Towards a Calculus For Concurrent Java

Vladimir Klebanov
Universität Koblenz

June 7, 2004

The Goal

A correct and complete DL calculus for concurrent Java.

This calculus would:

- Handle threads + native synchronization primitives
- allow to verify arbitrary Java programs (but what properties?)
- probably be feasible only for small systems

...Is Not Impossible

Previous work:

- Gries and Owicki 1976:
  Axiomatic proof technique for parallel programs
- KIV Augsburg:
  Verifying concurrent systems with symbolic execution
- Ábrahám, de Boer, Steffen et al.:
  An assertion-based proof system for multithreaded Java

The Issue With Concurrency

Program semantics may be dependent on scheduling.

Race Conditions:

1. Two or more threads have access to the same memory location
2. At least one thread is writing to this location
3. No mechanism in place to guarantee temporal ordering

The Issue With Concurrency (2)

For t threads of n statements each, there are

$$\frac{(tn)!}{(n!)^t}$$

possible execution orderings.

2 threads with 5 statements — makes 252 interleavings
➥ proof space reduction is needed

A Remedy

Idea: Verify two threads T_1 and T_2 separately, then prove correct composition.

Proofs for T_1 and T_2 can be composed if no statement of T_1 **interferes** with statement of T_2 and vice versa.

Gives $|T_1| \times |T_2|$ additional correctness conditions.

Non-Interference

When does statement S_1 not interfere with S_2?

1. preservation of pre-state

   Let S_1 be executed in state char. by ϕ
   Let S_2 be executed in state char. by ψ
   Non-interference condition:
   $\psi \wedge \phi \rightarrow \langle S_1 \rangle \psi$
2. assertion insensitivity (post-state)
3. syntactical disjointness

What if we do have interference?

What else?

- Operational semantics of concurrency primitives
- From source code to JVM

JVM Concurrency Pitfalls

Lots of them:

- Load and store not atomic for `long` and `double`
- Compiler statement reordering
- Order of updates to variables as seen by other threads may not be consistent
- No guarantee that any `synchronized` method will ever be executed

Compositional Verification

- Develop robustness conditions for specifications of larger software units
- Compositionality principles, "lifting"
- Possibly just for certain architectures: Server, Producer/Consumer, Controller

The Properties

- (Partial) correctness — the usual, refers to terminal states only
- μ-Calculus (liveliness and safety properties)
- Regular/Schematic Sequence Charts

Thank You!

TOC