Enabling the Runtime Assertion Checking of Concurrent Contracts for the Java Modeling Language

by Wladimir Araujo, Lionel C. Briand, Yvan Labiche

CCC Seminar Talk

Alexander Kogtenkov

ETH Zürich

April 15, 2015

Bibliography *Cited by*

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

[1] W. Araujo, L.C. Briand, and Y. Labiche. "On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Race Conditions and Deadlocks in Concurrent Object-Oriented Software". In: *ESEM 2011*. Sept. 2011, pp. 10–19.

[2] W. Araujo, L.C. Briand, and Y. Labiche. "On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Functional Faults in Concurrent Object-Oriented Software". In: *Software Engineering, IEEE Transactions on* 40.10 (Oct. 2014), pp. 971–992.

[3] E.T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *Software Engineering, IEEE Transactions on* PP.99 (2014), pp. 1–30.

[4] Antonio Carzaniga et al. "Cross-checking Oracles from Intrinsic Software Redundancy". In: ICSE 2014. Hyderabad, India: ACM, 2014, pp. 931–942.

[5] Mark Harman et al. *A Comprehensive Survey of Trends in Oracles for Software Testing*. Tech. rep. CS-13-01. University of Sheffield, Department of Computer Science, 2013.

[6] A. Jyoti and V. Arora. "Debugging and visualization techniques for multithreaded programs: A survey". In: *ICRAIE, 2014*. May 2014, pp. 1–6.

[7] Jorne Kandziora. *Runtime assertion checking of multithreaded Java programs - An extension of the STROBE framework*. Aug. 2014.

[8] Mauro Pezzè and Cheng Zhang. "Chapter One - Automated Test Oracles: A Survey". In: ed. by Atif Memon. Vol. 95. Advances in Computers. Elsevier, 2014, pp. 1–48.

Bibliography *Paper details*

W. Araujo, L.C. Briand, and Y. Labiche. “Enabling the runtime assertion checking of concurrent contracts for the Java modeling language”. In: *Software Engineering (ICSE), 2011 33rd International Conference on*. May 2011, pp. 786–795. DOI: 10.1145/1985793.1985903.

Wladimir Araujo

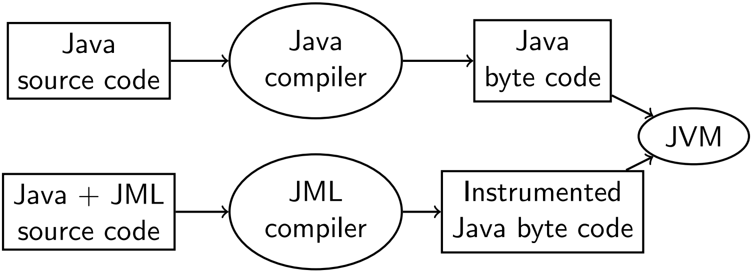
Juniper Networks

Lionel C. Briand

Simula Research Laboratory and University of Oslo

Yvan Labiche

Carleton University



Design by Contract in a nutshell

Theory: correct program + DbC $\Longleftrightarrow$ correct program

Run-time:

Sequential: regular $\equiv$ instrumented

Concurrent: *execution time affects execution paths*

How to restore equivalence?

Concurrent DbC *Challenges*

- Interference

  Objects involved in contracts may be changed before/after or during execution of a method

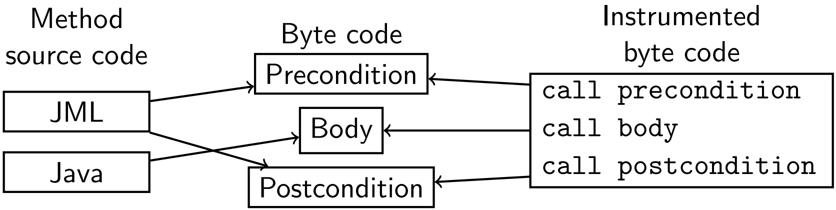
- Locking-related properties

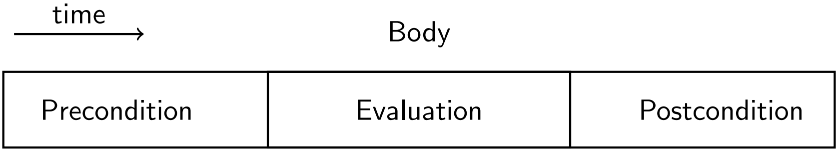
  Deadlocks

- Specification of thread-safety properties in presence of inheritance

  Thread-safety predicates may become meaningless in descendants

Interference *Sequential JML*





Example: Linked queue

```
     /* @ public normal_behavior
3       @ ensures \result <==> head.next == null */
     public boolean isEmpty () {
1        synchronized (head) {

2        return head.next == null;
     }}
```

...	...	"make list empty"	head.next == **null**
T1	1'	`lock (head)`	[head]
T1	2	**result** = (head.next == **null**)	[head], **result** == **true**
T1	1"	`unlock (head)`	**result** == **true**
T2	...	`insert (v)`	head.next != **null**
T1	3	**result** <==> head.next == **null**	*Postcondition violation!*

Example: Linked queue

```
  /* @ public normal_behavior
3    @ ensures \result <==> head.next == null */
  public boolean isEmpty () {
1     synchronized (head) {
      // @ ensures_safepoint:
2     return head.next == null;
  }}
```

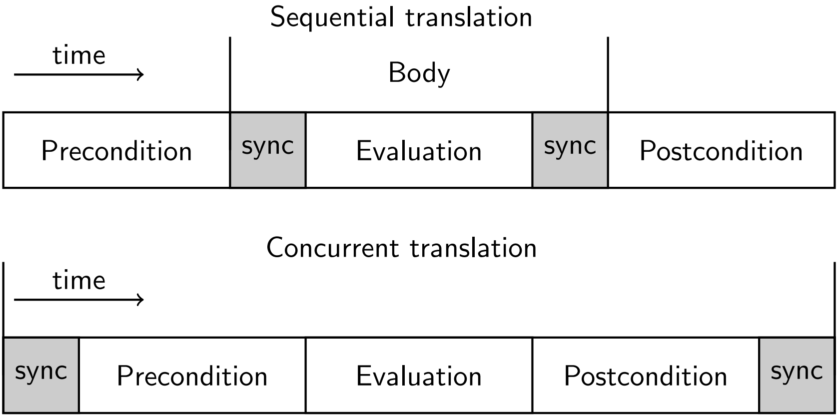
...	...	"make list empty"	head.next == **null**
T1	1'	`lock (head)`	[head]
T1	2	**result** = (head.next == **null**)	[head], **result** == **true**
T1	1"	`unlock (head)`	**result** == **true**
T2	...	`insert (v)`	
T1	3	**result** <==> head.next == **null**	

Example: Linked queue

```
  /* @ public normal_behavior
3    @ ensures \result <==> head.next == null */
  public boolean isEmpty () {
1     synchronized (head) {
      // @ ensures_safepoint:
2     return head.next == null;
  }}
```

...	...	"make list empty"	head.next == **null**
T1	1'	`lock (head)`	[head]
T1	2	**result** `=` `(head.next == ` **null**`)`	[head], **result** == **true**
T1	3	**result** `<==>` `head.next ==` **null**	[head], *Postcondition OK*
T1	1"	`unlock (head)`	**result** == **true**

Interference *Concurrent JML*



Mechanism	Specification	Implementation
Safepoints	`requires` `ensures`	`requires_safepoint` `ensures_safepoint`
Wait condition	`when`	`commit`
Thread safety	`requires_thread_safe` `ensures_thread_safe`	—

Scope

- Safepoints: internal state
- Thread safety: external state

Mechanism	Specification	Implementation
Safepoints	`requires` `ensures`	`requires_safepoint` `ensures_safepoint`
Wait condition	`when`	`commit`
Thread safety	`requires_thread_safe` `ensures_thread_safe`	—

Scope

- Safepoints: internal state
- Thread safety: external state

Thread safety + Safepoints = No Interference

Lock order specification

Syntax

`lock_order` *lock1* < *lock2*

`lock_order` *lock1* <= *lock2*

Semantics

Acquisition order	<	<=
lock1 before *lock2*	true	true
lock1 only	true	true
lock2 before *lock1*	false	false
lock2 only	false	false
neither *lock1* not *lock2*	false	true

Monitoring

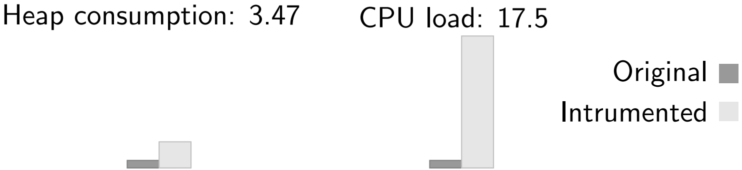
At every lock acquisition point

Before actually attempting to acquire a lock (to prevent deadlocks)

Equivalence assessment *Run-time overhead*

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

- Setup: Router driver of Juniper's E-series routers, 520 000 active subscribers, 1 500 transactions per second
  - 54 classes
  - 33 509 LOC
  - 34% concurrent behavior
- Measurement



Performance ratio between production and instrumented versions is **constant**.

Equivalence assessment *Indistiguishability*

Analysis of behavior:

- Standard test suite
  - 2 hours, no contract violations
- Faults introduced from Juniper's defect database
  - Instrumentation to print errors instead of assertion violations
  - Reproduced all 139 functional and concurrent faults
- Influence of instrumentation on thread interleavings
  - No additional locks
    - Safepoint evaluation (inside `synchronized`)
    - Commit points for wait conditions (inside `synchronized`)
    - Locking predicates (use thread-local objects in pre- and post-states)
  - Limited locking
    - `ConcurrentHashMap` optimized for concurrent access
    - Writes only in pre- and post-states of methods with `requries_/ensures_thread_safety`
    - Reads rarely acquire locks

Comparison to SCOOP

Property	Concurrent JML	SCOOP
Syntax	$>$ 10 keywords	`separate`
Internal state	Safepoints	No data races
External state	Thread-safety	Controlled arguments
Wait conditions	Explicit	Separate calls in preconditions
Synchronization	Explicit locking	Controlled arguments
Deadlock avoidance	Lock order	Partial: controlled arguments

Beyond the paper *Open questions*

For further discussion

- What is allowed between method entry and `requires_safe_point`?
- Blocks `concurrent_behavior` are combined using conjunction. What about modularity?
- What feature of concurrent JML would be nice to have in SCOOP?
- What can be improved or simplified in concurrent JML?
- Does lock order specification prevent from deadlocks?