

# A Dynamic Logic for Deductive Verification of Concurrent Programs

Vladimir Klebanov

`vladimir@uni-koblenz.de`

June 6, 2006

# Goal

A Dynamic Logic for concurrent Java programs +  
a deductive calculus, which is

- correct
- complete
- adequate (including the JMM)
- understandable
- handles unbounded systems
- without abstraction
- scales reasonably

# Advantages of Our Product

- handles unbounded systems
- handles data well
- does not require abstraction
- based on symbolic execution
- simple cases simple, difficult still doable
- builds on KeY, which has a very high sequential coverage

# Current Restrictions

Real limitations:

- No thread identities **in** programs
- No dynamic thread creation (but unbounded concurrency)
- All loops within atomic blocks

Not finished yet:

- Locking (synchronized **blocks**)
- Conditional variables (`wait()/notify()`)

# Concurrent Dynamic Logic

An example:

$$\langle \{\mathbf{123}\} g=1; \{ \} sta_2; \{ \} sta_3; \rangle \phi$$

may evolve to:

$$\{g := 1(\mathbf{2})\} \langle \{\mathbf{1} \ \mathbf{3}\} g=1; \{\mathbf{2}\} sta_2; \{ \} sta_3; \rangle \phi$$

or to...

(branches badly)

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \rightsquigarrow \{g := 1(p(k))\}$$

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \rightsquigarrow \{g := 1(p(k))\}$$

$$\langle \{\mathbf{123}\} g=1; \{ \} sta_2; \{ \} sta_3; \{ \} \rangle \phi$$

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \rightsquigarrow \{g := 1(p(k))\}$$

$$\langle \{\mathbf{123}\}_{g=1}; \{ \ }_{sta_2}; \{ \ }_{sta_3}; \{ \ } \rangle \phi$$
$$\langle \{3\}_{g=1}; \{0\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi$$

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \quad \rightsquigarrow \quad \{g := 1(p(k))\}$$

$$\begin{aligned} & \langle \{\mathbf{123}\}_{g=1}; \{ \} sta_2; \{ \} sta_3; \{ \} \rangle \phi \\ & \langle \{3\}_{g=1}; \{0\} sta_2; \{0\} sta_3; \{0\} \rangle \phi \\ & \{g := 1(p(1))\} \langle \{2\}_{g=1}; \{1\} sta_2; \{0\} sta_3; \{0\} \rangle \phi \end{aligned}$$

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \quad \rightsquigarrow \quad \{g := 1(p(k))\}$$

$$\begin{aligned} & \langle \{\mathbf{123}\}_{g=1}; \{ \}_{sta_2}; \{ \}_{sta_3}; \{ \} \rangle \phi \\ & \langle \{3\}_{g=1}; \{0\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \\ & \{g := 1(p(1))\} \langle \{2\}_{g=1}; \{1\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \\ & \{g := 1(p(2))\} \langle \{1\}_{g=1}; \{2\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \end{aligned}$$

# Idea: Make Scheduling Explicit

$$\langle g=1; \rangle \quad \rightsquigarrow \quad \{g := 1(p(k))\}$$

$$\begin{aligned} & \langle \{\mathbf{123}\}_{g=1}; \{ \}_{sta_2}; \{ \}_{sta_3}; \{ \} \rangle \phi \\ & \langle \{3\}_{g=1}; \{0\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \\ & \{g := 1(p(1))\} \langle \{2\}_{g=1}; \{1\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \\ & \{g := 1(p(2))\} \langle \{1\}_{g=1}; \{2\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \\ & \{g := 1(p(3))\} \langle \{0\}_{g=1}; \{3\}_{sta_2}; \{0\}_{sta_3}; \{0\} \rangle \phi \end{aligned}$$

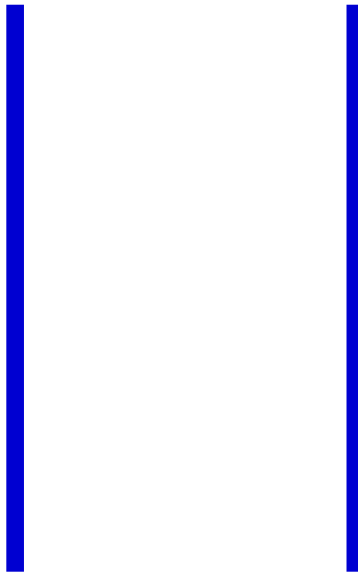
# Scheduling as Permutation

- dynamic thread creation
  - non-atomic loops
  - + some magic
- 

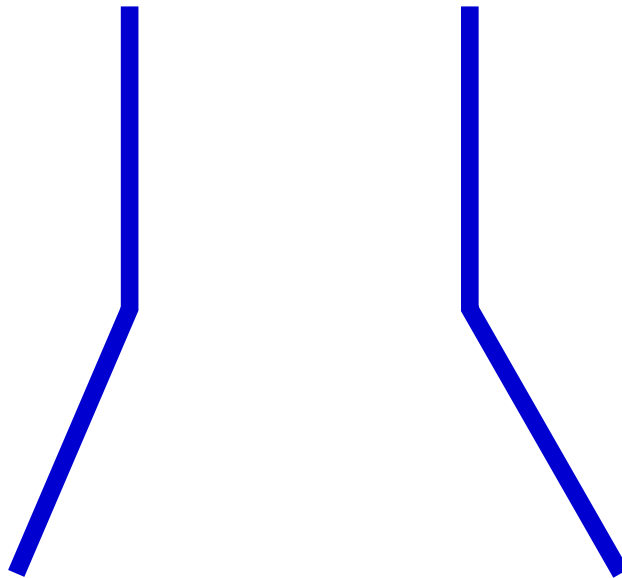
every thread passes every position exactly once.

↳ Scheduling with permutation functions.

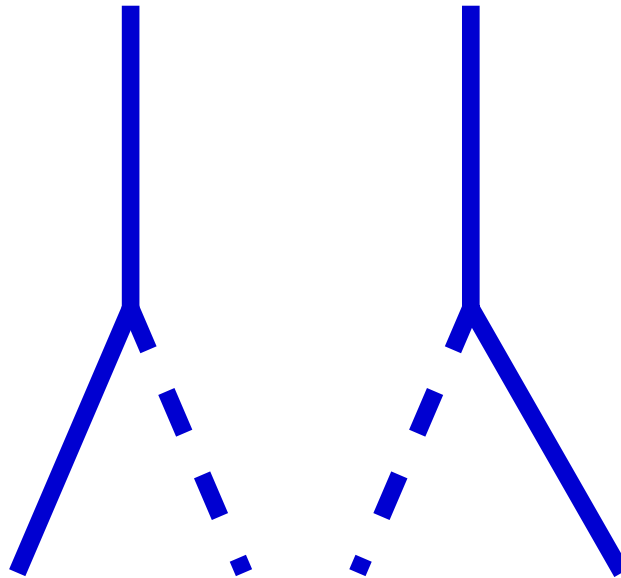
# Thread Symmetry (Same Data)



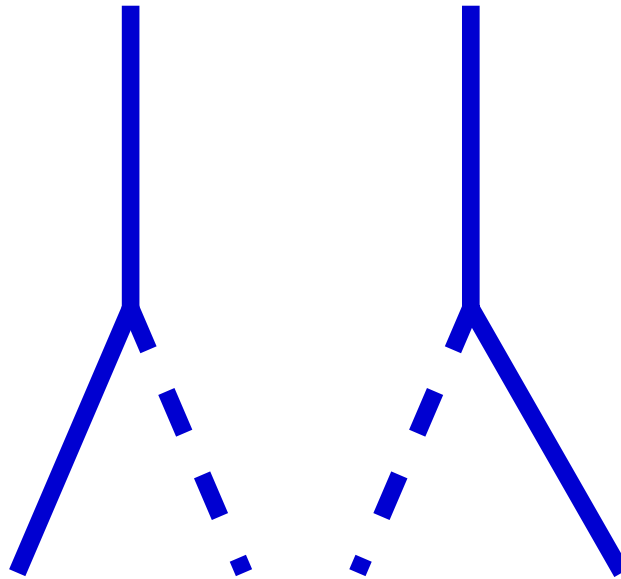
# Different Data—Symmetry Lost



## ... And Regained (with Symbolic Execution)



## ... And Regained (with Symbolic Execution)



Scheduling is separated from data.

# Symbolic Execution Rules

step

$$\begin{array}{c}
 \vdash P(r, c) = pos \\
 path(pos, p) \vdash \langle [S^{*(pos)}] \rangle \langle [r \mid \pi \{p_{pos:n-1}\} S\{p_{pos+1:k+1}\} \omega] \rangle \phi \\
 \neg path(pos, p) \vdash \langle [r \mid \pi \{p_{pos:n-1}\} S\{p_{pos+1:k+1}\} \omega] \rangle \phi \\
 \hline
 \vdash \underbrace{\langle [r \mid \pi \{p_{pos:n}\} S\{p_{pos+1:k}\} \omega] \rangle \phi}_{= p}
 \end{array}$$

and where  $pos$  is the position of  $S$  in  $p$

# An Invariant Rule

invariant

$$\frac{\begin{array}{l} \vdash \mathcal{U}INV(r, c_0) \\ INV(r, c_{\downarrow}) \vdash \phi \\ INV(r, c), \text{ path}(1, p), c(1) > 0 \vdash \langle\langle S_1^{*(1)} \rangle\rangle INV(r, c_1) \\ \vdots \\ INV(r, c), \text{ path}(q, p), c(q) > 0 \vdash \langle\langle S_q^{*(q)} \rangle\rangle INV(r, c_q) \end{array}}{\vdash \mathcal{U}\langle\langle r | c_0 | p \rangle\rangle \phi}$$

# An Example

Prove:

$$\{\text{sum} := 0\} \langle \{n\} \ll \text{sum} = \text{sum} + e; \gg \{ \} \rangle (\text{sum} = \sum_{i=1}^n e(i))$$

## An Example

Prove:

$$\{\text{sum} := 0\} \langle \{n\} \ll \text{sum} = \text{sum} + e; \gg \{\} \rangle (\text{sum} = \sum_{i=1}^n e(i))$$

After one step:

$$\{\text{sum} := \text{sum} + e(p(1))\} \\ \langle \{n-1\} \ll \text{sum} = \text{sum} + e; \gg \{1\} \rangle (\text{sum} = \sum_{i=1}^n e(i))$$

## An Example (cntd.)

After  $n$  steps:

$$\left\{ \text{sum} := \sum_{i=1}^n e(p(i)) \right\}$$
$$\langle \{0\} \ll \text{sum} = \text{sum} + e; \gg \{n\} \rangle \left( \text{sum} = \sum_{i=1}^n e(i) \right)$$

# Current Status and Future Directions

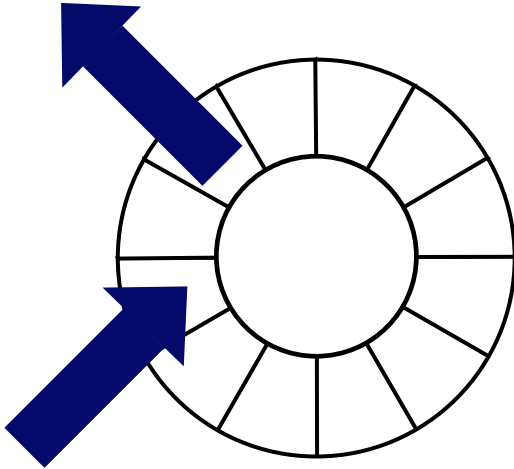
Done:

- Basic calculus
- Prototypical implementation

Next steps:

- Locking (mutual exclusion)
- Conditional variables (`wait()/notify()`)
- JMM-Faithfulness
- Tackle restrictions

## Further Experiments



- Blocking concurrent queue
- `java.util.concurrent-Library`

**Thank You!**

Questions?

## Related Work (1)

**Model checkers** for Java software...

- handle data badly
- handle unbounded systems badly
- abstraction is a black art
- not JMM-faithful
- not understandable
- we don't do it

But: good advances in scalability, state space reduction techniques.

## Related Work (2)

- Calculi for other concurrent languages: JCSP, Erlang, etc.
- Concurrent Dynamic Logic [Peleg]

## Related Work (3)

The “Kiel Calculus”:

- bad sequential coverage
- requires Hoare-style annotations
- VCG-style reasoning
- verification conditions are fed into PVS (↪doom!)
- not JMM-faithful

# TOC

Goal ❖

Advantages of Our Product ❖

Current Restrictions ❖

Concurrent Dynamic Logic ❖

Idea: Make Scheduling Explicit ❖

Scheduling as Permutation ❖

Thread Symmetry (Same Data) ❖

Different Data—Symmetry Lost ❖

... And Regained (with Symbolic Execution) ❖

Symbolic Execution Rules ❖

An Invariant Rule ❖

An Example ❖

An Example (cntd.) ❖

Current Status and Future Directions ❖

Further Experiments ❖

Thank You! ❖

Related Work (1) ❖

Related Work (2) ❖

Related Work (3) ❖