Modeling and Verification of Component Behavior

Pavel Parízek

DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsrg.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics

Outline

- Behavior protocols
  - Modeling of component behavior
  - Verification of communication correctness
    - Behavior compliance

- My research
  - Behavior verification and analysis of Java implementation of primitive components

Behavior Protocols

- Formalism for modeling of component behavior
- Frame protocol (FP)
  - Valid sequences of events on component's interfaces
  - Component behavior specification
- Example
  - File Server

```
(?fs.listDirectory{!log.log*} + ?data.read{!log.log*})*
```

  - Controller

```
!player.play* | !player.pause* | !player.resume* |
!player.getStatus* | !fs.listDirectory*
```

Behavior Protocols – Syntax

- Atomic events:
  - Emitting a method invocation: `!interface.method↑`
  - Accepting a method invocation: `?interface.method↑`
  - Emitting a return from method: `!interface.method↓`
  - Accepting a return from method: `?interface.method↓`

- Operators:
  - Sequence: **;**
  - Alternative: **+**
  - Repetition: *****
  - And-parallel composition: **|**

- Syntactic abbreviations (method call, etc.)
  - `?i.m` = `?i.m↑ ; !i.m↓`
  - `?i.m{prot}` = `?i.m↑ ; prot ; !i.m↓`
  - Similarly for `!i.m` and `!i.m{prot}`

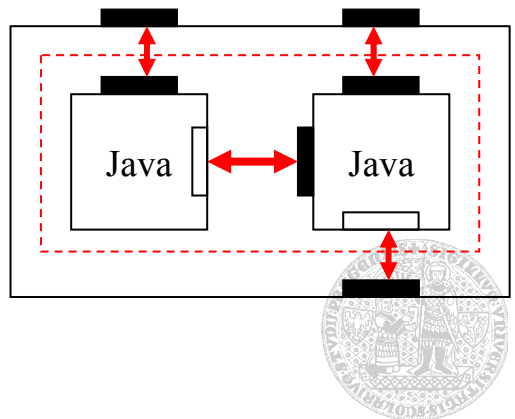
Behavior Compliance

- Basic idea
  - **Compliant** frame protocols model **error-free communication**
- Supported communication errors
  - Bad activity
    - Unexpected call ~ `!i.m` with no matching `?i.m`
    - Example

      `(!player.play* | !player.pause*) ; !player.getStatus`

      vs.

      `(?player.play* | ?player.pause*)`
  - No activity
    - Deadlock ~ all components wait for a call (`?i.m`)
- Application
  - For a composite component, checking whether
    - Sub-components communicate with no errors
    - Cooperating sub-components do what the parent component expects
  - Tool: **Behavior Protocol Checker (BPC)**
    - Based on model checking (state space traversal)



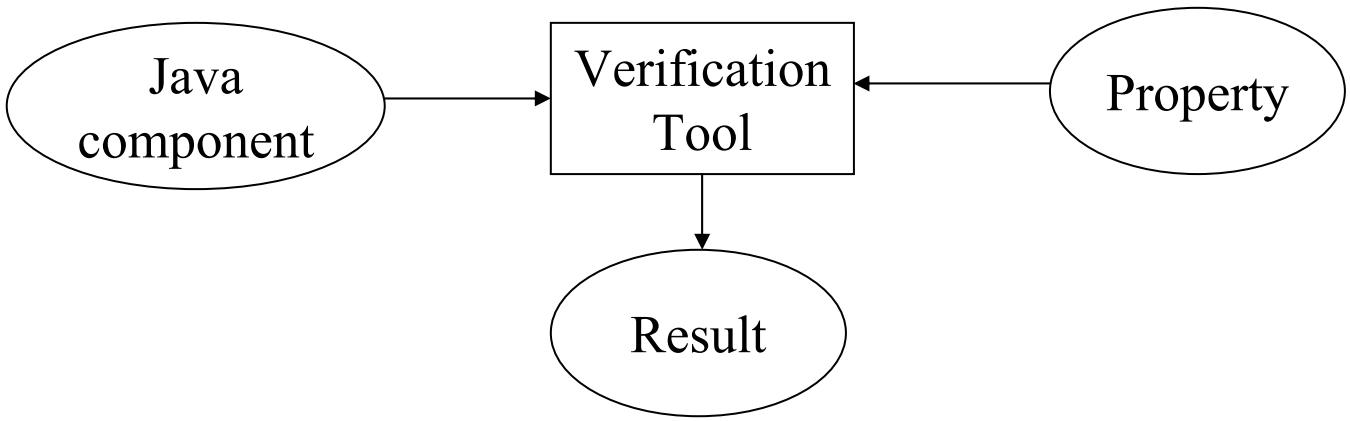
Outline

- Behavior protocols
  - Modeling of component behavior
  - Verification of communication correctness
    - Behavior compliance

- Verification and analysis of Java implementation of primitive components
  - Checking Java code against specific properties

Verification of Primitive Components

- Purpose
  - To find whether Java code of a primitive component satisfies all required properties



- Supported properties
  - Correspondence with behavior specification
    - Obeying of a frame protocol
  - Absence of concurrency errors
    - Deadlocks, race conditions

Obeying of Frame Protocol

- Formal definition
  - Primitive component has to be able to accept/issue those method call-related event sequences on its frame that are specified by its frame protocol

- Example

  Frame protocol: `?fs.listDirectory{!log.log*}`

Correct

```
public void listDirectory {
   ...
   log.log("hello");
   ...
   log.log("world");
   ...
}
```

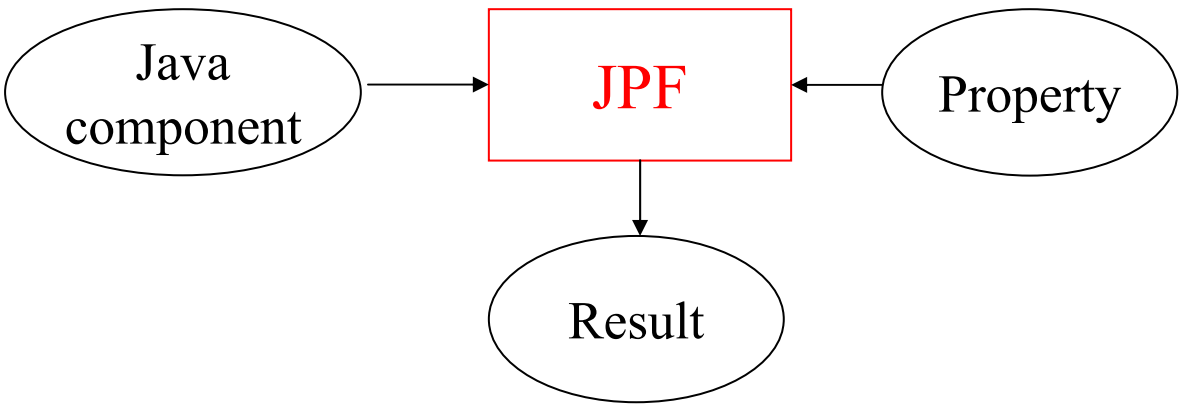
Wrong

```
public void listDirectory {
   ...
   log.log("hello");
   ...
   log.log("world");
   ...
   log.flush();
}
```

Java PathFinder

- Question
  - **How to check Java code against the properties ?**

- Our approach
  - Use of Java PathFinder (JPF) as the core verification tool
    - State-of-the-art model checker for Java bytecode programs
    - Highly extensible and customizable
    - Provides API for monitoring of state space traversal



Result = YES / NO+counterexample

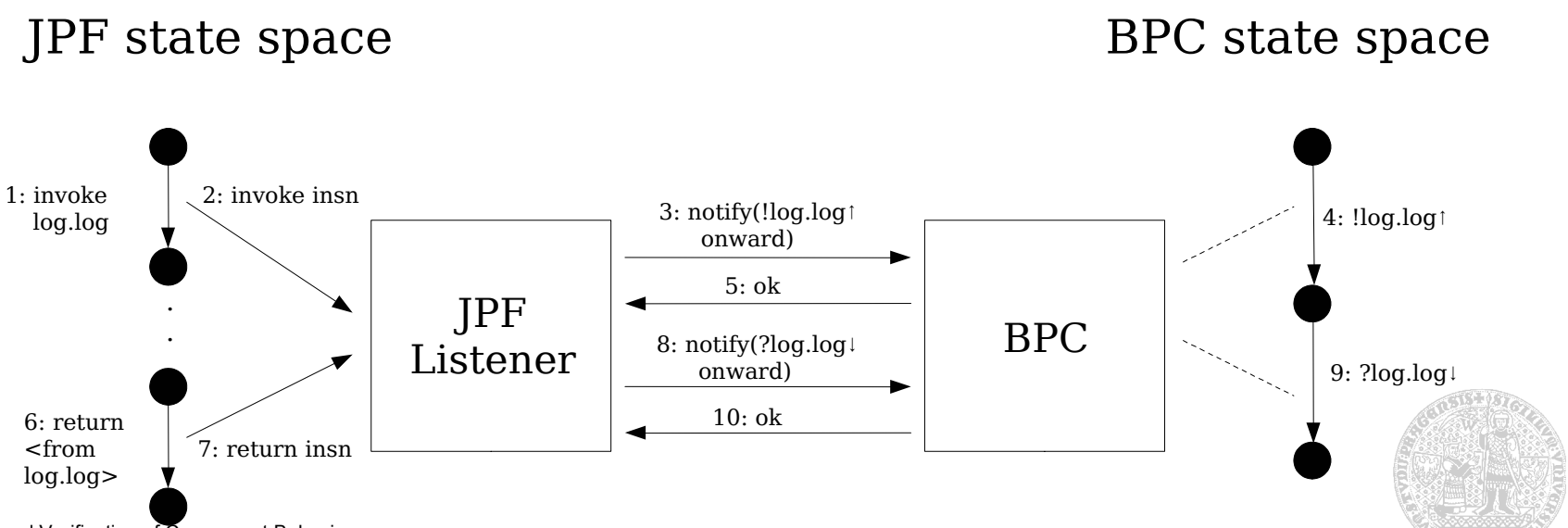
Checking Primitive Components with JPF - Issues

- No support for the property ‘obeying of frame protocol’ in JPF
- Problem of missing environment
- State explosion

Checking the Property of Obeying with JPF

- Problem
  - JPF supports only low-level properties by default
    - Absence of deadlocks, assertion violations
  - Obeying of frame protocol is a high-level property

- Our solution
  - Combination of JPF with the Behavior Protocol Checker (BPC)
    - Coordination of state space traversal
    - Mapping between state spaces

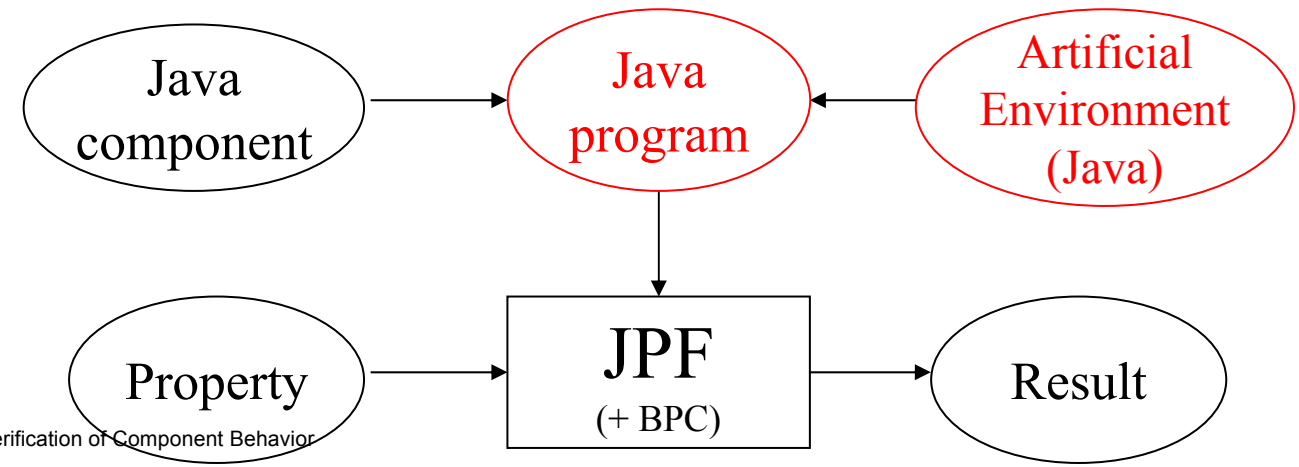


Outline

- Behavior protocols
  - Modeling of component behavior
  - Verification of communication correctness
    - Behavior compliance
- Verification and analysis of Java implementation of primitive components
  - Checking Java code against specific properties
    - Correspondence of implementation with specification
    - Absence of concurrency errors
  - Issues
    - **Problem of missing environment**
    - State explosion

Problem of Missing Environment

- Problem
  - JPF works only for complete Java programs (with `main`)
    - Isolated primitive component is not such a program
- Our solution
  - Construction of artificial environment for an isolated component
    - Component + environment = complete Java program
  - Artificial environment
    - Set of Java classes that simulate behavior of other components



Artificial Environment – Java Code

```
public class EnvPlayThread
    extends Thread {
  Player pl;
  ...

  public void run() {
    while (true) {
      pl.play();
    }
    ...
  }
}

public class EnvPauseThread {}
public class EnvResumeThread {}
...
```

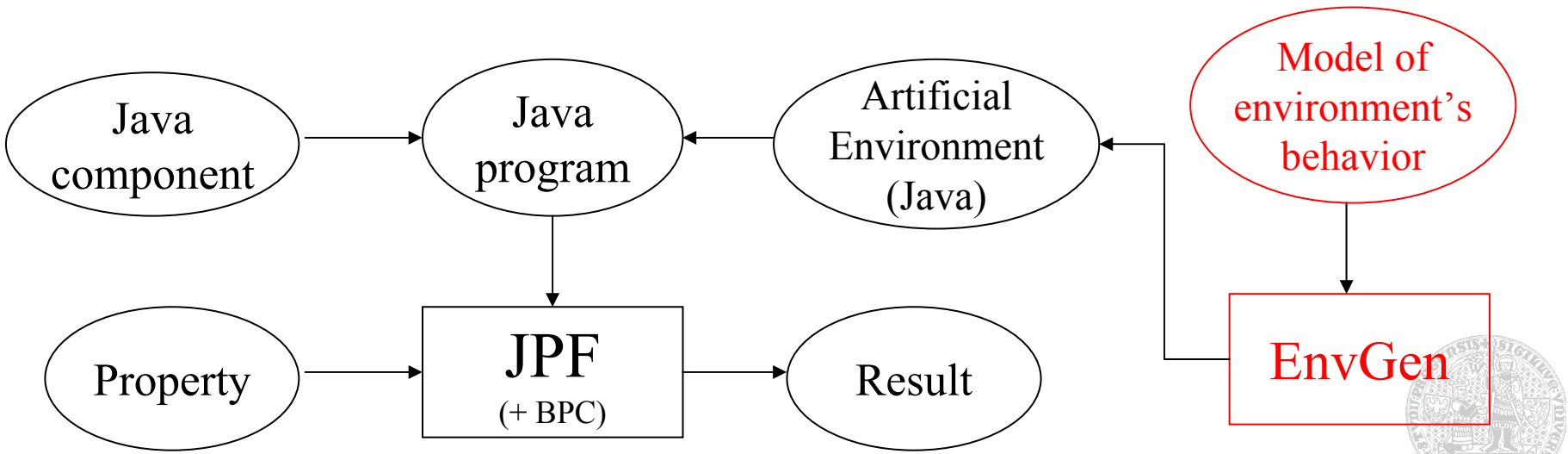
```
public static void main(String[]) {
  Player pl = new PlayerImpl();

  Thread th1 =
    new EnvPlayThread(pl).start();
  Thread th2 =
    new EnvPauseThread(pl).start();
  ...

  // wait for threads to finish
  th1.join();
  th2.join();
  ...
}
```

Automated Generation of Environment

- Environment Generator for JPF (**EnvGen**)
  - Input
    - Model of the environment's behavior
    - Sets of method parameter values
  - Output
    - Stub implementations of the required interfaces
    - Driver program that calls methods of the component according to the model



Model of Environment's Behavior

- Purpose
  - Modeling interaction of a component with a particular real environment (e.g. rest of an application)
    - What component's methods should the environment call (and when)
    - What calls from the component have to be accepted by the environment (and when)

- Definition
  - Environment for a component C can be seen as a component E
  - E has a frame protocol FP_E ➔ environment protocol of C (EP_C)
    - EP_C has to be compliant with FP_C

- Example
  - $FP_{Controller}$ = `!player.play* | !player.pause* | ...`
  - $EP_{Controller}$ can be `?player.play* | ?player.pause* | ...`

Specific Environment Protocols

- Inverted frame protocol (EP^{inv})
  - Models an environment that exercises the component in all possible ways it was designed for
    - JPF checking prone to state explosion
  - Constructed from FP by replacing `?i.m` with `!i.m` and vice versa
  - Example

```
!player.play* | !player.pause* | !player.resume* |
!player.getStatus* | !fs.listDirectory*
```

- Context protocol (EP^{ctx})
  - Models use of the component in a particular application
    - Typically simpler than the inverted frame protocol
  - Construction based on state space traversal ➔ prone to state explosion
  - Example

```
(!player.play ; (!player.pause ; !player.resume)*)* |
!player.getStatus* | !fs.listDirectory*
```

- Calling & trigger protocol-based (EP^{trig})

Calling & Trigger Protocol

EP^{trig} = **<calling & trigger protocol> | ?m1* | … | ?mN***

Precise interleaving of events on component's provided intefaces and triggers for callbacks

Events on required interfaces, except triggers of callbacks

- Models use of the component in a particular application
  - Typically simpler than the inverted frame protocol
- Constructed via syntactical expansion and substitution of fragments of frame protocols of other components

	EP^{inv}	EP^{ctx}	EP^{trig}
Construction	OK	State expl.	OK
JPF checking	State expl.	OK	OK

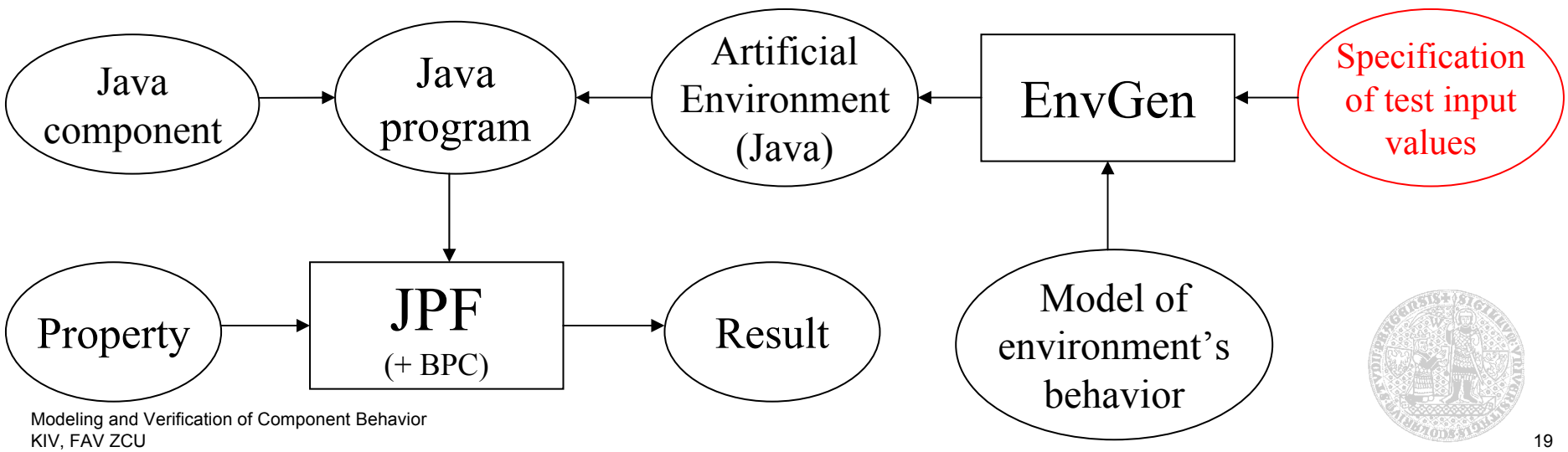
→ **Model of choice**

Sets of Method Parameter Values

- Provided by the user in a special Java class ("container")
  - Automated derivation is subject to future work

- Example:

```
putStringSet("Player", "", "play",
                new String[]{"movie1.avi", "song2.mp3"});
putStringSet("FileServer", "FileSystem", "listDirectory",
                new String[]{"movie1.avi", "song2.mp3"});
putIntSet("FileServer", "DataAccess", "read",
                                    new int[]{0, 1024});
```



Outline

- Behavior protocols
  - Modeling of component behavior
  - Verification of communication correctness
    - Behavior compliance
- My research: verification and analysis of Java implementation of primitive components
  - Checking Java code against specific properties
    - Correspondence of implementation with specification
    - Absence of concurrency errors
  - Issues
    - Problem of missing environment
    - **State explosion**

State Explosion

- Problem
  - Verification of Java code of primitive components with JPF is prone to state explosion
    - For complex components or components with complex environment
- Caused by
  - **High number of threads**
    - in a Java program (component + artificial environment)
  - Big data domains
    - Not a problem in our case ➔ small sets of method parameter values are typically specified by the user
- Consequence: **JPF runs out of memory**
  - Failure to check whether all properties are satisfied by Java code

Addressing State Explosion

- Common techniques
  - Abstraction, reduction of state space size
    - Goal: verification of correctness
  - Heuristics for state space traversal
    - Goal: discovery of errors in limited resources

- Our approach: **heuristic reduction of the level of parallelism in environment**
  - Goal: discovery of concurrency errors in limited time and memory
    - Other kinds of properties are subject to future work
  - Specific techniques
    - Reduction of parallelism in environment protocol
    - Construction of reasonable artificial environment

Reducing Parallelism in Environment Protocol

- Identification of methods whose parallel execution would likely cause concurrency errors
  - via static analysis of Java byte code
    - Search for suspicious concurrency patterns
  - Example: **access to a variable guarded by locks with different types**

```
X x;                          Y y;
synchronized (x) {            synchronized (y) {
  this.attr = ..                .. = this.attr;
}                             }
```

- Reducing the level of parallelism in environment protocol
  - Parallel composition is preserved between method calls that feature potential errors

    `p1 | p2 | p3 | p4` ➔ `p2 ; p4 ; (p1 | p3)`

Construction of Reasonable Environment

- Degree of mutual interaction among methods via concurrency-related constructs of Java is measured
  - via a software metric and static analysis

```
m1() {
  id = 1;
}
```

```
m2() {
  print(id);
}
```

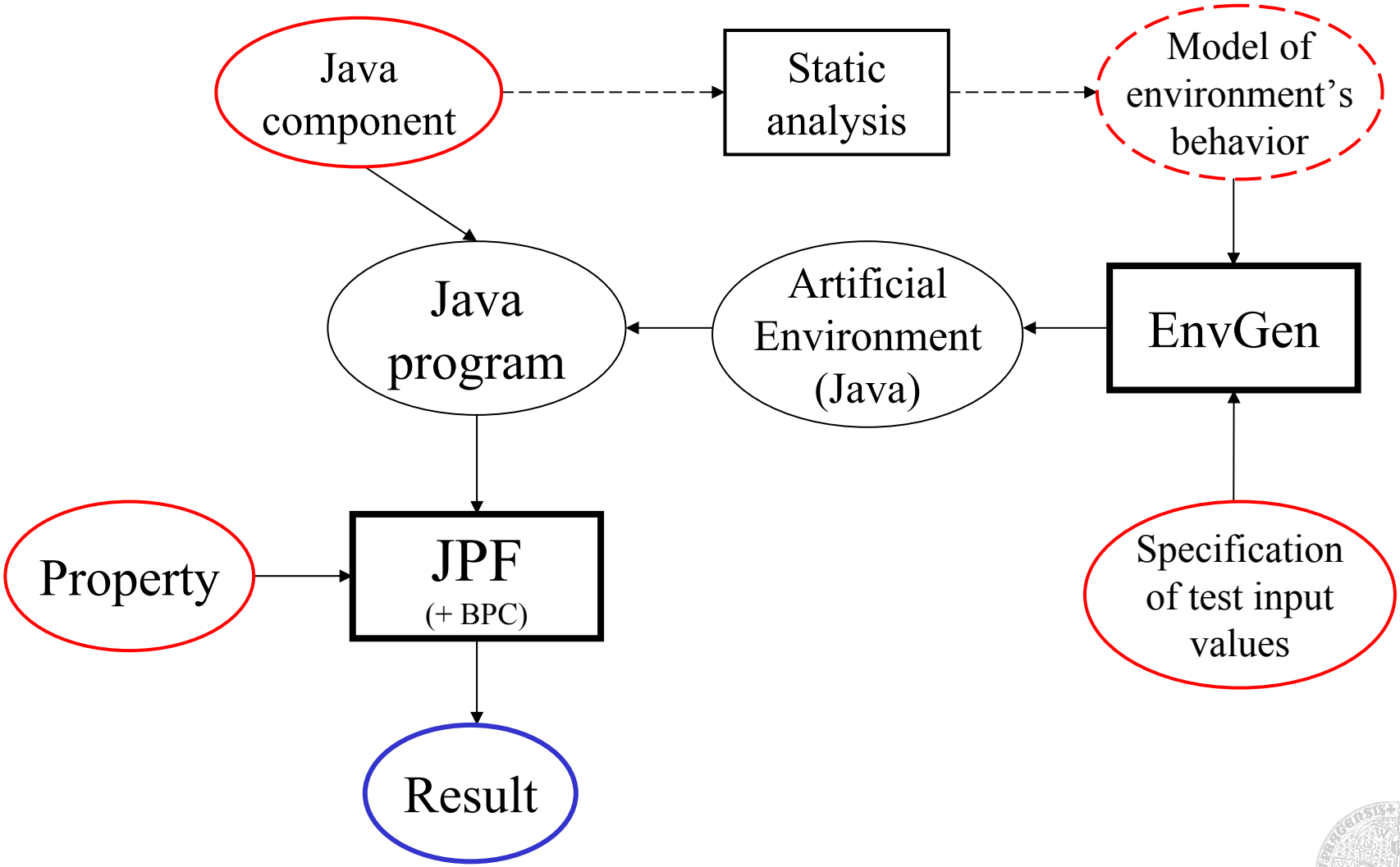
```
m3() {
  name = "abc";
  i = count;
}
```

```
m4() {
  print(name);
  count++;
}
```

- Environment protocol is constructed
  - Parallel execution specified only for methods that feature concurrent interaction
    - No concurrent interaction ➔ no chance of concurrency error
  - Methods sets are ordered according to the degree of interaction
    - High interaction makes concurrency errors more likely
    - Environment generator and JPF respect the order defined in EP

EP = **(!m3 | !m4) + (!m1 | !m2)**

Whole Picture



Future Work

- Checking of Java code against frame protocol
  - Addressing state explosion in JPF checking
    - Probably involving static program analysis
  - Support for thread behavior protocols (TBP)
    - Explicit threads
    - Java-like synchronization
    - State variables
- Automated derivation of method parameter values
  - Options: static analysis, symbolic execution
- Construction of reasonable environment for other kinds of errors and properties
  - Assertion violations, obeying of frame protocol, ...