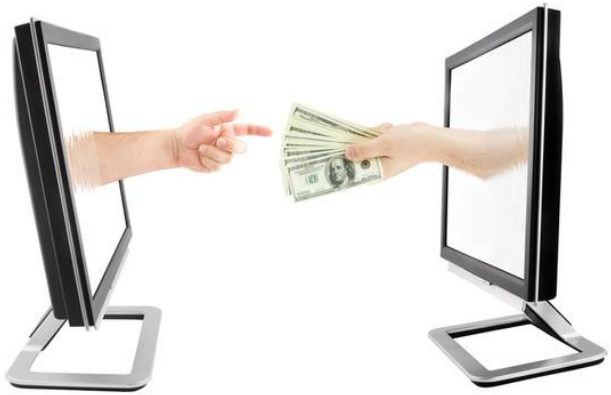
Asserting and Checking Determinism for Multithreaded Programs

Jacob Burnim & Koushik Sen, ESEC-FSE 2009

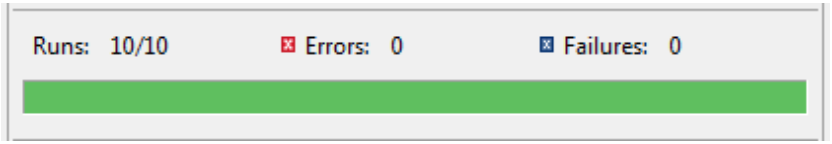
Multithreaded Programs



```
transfer (amount, account_from, account_to){
      if (account_from.balance < amount)
                  return NOPE;
      account_to.balance += amount;
      account_from.balance -= amount;
      return YEP;
}
```

Multithreaded Programs

```
transfer (amount, account_from, account_to) {
        if (account_from.balance < amount)
                return NOPE;
        account_to.balance += amount;
        account_from.balance -= amount;
        return YEP;
}
```



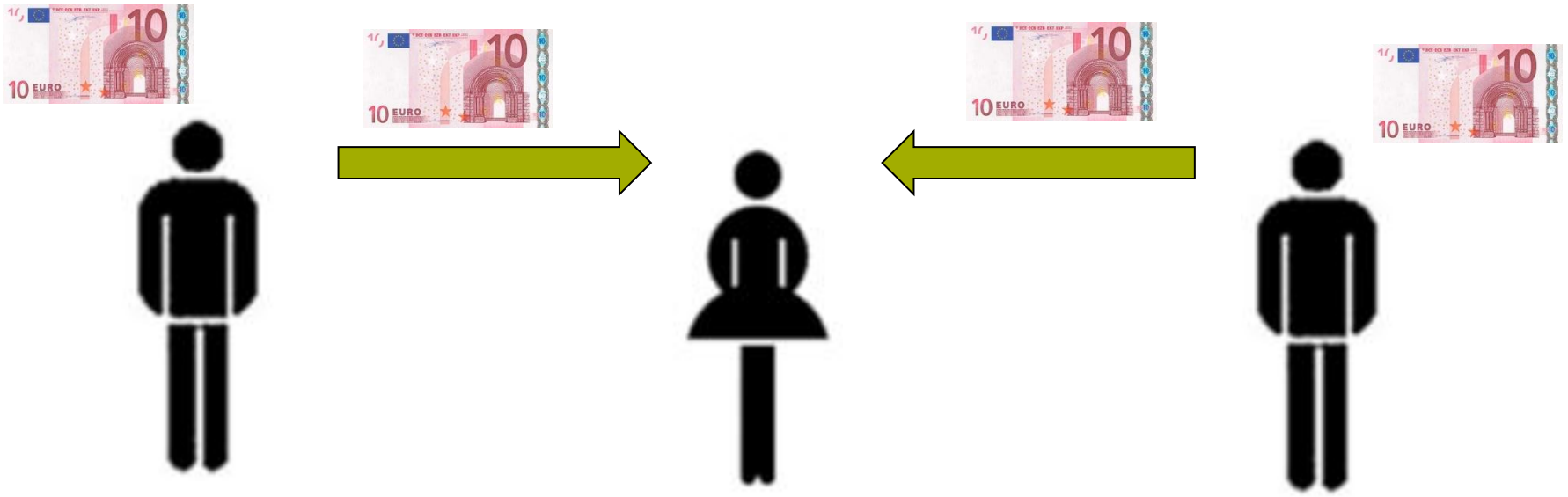
JUnit

- transferTest1()
- transferTest2()
- transferTest3()
- transferTest4()
- …
- transferTest10()

Multithreaded Programs

Multithreaded Programs

Multithreaded Programs



Multithreaded Programs

Multithreaded Programs

```
transfer1 (amount, account_from, account_to){
     if (account_from.balance < amount)
          return NOPE;
     account_to.balance += amount;
     account_from.balance -= amount;
     return YEP;
}
```

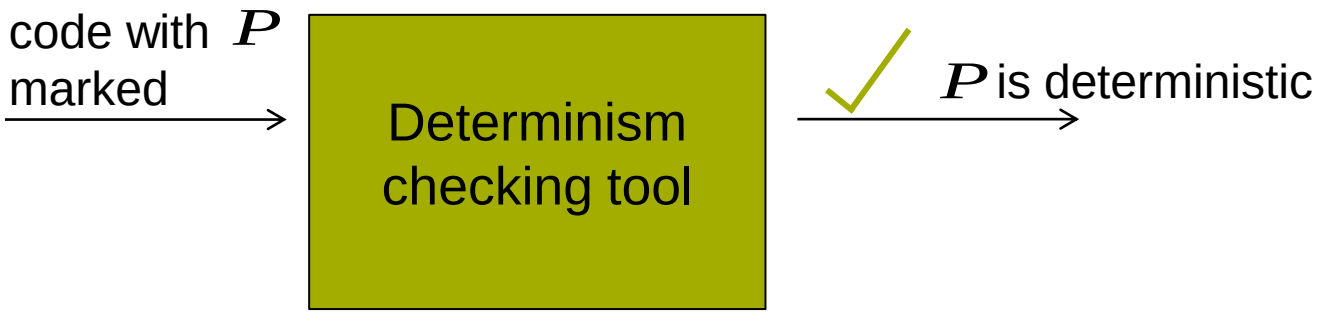
Problems in Concurrent Programs

- Difficult to notice concurrency errors
- Some thread interleavings may produce correct results, while others do not.

Key problem: non-determinism
(from a particular initial state there are executions of the code that produce different final states)

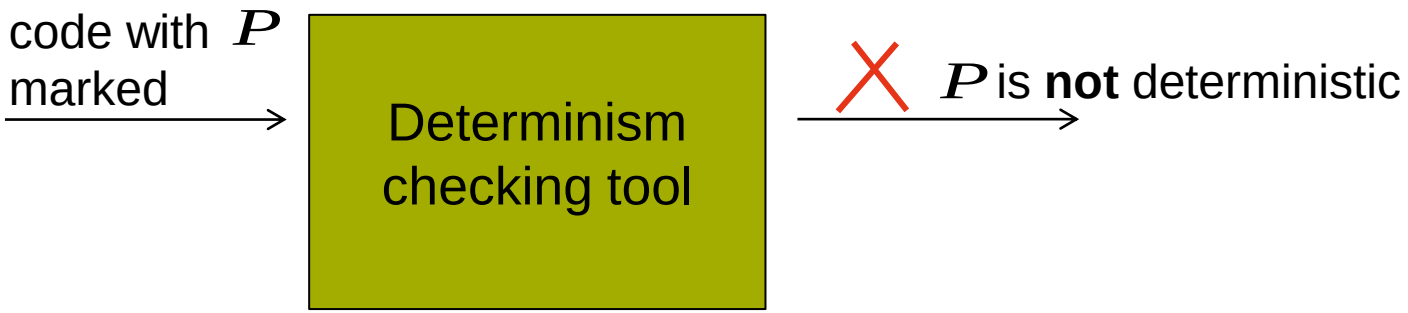
Idea: only test determinism

- Program block P is supposed to be deterministic
- If P really is deterministic:



Idea: only test determinism

- Program block P is supposed to be deterministic
- If P is **not** deterministic:



Determinism Checking Framework (DCF)

- Program block P is supposed to be deterministic

mark *P*:

```
main(){
    ...
    deterministic {
        P
    }
    ...
}
```

- DCF never answers 'P is determnistic'

Determinism Checking Framework (DCF)

- Program block P is supposed to be deterministic

mark *P*:

```
main(){
    ...
    deterministic {
        P
    }
    ...
}
```

Possible output:

Execution 1: ok (so far)

- DCF never answers 'P is determnistic'

Determinism Checking Framework (DCF)

- Program block P is supposed to be deterministic

mark *P*:

```
main(){
    ...
    deterministic {
        P
    }
    ...
}
```

Possible output:

Execution 1: ok (so far)
Execution 2: ok (so far)

- DCF never answers ‘P is determnistic‘

Determinism Checking Framework (DCF)

- Program block P is supposed to be deterministic

mark *P*:

```
main(){
    ...
    deterministic {
        P
    }
    ...
}
```

Possible output:

Execution 1: ok (so far)
Execution 2: ok (so far)
Execution 3: ok (so far)

- DCF never answers 'P is determnistic'

Determinism Checking Framework (DCF)

- Program block P is supposed to be deterministic

mark *P*:

```
main(){
    ...
    deterministic {
        P
    }
    ...
}
```

Possible output:

Execution 1: ok (so far)
Execution 2: ok (so far)
Execution 3: ok (so far)
Execution 4: ✗ not deterministic
determinism violation noticed!

- DCF never answers 'P is determnistic'

Deterministic Specification

Give programers possibility to say:

This result is supposed to be deterministic!

```
...
deterministic {
    P
}
...
```

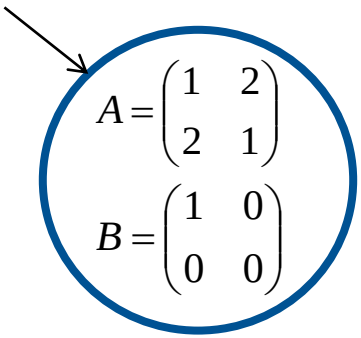
Deterministic Specification

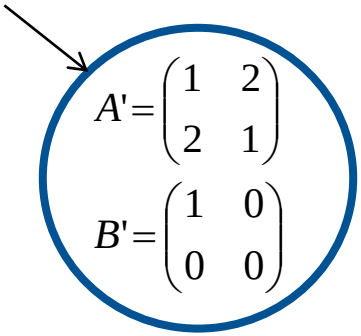
```
…
deterministic {
    C = parallel_matrix_multiply_int(A, B);
}
…
```

Deterministic Specification

```
…
deterministic {
    C = parallel_matrix_multiply_int(A, B);
}
…
```

Same input

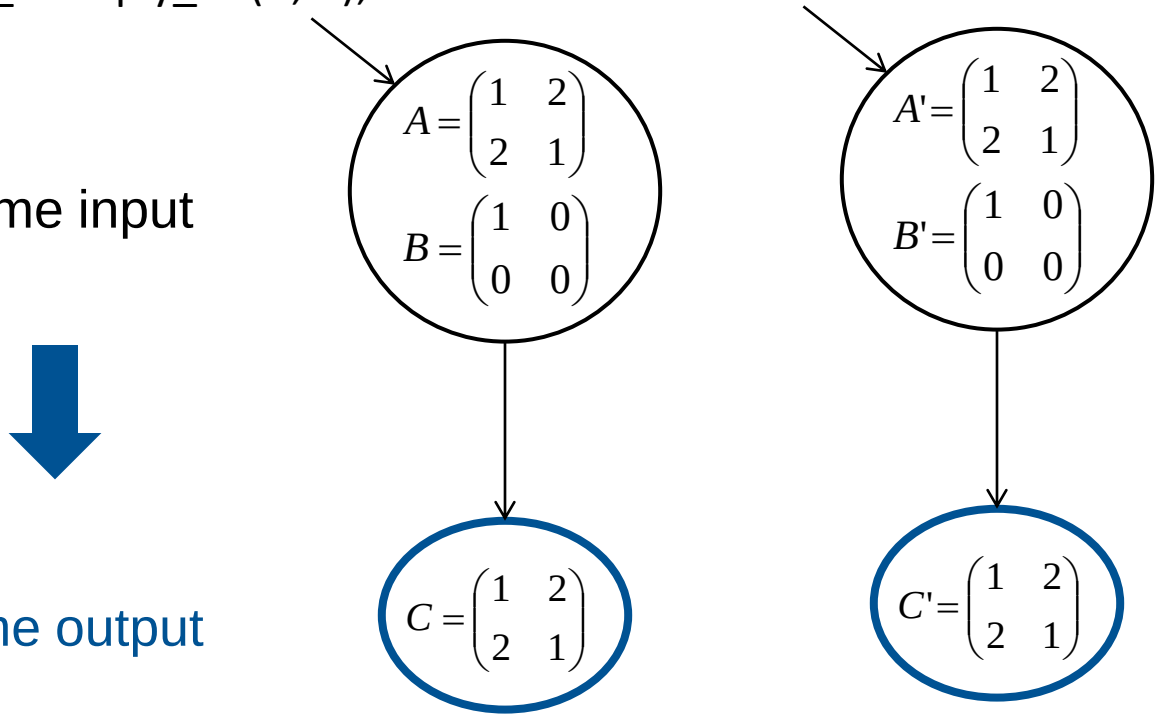




Deterministic Specification

```
…
deterministic {
    C = parallel_matrix_multiply_int(A, B);
}
…
```

Same input

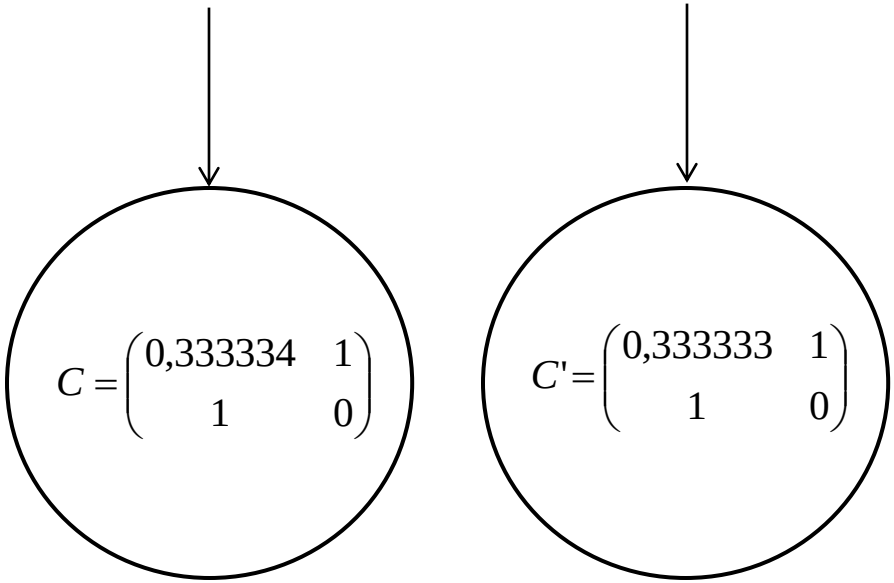


Same output

Semantic Determinism

```
…
deterministic {
    C = parallel_matrix_multiply_float (A, B);
}
…
```

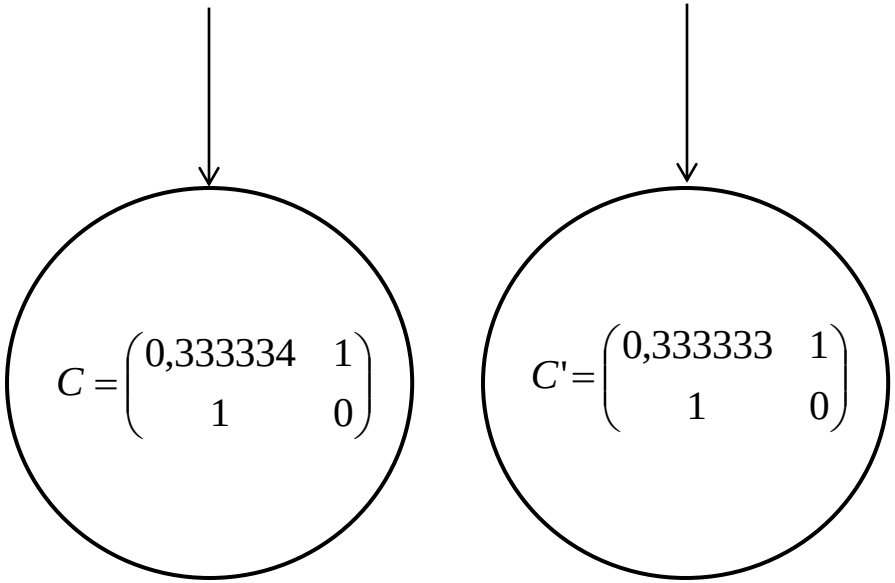
Same output?



Semantic Determinism

```
…
deterministic {
    C = parallel_matrix_multiply_float (A, B);
}
…
```

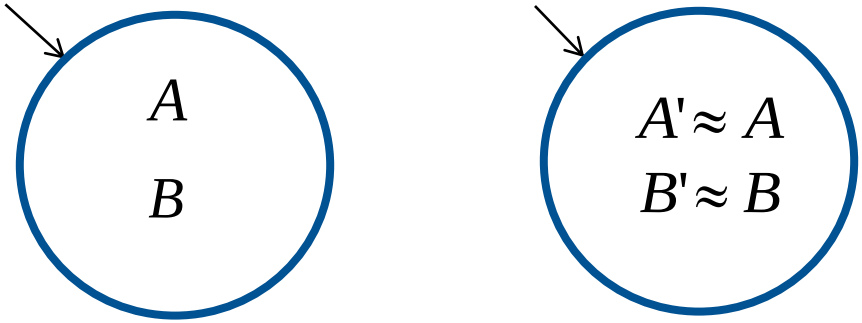
Same output?



Not exactly, but what does equivalence mean for floats?

Semantic Determinism

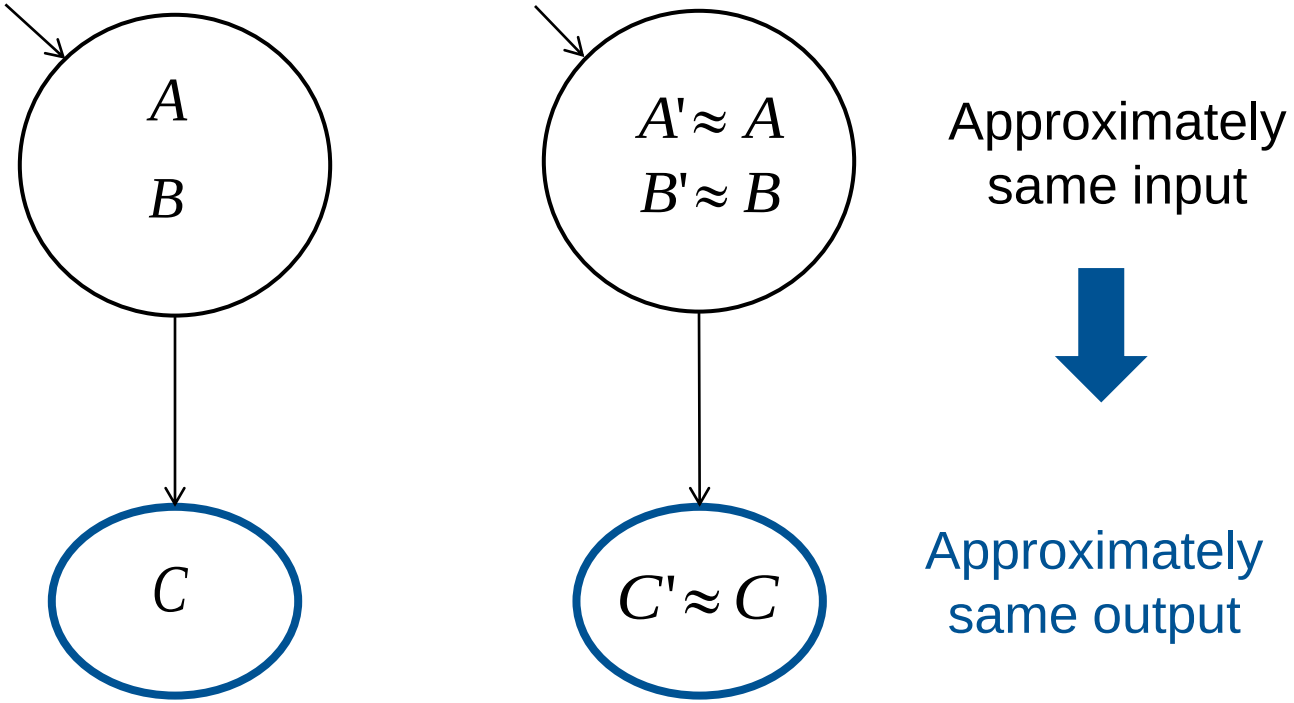
```
…
deterministic {
    C = parallel_matrix_multiply_float (A, B);
}
…
```



Approximately same input

Semantic Determinism

```
…
deterministic {
      C = parallel_matrix_multiply_float (A, B);
}
…
```



Semantic Determinism

Relaxing the deterministic specification:

Use relation between states from different executions of the code (**bridge predicates**)

...
deterministic assume ($|A - A'| < 10^{-6} \wedge |B - B'| < 10^{-6}$) {
    C = parallel_matrix_multiply_float (A, B);
} **assert** ($|C - C'| < 10^{-6}$)
...

Checking Deterministic Specifications

deterministic assume ($A = A' \wedge B = B'$) {
    C = parallel_matrix_multiply_int (A, B);
} **assert** ($C = C'$)

Execution 1 ⟶ store:

- Nothing zu check

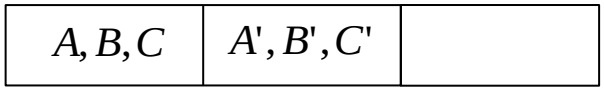
Checking Deterministic Specifications

deterministic assume $(A = A' \wedge B = B')$ {
    C = parallel_matrix_multiply_int (A, B);
} **assert** $(C = C')$

Execution 1 ⟶ store:

- Nothing zu check

Execution 2 ⟶ store:



- Check $(A = A' \wedge B = B') \Rightarrow (C = C')$

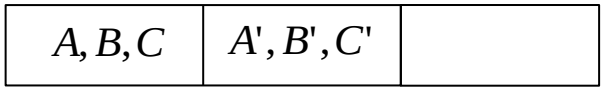
Checking Deterministic Specifications

deterministic assume $(A = A' \wedge B = B')$ {
    C = parallel_matrix_multiply_int (A, B);
} **assert** $(C = C')$

Execution 1 ⟶ store:

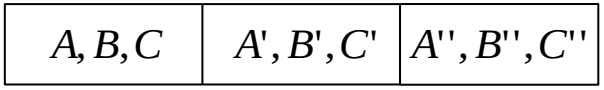
- Nothing zu check

Execution 2 ⟶ store:



- Check $(A = A' \wedge B = B') \Rightarrow (C = C')$

Execution 3 ⟶ store:



- Check $(A = A'' \wedge B = B'') \Rightarrow (C = C'')$
- Check $(A' = A'' \wedge B' = B'') \Rightarrow (C' = C'')$

Java Library-Example

…

Predicate equals = new Equals();

Deterministic.open();

Deterministic.assume(numberOfRowsA, equals);

Deterministic.assume(numberOfRowsB, equals);

…

P

Determinsitic.assert(matrix, equals);

Determinsitic.close();

…

corresponds to:

determinsitic

assume (numberOfRowsA = numberOfRowsA‘ and numberOfRowsB = numberOfRowsB‘ and …) {

P

} **assert** (matrix = matrix‘)

Java Library-Example

- Built-in predicate, i.e. already implemented
- uses equals()-method

```
…
Predicate equals = new Equals();
Deterministic.open();
Deterministic.assume(numberOfRowsA, equals);
Deterministic.assume(numberOfRowsB, equals);
…
P

Determinsitic.assert(matrix, equals);
Determinsitic.close();
…
```

corresponds to:

determinsitic

assume (numberOfRowsA = numberOfRowsA‘ and numberOfRowsB = numberOfRowsB‘ and …) {

P

} **assert** (matrix = matrix‘)

Java Library-Example

…

Predicate equals = new Equals();

Deterministic.open();

Deterministic.assume(numberOfRowsA, equals);

Deterministic.assume(numberOfRowsB, equals);

…

P

Determinsitic.assert(matrix, equals);

Determinsitic.close();

…

corresponds to:

determinsitic

assume (numberOfRowsA = numberOfRowsA‘ and numberOfRowsB = numberOfRowsB‘ and …) {

P

} **assert** (matrix = matrix‘)

Java Library-Example

```
…
Predicate equals = new Equals();
Deterministic.open();
Deterministic.assume(numberOfRowsA, equals);
Deterministic.assume(numberOfRowsB, equals);
…
P

Determinsitic.assert(matrix, equals);
Determinsitic.close();
…
```

corresponds to:

determinsitic

assume (numberOfRowsA = numberOfRowsA‘ and numberOfRowsB = numberOfRowsB‘ and …) {

P

} **assert** (matrix = matrix‘)

Java Library-Example

…

Predicate equals = new Equals();

Deterministic.open();

Deterministic.assume(numberOfRowsA, equals);

Deterministic.assume(numberOfRowsB, equals);

…

P

Determinsitic.assert(matrix, equals);

Determinsitic.close();

…

corresponds to:

determinsitic

assume (numberOfRowsA = numberOfRowsA‘ and numberOfRowsB = numberOfRowsB‘ and …) {

P

} **assert** (matrix = matrix‘)

Evaluation-Ease of Use

- 13 Java benchmark programs
- Authors added one deterministic block to each program manually
- Time needed: 5-10 minutes for most programs (without knowing the code)
- Lines of code added: 4-10 (+34 for one program: implementation of `equals` and `hashcode`)

Seems easy to use!

Evaluation-Effectiveness

- CALFUZZER was used to find data races and high-level races in the 13 benchmark programs
- **Data race**: Two threads access a shared memory location simultaneously and at least one is writing.
- **High-level race**: Like data races, but a simultaneous access to the shared memory location is not possible because of a lock. However actions, that should be executed atomically, can be interleaved.

- Goal: DCF finds all real determinism violations and marks no benign races.

Evaluation-Effectiveness

Benchmark suite	# Programs in benchmark suite	Races found by CALFUZZER (normal data races + high-level races)	Determinism Violations marked by DCF (normal data races + high-level races)
Java Grande Forum	8	9 + 2	1(*) + 0
Parallel Java Library	4	25 + 1	0 + 1(**)
-	1	6 + 2	0 + 0

All known determinism violations (* & **) where recognized by DCF when running 10 trials per found race.

Conclusion

Main contribution:

- ***Bridge predicates***, that relate program states from different executions
- ***Deterministic Specifications*** to express deterministic behaviour
- Method for ***checking*** deterministic specifications

Conclusion

Main contribution:

- ***Bridge predicates***, that relate program states from different executions
- ***Deterministic Specifications*** to express deterministic behaviour
- Method for ***checking*** deterministic specifications

Questions?