Human-Machine Pair Programming for Future Software Engineering
人間とマシンのペアプログラミング

Shaoying Liu (劉少英)
広島大学・大学院理工系科学研究科
Email: sliu@hiroshima-u.ac.jp
HP: https://home.hiroshima-u.ac.jp/sliu/

Overview

1. Why Human-Machine Pair Programming (HMPP)（何故人間とマシンのペアプログラミング）?
2. Theoretical Foundation and Framework for HMPP（HMPPの理論基礎とフレームワーク）
3. Knowledge Classification（知識分類）
4. HMPP for Agile-SOFL（Agile-SOFLにおけるHMPP）
5. Challenges（課題）
6. Conclusions（まとめ）
7. Future Work（将来研究）

1. Why Human-Machine Pair Programming
（何故人間とマシンのペアプログラミング）?

(1) What is pair programming (PP)（ペアプログラミングとは）?

Definition: **Pair programming** is one of the techniques in the Extreme Programming **agile software development** method in which two programmers work together at one workstation.

Agile Development
アジャイル開発

Agile software development is an **evolutionary development** technique that emphasizes the following values:

- ***Individuals and Interactions*** *over processes and tools*
- ***Working Software*** *over comprehensive documentation*
- ***Customer Collaboration*** *over contract negotiation*
- ***Responding to Change*** *over following a plan*

Why Not Other Methods
（何故他の開発手法を使わないの）?

Model-Driven Development (MDD)

Model (e.g., UML) → Code → Testing

Formal Methods (FM)

Specification → Code → Verification

時間消耗,
文書管理,
保証なし,
作成困難,
顧客との協力
制限

Component-Based Development (CBD)

Components → Code → Testing

適切なコンポ
ーネントの検
索、選択、検証
などが困難.
一定の効果が
ある。

Why Are Agile Methods Popular?

- Consistent with the **nature of evolution** (changes) in software development (design-oriented)
- Timely and **comprehensible communication and interactions** between the developer and the customer
- One level documentation (code) for time saving

Is Agile Methods Perfect?

No! It is impossible!!!

- **Frequent changes of code**
- **Architecture-related errors**
- **Data structure-related errors**
- **Algorithm-related errors**

Insufficient time and consideration for requirements analysis and design.

How Can We Improve It ?

- **Specification-Based Programming** (for improving understanding of requirements and design)
- **Pair Programming** (for improving interactions and cooperation)

Characteristics of Pair Programming

1)Two programmers work together. One is called *driver* and the other is called *observer* or *navigator*.

2)The driver writes code, while the observer reviews each line of code as it is typed in.

3)The two programmers switch roles frequently.

Problems:

- There is a lack of clear principle to govern the process of working together by the two programmers
- The work can be considerably affected when the collaboration of the two programmers does not go well.

The dark side of pair programming.

- It is more costly than one-person programming.

Question

How to take advantage of both specification-based programming and pair programming and limit their disadvantages?

Solution:

Human-Machine Pair Programming(HMPP) (for automatic monitoring, predicating, and incremental program review)

(2) What is human-machine pair programming (HMPP)?

Definition: **HMPP** means that a programmer and computer work together to construct a program, where the programmer plays the role of *driver* and the computer plays the role of *observer*.

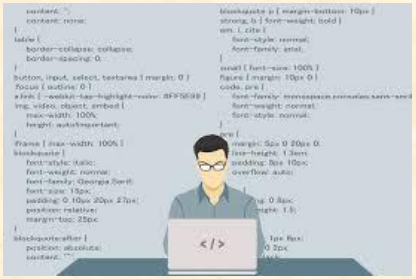
program = specification or

code or

combination of both

Characteristics:

A) One programmer works together with computer in programming.



B)The programmer constructs the program, while the computer analyses the program that has been typed in to detect bugs and to predict code fragments.

C) It is much less costly than conventional pair programming and human mistakes will be significantly reduced.

2. Theoretical Foundation and Framework for HMPP

The theoretical foundation defines the roles of computer in HMPP and the principle of fulfilling each role.

The framework provides an architecture and procedure for realizing the roles of computer (design for tool support) in HMPP.

An Evolutionary View of Programming

The research on HMPP focuses on the observer role of computer and on dealing with the problem of how to construct a correct program S through a series of evolutions of partial programs:

$$S1 \ll S2 \ll \cdots \ll Sn = S$$

where each Si (i = 1..n) is a partial program (program segment), and S is the completed program.

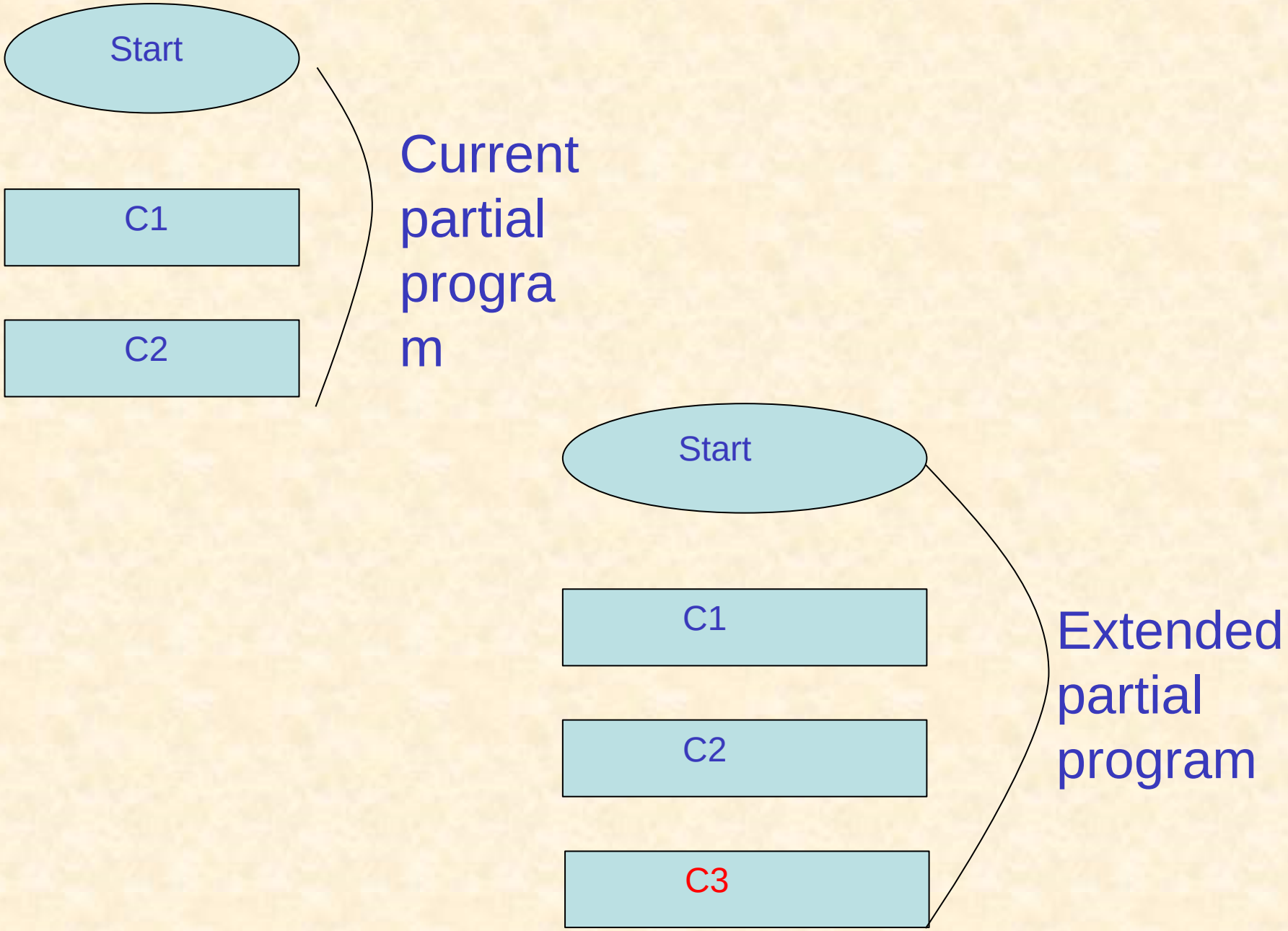
$S1 \ll S2$ means that S1 is evolved to S2, or S2 is an evolution of S1.

Program Evolution

Definition: A partial program S is a sequence of commands, denoted by S = [C1, C2, …, Cm], where each Ci (i = 1, …, m) is a command (a specification or code).

Definition: Let S1 = [C1, C2, …, Cm] and S2 = [C1', C2', …, Cn']. Then, S1 is extended to S2, denoted by S1 $\leqq$ S2, if and only if they satisfy the condition:

Ci = Ci' (for i = 1, …, m) and n > m

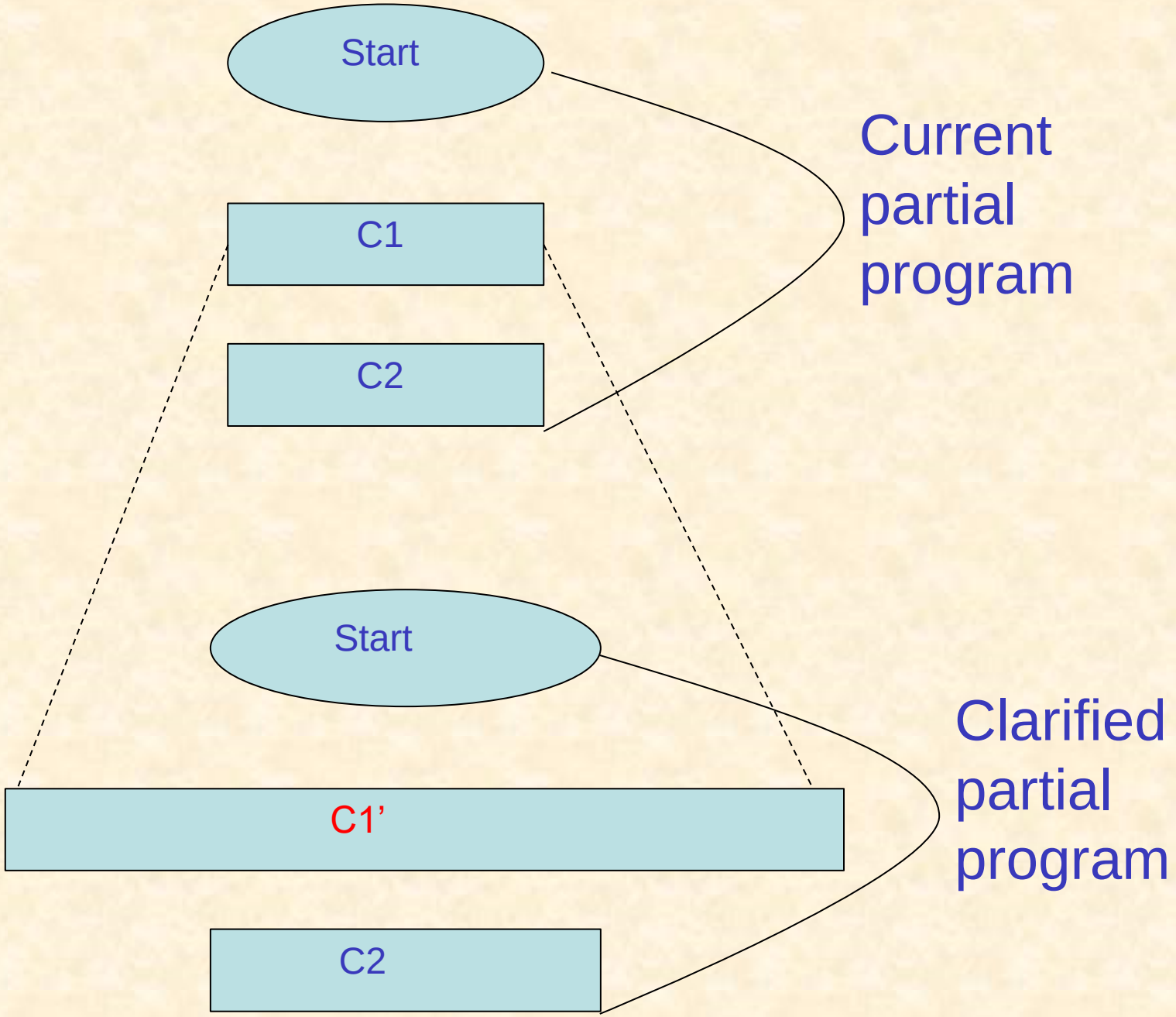

Start
C1
C2
Current partial program
Start
C1
C2
C3
Extended partial program

Definition: Let S1 = [C1, C2, …, Cm] and S2 = [C1', C2', …, Cn']. Then, S1 is clarified to S2, denoted by S1 Π S2, if and only if they satisfy the condition:

n = m and for some Ci (1 <= i <= m), Ci is redescribed more clearly by Ci'.

In this case, we also say S2 is a clarification of S1.

The clarification also defines the human's responsibility.


Start
C1
C2
Current partial program
Start
C1'
C2
Clarified partial program

Definition: Let S1 and S2 be two partial programs. Then, S1 is evolved to S2, denoted by S1 $\ll$ S2, if and only if they satisfy the condition:

S1 is either extended to S2 or

S1 is clarified to S2.

Definition: Let S1 be a partial program of a correctly completed program. Then, S1 must satisfy a set of desired properties denoted by P_S1.

2.1 Theoretical Foundation

The roles of the computer observer:

1) Software Construction Monitoring (SCM)

1.1) Learning patterns for making faults from programming (where a fault is a syntactic expression that violates some property of the current partial program.)

1.2) Verifying properties of partial programs to detect potential faults

1.3) Reporting potential faults and the related information

2) Software Construction Predicting (SCP)

2.1) Self-correction of the mistakes in partial programs to remove faults

2.2) Predicting program fragments to enhance the robustness (or other properties, such as safety or security) of the program

2.3) Predicting program fragments towards completing the program

2.4) Reporting the predicted program fragments

3) Incremental Program Review (IPR)

3.1) Transforming programs to graphical representations for comprehension

3.2) Guiding the programmer to review the properly selected program fragments

3.3) Carrying out knowledge-based peer review

1.1) Learning patterns for making faults from programming

Analyse the process of editing the program to determine the patterns for making faults based on how many times the same expression or statement is repeatedly modified.

Example1: assume the decision

if (x > y && y < z) {…}

is modified twice as follows:

x > y || y < z → x > z & y < z → x > y && y < z

Then, the computer will learn the pattern indicating that decisions containing the logical operator || is likely to contain mistakes.

Example2: assume that each of the three decisions

```
if (x.age >= 20) { …;}
…
if (x > y && x < z) {…;}
...
while (amount < balance) {…;}
```

is modified twice during the editing of the program, then the computer will lean the pattern indicating that the decisions in conditional and iteration statements are likely to contain faults.

The same principle can be applied to other syntactic phenomena, such as variable declaration, function calls, assignments, nested statement, and components.

1.2) Verifying properties of partial programs to detect potential faults

Let S1 be the current partial program. Let P_1, P_2, ..., Pn be properties S1 must satisfy.

SCM aims to automatically, dynamically check whether the current partial program S1 satisfies these properties.

A property may be formed based on the specification or formed based on some implementation rules (e.g., avoiding exceptions).

Challenges

- How to find and define all the properties P_1, P_2, ..., Pn that S1 must satisfy?

- Given a relevant property Pi ($i \in \{1,2,...,n\}$), how can S1 be automatically and efficiently checked to determine whether it satisfies Pi or not?

Potential Techniques

For defining properties:

1. Specification-Based property definition
2. Exception-based property definition

For verifying the properties:

1. Specification-based static analysis
2. Predicate-based testing

1.3) Reporting potential faults and the related information

How to report the detected faults and potential faults will affect the accuracy and efficiency of human understanding and the human-machine interaction.

The issues to address:

(1) Format of the reported message (the structure of the message)

(2) Presentation style of the reported message (the notation for expressing the message)

(3) The level of the detail of the fault description

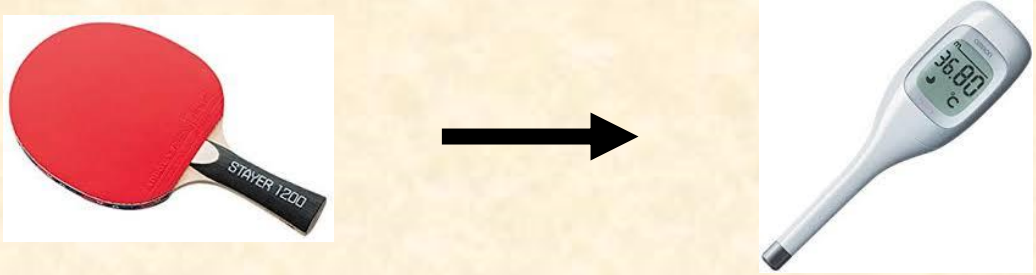
Challenges

(1) How to ensure that all of the reported faults are real faults for S1 ?

(2) How to ensure that the programmer will accurately and efficiently understand the reported faults given in the adopted format, style, and detail level?

2.1) Self-correction of the mistakes in partial programs to remove faults

After the programmer types in a line of program, the SCP system will automatically find some obvious mistakes and automatically correct them, respectively. After the correction, the results should be highlighted to remind the programmer of the changes.



The items that can be possibly self-corrected:

(1) Variable names
(2) Operators
(3) Function definitions
(4) Statements
(5) Decision and conditions
(6) Access restrictions (private, protected, public)
(7) Others

Possible techniques for self-correction:

(1) Change the syntax of the target

(e.g., F0rmal → Formal)

(2) Add necessary items to the target

(e.g., month >= 1 and month <= 12)

(3) Remove items from the target

(e.g., a < 10 && a < 20 → a < 20)

2.2) Predicting program fragments to enhance the robustness of the program

The robustness of a program is often concerned with input from the human-machine interface (e.g., GUI) and exceptions. The SCP system should automatically identify those parts and determine whether there is a need to improve the current partial program. If yes, then an appropriate program fragment should be added.

In general, the means for improving the robustness is exception handling. There might be other ways, such as adding conditional statements (e.g., a person's age must be greater than 0 and less than 150; the amount for withdrawal from ATM must be great then 0 and less than a specified limit).

2.3) Predicting program fragments towards completing the program

Suppose a completed program S is composed of n commands C_1, C_2, ..., Cn. Abstractly, it is represented as S = [C_1,C_2,...,Cn].

Let the current partial program be CV_S = [C1, C2]. Then predicting program fragments means to make an extension of CV_S, for example CV_S' = [C1, C2, C3, C4].

A sequential program usually defines a mathematical function: given an input, it will produce an output.

When the current version of the program is written, the SCP system should automatically understand what should be written next and therefore propose a program fragment for this purpose.

Best  Best regards

Possible situations for proposing a program fragment:

(1) Consider the logical expression (decision, condition) in if-then-else statements in order to ensure that for every possible situation, the corresponding processing statements are provided.

Example:

Current statement: if (amount <= balance && amount <= w_limit && amount > 0){ }

Proposed program fragment:

```
if (amount <= balance && amount <= w_limit && amount <= 0){
…;}
if (amount <= balance && amount > w_limit && amount > 0){…; }
if (amount > balance && amount <= w_limit && amount > 0){…;}
```

(2) Consider an iteration statement to ensure that for every possible exit of the iteration statement, the corresponding processing statements are provided.

Example:

current statement: while (p1 && p2 && p3) { …;}

proposed program fragment:

```
if (!p1) {…;}
if (!p2) {…:}
if (!p3) {…;}
```

(3) Consider a class to ensure that the class contains all of the necessary functions (or methods).

Example:

current class:

```
class account { string name;
                string acc_no;
                int balance;
                int w_limit;
                Deposit(int amount){…;}
              }
```

proposed program fragment:

```
class account {…; //existing code
                string getName(){return name;}
                void setName(string na){name = na;}
                …
                int Withdraw(?){?}
                int Inquire(){?}
}
```

2.4) Reporting the predicted program fragments

How should the predicated program fragment be presented to ensure that the human programmer can accurately and efficiently understand it for deciding how it can be adopted in the current version of the program.

The issues to address:

(1) Presentation style of the proposed program fragment (code, pseudocode, or diagram?)

(2) The level of the detail of the proposed program fragment.

3.1) Transforming programs to graphical representations for comprehension

Examples:

(a) Data flow diagrams

(b) Control flow diagrams

(c) Variable dependency graph

3.2) Guiding the programmer to review the properly selected program fragments

Activities:

(a) Select program fragments for review (e.g., where in the newly constructed program parts needs to be reviewed? Complexity? Importance?)

(b) Raise questions about the selected fragments to guide the review and support the review process

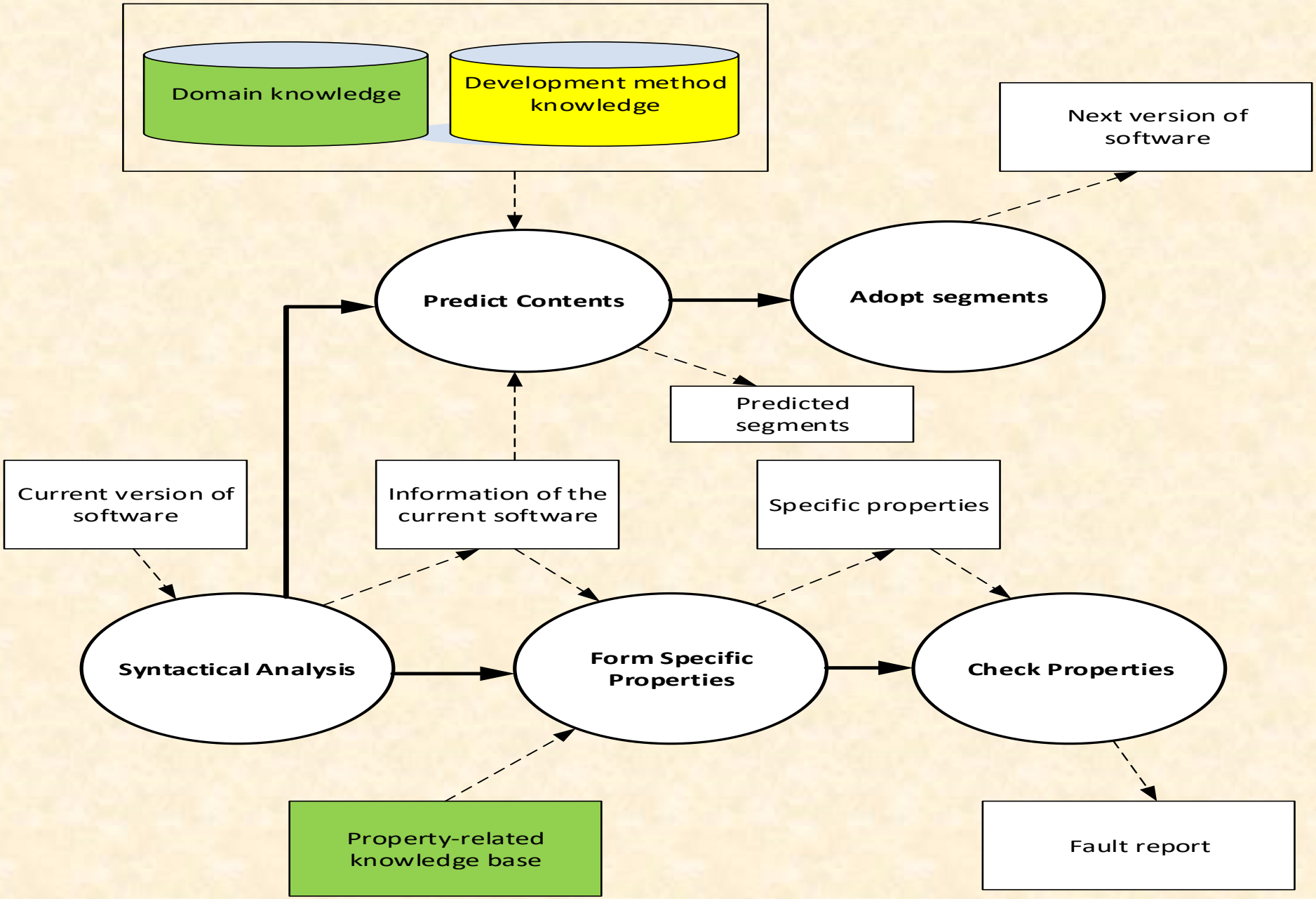
3.3) Carrying out knowledge-based peer review

Activities:

(a) Build a knowledge base of bug patterns for programs on computer. Each bug pattern is a faulty expression, and it can be formed based on the domain, the specification, the exceptions, peers, and other sources.

(b) Support the application of the knowledge to the current partial program.

Framework for HMPP



3. Knowledge Classification

Get back to the basics:

What are data? (1, 7, 8)

What is information? (one-hour ?, seven-hour ?, eight-hour ?)

What is knowledge? (7 + 1 > 8)

Knowledge for HMPP:

(1) Domain knowledge (e.g., ATM, Railway card, Air ticket reservation, Railway control system)

(2) Method knowledge (e.g., rules and procedures suggested by a specific programming method or software development method)

(3) Property-related knowledge (e.g., various properties of variables, expressions, statements, modules)

(4) Fault-related knowledge (e.g., common faults, specific faults)

(5) Standard-related knowledge (e.g., the depth of nested conditional statements should be less than 5; variable declarations should be given together before the statements in the body of a function)

(6) Language-related knowledge (e.g., a function can only return one value; no multiple inheritances of class is allowed)

(7) Self-correction knowledge (e.g., class Derived:: public Base should be corrected to: class Derived: public Base)

(8) Robustness-related knowledge (e.g., input value should be converted into a consistent type; a person's age should be between 0 and 150 or 200)

(9) Extension knowledge (e.g., after the while (p1 || p2 || p3) { …;}, there should be appropriate program statements to deal with the situations when each of the condition in the loop decision is false)

(10) Others

4. HMPP for Agile-SOFL

(1) When a specific HMPP system is built, it is always supposed to support a specific software construction or programming method.

(2) Building a HMPP system to support multiple programming methods is possible, but its efficiency and effectiveness would be considerably damaged.

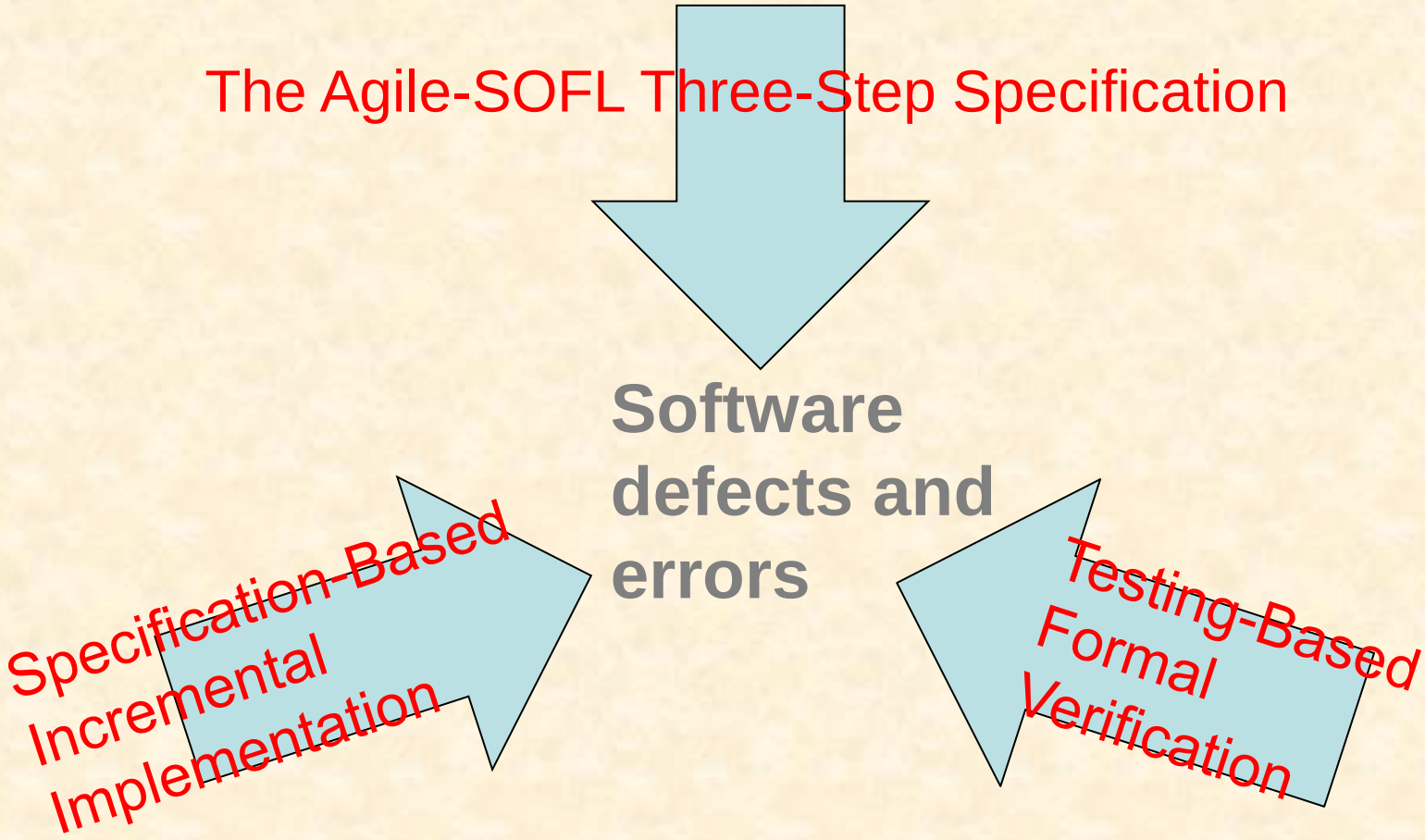
(3) Building a domain-specific and method-specific HMPP system would be the best way to gain efficiency and effectiveness in supporting programming.

Agile-SOFL: a specification-based programming agile method

Characteristics:

1. A three-step approach to building comprehensible hybrid specification for analyzing requirements and defining what to be done by the potential system.
2. Testing-Based Formal Verification (TBFV) for program verification.
3. Incremental implementation together with the application of TBFV in small cycles

Principle of Agile-SOFL



HMPP for Agile-SOFL

SOFL = Structured Object-Oriented Formal Language

The main functions to support:

(1) Construction of hybrid specifications (combination of semi-formal specification, GUI design, and formal specification)

(2) Module-based incremental programming

(3) Program testing and debugging

5. Challenges

- Identification and definition of all the necessary properties for partial programs.
- Theory and techniques for learning frequently occurred faults from programming activities (e.g., editing, testing, maintenance)
- Identification and definition of domain knowledge, programming method knowledge, and knowledge for predicting program fragments.
- Knowledge representation techniques for efficiently searching and applying the knowledge in the knowledge base.
- Theory and techniques for efficient and effective interactions between human and machine.

6. Conclusions

- Human-Machine Pair Programming (HMPP) is a promising technology for software development, but the research on it is just beginning.
- HMPP combines AI with software development technologies and will significantly improve software productivity and quality, but it will not completely replace humans in software development.
- HMPP can only be realized with high-quality tool support and building the tool support needs to rely on a solid theoretical foundation. Formal methods can be a good means to help study the theoretical foundation.

7. Future Work

(1) Tackle the challenging issues given previously.

(2) Build intelligent tools to support HMPP.

(3) Apply HMPP to software engineering in practice.

Related publications

(1) Shaoying Liu and Shin Nakajima, "Automatic Test Case and Test Oracle Generation based on Functional Scenarios in Formal Specifications for Conformance Testing", **IEEE Transactions on Software Engineering**, **DOI:** 10.1109/TSE.2020.2999884, 2020.

(2) Shaoying Liu, "A Three-Step Hybrid Specification Approach to Error Prevention", **Journal of Systems and Software**, Elsevier, Vol. 178, 110975, 2021, DOI: https://doi.org/10.1016/j.jss.2021.110975.

(3) Shaoying Liu, "Software Construction Monitoring and Predicting for Human-Machine Pair Programming", **Proceedings of 8th International Workshop on SOFL +MSVL 2018 for Reliability and Security**, LNCS 11392, Springer, Gold Coast, Australia, Nov. 16, 2018, pp. 3-20.

(4) Shaoying Liu, "Agile Formal Engineering Method for Software Productivity and Reliability", **The 14th Central and Eastern European Software Engineering Conference Russia (CEE-SECR 2018)**, ACM press, Moscow, Russia Federation, Oct. 12-13, 2018, pp. 64-69.

(5) Siyuan Li and Shaoying Liu, "A Software Tool to Support Scenario-Based Formal Specification for Error Prevention", **The 7th International Conference on SOFL+MSVL (SOFL+MSVL 2017)**, LNCS 10795, Springer, Xi'an, China, Nov. 16, 2017, pp. 187-199.

Thank you !