Verification of Software Components: Checking Implementation (Java) against Specification (Behavior Protocols)

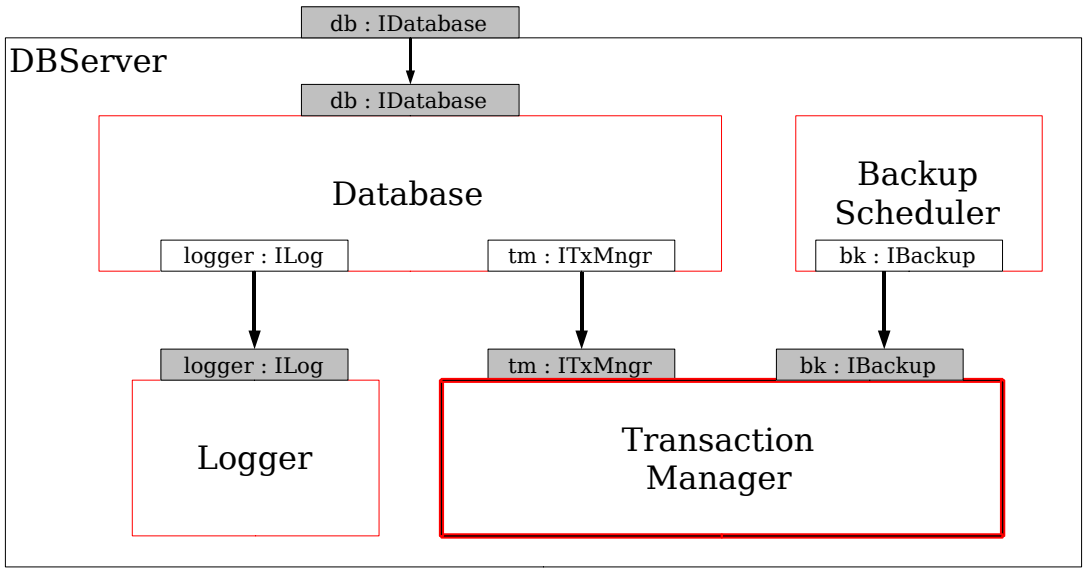
Pavel Parízek

DISTRIBUTED SYSTEMS RESEARCH GROUP
http://nenya.ms.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics

Behavior Compliance

- Works fine
  - Assumption: each primitive component (Java code) is compliant with its frame protocol
    - It has to accept/issue exactly those method call related event sequences on its frame that are specified by the frame protocol



Verification of Primitive Components

- Goal
  - Design an algorithm and a tool for checking compliance between Java implementation of a primitive component and its frame protocol
    - via Java source code / byte code analysis

- Previous experience
  - Checking of compliance between behavior protocols
    - Behavior protocol checker (BPC)

Checking of Java Code

- Java PathFinder
  - Model checker for Java programs
  - Highly customizable and extensible

- Features
  - Able to check only low-level properties
    - Deadlocks, uncaught exceptions
  - Non-deterministic value choice
  - Publisher/listener pattern
    - Listeners can watch the course of the state space traversal and check for specific properties

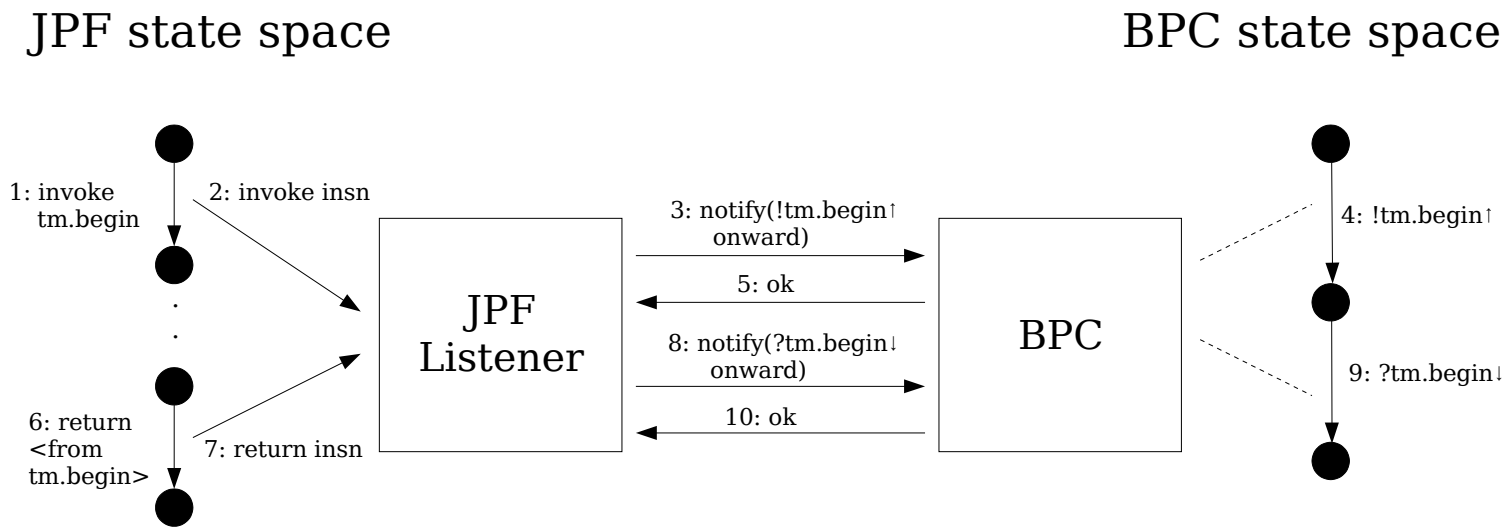
Our Solution

- Checking by JPF is not directly possible
  - JPF is able to check only low-level properties
    - Compliance between Java code and frame protocol is a high-level property
    - Extension of JPF is necessary
  - JPF accepts only complete Java programs
    - Isolated primitive component is not such a program ➔ problem of missing environment

- Basic idea
  - Cooperation of the Java PathFinder (JPF) model checker with the Behavior Protocol Checker (BPC)
  - Automated generation of component environment

Cooperation of JPF and BPC

- Both model checkers cooperate during traversal of their own state spaces
  - JPF listener communicates with BPC



- Coordination of backtracking
  - JPF is allowed to backtrack if and only if BPC agrees
  - Both checkers must be in an already visited state or in an end state for backtracking to take place

Cooperation of JPF and BPC – cont.

- Issue
  - JPF and BPC work on different levels of abstraction
    - JPF at the level of byte code instructions
    - BPC at the level of behavior protocols

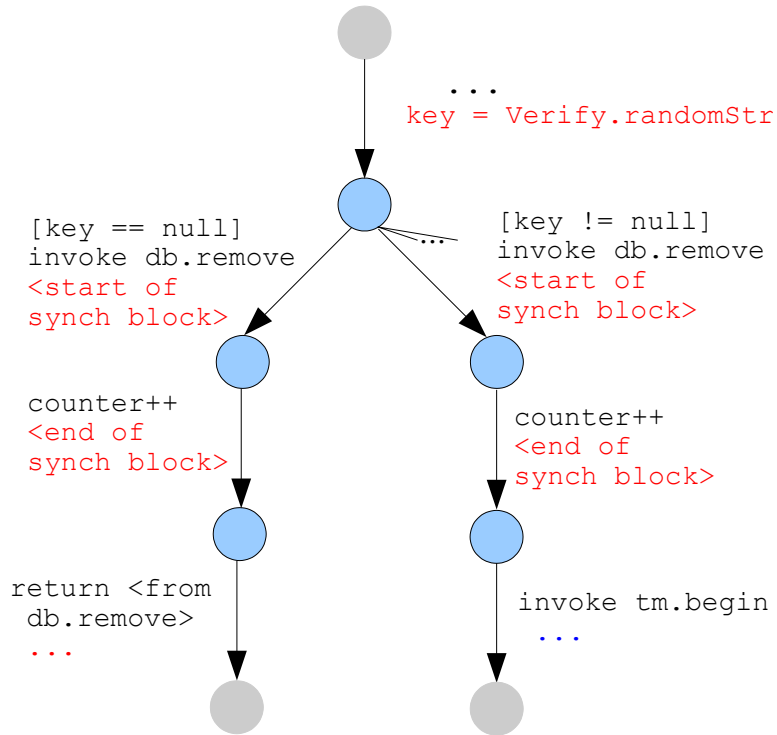
- We need to have a mapping from the JPF state space into the BPC state space
  - Program code ➔ behavior protocols

Mapping between Checkers' State Spaces

- Unique association of frame call points in Java byte code with JPF states
  - Frame call point
    - Invoke or return byte code instruction related to a method of a provided or required interface
  - Consequence: no JPF state is associated with two or more frame call points
    - (by definition, each BPC state is uniquely associated with a frame call point)
- Correspondence between end states in both checkers' state spaces
  - An end state in the program code state space (JPF) has to be mapped to an end state in the behavior protocol state space (BPC)

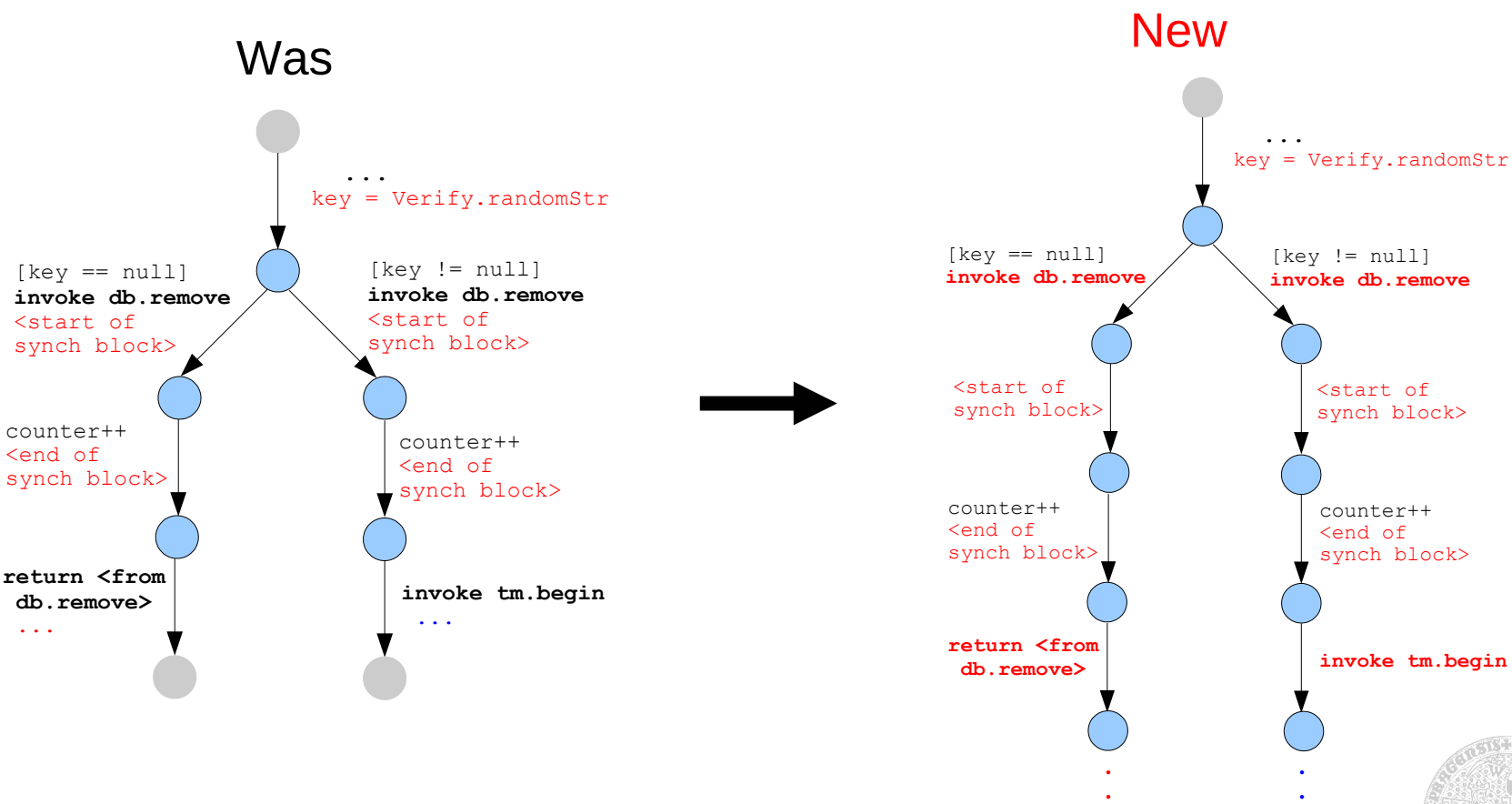
From Java Program Code to JPF State Space

```
public class DatabaseImpl
    implements IDatabase
{
  private ITxMngr tm;

  public void remove(String key)
  {
    synchronized (this) {counter++;}
    if (key == null) return;
    tm.begin();
    ...
  }
}

public static void main(...) {
  IDatabase db = new DatabaseImpl();

  String key = Verify.randomStr();
  db.remove(key);
  ...
}
```



Frame Call Points in JPF State Space

- Frame call point
  - Triggers a JPF transition

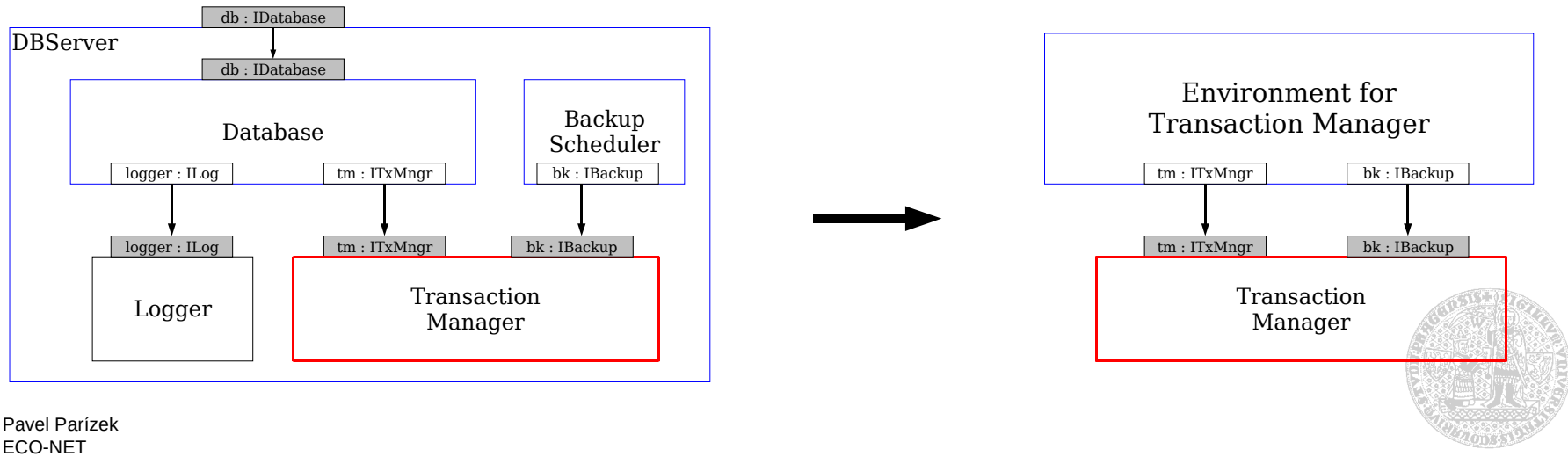


Cooperation between JPF and BPC: Summary

- Goal
  - **Checking compliance between Java implementation of a primitive component and its frame protocol**
- Solution
  - **Cooperation between JPF and BPC**
    - Mapping from program code state space (in JPF) to behavior protocol state space (in BPC)
- Remaining issue
  - **Problem of missing environment**

Problem of Missing Environment

- Problem
  - JPF checks only complete Java programs, but an isolated primitive component is not such a program
- Solution
  - Construction of an environment for a primitive component
    - Set of Java classes (one of them featuring the `main` method)
    - Simulating the behavior of other components in a given architecture
- Component + environment = complete Java program



Requirements for an Environment

- Completeness
  - Representation of behavior of all the other components bound to the target component
  - Exercise all the control flow paths in the target component's code
    - Sequences and parallel interleavings of method calls
    - Different combinations of method parameter values
- Feasibility
  - Reasonable state space size of the Java program composed of the component and its environment

“Ideal” Environment for Transaction Manager

- Satisfies all requirements on the environment
  - Calling methods of the component in parallel
  - Checking all control flow paths (different combinations of parameter values)

```
public class EnvDbThread
   extends Thread {
 ITxMngr tm;
 ...

 public void run() {
  String id = tm.begin(rndStr());
  if (rndBool()) tm.commit(id);
  else tm.abort(id);
  ...
 }
}

public class EnvBkThread
   extends Thread {
 ...
}
```

```
public static void main(String[]) {
 TransactionMngrImpl tm =
   new TransactionMngrImpl();

 tm.start();

 new EnvDbThread(tm).start();
 new EnvDbThread(tm).start();
 new EnvBkThread(tm).start();

 // wait for threads to finish

 tm.stop();
}
```

Construction of the Environment

- We aim at automated environment generation
  - Better than manual construction
    - Hard and tedious work ➔ not a way to go
- Environment Generator for JPF
  - Input
    - Model of the environment's behavior
    - Definition of component's provided and required interfaces
    - Sets of possible method parameter values and return values
  - Output
    - Stub implementations of the required interfaces
    - Driver program that calls methods of the provided interfaces according to the model of environment's behavior

Modeling the Environment's Behavior

- The model should force the environment
  - To call a certain method of a particular provided interface at the moment the component expects it
  - To accept a certain method call issued on a particular required interface at the moment the component "wishes" to do so
- Idea
  - Exploit model of component's behavior (i.e. frame protocol)
  - Use a behavior protocol as a model of environment

- Options
  - Inverted frame protocol
    - Constructed from the frame protocol by replacing all the accept events with emit events and vice versa
    - Models the most general valid environment
  - Context protocol
    - Specifies actual use of the target component by the other components in the particular hierarchical architecture
    - Models the simplest valid environment

Context Protocol vs. Inverted Frame Protocol

- Context protocol is more suitable
  - Component application typically exploits a subset of functionality provided by the target component
  - Completeness and feasibility at the same time
- Example (for Transaction Manager)
  - Inverted frame protocol

```
( !tm.start ; !tm.begin* ; ( !tm.begin* | !tm.commit* | !tm.abort* ) ; !tm.stop ) | !bk.backup*
```

  - Context protocol

```
( !tm.start ; ( !tm.begin ; (!tm.commit + !tm.abort))* ; !tm.stop ) | !bk.backup*
```

Calling Protocol

- Problems
  - Computing the context protocol could be very time consuming task for some inputs
  - No Java construct for acceptance of a method call depending upon execution history
    - Accept events not directly reflected in the environment's code

- Solution: **calling protocol**
  - Should be as close to the context protocol as possible to allow for feasible verification
  - Computed efficiently via syntactical expansion of frame protocols of other components in a particular hierarchical architecture
  - Only calls on the component's provided interfaces are modeled
    - Accept events are reflected implicitly in the environment's code

Modeling Environment Behavior: Evaluation

- Assuming
  - Environment's behavior is modeled via calling protocol
- Then
  - Component environment can be used for checking of arbitrary properties of primitive components
    - i.e. not only for compliance of Java code to a frame protocol
  - Verification has to be done for each architecture the component is used in
    - Calling protocol is specific to a particular hierarchical component architecture (i.e. depends on the context)

Environment and Feasibility of Code Checking

- Issue
  - Use of environment modeled by the calling protocol can still lead to state explosion
- Our approach (current research)
  - Reducing complexity of the environment
    - Heuristic transformations of the calling protocol
    - Static analysis of Java byte code

Reduction of Environment's Complexity

- State explosion
  - Caused mainly by parallelism and repetition
- Reduction of level of parallelism
  - Some parallel operators are replaced be sequencing
    - e.g. $p_1 \mid p_2 \mid p_3 \mid p_4 \rightarrow p_2;p_4;(p_1 \mid p_3)$
  - via static code analysis and concurrency metrics
- Reduction of repetition
  - Repetition operator is replaced by a sequence
    - e.g. `prot*` → `NULL + ( prot ; prot )`

Conclusion

- Goal
  - Checking compliance between Java code of a primitive component and its frame protocol
- Our approach
  - Cooperation of the Java PathFinder with Behavior Protocol Checker
    - Issue: mapping between state spaces
  - Automated environment generation
    - Issue: modeling of environment's behavior
  - Heuristic reduction of environment's complexity
    - Current research