

JavaCard における 形式的技法適用例

永宮 悠大

群馬大学大学院工学研究科情報工学専攻 藤田研究室

Email : nagamiya@comp.cs.gunma-u.ac.jp

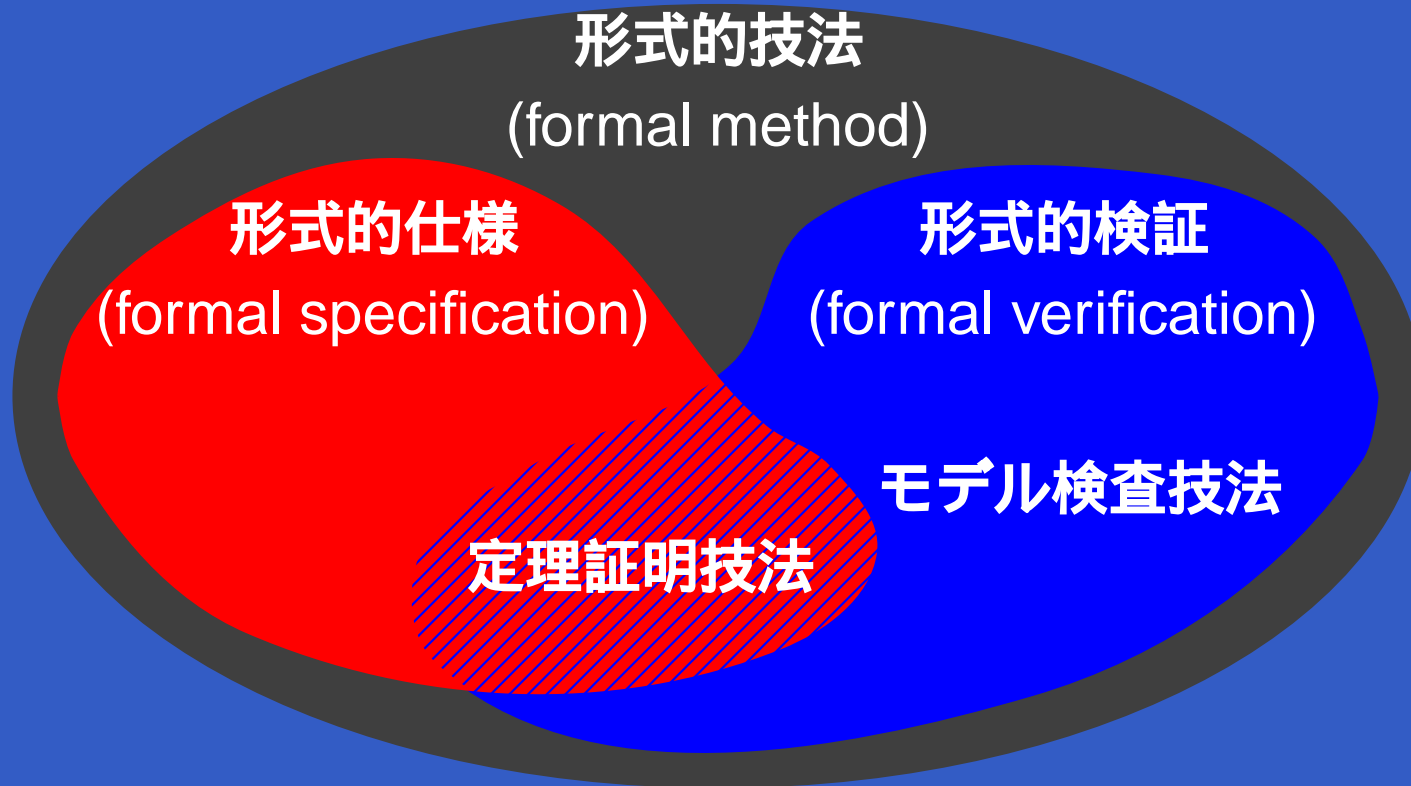
概要

1. 形式的技法とは?
2. なぜ形式的技法が必要か?
3. Java の形式化と検証
4. JavaCard の形式化と検証
5. まとめと今後の課題

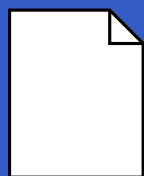
1. 形式的技法とは?

形式的技法 (formal method):

論理学を基盤としたシステムの設計・検証技法

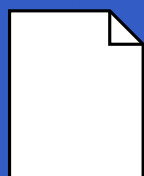


形式的仕様記述



自然言語で記述した
設計仕様

- 曖昧さが生じる



論理式で記述した
仕様

- システムについての性質を論理的な枠組で議論できる
- その性質を証明することができる



形式化した仕様

- 証明がコンピュータで処理できる形になる

検証

定理証明技法のフレームワーク

- 設計仕様 R を「同値な」形式的仕様 ϕ_R に変換
- ϕ_R を実装するプログラム P を記述
- $P \vdash \phi_R$ を証明

モデル検査技法のフレームワーク

- システムを状態遷移モデル \mathcal{M} にモデル化
- 検証したい性質 ϕ を記述
- $\mathcal{M} \models \phi$ を証明

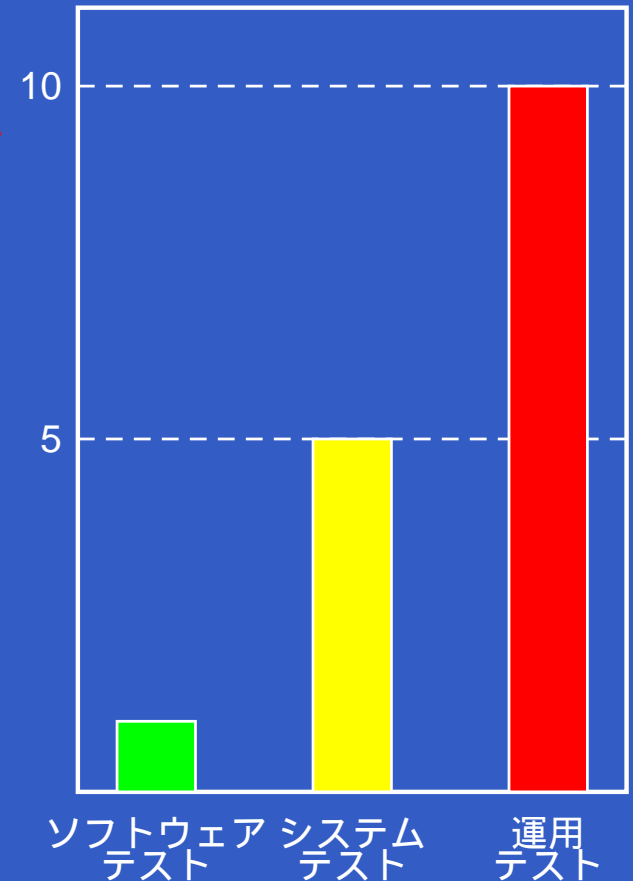
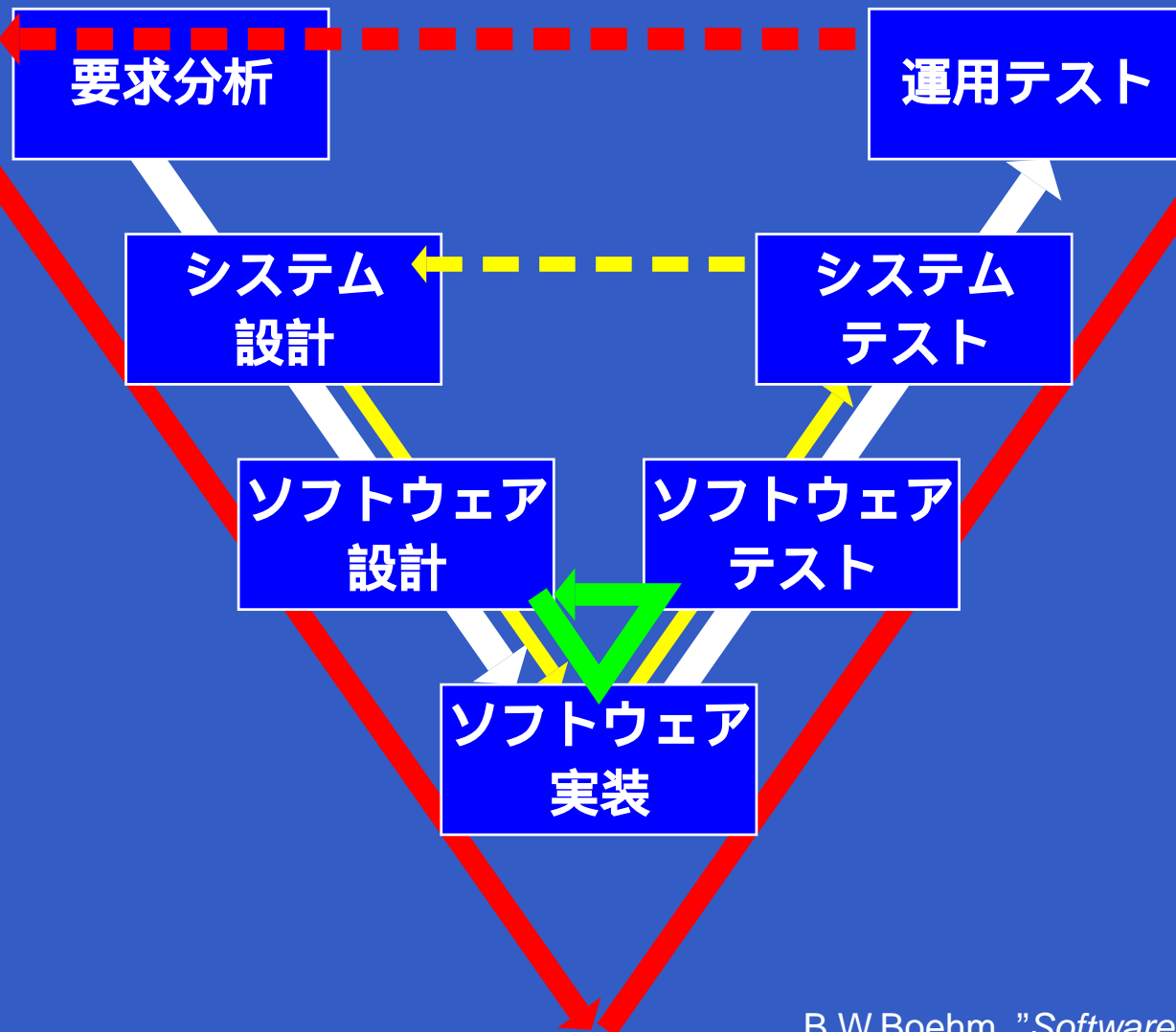
定理証明技法 vs モデル検査技法

	定理証明技法	モデル検査技法
検証方法	論理的証明	全数検査
用意するもの	ソースコード 形式的仕様	状態遷移モデル 検証したい性質
自動化のレベル	半自動	全自動
誤りの原因究明	×	(反例が出力)
対象	逐次的 停止する	並列的 停止しない
適用できる工程	上流	上流
仕様品質の向上		×

2. なぜ形式的技法が必要か?

- 増大するリコール問題
 - 携帯電話の回収騒動 (2001 年)
 - SO502iWM 3万台回収
 - SO503i 42万台回収
 - P503i 23万台回収
 - Pentium CPU の FDIV エラー (1994 年)
 - 浮動小数点除算の結果が正常と異なる
 - 対象 CPU の無償交換
- 顕在化する手戻りの発生

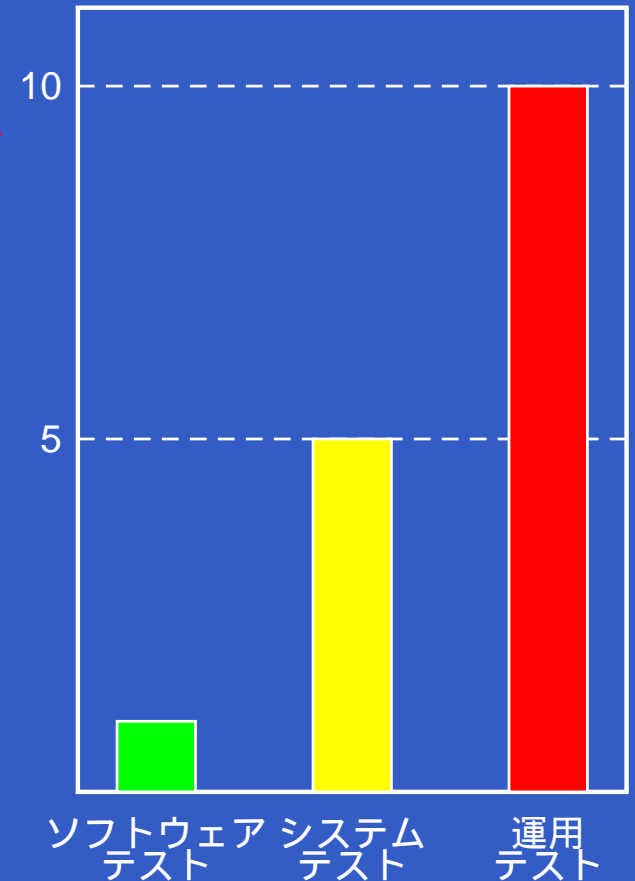
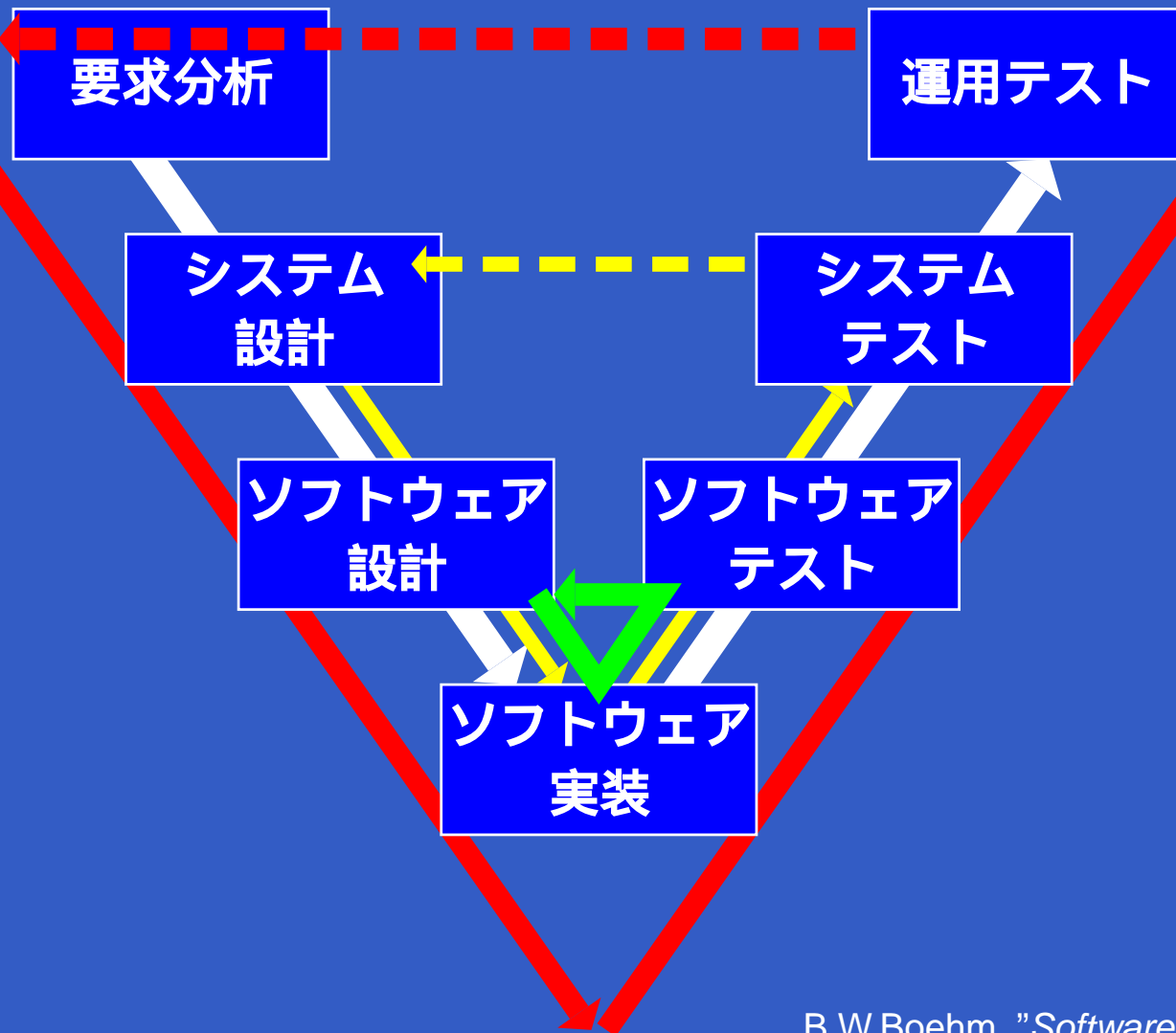
手戻りとコストの関係



発見工程別相対修正コスト

B.W.Boehm, "Software Engineering", IEEE Trans. on Comp, 1976

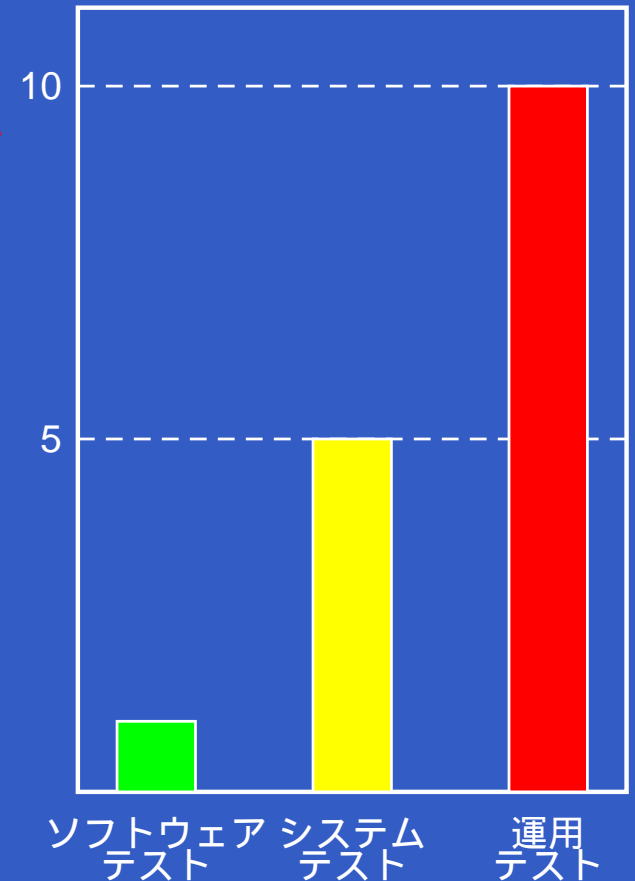
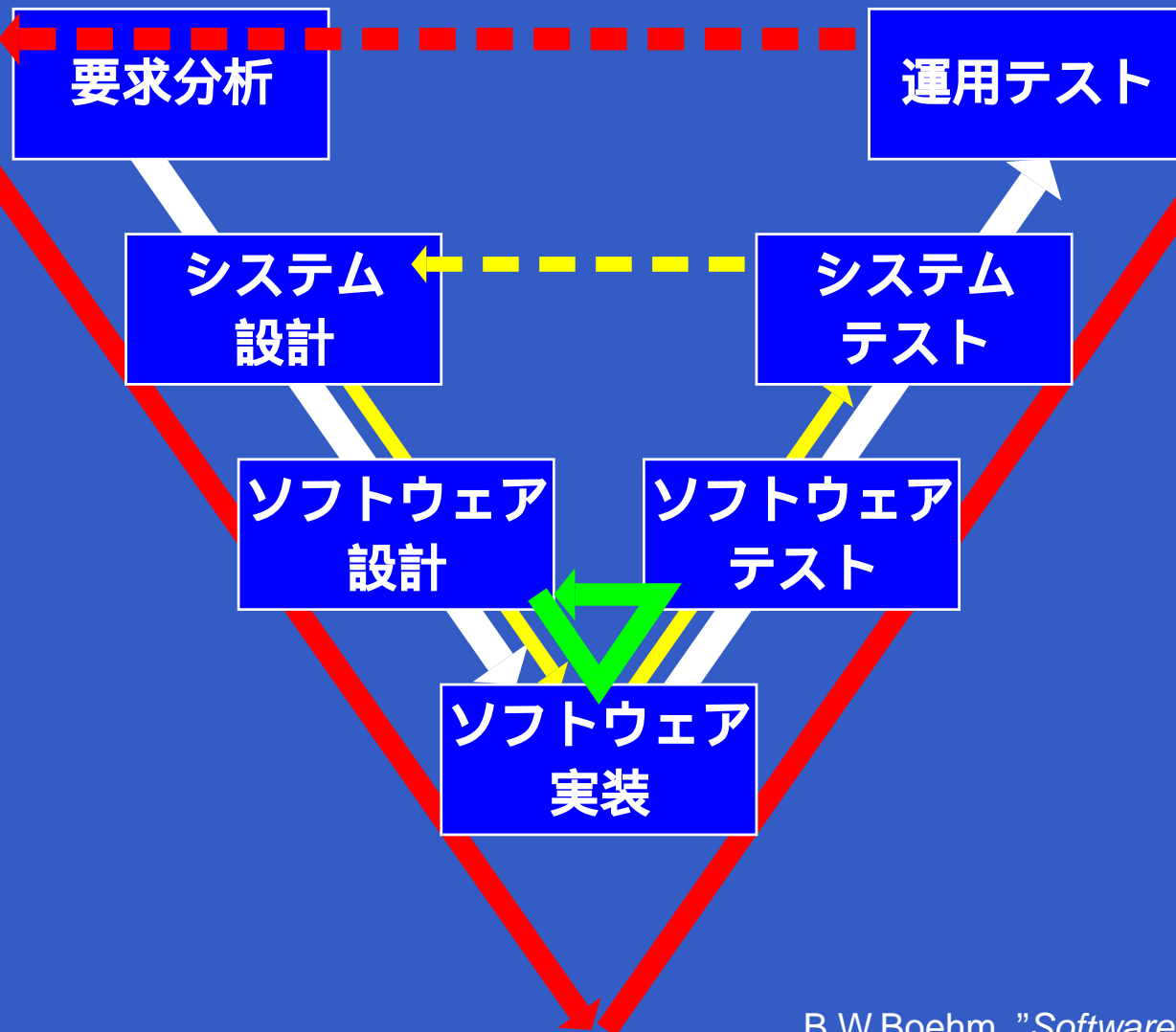
手戻りとコストの関係



発見工程別相対修正コスト

B.W.Boehm, "Software Engineering", IEEE Trans. on Comp, 1976

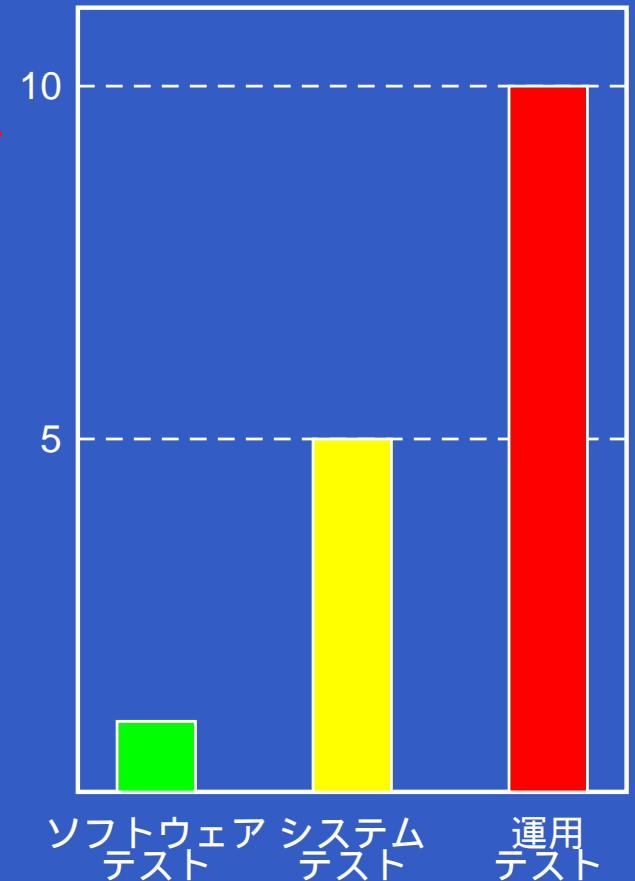
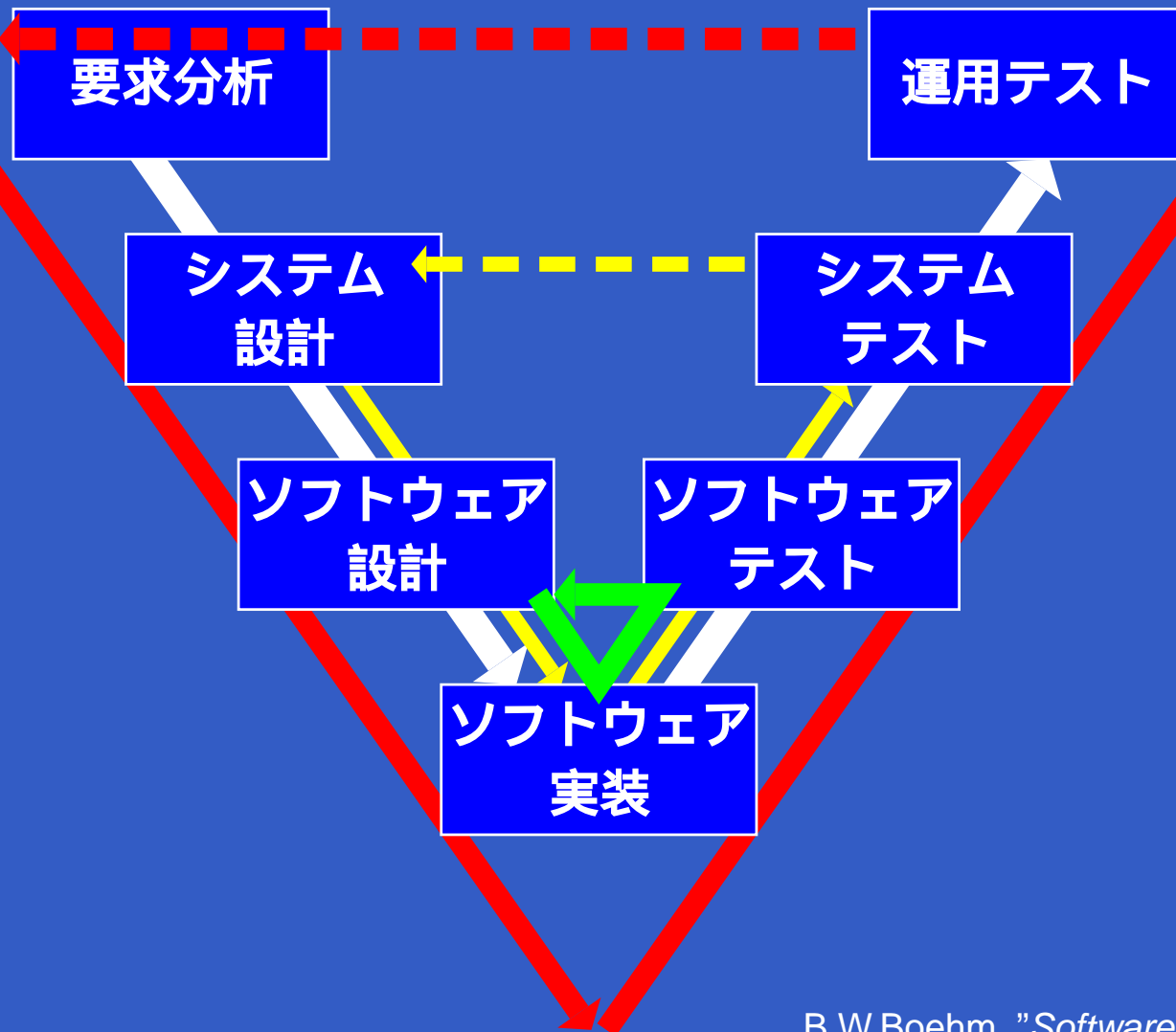
手戻りとコストの関係



発見工程別相対修正コスト

B.W.Boehm, "Software Engineering", IEEE Trans. on Comp, 1976

手戻りとコストの関係



発見工程別相対修正コスト

B.W.Boehm, "Software Engineering", IEEE Trans. on Comp, 1976

機能安全に関する国際規格 IEC61508

安全とは:

- **本質安全:**
ハザード (危険の原因) を取り除く措置で得られる安全
- **機能安全:**
ハザードの存在を認めリスクを軽減する措置で得られる安全

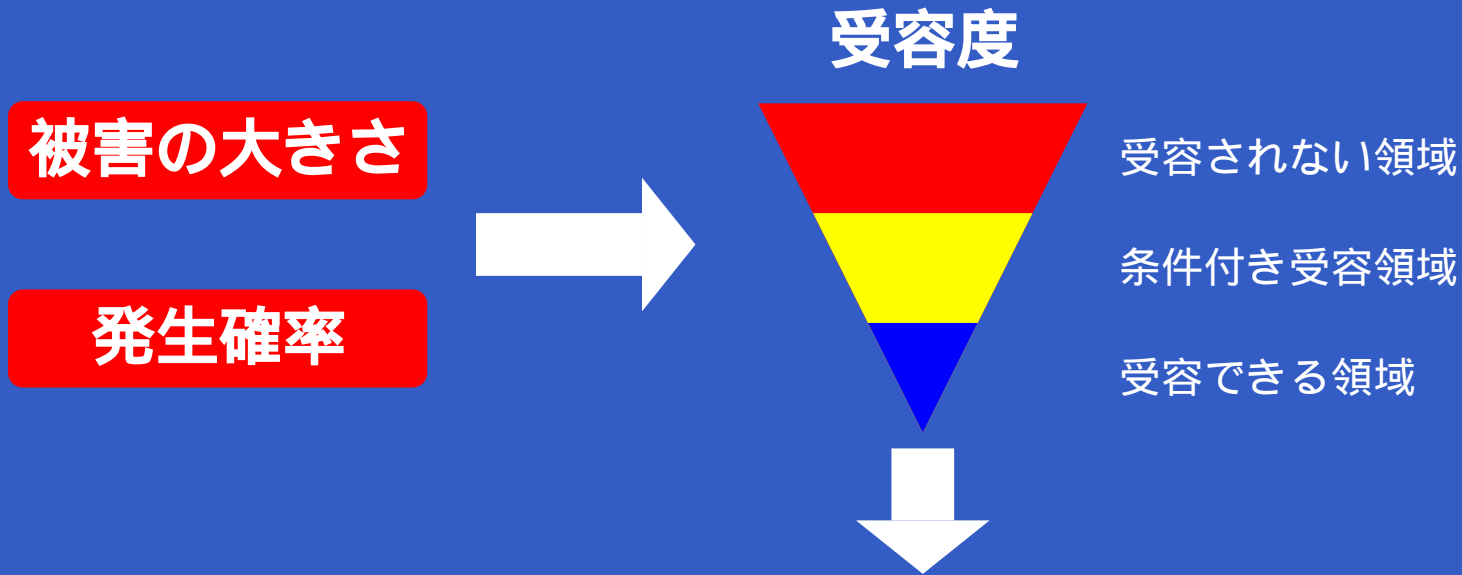
対象:

- 人命が関わったり人が負傷する可能性のあるシステム
- 障害によって社会に多大な損害が出るシステム

認証機関:

- Sira(イギリス), TÜV(ドイツ)

機能安全の考え方



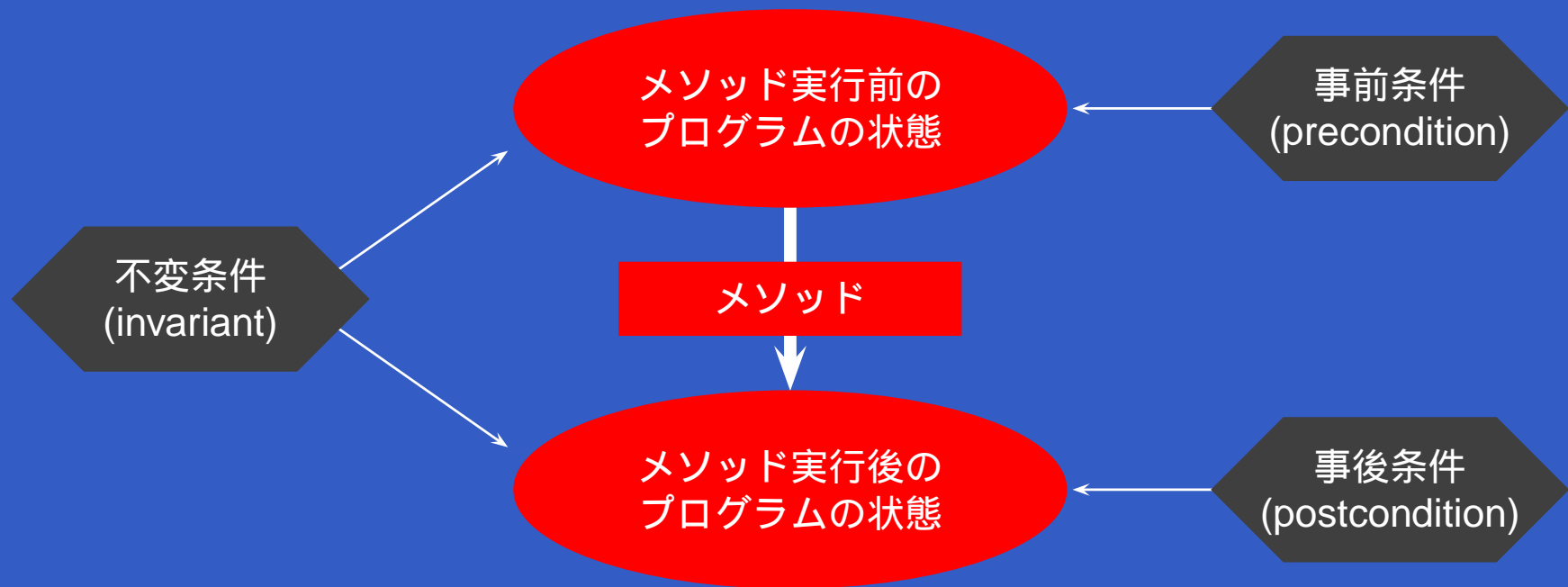
	SIL1	SIL2	SIL3	SIL4
アサーション・プログラミング	R	R	R	HR
構造化技法	HR	HR	HR	HR
準形式的技法	R	R	HR	HR
形式的技法	-	R	R	HR

R: 推奨 HR: 強く推奨 NR: 推奨しない

3. Java の形式化と検証

- 形式的仕様記述言語
 - JML(Java Modeling Language)
- 定理証明技法
 - Krakatoa + Coq , LOOP + PVS , Bali + Isabelle/HOL , ...
- ソースコードの静的検査
 - ESC/Java , ChAsE , Calvin , ...
- 型検査
 - JVM のバイトコード検証

Java Modeling Language



```
/*@ public normal_behavior
```

```
@ requires x >= 0;
```

```
@ ensures \result >= 0 && \result * \result <= x
```

```
@      && x < (\result + 1) * (\result + 1);
```

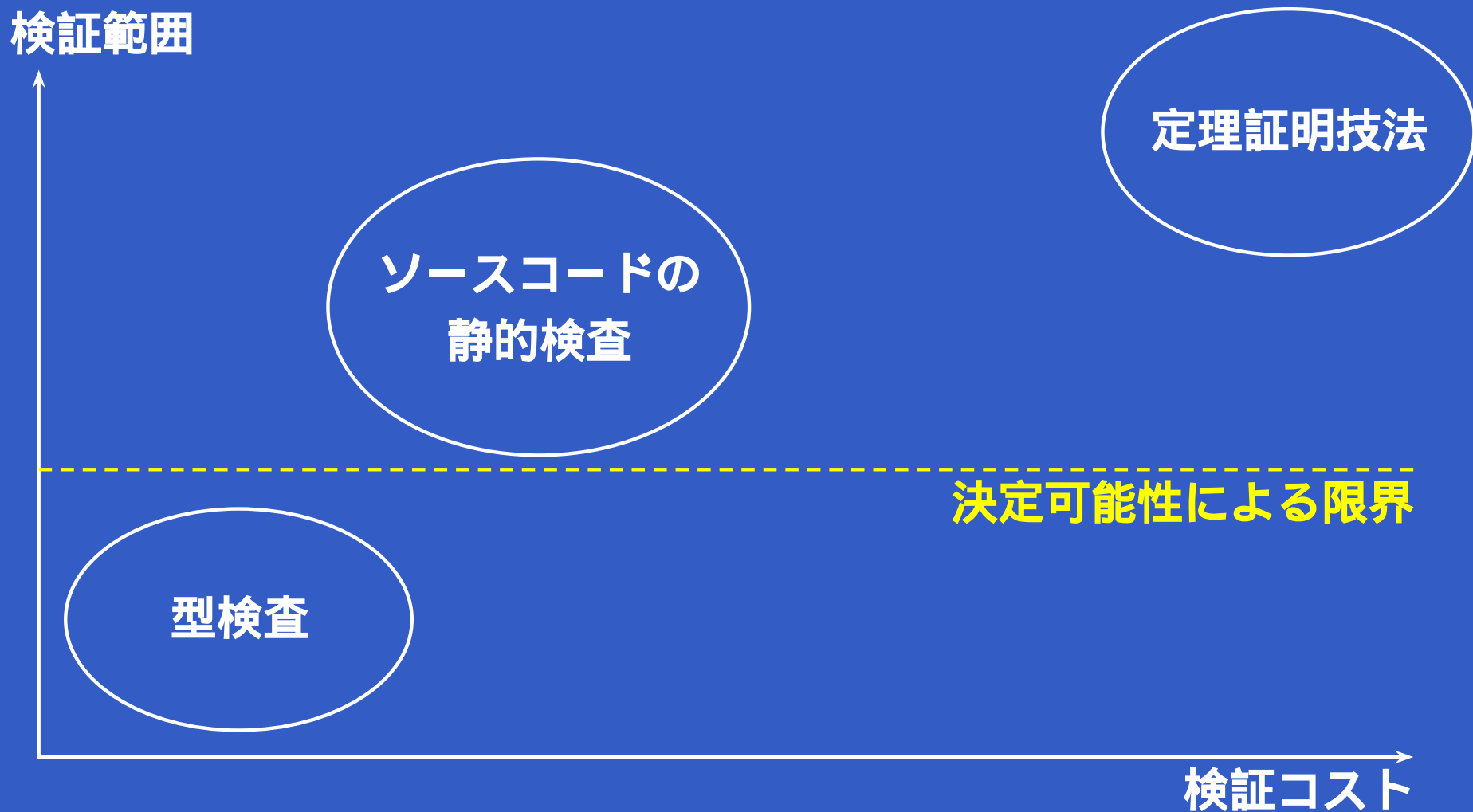
```
@*/
```

```
public static int sqrt(int x){}
```

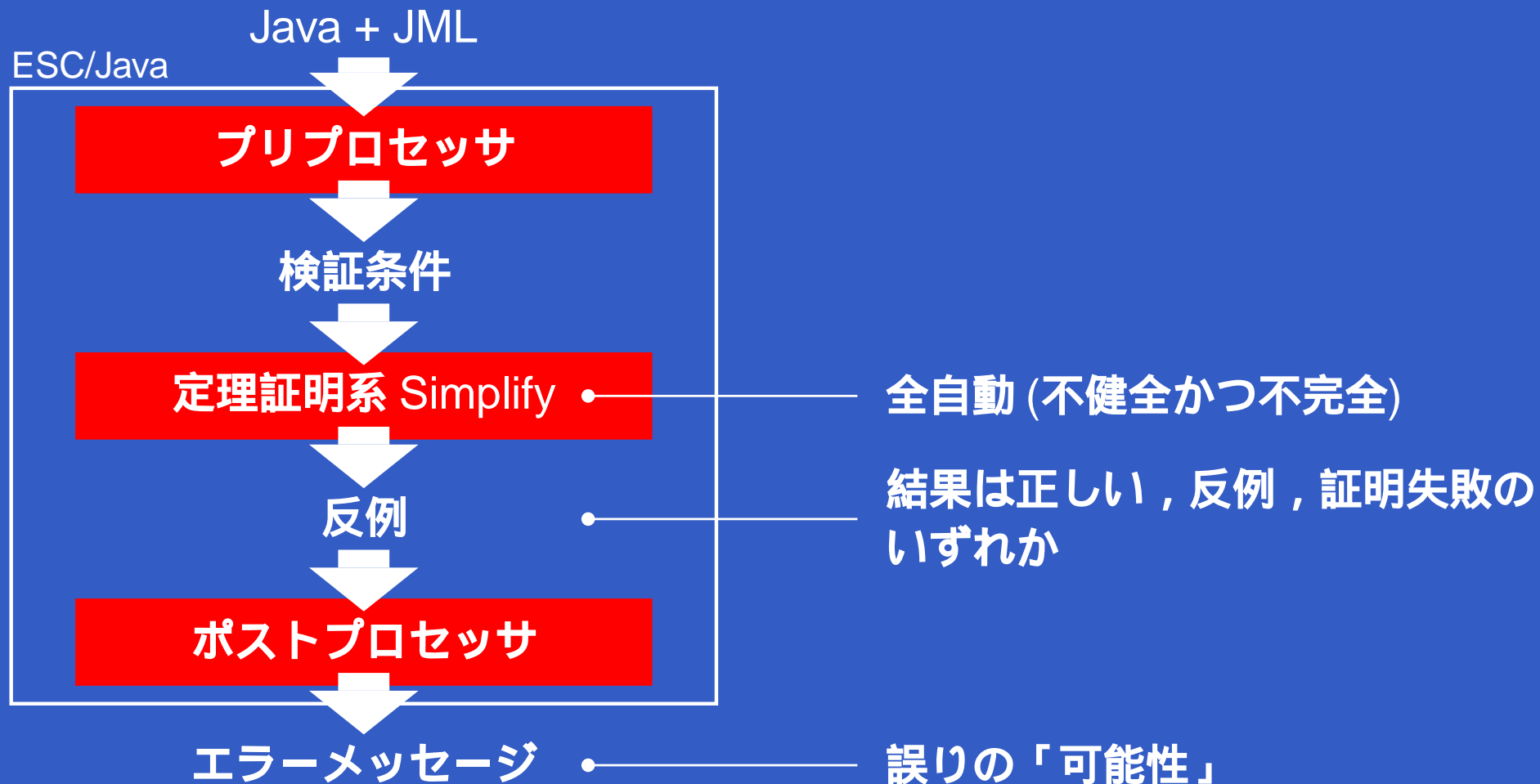
事前条件

事後条件

Javaの検証の種類と性質



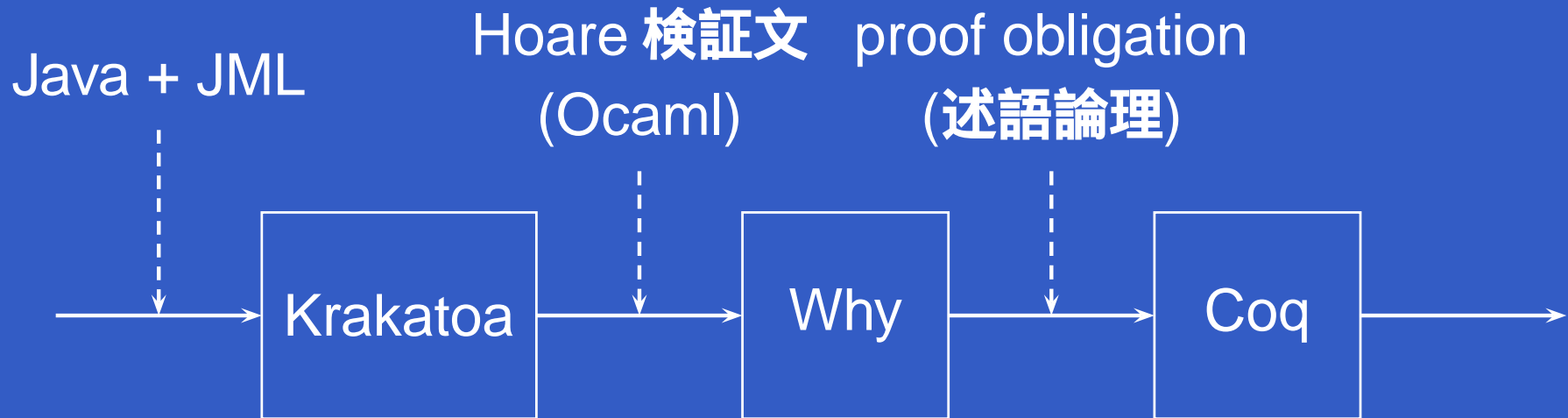
ESC/Java による誤りの検出



11: Warning: Possible null dereference (Null)

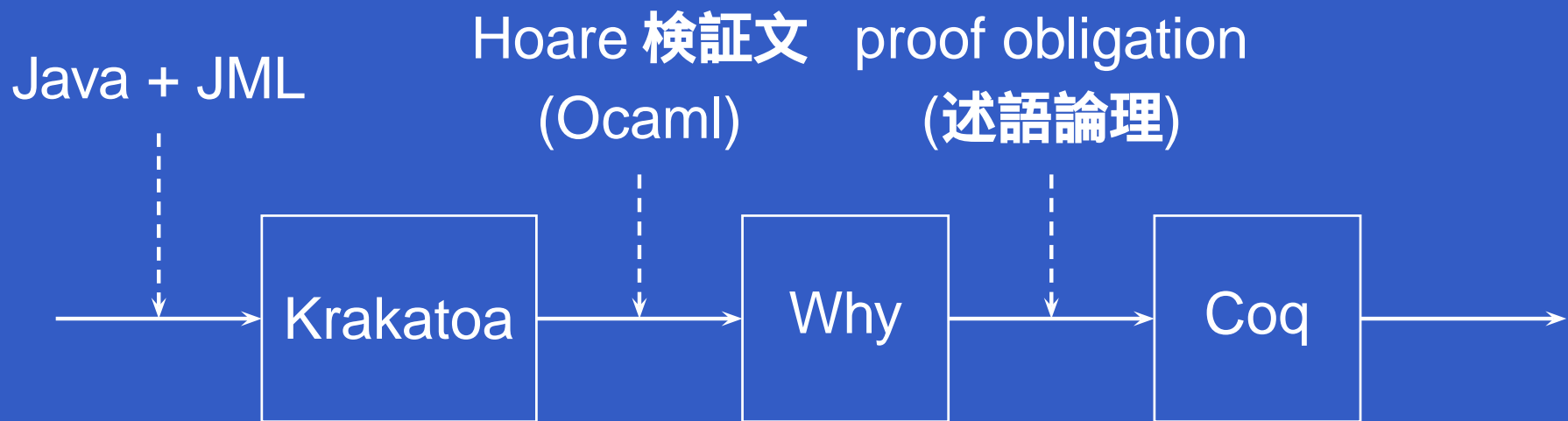
14: Warning: Postcondition possibly not established (Post)

Krakatoa による検証



- Krakatoa: Java/JML ソース \mapsto Hoare 検証文
- Why: Hoare 検証文 \mapsto_{Hoare} proof obligation
- Coq: proof obligation を対話的に証明

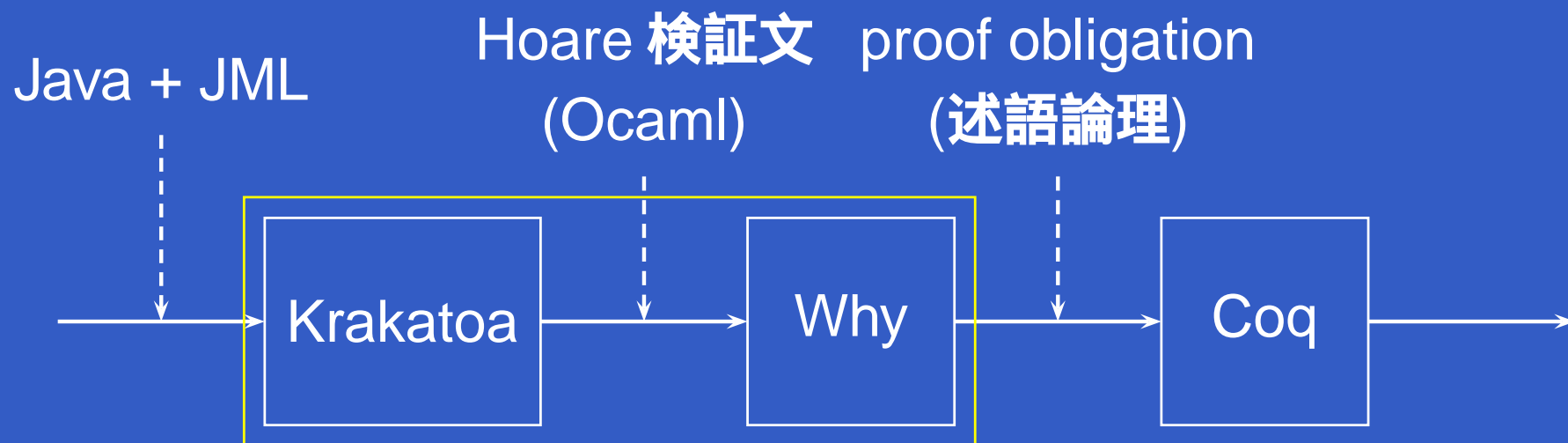
Krakatoa による検証



- Krakatoa: Java/JML ソース \mapsto Hoare 検証文
- Why: Hoare 検証文 \mapsto_{Hoare} proof obligation
- Coq: proof obligation を対話的に証明

プログラムが仕様を満たす \Rightarrow proof obligation が証明できる

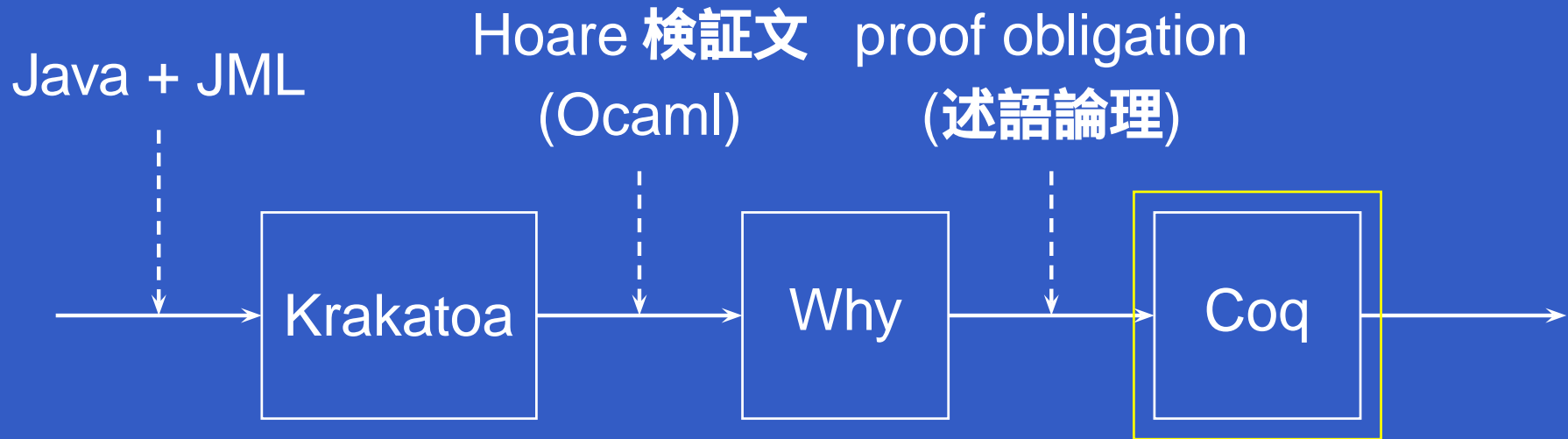
Krakatoa による検証



- Krakatoa: Java/JML ソース \mapsto Hoare 検証文
- Why: Hoare 検証文 \mapsto_{Hoare} proof obligation
- Coq: proof obligation を対話的に証明

プログラムが仕様を満たす \Rightarrow proof obligation が証明できる

Krakatoa による検証



- Krakatoa: Java/JML ソース \mapsto Hoare 検証文
- Why: Hoare 検証文 \mapsto_{Hoare} proof obligation
- Coq: proof obligation を対話的に証明

プログラムが仕様を満たす \Rightarrow proof obligation が証明できる

4. JavaCard の形式化と検証

JavaCard:

- Sun Microsystems 社が開発した Java 実行環境を持つ IC カード
- Java アプリケーションを複数個書き込むことができる

JavaCard の利用例:

- 電子マネー, クレジットカード, 身分証, ...

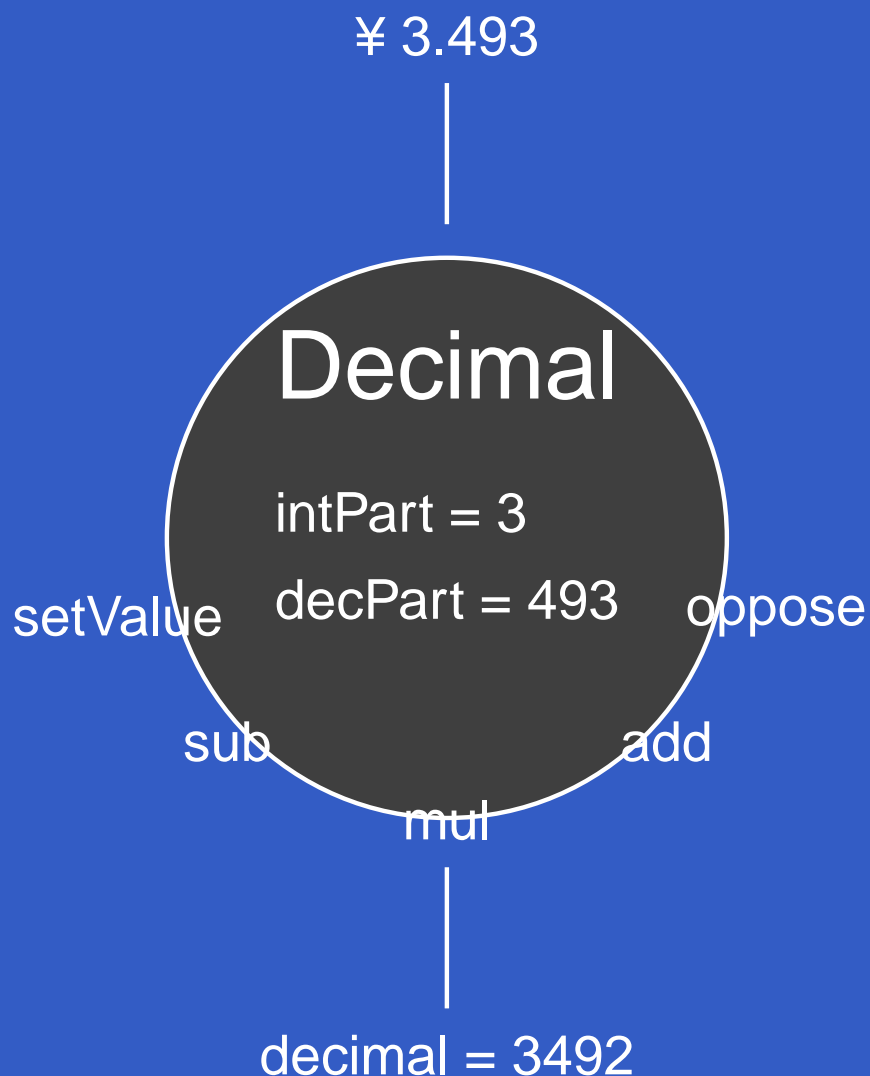
形式的技法の必要性:

- 高い信頼性が要求される
- JCVM はバイトコード検証機能を持たない

Gemplus の電子マネー

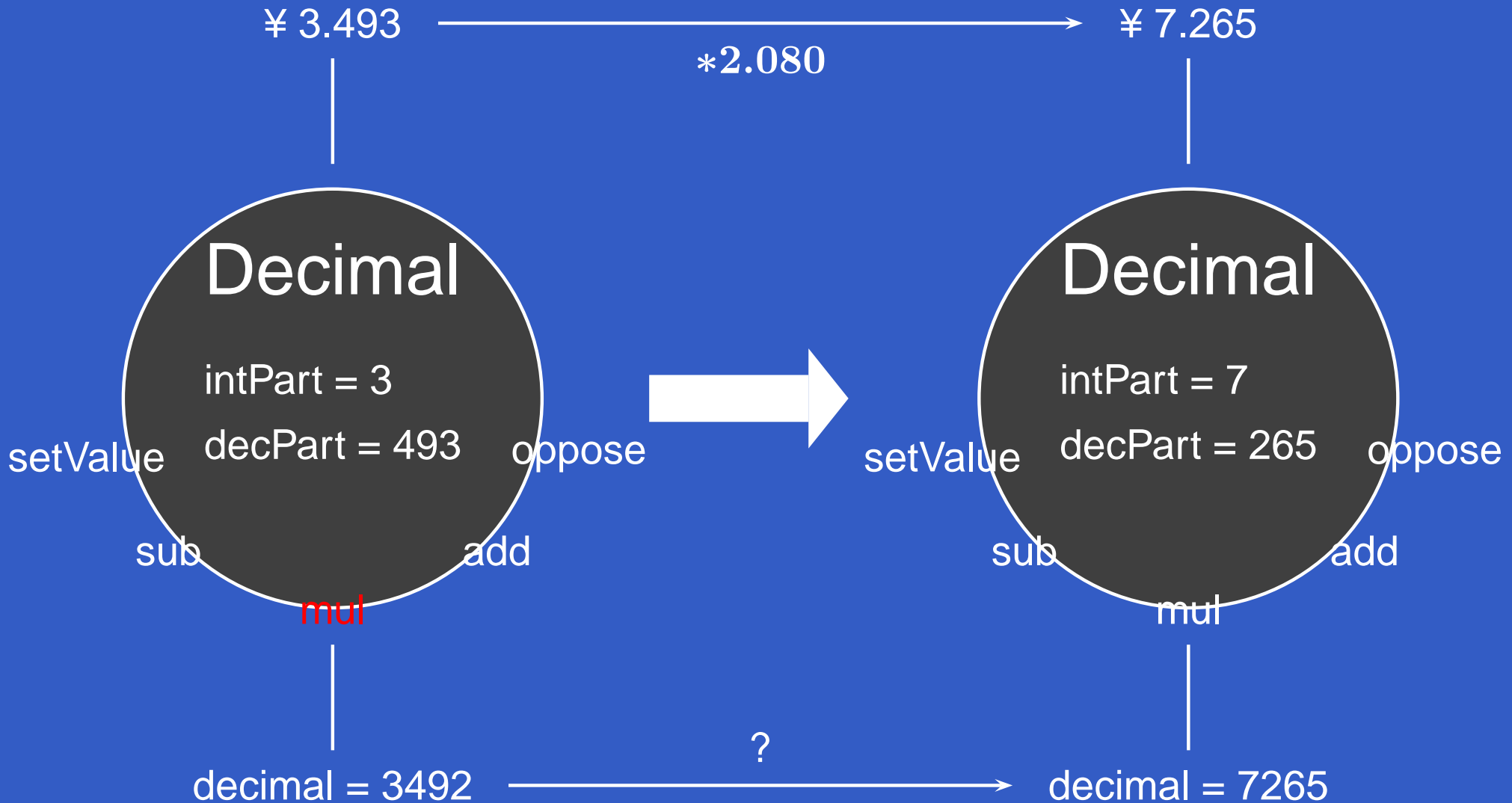
- JavaCard の検証の現実的な例として開発されたアプリケーション
- 入金 , 出金 , 外貨両替
- アプレット間通信
- 本人認証 (ID とパスワード)
- ロイヤリティ・ポイント

Decimal クラスによる残高の表現



- 浮動小数点型は存在しない
- `intPart` と `decPart` は short 型の変数
- `decPart` の有効桁数は 3 桁
- `0 <= decPart && decPart < PRECISION`
- `PRECISION` は short 型の定数値 1000
- `decimal <- intPart * PRECISION + decPart`

mul メソッドの実装と仕様を考える



浮動小数点を使わない少数の掛け算

$$a.b * e.f = a * e + a * 0.f + 0.b * e + 0.b * 0.f$$

- a と e は intPart に対応, b と f は decPart に対応
- 最初の三つの掛け算は add メソッドを用いて計算

```
for (short i = (short)0; i < e; i++) {  
    add(a,b);  
}
```

$a.b$ を e 回加える

```
for (short i = (short)0; i < a; i++) {  
    add(0,f);  
}
```

$0.b$ を a 回加える

- $0.b * 0.f$ (rest-part) はどうやって計算する?

rest-part の計算

- b と f は 1000 未満なので $0.b * 0.f = (b * f) / 10^6$
 $0.999 * 0.999 = 0.998001$
 $= 998001 / 10^6$
- decPart の有効桁数は 3 桁なので $\lfloor (b * f) / 10^3 \rfloor$ を計算すればよい
 $\lfloor 998001 / 10^3 \rfloor = \lfloor 998.001 \rfloor$
 $= 998$
- 計算結果を decPart に格納

rest-partの実装における問題点

```
add(0, ((b * f) / 1000));
```

- **b と f は short 型でも b * f の計算結果が short 型の範囲を越える可能性がある**

rest-partの実装における問題点

```
add(0, ((b * f) / 1000));
```

- **b と f は short 型でも $b * f$ の計算結果が short 型の範囲を越える可能性がある**

```
add(0, ((b / 1000) * (f / 1000)));
```

- **Java では $(b * f) / 1000$ と $(b / 1000) * (f / 1000)$ が等しくならない可能性がある (丸め誤差)**

例. $(347 / 10) * 10 == 340$

rest-partの実装における問題点

```
add(0, ((b * f) / 1000));
```

- **b と f は short 型でも $b * f$ の計算結果が short 型の範囲を越える可能性がある**

```
add(0, ((b / 1000) * (f / 1000)));
```

- **Java では $(b * f) / 1000$ と $(b / 1000) * (f / 1000)$ が等しくならない可能性がある (丸め誤差)**

例. $(347 / 10) * 10 == 340$

⇒ B.Jacobs らが解決策を考案

Jacobs のアルゴリズム

$$\begin{aligned} & (a * b) / 1000 \\ = & ((a_1 \ a_2 \ a_3) * b) / 1000 \\ = & \begin{pmatrix} 100 * a_1 * b \\ + \ 10 * a_2 * b \\ + \ a_3 * b \end{pmatrix} / 1000 \\ = & \begin{pmatrix} (c_1 \ c_2 \ c_3 \ c_4 \ 0 \ 0) \\ + \ (c'_1 \ c'_2 \ c'_3 \ c'_4 \ 0) \\ + \ (c''_1 \ c''_2 \ c''_3 \ c''_4) \end{pmatrix} / 1000 \\ = & \left(\begin{pmatrix} (c_1 \ c_2 \ c_3 \ 0 \ 0 \ 0) \\ + \ (c'_1 \ c'_2 \ 0 \ 0 \ 0) \\ + \ (c''_1 \ 0 \ 0 \ 0) \end{pmatrix} + \begin{pmatrix} (c_4 \ 0 \ 0) \\ + \ (c'_3 \ c'_4 \ 0) \\ + \ (c''_2 \ c''_1 \ c''_4) \end{pmatrix} \right) / 1000 \\ = & \begin{pmatrix} (c_1 \ c_2 \ c_3) \\ + \ (c'_1 \ c'_2) \\ + \ (c''_1) \end{pmatrix} + \begin{pmatrix} (c_4 \ 0 \ 0) \\ + \ (c'_3 \ c'_4 \ 0) \\ + \ (c''_2 \ c''_1 \ c''_4) \end{pmatrix} / 1000 \\ = & \begin{pmatrix} d_1 \\ + \ d_2 \\ + \ d_3 \end{pmatrix} + \begin{pmatrix} e_1 \\ + \ e_2 \\ + \ e_3 \end{pmatrix} / 1000 \end{aligned}$$

mul メソッドの実装

```
private void mul(short e, short f) {  
    short a = intPart;  
    short b = decPart;  
  
    intPart = (short) 0;  
    decPart = (short) 0;
```

```
    for (short i = (short) 0; i < e; i++) {  
        add(a, b);  
    }
```

$a * b + 0.b * e$

```
    for (short i = (short) 0; i < a; i++) {  
        add((short) 0, f);  
    }
```

$a * 0.f$

```
    short a1 = (short) (b / 100);  
    short a2 = (short) ((b - a1 * 100) / 10);  
    short a3 = (short) (b - a1 * 100 - a2 * 10);
```

```
    short d1 = (short) ((a1 * f) / 10);  
    short d2 = (short) ((a2 * f) / 100);  
    short d3 = (short) ((a3 * f) / 1000);  
    short gross = (short) (d1 + d2 + d3);
```

$0.b * 0.f$

```
    short e1 = (short) ((a1 * f - d1 * 10) * 100);  
    short e2 = (short) ((a2 * f - d2 * 100) * 10);  
    short e3 = (short) (a3 * f - d3 * 1000);  
    short rest = (short) (e1 + e2 + e3);
```

```
    add((short) 0, (short) (gross + (rest / 1000)));  
}
```

mul メソッドの仕様

```
/*@ requires 0 <= f && f < PRECISION &&
@           0 <= e && e <= MAX_DECIMAL_NUMBER &&
@           (e+1) * (intPart+1) < MAX_DECIMAL_NUMBER;
@ modifies decimal
@ ensures decimal =
@           e * \old(intPart) * PRECISION
@           +
@           \old(intPart) * f
@           +
@           e * \old(decPart)
@           +
@           (f * \old(decPart)) / PRECISION;
@ signals(Exception e) false;
@*/
```

検証結果

ESC/Java:

- Decimal クラス全体を検証
- いくつかの事後条件で Simplify エラーによる証明失敗

Krakatoa:

- mul メソッドを検証
- いくつかの proof obligation が証明できなかった
 - LOOP では証明されている [Jacobs, 2004]

5. まとめと今後の課題

まとめ:

- ESC/Java と Krakatoa で JavaCard の具体例を検証
- 形式的技法は安全性・信頼性を重視するシステムで必要

今後の課題:

- Krakatoa による mul メソッドの検証
- 定理証明系によるフィードバック
- 抽象化のレベルと信頼度のレベルの対応付け
 - ESC/Java と Krakatoa の中間?

Thank You!