

JavaBIP meets VerCors

Towards the Safety of Concurrent Software Systems in Java

Simon Bliudze, Petra van den Bos, Marieke Huisman,
Robert¹ Rubbens, Larisa Safina

March 30th, 2023

UNIVERSITY
OF TWENTE.



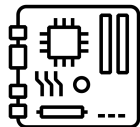
Inria

¹(Bob)

Designing concurrent systems



- Systems contain many interacting components
- Need to handle complexity



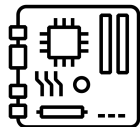
(c) Mohamed Mb, M.

Tohirin, Marissa Coffey

Designing concurrent systems



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

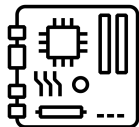


(c) Mohamed Mb, M.

Tohirin, Marissa Coffey



- 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**



(c) Mohamed Mb, M.

Tohirin, Marissa Coffey



- 1 Model-based coordination framework: JavaBIP
- 2 Deductive verification: VerCors
- 3 JavaBIP + VerCors = Verified JavaBIP
- 4 Casino case study

Model-based coordination framework: JavaBIP



- Component = state machine
- Implementation = states and transitions
- Interaction = synchronization of transitions



```
1  @Component(initial=IDLE, name=DISPLAY_SPEC)
2  class CoffeeMachineDisplay {
3      @Transition(
4          name=SHOW_COFFEE_MSG,
5          source=IDLE,
6          target=SHOW_PROGRESS)
7      void showCoffeeMessage() {
8          System.out.println("Dispensing coffee");
9      }
10 }
```



```
1  class CoffeeMachineDisplay {  
2      @Transition(name=SHOW_COFFEE_MSG, ...  
3  }
```



```
1 class CoffeeMachineDisplay {  
2     @Transition(name=SHOW_COFFEE_MSG, ...  
3 }
```

```
1 class CoffeeMachineProducer {  
2     @Transition(name=PRODUCE_COFFEE, ...  
3 }
```



```
1 class CoffeeMachineDisplay {  
2     @Transition(name=SHOW_COFFEE_MSG, ...  
3 }
```



(interaction specification requires synchronization)



```
1 class CoffeeMachineProducer {  
2     @Transition(name=PRODUCE_COFFEE, ...  
3 }
```

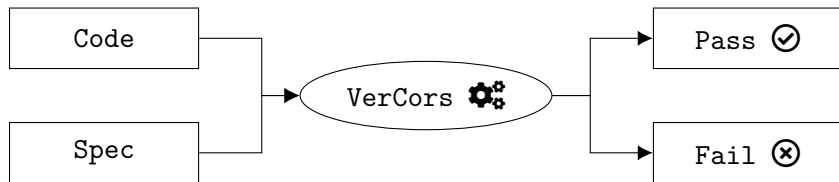
Deductive verification: VerCors



- Deductive verifier
- Java, C, PVL
- Focus on concurrency
- Contract specifications



- Deductive verifier
- Java, C, PVL
- Focus on concurrency
- Contract specifications





```
1  //@ requires 0 <= r && r <= 255;
2  //@ requires 0 <= g && g <= 255;
3  //@ requires 0 <= b && b <= 255;
4  //@ ensures 0 <= \result && \result <= 255;
5  int averagePixel(int r, int g, int b) {
6      return (r + g + b) / 3;
7  }
```



```
1  //@ requires 0 <= r && r <= 255;
2  //@ requires 0 <= g && g <= 255;
3  //@ requires 0 <= b && b <= 255;
4  //@ ensures 0 <= \result && \result <= 255;
5  int averagePixel(int r, int g, int b) {
6      return (r + g + b) / 3;
7  }
```

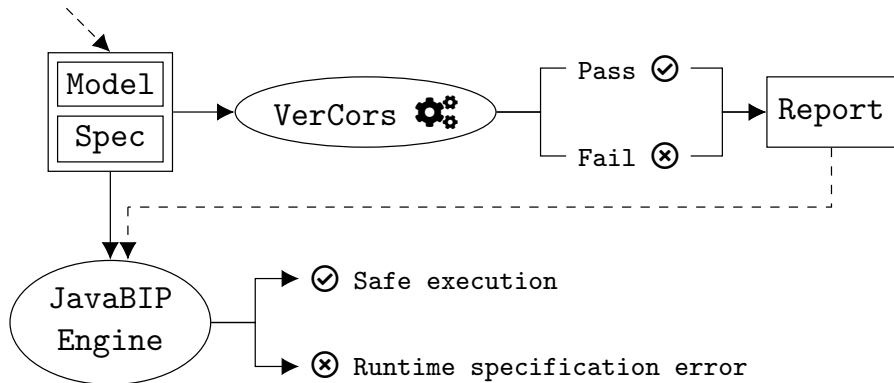
Other contract features:

- Permissions
- Lock invariants
- Recursive modeling functions
- Ghost state



JavaBIP + VerCors = Verified JavaBIP

Verified JavaBIP: workflow





```
1  @Component(initial=IDLE, name=MYCOMPONENT_SPEC)
2  class MyComponent {
3      @Transition(
4          name=MY_TRANSITION,
5          source=S,
6          target=T)
7      void myTransition() ...
```



```
1  @Component(initial=IDLE, name=MYCOMPONENT_SPEC)
2  @StatePredicate(state=IDLE, expr="I") // <---
3  class MyComponent {
4      @Transition(
5          name=MY_TRANSITION,
6          source=S,
7          target=T,
8          requires="P", // <--
9          ensures="Q") // <--
10 void myTransition() ...
```



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



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



- 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



■ 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 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 Evaluate non-verified properties at points of interest

Casino case study



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

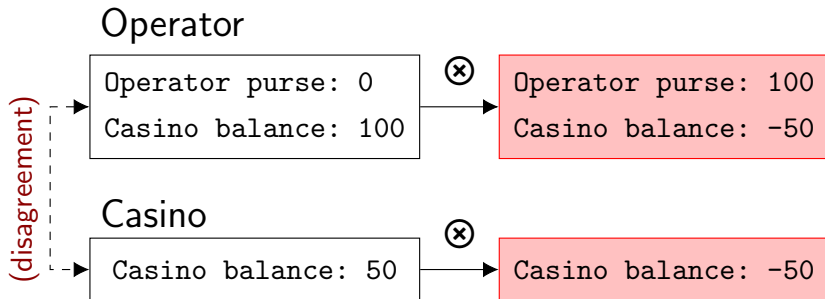


(c) <https://verifythis.github.io>

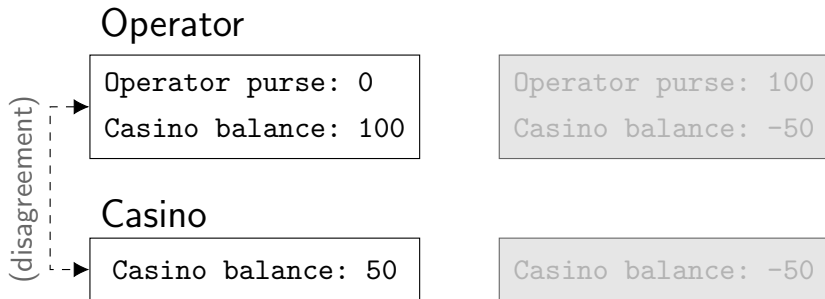


- Casino:
 - Takes bets
 - Pays out on correct guesses
- Operator
 - Owns casino
 - adds/withdraws money from casino balance
- Player:
 - Uses casino
 - Place bets
 - Lose/win money

Casino case study: problem



Casino case study: solution





- 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



- 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 (FASE 2023)

At: hal.inria.fr/hal-03911393

Robert Rubbens

Formal Methods & Tools, University of Twente

r.b.rubbens@utwente.nl



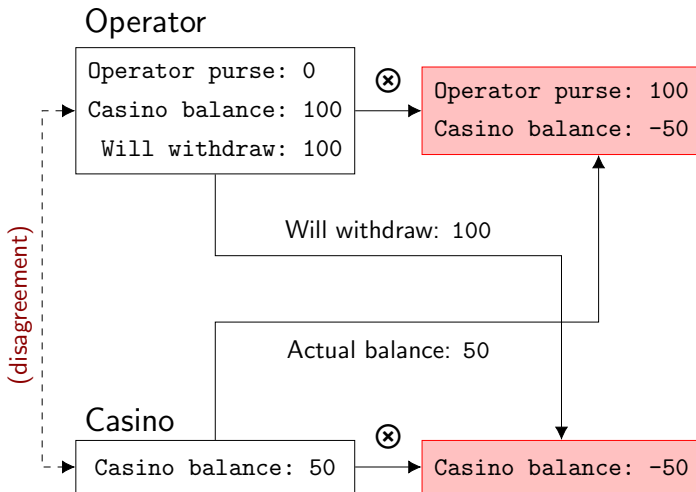
Bonus slides



In JavaBIP initialization, likely in main()

```
1  // Synchronize exclusively:
2  synchron(Casino.class, RECEIVE_BET)
3    .to(Player.class, PLACE_BET);
4
5  // Requires any of:
6  port(Operator.class, DECIDE_BET)
7    .requires(Casino.class, CASINO_WIN);
8  // Accepts only of:
9  port(Casino.class, CASINO_WIN)
10    .accepts(Operator.class, DECIDE_BET);
11
12 // Data flow
13 data(Operator.class, OUTGOING_FUNDS)
14    .to(Casino.class, INCOMING_FUNDS);
```

Casino case study: problem (+ data)





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**