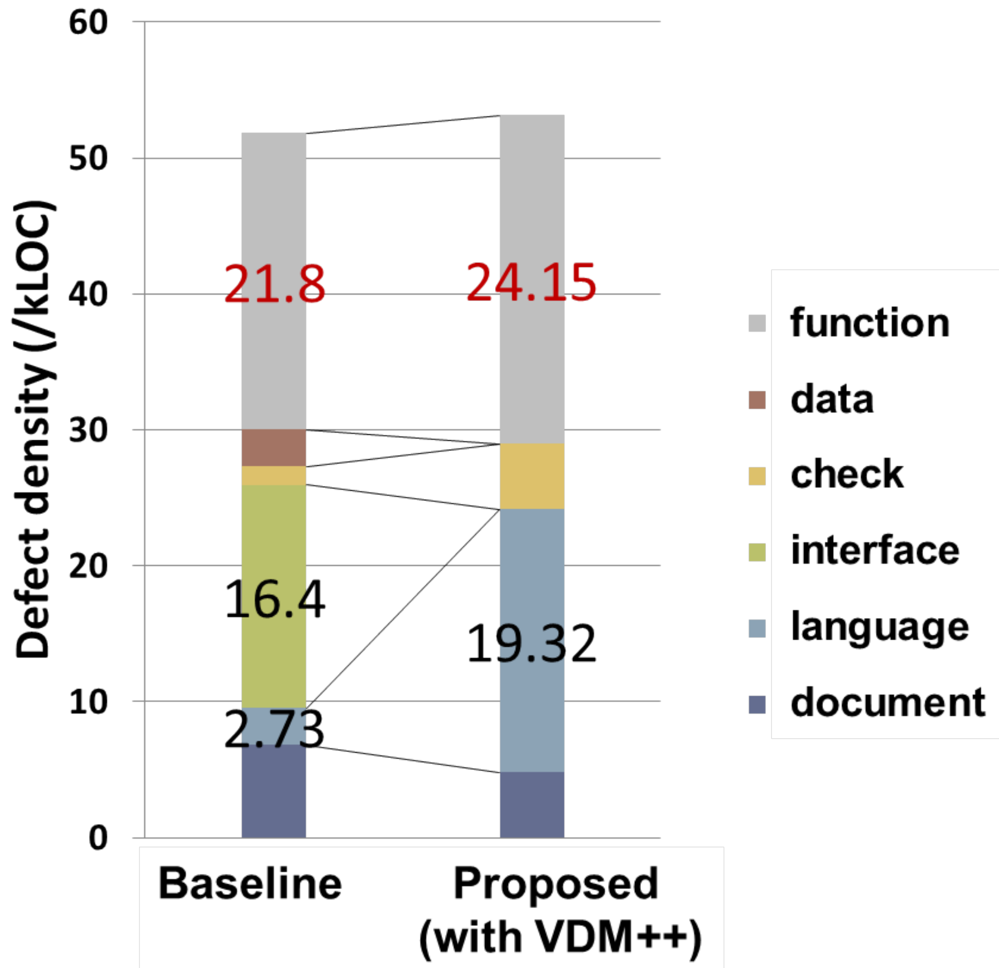
[Step 3:] Prevention of F-2 type defects

- Write explicit VDM++ specification for selected part
- Use animation of VDMTools.

Time Distribution



Defect Density



Interface type

- none

Function type

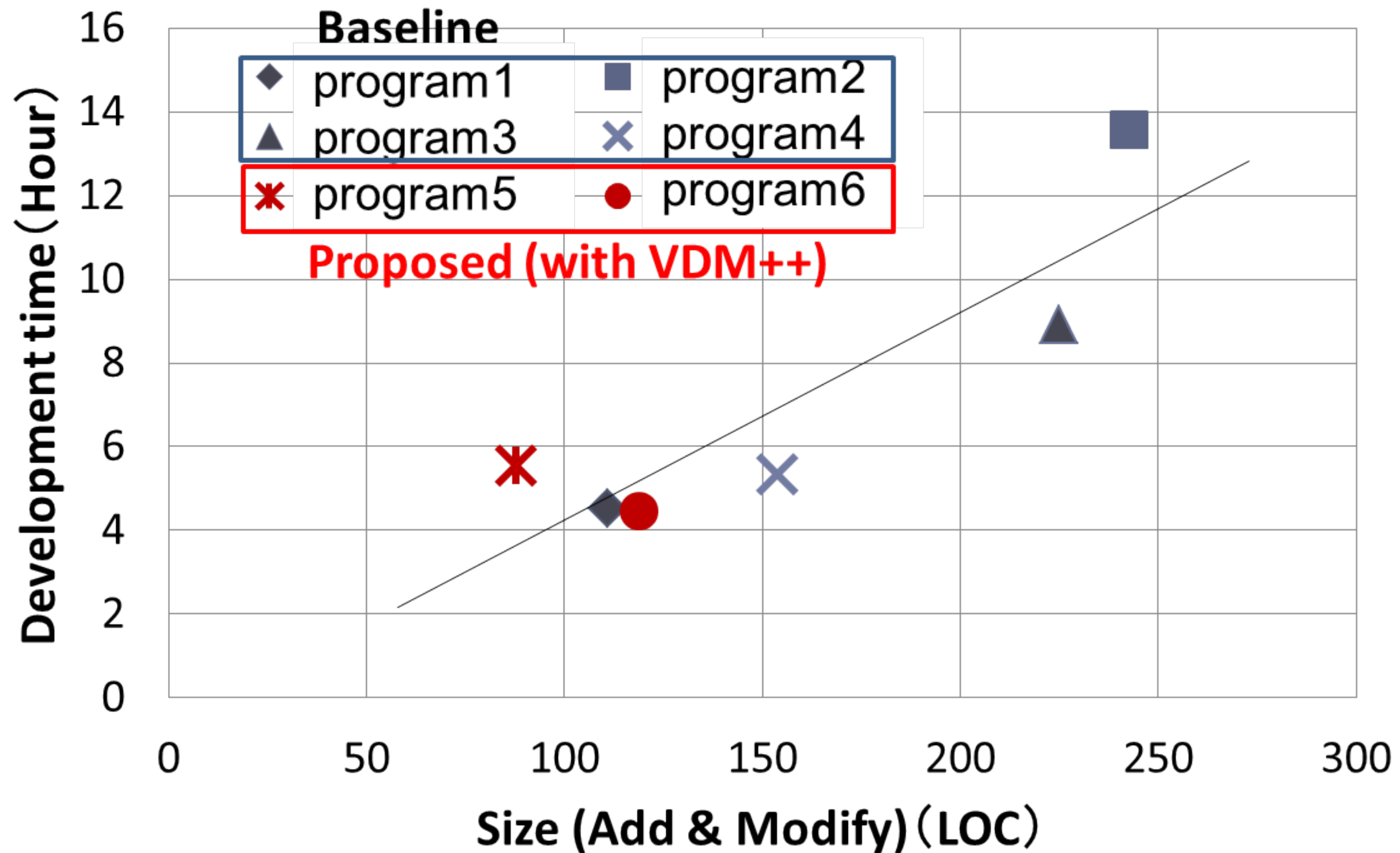
(slightly increased, but removed in..)

- baseline
  - mainly (87.5%) removed in Test
- proposed
  - mainly remove in design review
  - only 20% in Test

Total

- no reduction ...
  - language proficiency?

Productivity



Outline

- Background
- Personal Software Process
- Formal Method
- Process Improvement with Formal Methods
  - Case Report
- Concluding Remarks

Concluding Remarks

- Introduced VDM in a guided manner
  - He could use our baseline data in improving the process
- Effective process improvement
  - spent more in design and less in test
  - reduced the number of defects he had focused on without decreasing his productivity

  He felt that, without a disciplined process like PSP, he could not have made a process improvement plan with formal methods.
- Future work:
  - Other processes such as TSP, other formal methods, ...