JavaBIP meets VerCors

Towards the Safety of Concurrent Software Systems in Java

Simon Bliudze, Petra van den Bos, Marieke Huisman,
Robert[1] **Rubbens**, Larisa Safina

March 30th, 2023

[1] (Bob)

Designing concurrent systems

- Systems contain many interacting components
- Need to handle complexity

(c) Mohamed Mb, M. Tohirin, Marissa Coffey

Designing concurrent systems

- Systems contain many interacting components
- Need to handle complexity
- Model-based coordination framework: JavaBIP
- Separate interaction & implementation

(c) Mohamed Mb, M. Tohirin, Marissa Coffey

Designing concurrent systems

- Systems contain many interacting components
- Need to handle complexity
- Model-based coordination framework: JavaBIP
- Separate interaction & implementation
- Weakness: *assumptions are not checked*
- **Solution: combine JavaBIP & VerCors**
  - Using **contracts**
  - Deductive & runtime verification

(c) Mohamed Mb, M. Tohirin, Marissa Coffey

Outline

1. Model-based coordination framework: JavaBIP

2. Deductive verification: VerCors

3. JavaBIP + VerCors = Verified JavaBIP

4. Casino case study

Model-based coordination framework: JavaBIP

JavaBIP model: example

$$\begin{aligned}
\text{Model} &= \text{interacting components}\\
\text{interaction} &= \text{simultaneously execute transitions}\\
\text{component} &= \texttt{class}\\
\text{transition} &= \text{method} + \text{start \& end state}
\end{aligned}$$

JavaBIP model: example

Model = interacting components

interaction = simultaneously execute transitions

component = `class`

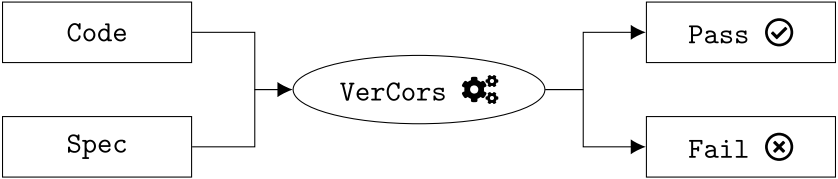
transition = method + start & end state

```java
@Component(initial=IDLE, name=DISPLAY_SPEC)
class CoffeeMachineDisplay {
  @Transition(
    name=SHOW_COFFEE_MSG,
    source=IDLE,
    target=SHOW_PROGRESS)
  void showCoffeeMessage() {
    System.out.println("Dispensing coffee");
  }
}
```

Deductive verification: VerCors

VerCors

- Auto-active deductive verifier
- Supports concurrent Java, C, PVL
- Contract specifications: pre- and postconditions

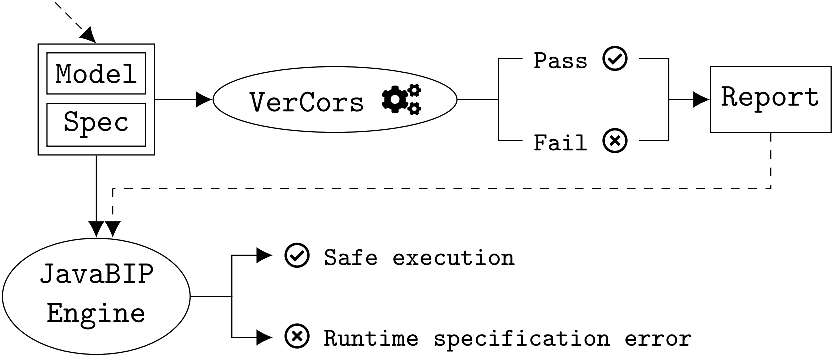


VerCors: contract example

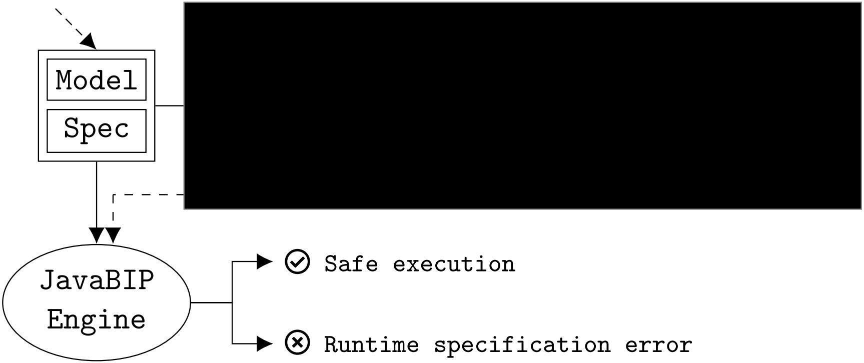
```
//@ requires 0 <= r && r <= 255;
//@ requires 0 <= g && g <= 255;
//@ requires 0 <= b && b <= 255;
//@ ensures 0 <= \result && \result <= 255;
int averagePixel(int r, int g, int b) {
  return (r + g + b) / 3;
}
```

JavaBIP + VerCors = Verified JavaBIP

Verified JavaBIP: workflow



Verified JavaBIP: workflow

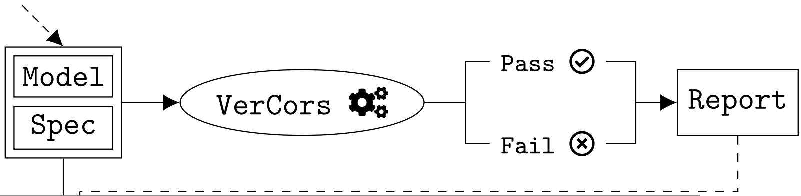


[redacted]

✓ Safe execution

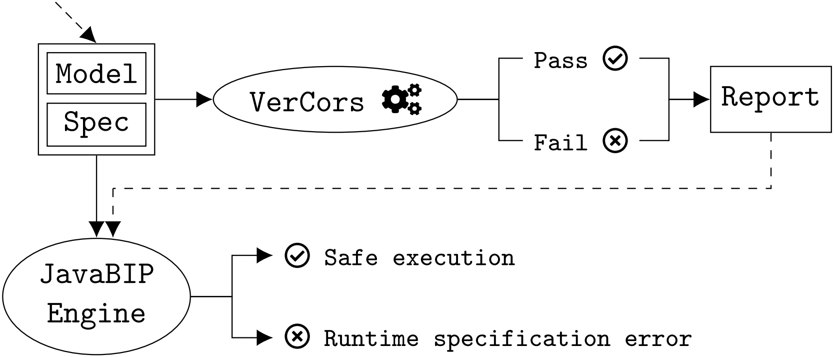
⊗ Runtime specification error

Verified JavaBIP: workflow



[redacted]

Verified JavaBIP: workflow



JavaBIP: original annotations

```
@Component(initial=IDLE, name=MYCOMPONENT_SPEC)
class MyComponent {
  @Transition(
    name=MY_TRANSITION,
    source=S,
    target=T)
  void myTransition() ...
```

Verified JavaBIP: extended with contracts

```
@Component(initial=IDLE, name=MYCOMPONENT_SPEC)
@StatePredicate(state=IDLE, expr="I") // <---
class MyComponent {
  @Transition(
    name=MY_TRANSITION,
    source=S,
    target=T,
    requires="P", // <--
    ensures="Q")  // <--
  void myTransition() ...
```

Verified JavaBIP: benefits

- Check model assumptions deductively
  - ⊕ Optimize runtime verification by reusing partial verification results

Verified JavaBIP: benefits

- Check model assumptions deductively
  - ⊕ Optimize runtime verification by reusing partial verification results
- Detect model assumption violations at runtime

Verified JavaBIP: benefits

- Check model assumptions deductively
  - ✚ Optimize runtime verification by reusing partial verification results
- Detect model assumption violations at runtime
  - ✚ Guarantee safety at runtime
  - ✚ Speed up prototyping of contracts

Casino case study

Casino case study

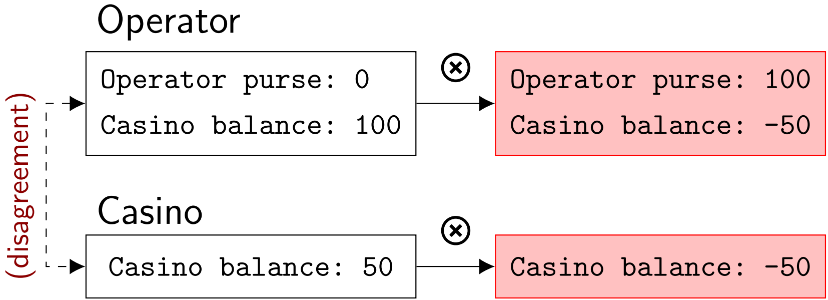
- Case study based on VerifyThis Long Term Challenge
- Original program: solidity casino smart contract
- Rewritten as JavaBIP model

(c) `https://verifythis.github.io`

Casino case study: general structure

- Casino:
  - Takes bets
  - Pays out on correct guesses
- Operator
  - Owns casino
  - adds/withdraws money from casino balance
- Player:
  - Uses casino
  - Place bets
  - Lose/win money

Casino case study: problem



(disagreement)

Operator

Operator purse: 0

Casino balance: 100

⊗

Operator purse: 100

Casino balance: -50

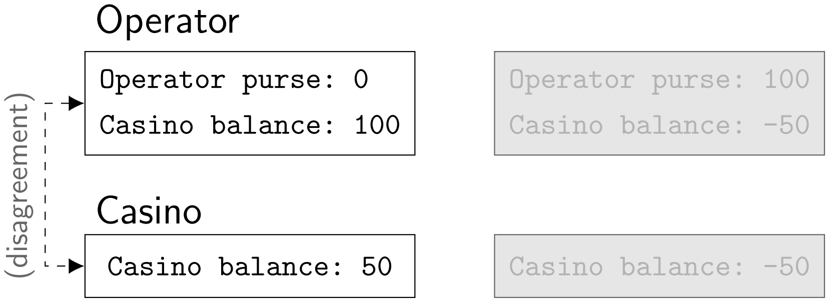
Casino

Casino balance: 50

⊗

Casino balance: -50

Casino case study: solution



Operator

Operator purse: 0

Casino balance: 100

Operator purse: 100

Casino balance: -50

(disagreement)

Casino

Casino balance: 50

Casino balance: -50

Conclusion

- Model-based coordination frameworks use unchecked assumptions
- Contracts facilitate combination of JavaBIP with VerCors to:
  - Verify JavaBIP models deductively
  - Check contracts at runtime
  - Optimize away runtime checks
- Casino case study to illustrate tool

Conclusion

- Model-based coordination frameworks use unchecked assumptions
- Contracts facilitate combination of JavaBIP with VerCors to:
  - Verify JavaBIP models deductively
  - Check contracts at runtime
  - Optimize away runtime checks
- Casino case study to illustrate tool

Paper: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java
DOI: `10.1007/978-3-031-30826-0_8`

Robert Rubbens
Formal Methods & Tools, University of Twente
r.b.rubbens@utwente.nl

Bonus slides

Interaction specification

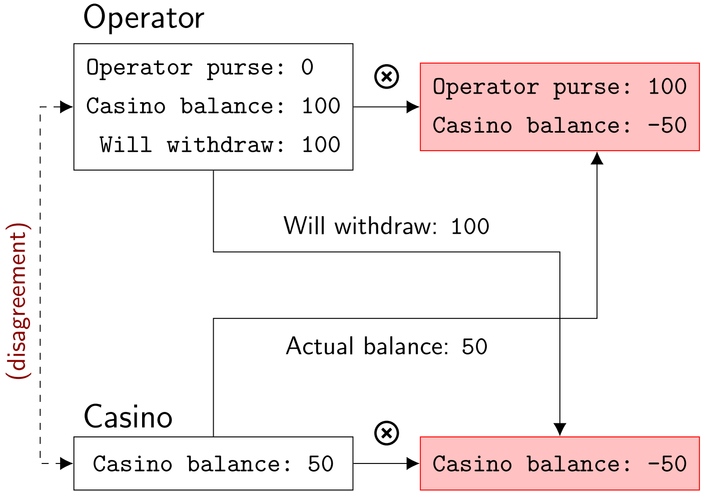
In JavaBIP initialization, likely in `main()`

```java
// Synchronize exclusively:
synchron(Casino.class, RECEIVE_BET)
  .to(Player.class, PLACE_BET);

// Requires any of:
port(Operator.class, DECIDE_BET)
  .requires(Casino.class, CASINO_WIN);
// Accepts only of:
port(Casino.class, CASINO_WIN)
  .accepts(Operator.class, DECIDE_BET);

// Data flow
data(Operator.class, OUTGOING_FUNDS)
  .to(Casino.class, INCOMING_FUNDS);
```

Casino case study: problem (+ data)



Operator

Operator purse: 0
Casino balance: 100
Will withdraw: 100

⊗

Operator purse: 100
Casino balance: -50

Will withdraw: 100

Actual balance: 50

(disagreement)

Casino

Casino balance: 50

⊗

Casino balance: -50

VerCors vs. JavaBIP

VerCors

Strong points:

- Analyze **data**
- **No assumptions**

Weak points:

- Only **local** analysis
- **No partial** analysis

JavaBIP

Strong points:

- Design **system-wide** behaviour
- **Partial** execution

Weak points:

- **Little data** reasoning
- **Assumptions**

Verified JavaBIP: implementation

- In VerCors:
  1. Parse Verified JavaBIP annotations
  2. Encode contracts using JavaBIP semantics into `COL`
  3. Verify `COL` program
  4. Translate back any errors to input
  5. Produce verification report
- In the JavaBIP engine:
  1. Parse Verified JavaBIP annotations
  2. If supplied, import verification report
  3. Runtime verification
     - Check non-verified properties at points of interest