Clara: a Framework for Statically Evaluating Finite-state Runtime Monitors

Eric Bodden, Patrick Lam and Laurie Hendren

tutorial vs. this talk

Integrate results of three communities

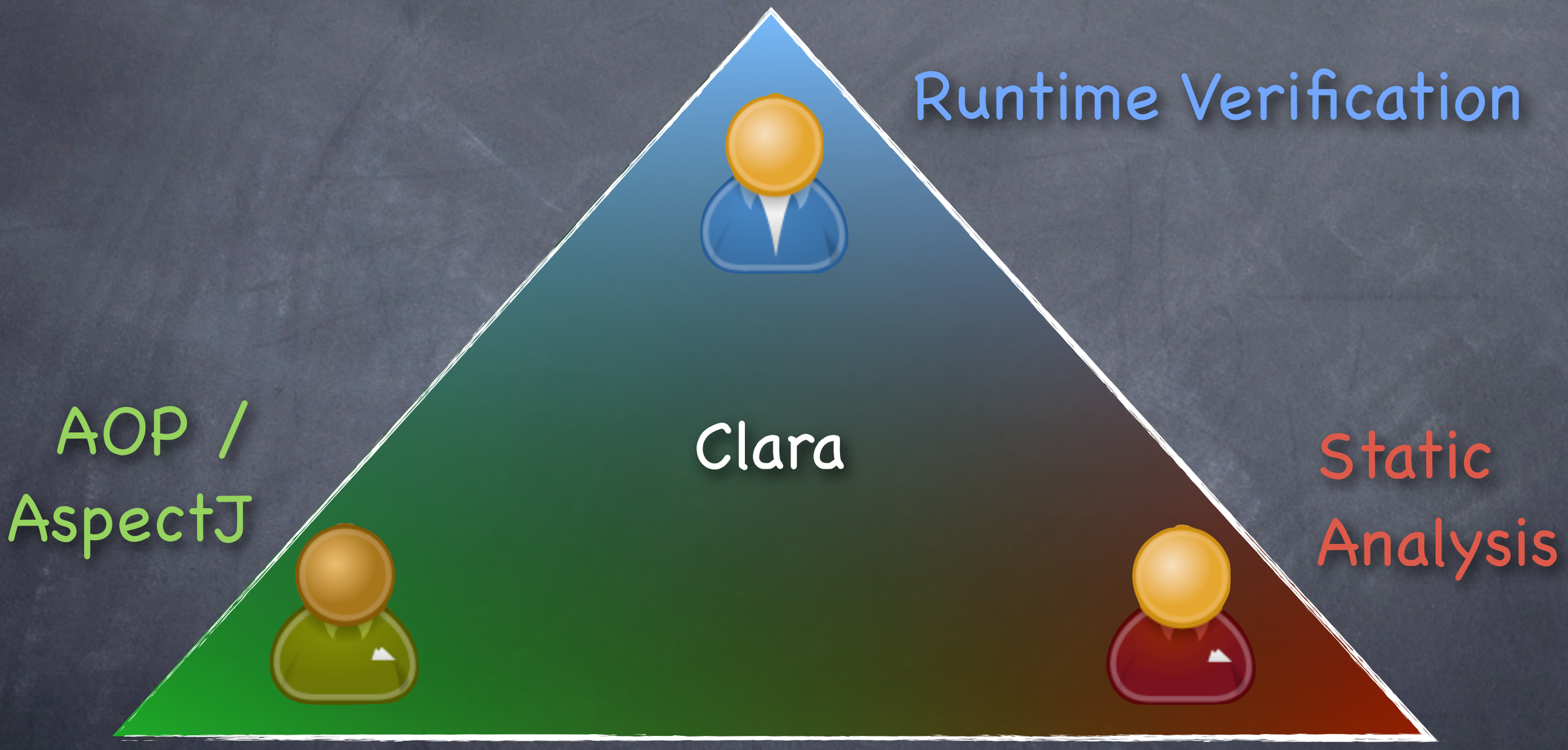
Runtime Verification

AOP / AspectJ

Static Analysis

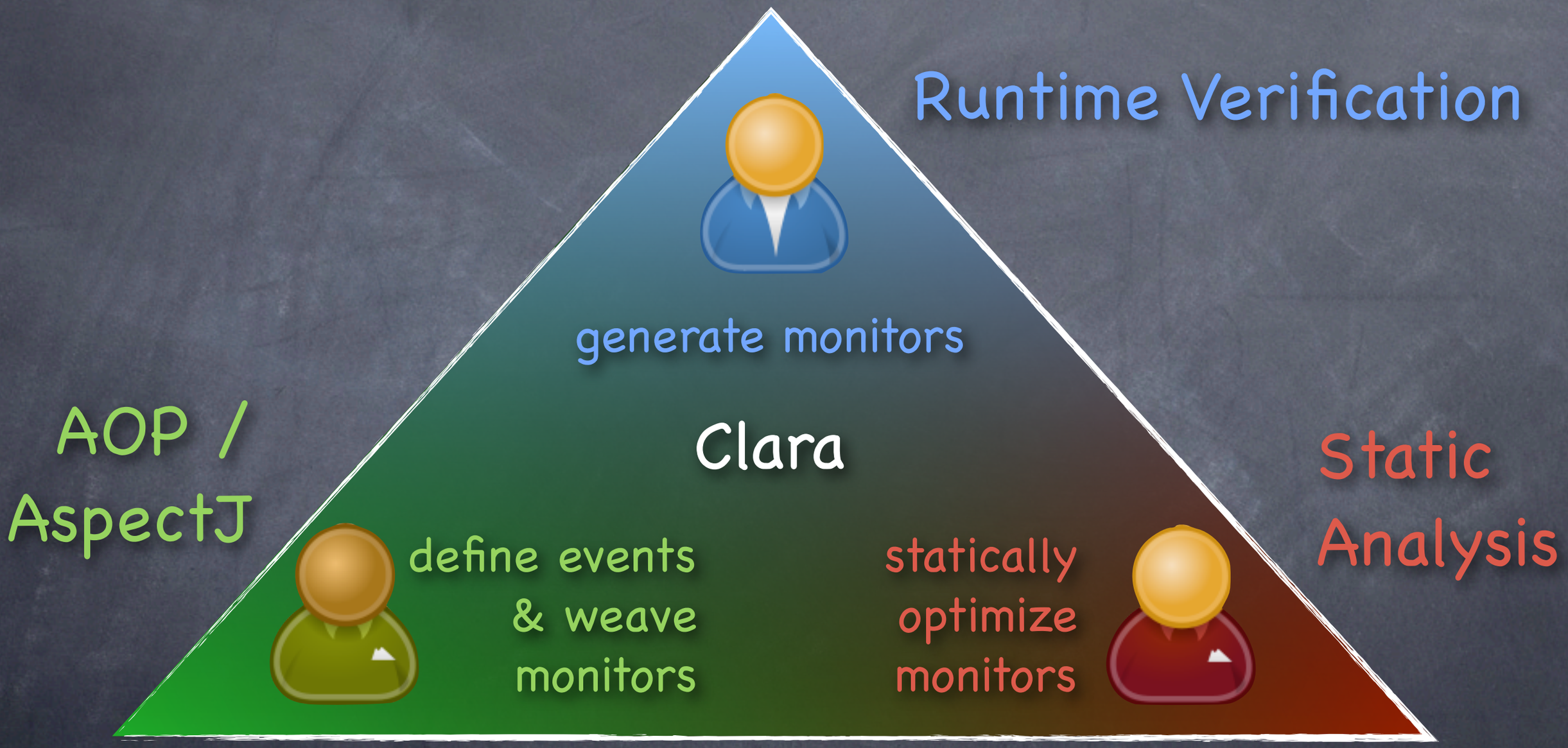
3

Integrate results of three communities



3

Integrate results of three communities



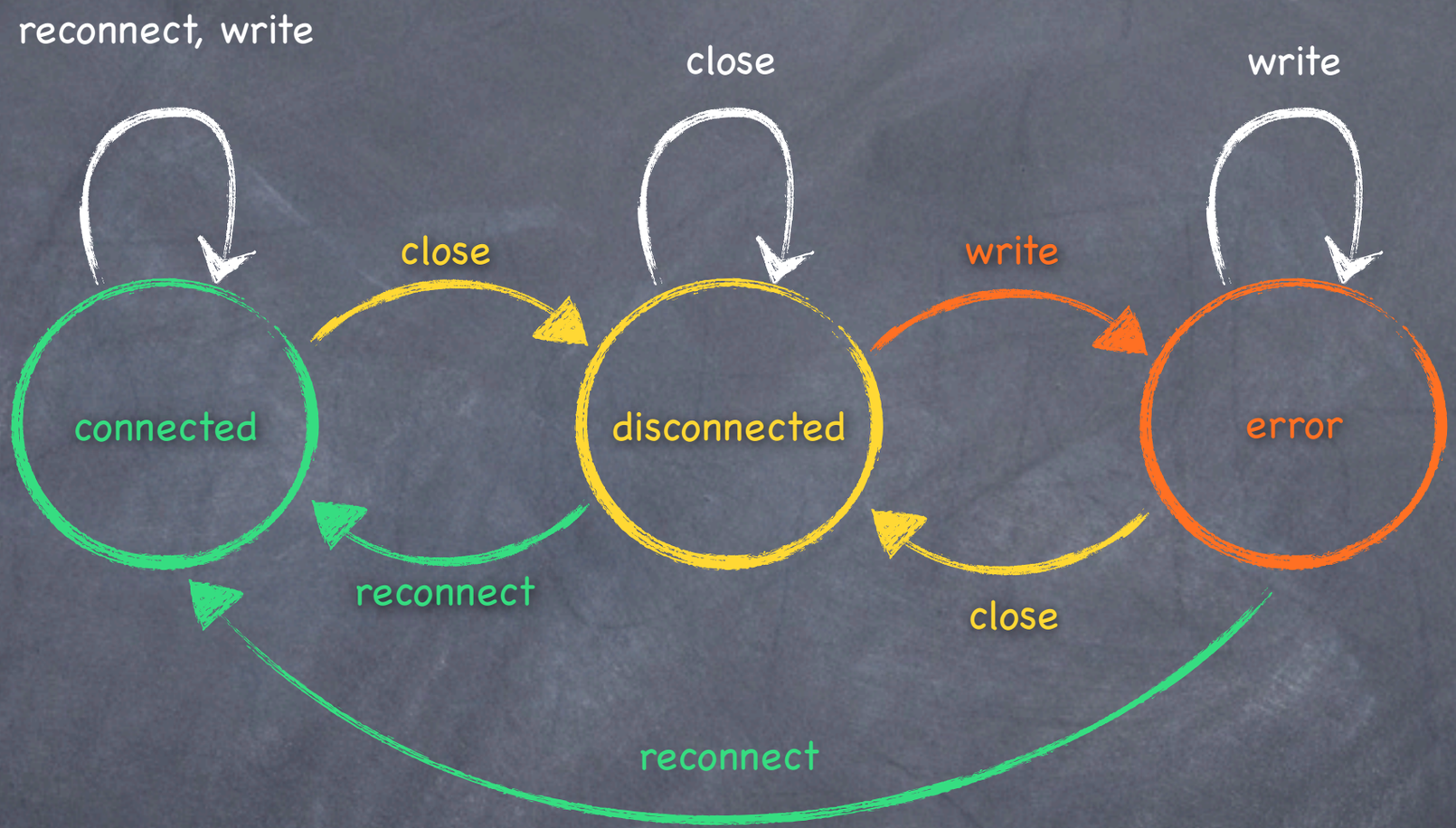
3

Finite-state properties

"After closing a connection c,
don't write to c until c is reconnected."

4

Finite-state properties



"After closing a connection c,
don't write to c until c is reconnected."

4

Runtime verification of finite-state properties (in AspectJ)

```
Set closed = new WeakIdentityHashSet();

after(Connection c) returning:
   call(* Connection.close()) && target(c) {
   closed.add(c);
}

after(Connection c) returning:
   call(* Connection.reconnect()) && target(c) {
   closed.remove(c);
}

after(Connection c) returning:
   call(* Connection.write(..)) && target(c) {
   if(closed.contains(c))
      error("May not write to "+c+", as it is closed!");
}
```


5

JavaMOP

various spec. languages

LTL

J-LO

Larva

DATEs

relational aspects

abc

Monitoring Aspects

LSCs

S2A

tracematches

Java-STAIRS aspects

MOFScript

MSCs

M2Aspects

6

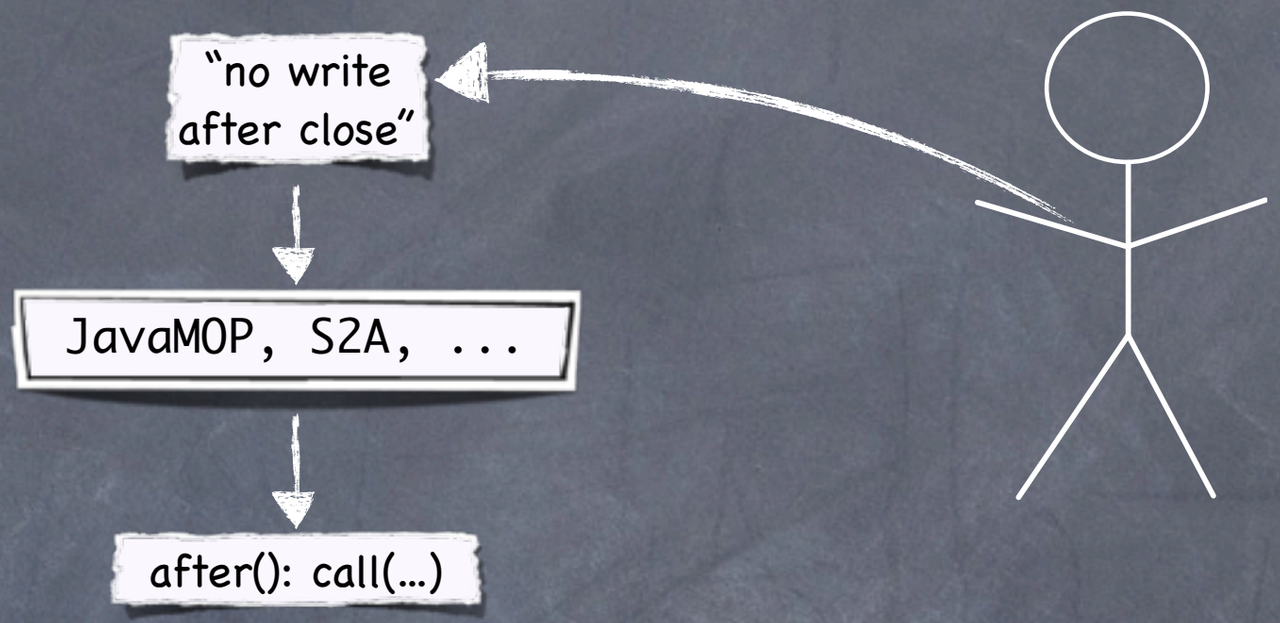
Runtime verification of finite-state properties

7

Runtime verification of finite-state properties

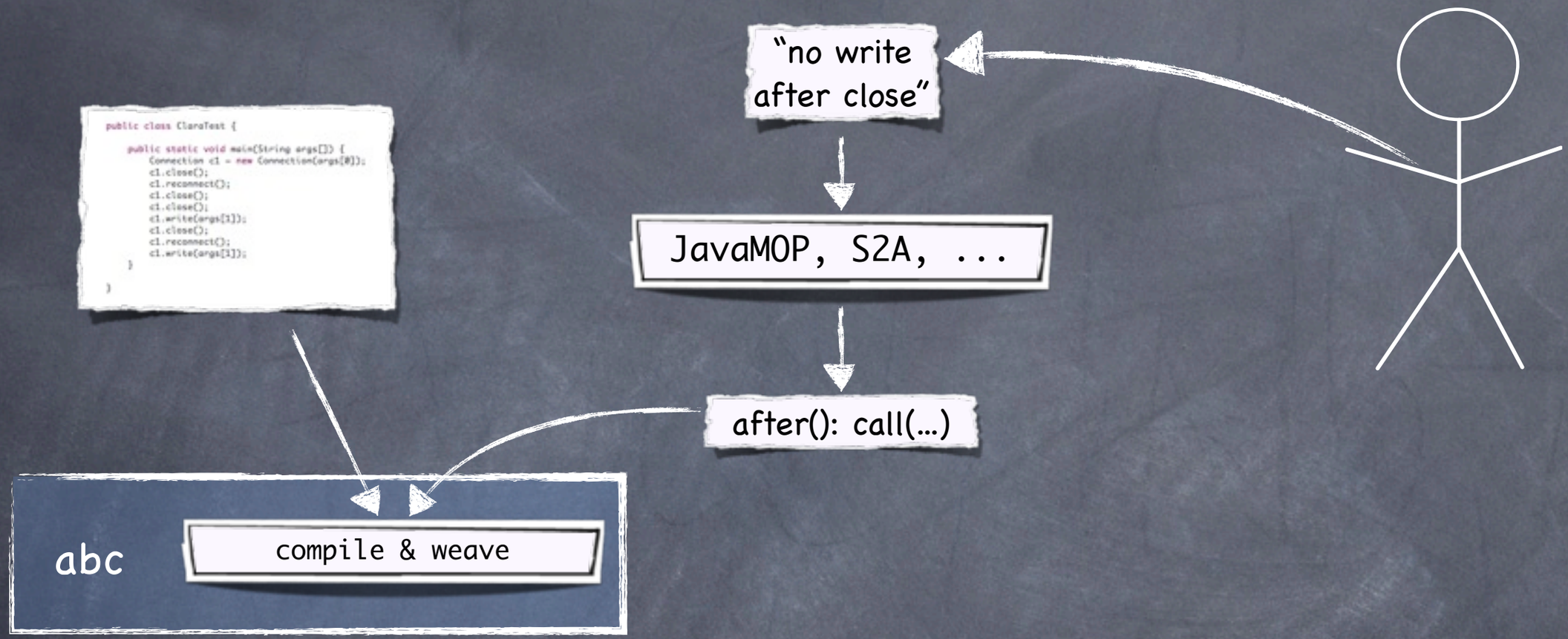
7

Runtime verification of finite-state properties



7

Runtime verification of finite-state properties



7

Runtime verification of finite-state properties

```
public class ClaraTest {

    public static void main(String args[]) {
        Connection c1 = new Connection(args[0]);
        c1.close();
        c1.reconnect();
        c1.close();
        c1.close();
        c1.write(args[1]);
        c1.close();
        c1.reconnect();
        c1.write(args[1]);
    }

}
```

"no write after close"

JavaMOP, S2A, ...

after(): call(...)

abc

compile & weave

```
public class ClaraTest {

    public static void main(String args[]) {
        Connection c1 = new Connection(args[0]);
        c1.close();
        c1.reconnect();
        c1.close();
        c1.close();
        c1.write(args[1]);
        c1.close();
        c1.reconnect();
        c1.write(args[1]);
    }

}
```


7

Runtime verification of finite-state properties

8

Runtime verification of finite-state properties

No static guarantees

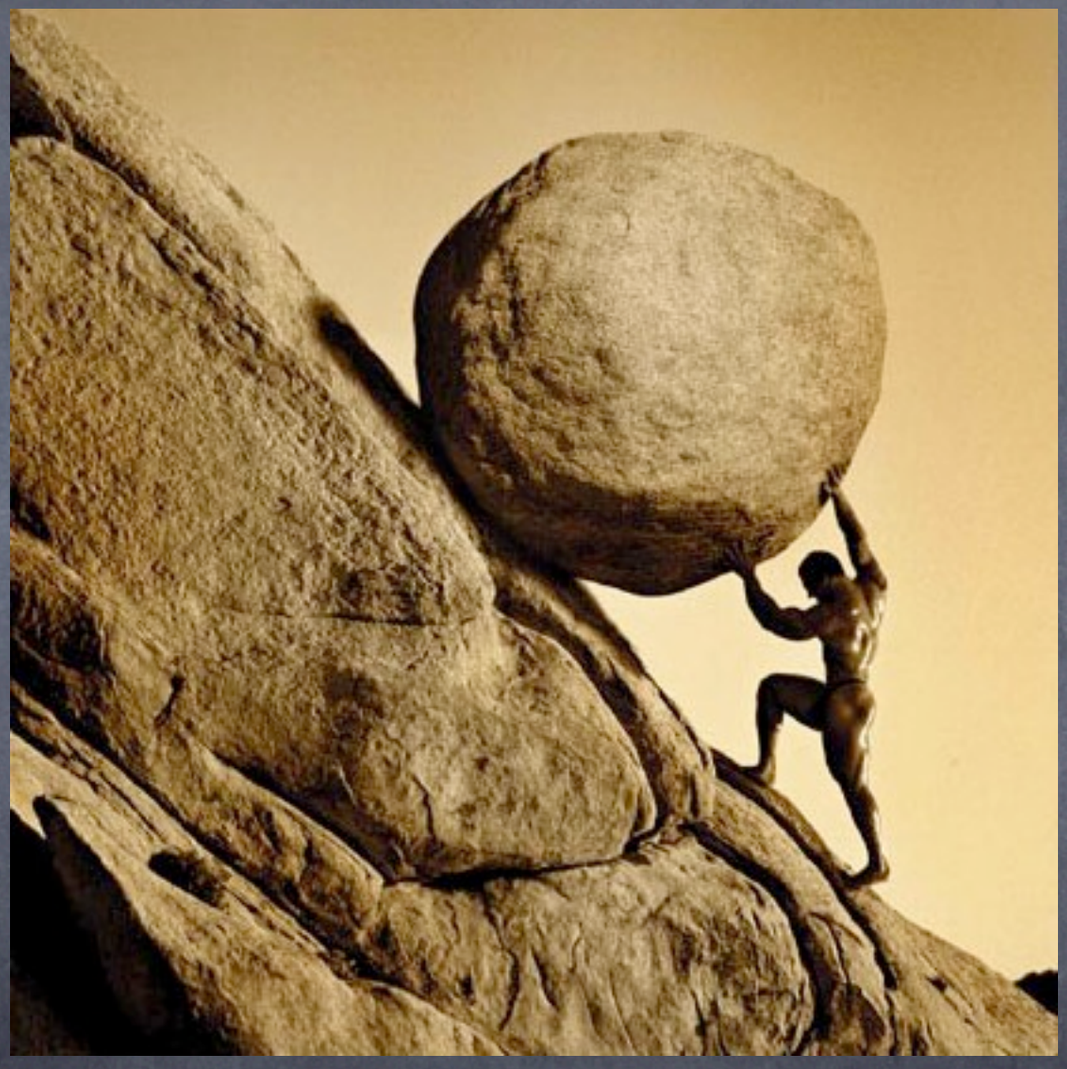
8

Runtime verification of finite-state properties

Potentially large runtime overhead

8

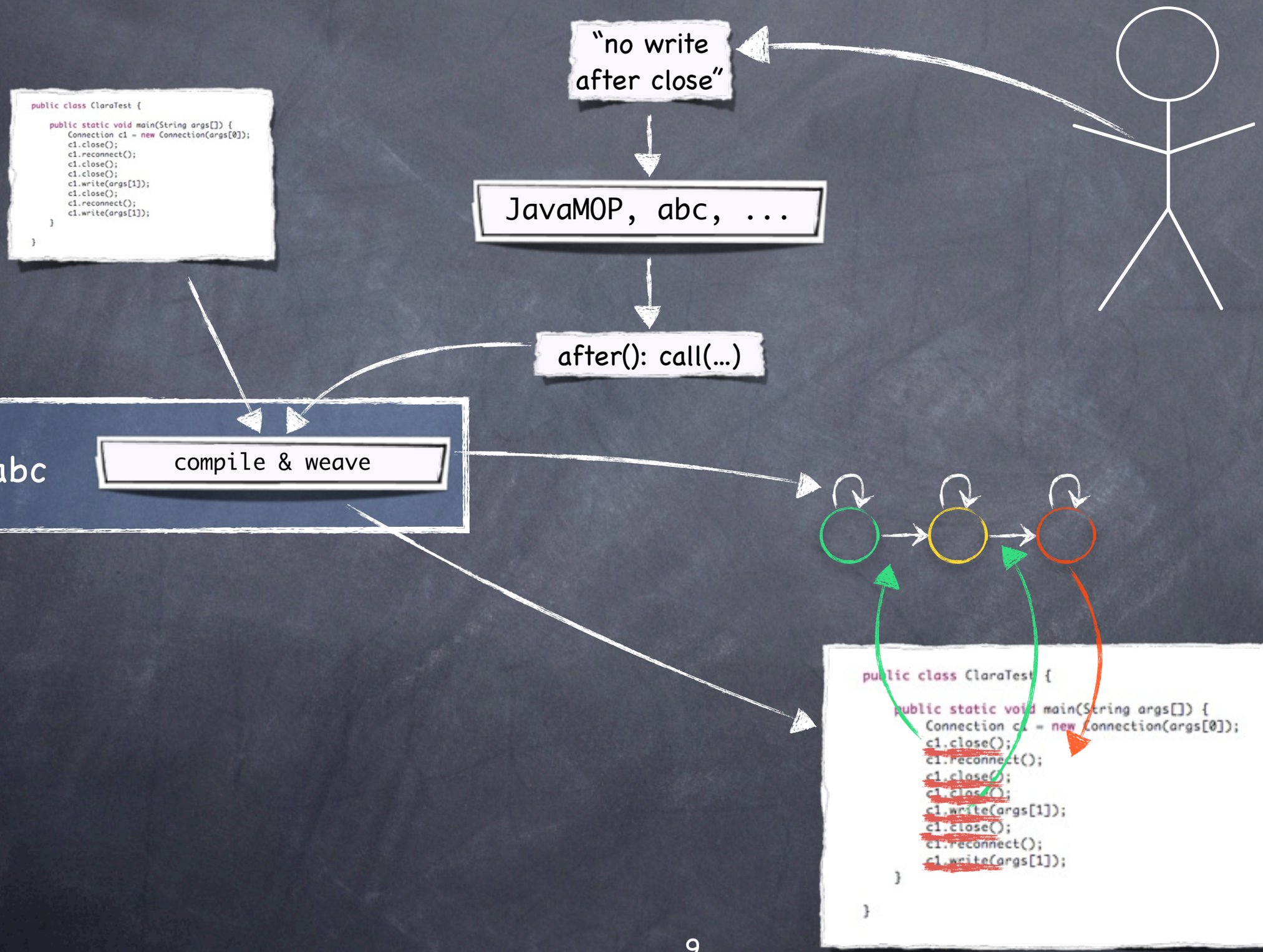
Runtime verification of finite-state properties



When to finish testing?

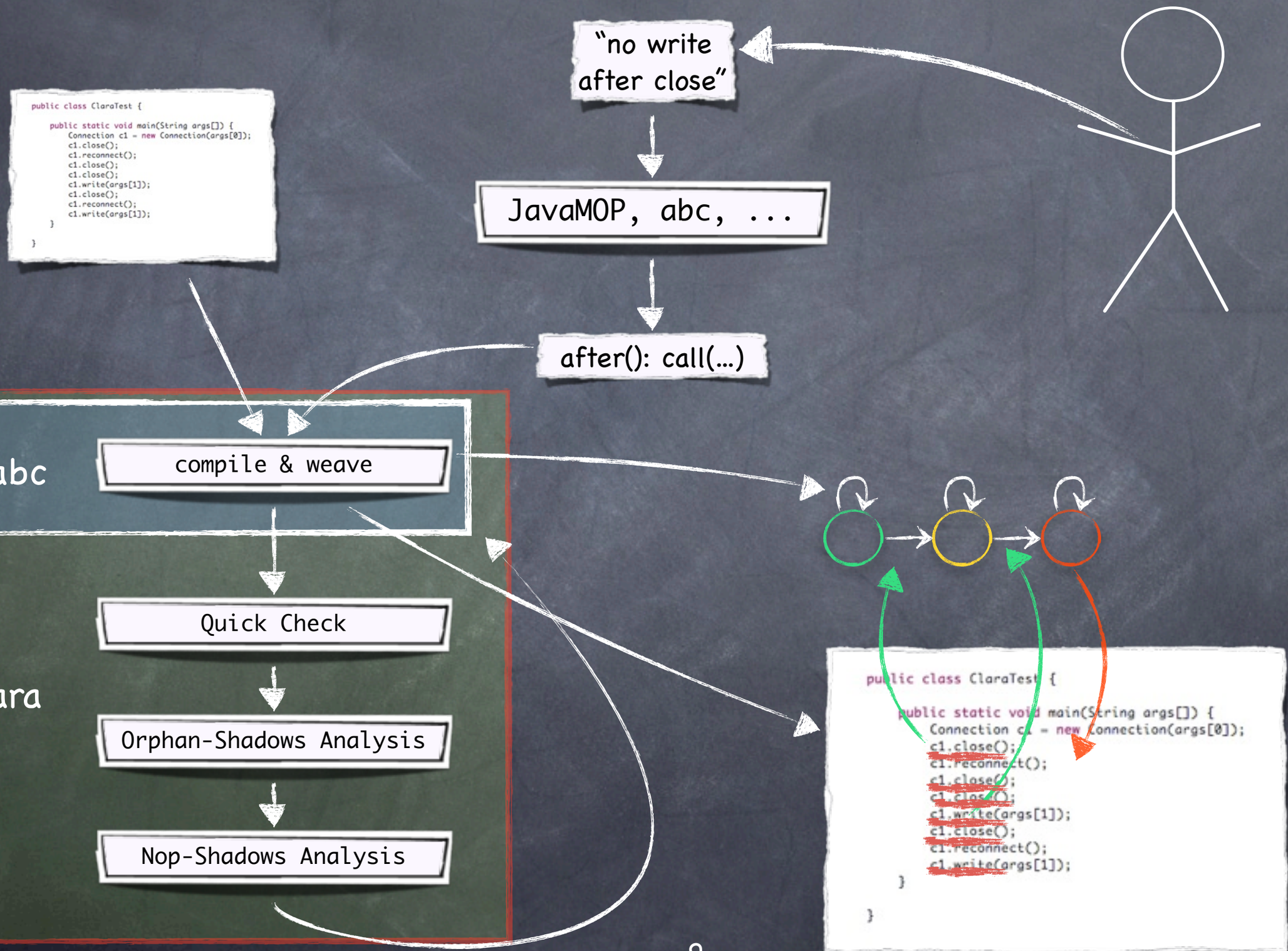
8

The Clara Framework



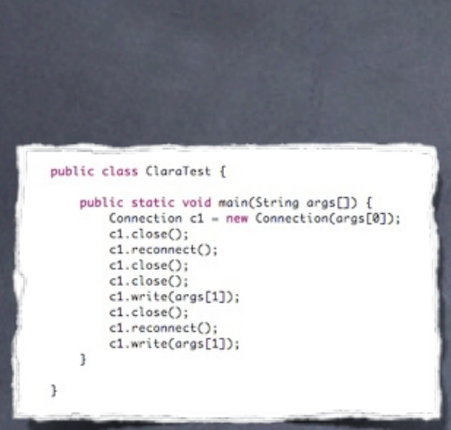
9

The Clara Framework



9

The Clara Framework



"no write after close"

JavaMOP, abc, ...

after(): call(...)

abc

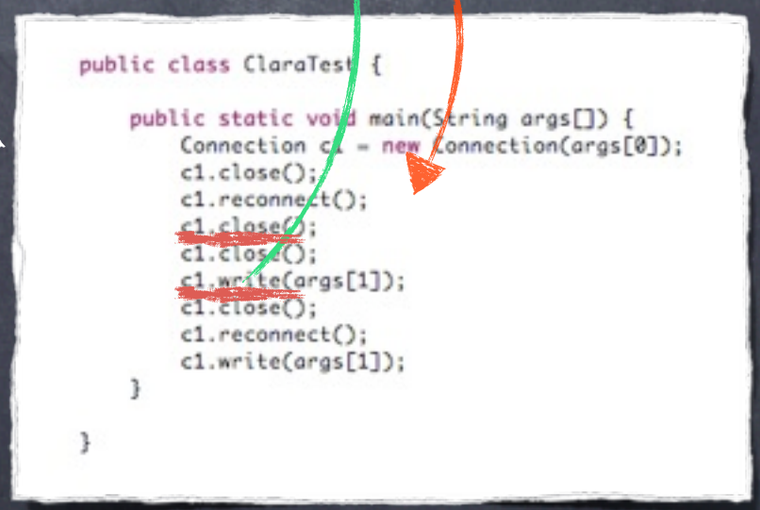
compile & weave

Quick Check

Clara

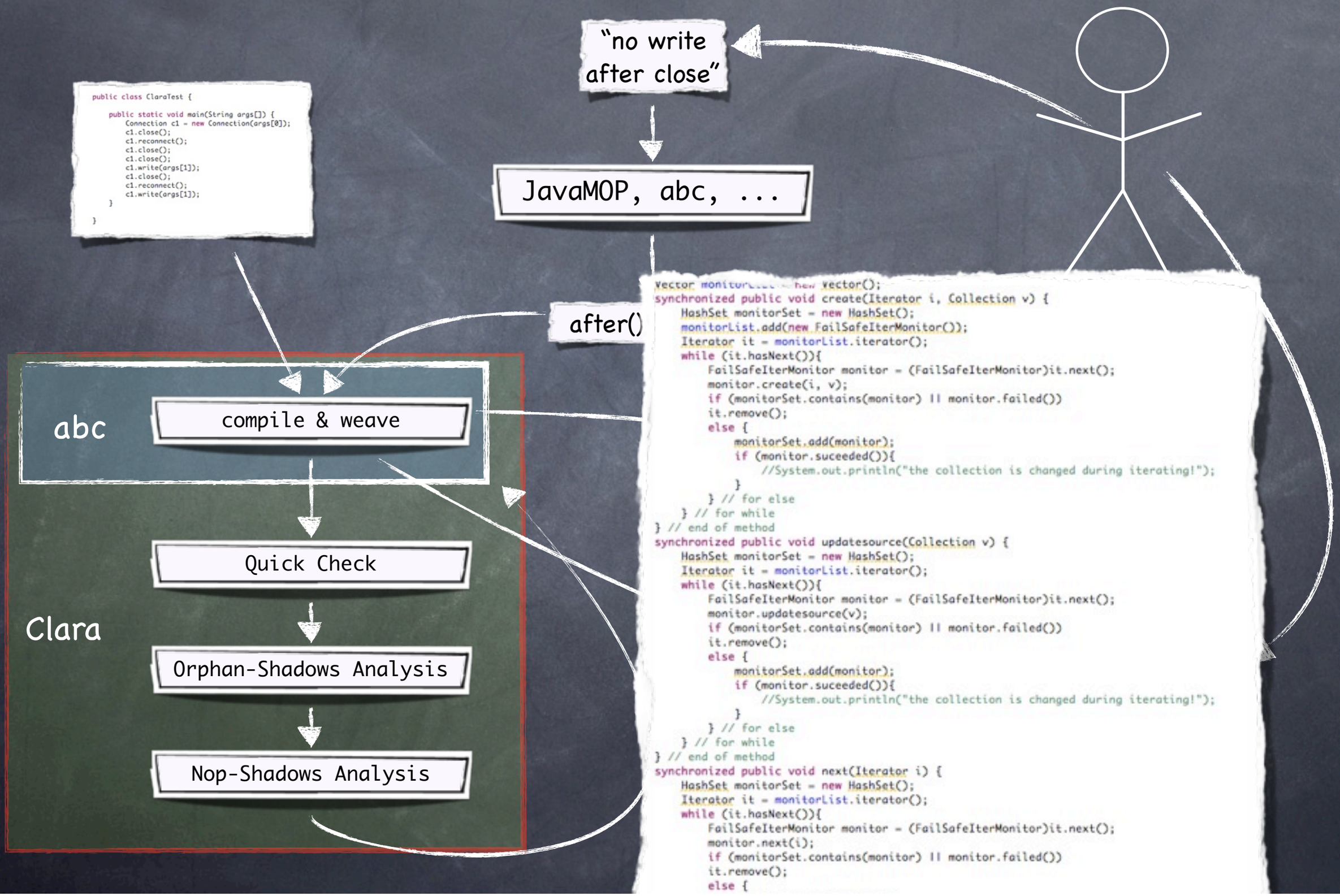
Orphan-Shadows Analysis

Nop-Shadows Analysis



9

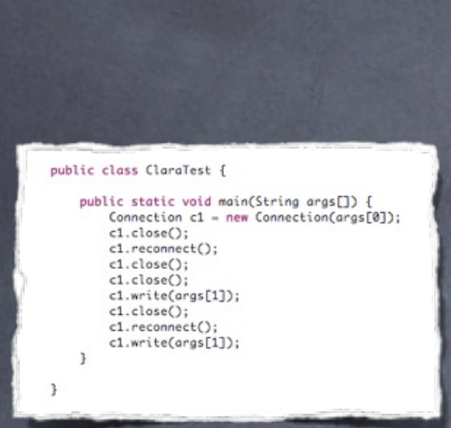
The Clara Framework




```
public class ClaraTest {

    public static void main(String args[]) {
        Connection c1 = new Connection(args[0]);
        c1.close();
        c1.reconnect();
        c1.close();
        c1.close();
        c1.write(args[1]);
        c1.close();
        c1.reconnect();
        c1.write(args[1]);
    }

}
```

```
Vector monitorList = new Vector();
synchronized public void create(Iterator i, Collection v) {
    HashSet monitorSet = new HashSet();
    monitorList.add(new FailSafeIterMonitor());
    Iterator it = monitorList.iterator();
    while (it.hasNext()){
        FailSafeIterMonitor monitor = (FailSafeIterMonitor)it.next();
        monitor.create(i, v);
        if (monitorSet.contains(monitor) || monitor.failed())
        it.remove();
        else {
            monitorSet.add(monitor);
            if (monitor.suceeded()){
                //System.out.println("the collection is changed during iterating!");
            }
        } // for else
    } // for while
} // end of method
synchronized public void updatesource(Collection v) {
    HashSet monitorSet = new HashSet();
    Iterator it = monitorList.iterator();
    while (it.hasNext()){
        FailSafeIterMonitor monitor = (FailSafeIterMonitor)it.next();
        monitor.updatesource(v);
        if (monitorSet.contains(monitor) || monitor.failed())
        it.remove();
        else {
            monitorSet.add(monitor);
            if (monitor.suceeded()){
                //System.out.println("the collection is changed during iterating!");
            }
        } // for else
    } // for while
} // end of method
synchronized public void next(Iterator i) {
    HashSet monitorSet = new HashSet();
    Iterator it = monitorList.iterator();
    while (it.hasNext()){
        FailSafeIterMonitor monitor = (FailSafeIterMonitor)it.next();
        monitor.next(i);
        if (monitorSet.contains(monitor) || monitor.failed())
        it.remove();
        else {
```

The Clara Framework



"no write after close"

JavaMOP, abc, ...

after(): call(...)

abc

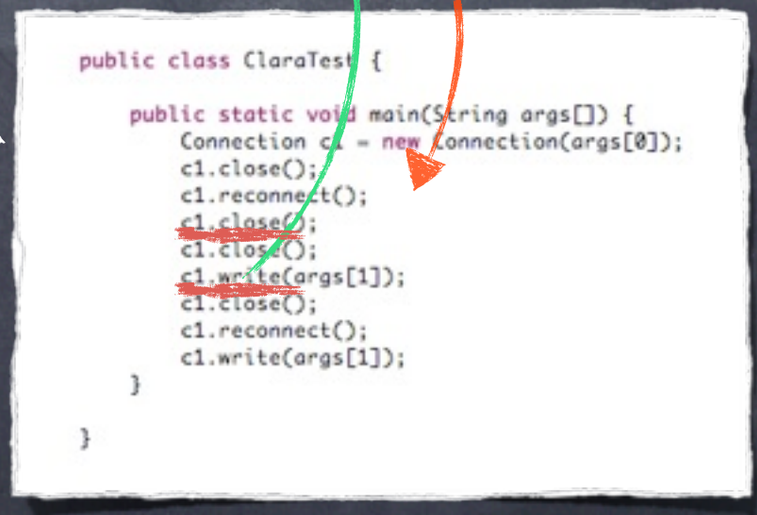
compile & weave

Quick Check

Clara

Orphan-Shadows Analysis

Nop-Shadows Analysis



9

Dependency State Machines

```
Set closed = new WeakIdentityHashSet();

after(Connection c) returning:
    call(* Connection.close()) && target(c) {
    closed.add(c);
}

after(Connection c) returning:
    call(* Connection.reconnect()) && target(c) {
    closed.remove(c);
}

after(Connection c) returning:
    call(* Connection.write(..)) && target(c) {
    if(closed.contains(c))
        error("May not write to "+c+", as it is closed!");
}
```


10

Dependency State Machines

abstract

concrete

```
Set closed = new WeakIdentityHashSet();

dependent after disconnect(Connection c) returning:
    call(* Connection.close()) && target(c) {
    closed.add(c);
}

dependent after reconnect(Connection c) returning:
    call(* Connection.reconnect()) && target(c) {
    closed.remove(c);
}

dependent after write(Connection c) returning:
    call(* Connection.write(..)) && target(c) {
    if(closed.contains(c))
        error("May not write to "+c+", as it is closed!");
}
```


11

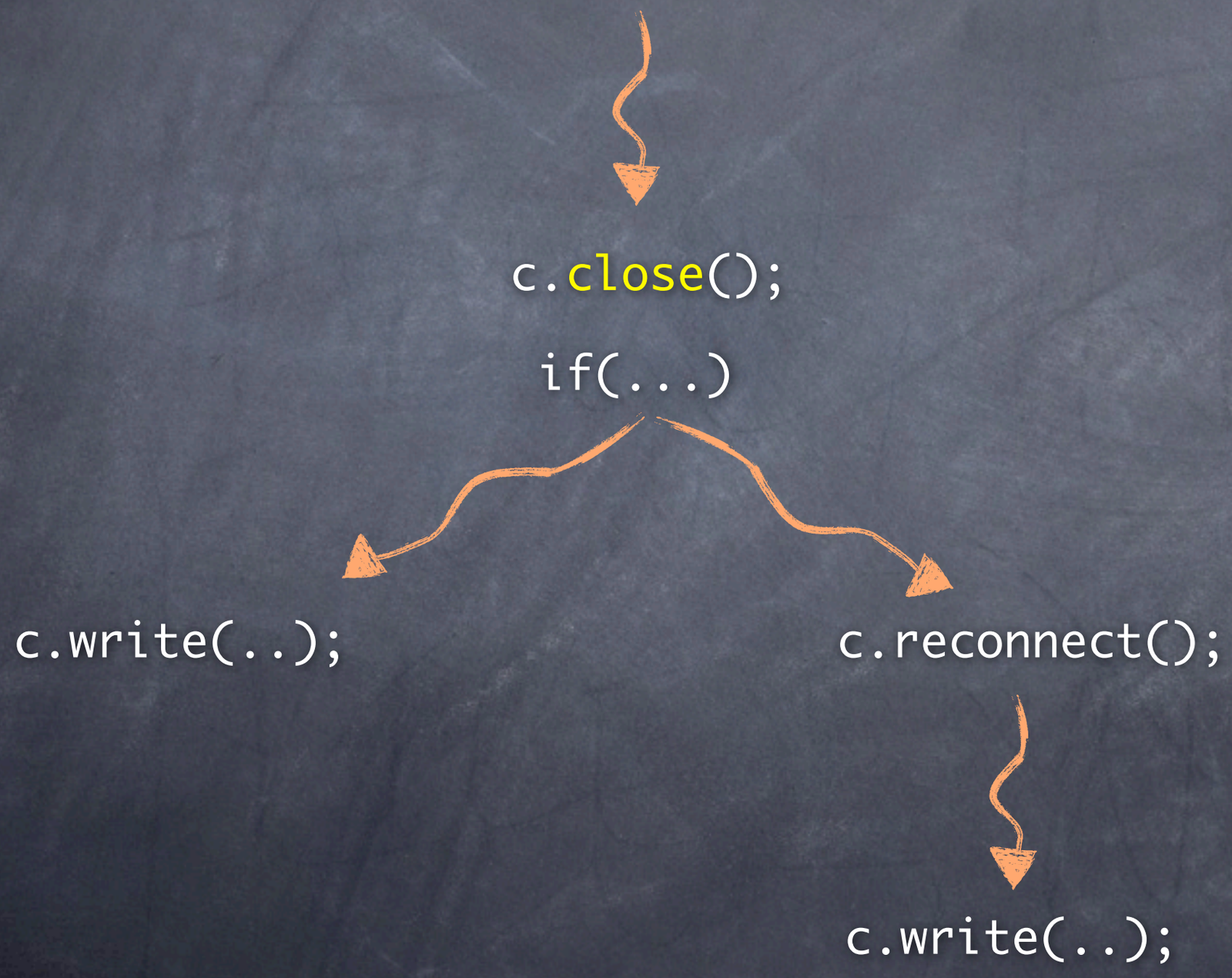
```
Set closed = new WeakIdentityHashSet();

dependent after disconnect(Connection c) returning:
    call(* Connection.close()) && target(c) {
    closed.add(c);
}

dependent after reconnect(Connection c) returning:
    call(* Connection.reconnect()) && target(c) {
    closed.remove(c);
}

dependent after write(Connection c) returning:
    call(* Connection.write(..)) && target(c) {
    if(closed.contains(c))
        error("May not write to "+c+", as it is closed!");
}
```

```
dependency{
    disconnect, write, reconnect;
    initial   connected: disconnect -> connected,
                         write -> connected,
                         reconnect -> connected,
                         disconnect -> disconnected;
              disconnect: disconnect -> disconnected,
                         write -> error;
    final     error: write -> error;
}
```

finite-state property

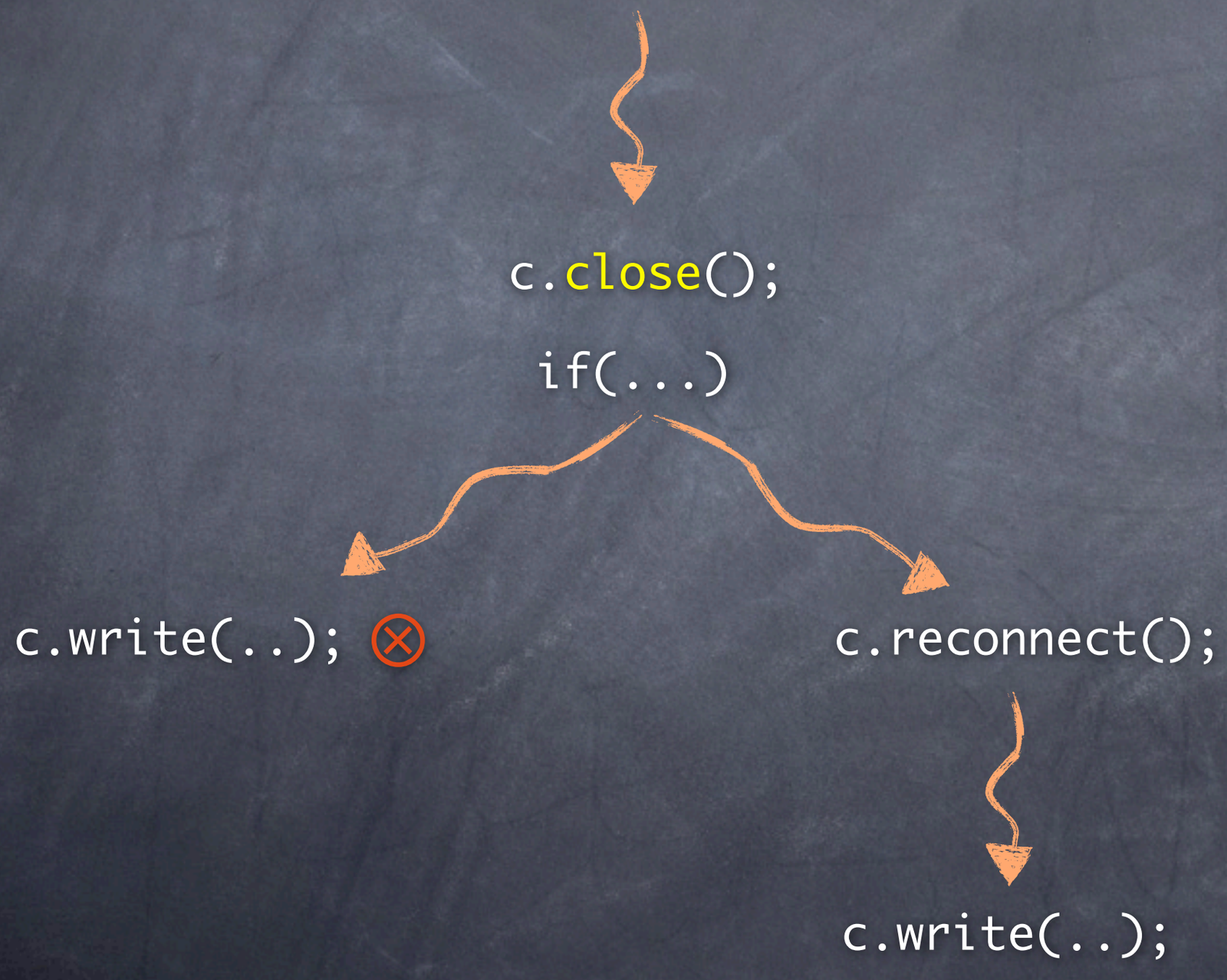
12

Semantics of Dependency State Machines



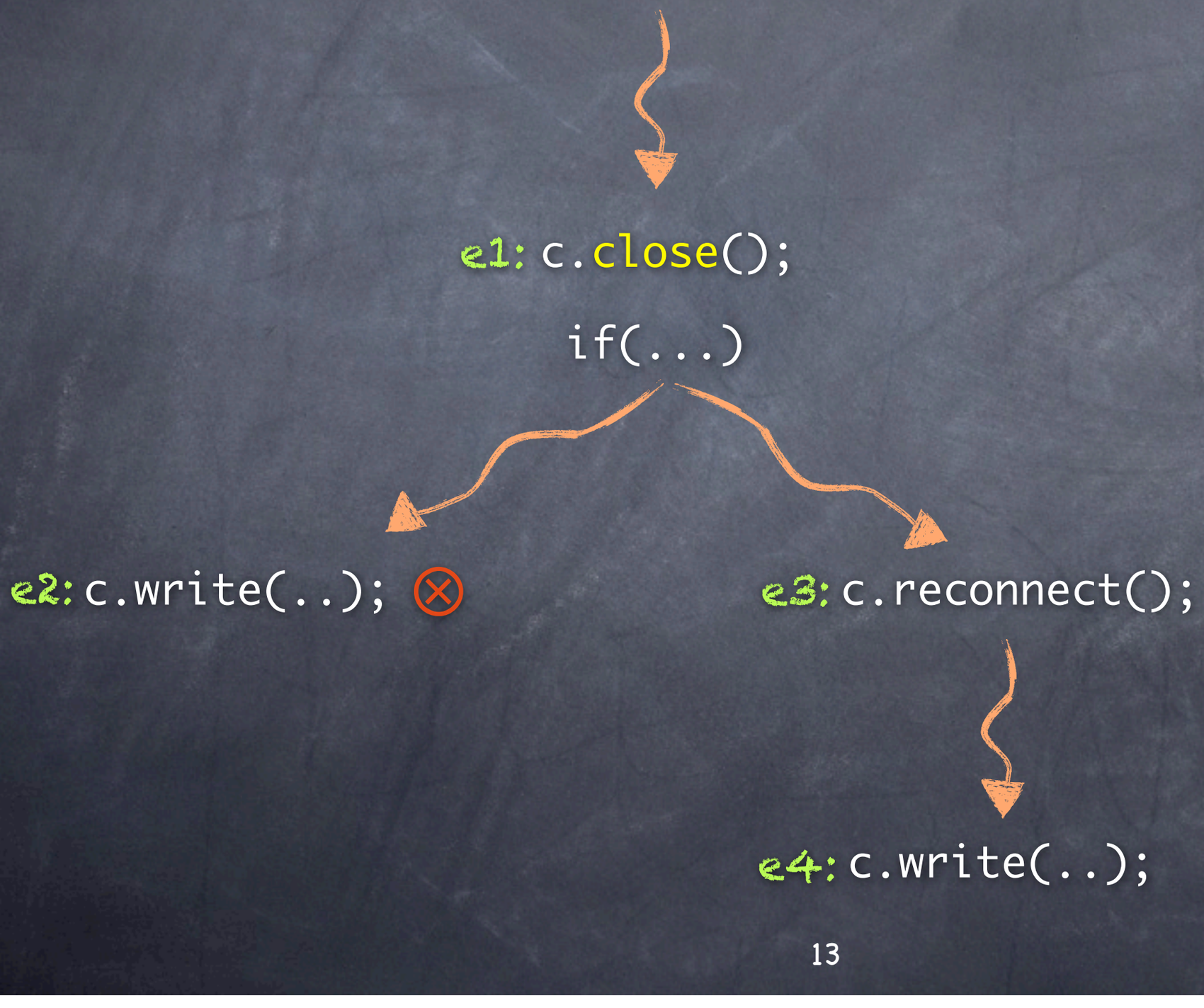
13

Semantics of Dependency State Machines



13

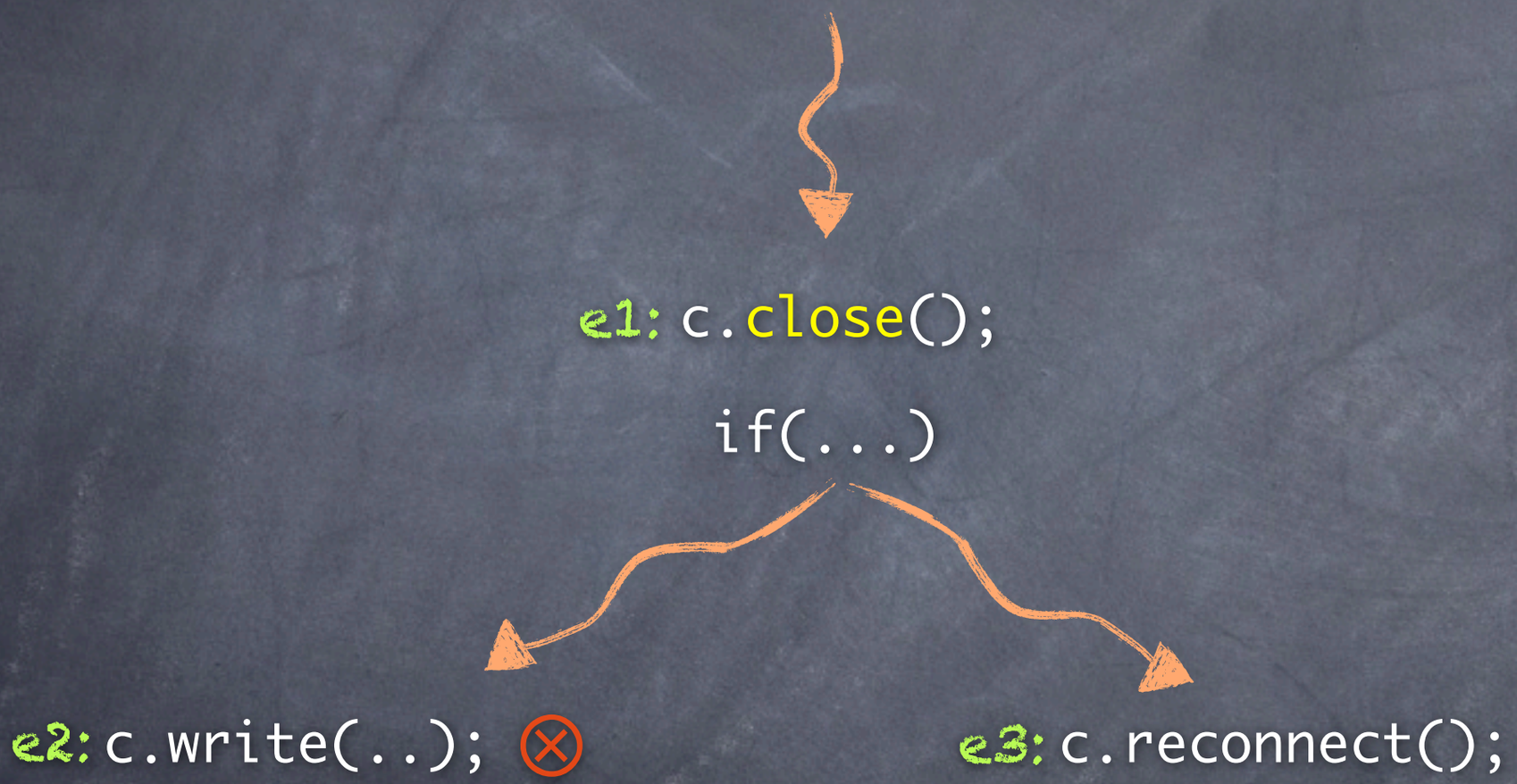
Semantics of Dependency State Machines



13

Semantics of Dependency State Machines

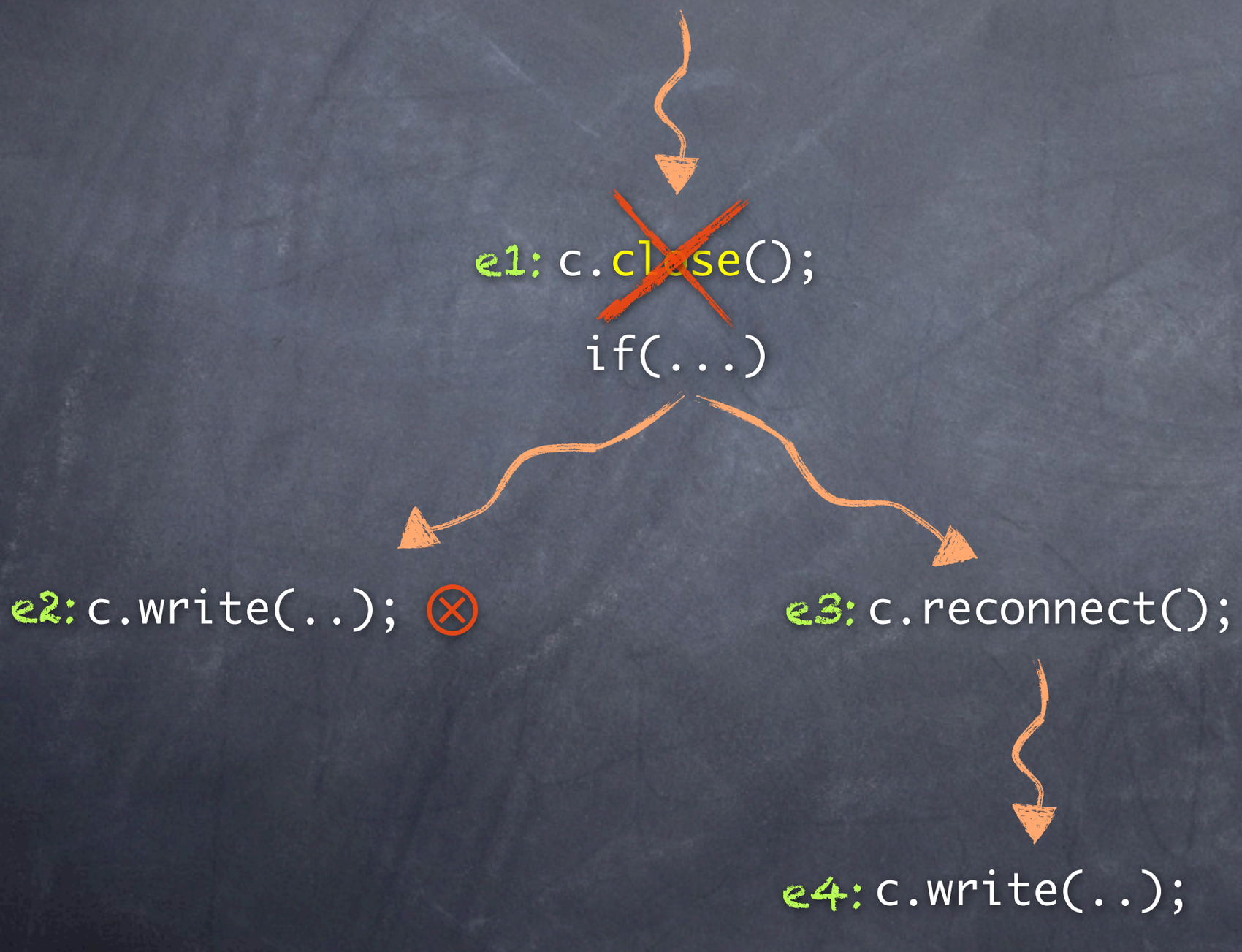
advice "close" must execute at e1 if



13

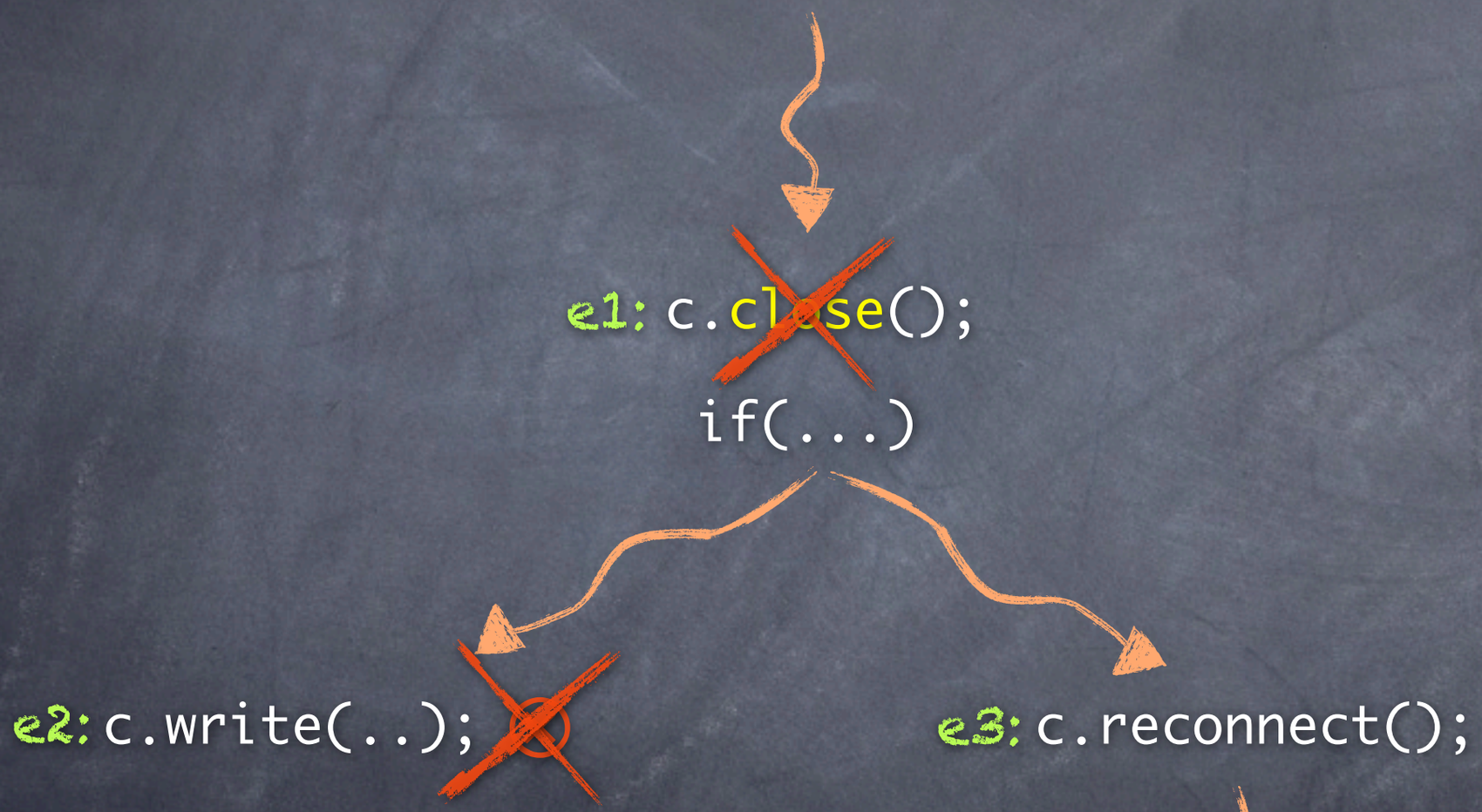
Semantics of Dependency State Machines

advice "close" must execute at e1 if omitting "close" at e1



13

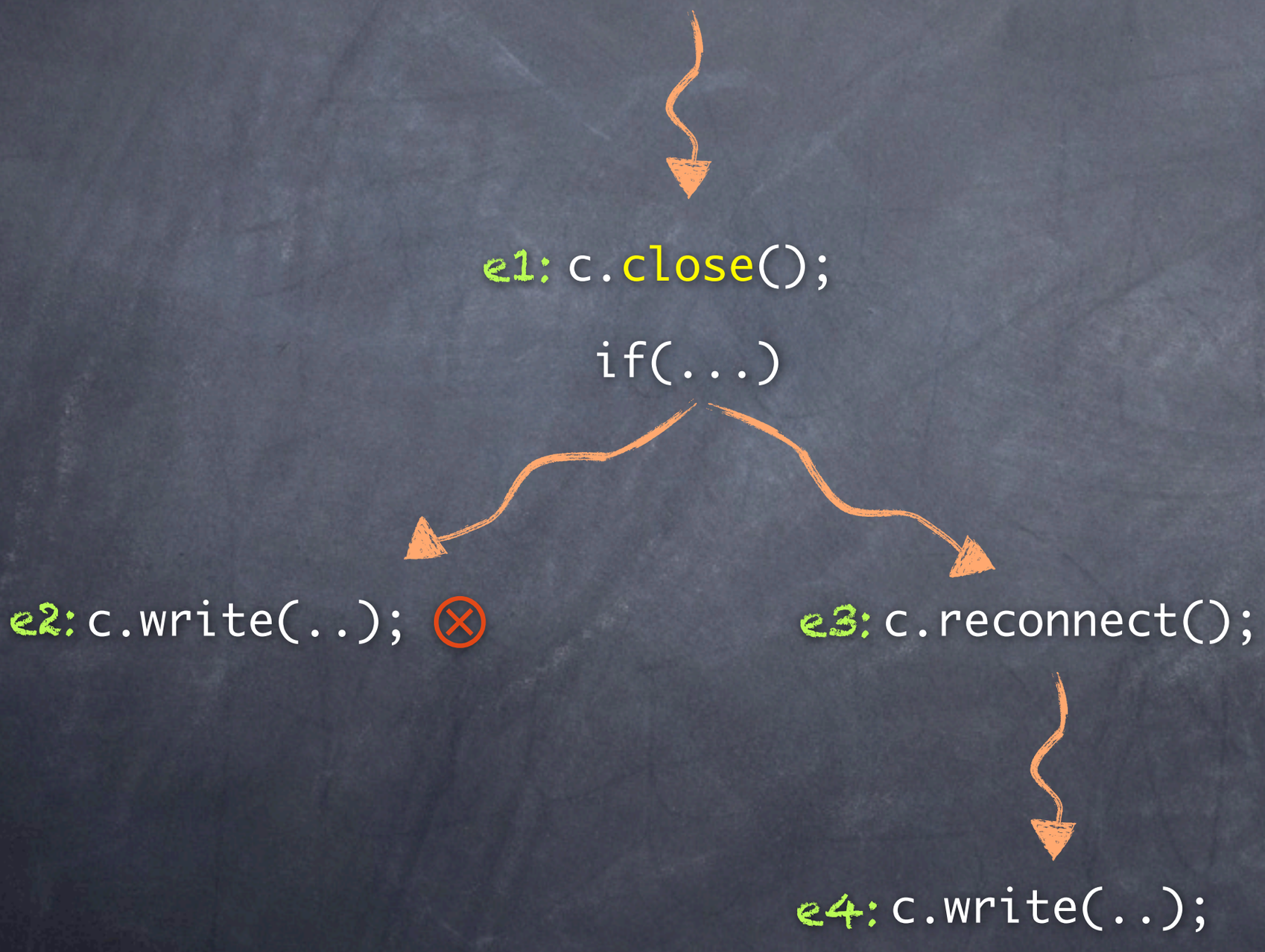
Semantics of Dependency State Machines



advice "close" must execute at e1 if omitting "close" at e1 may change the events at which a DSM reaches an error state

13

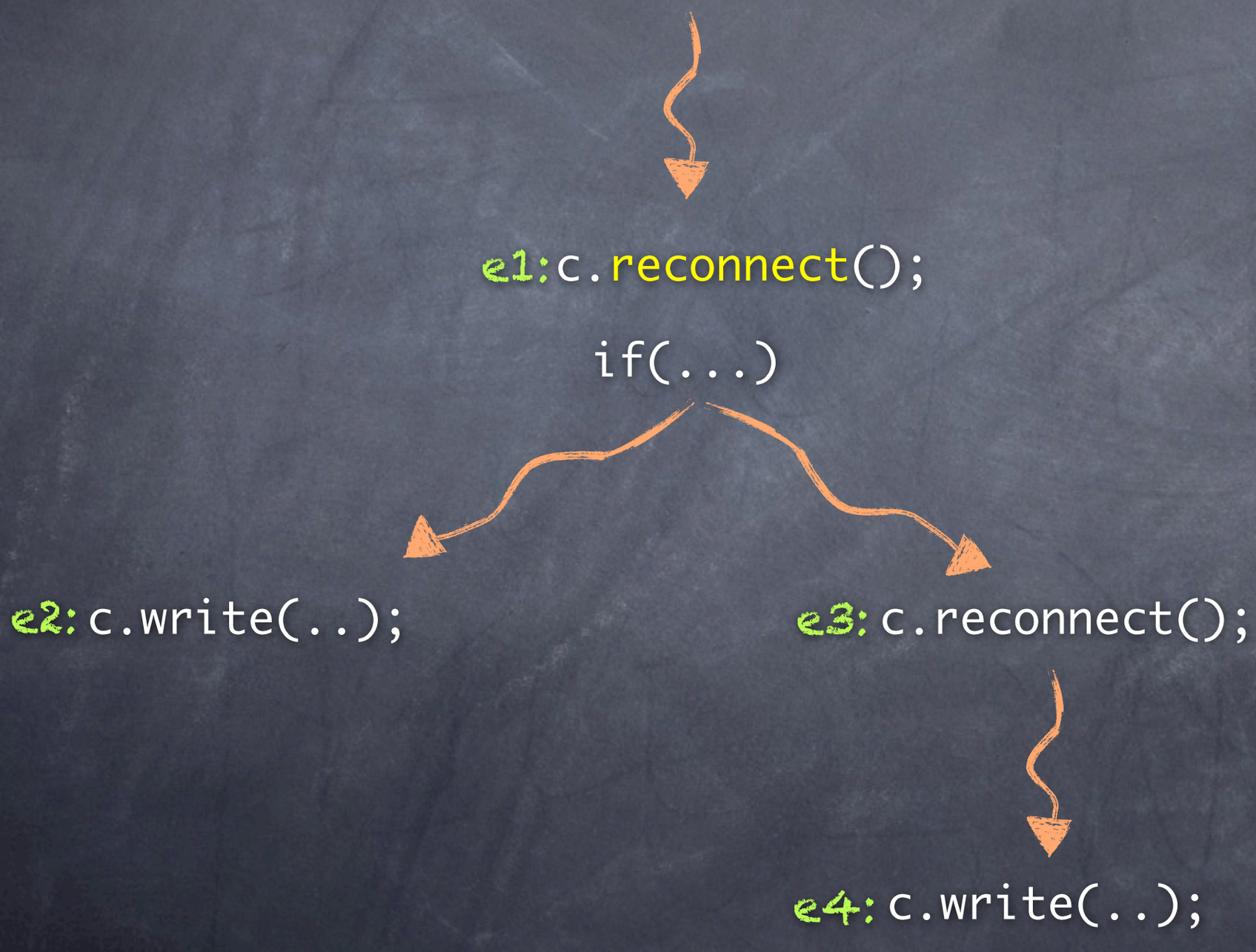
Semantics of Dependency State Machines



advice "close" must execute at e1 if omitting "close" at e1 may change the events at which a DSM reaches an error state

13

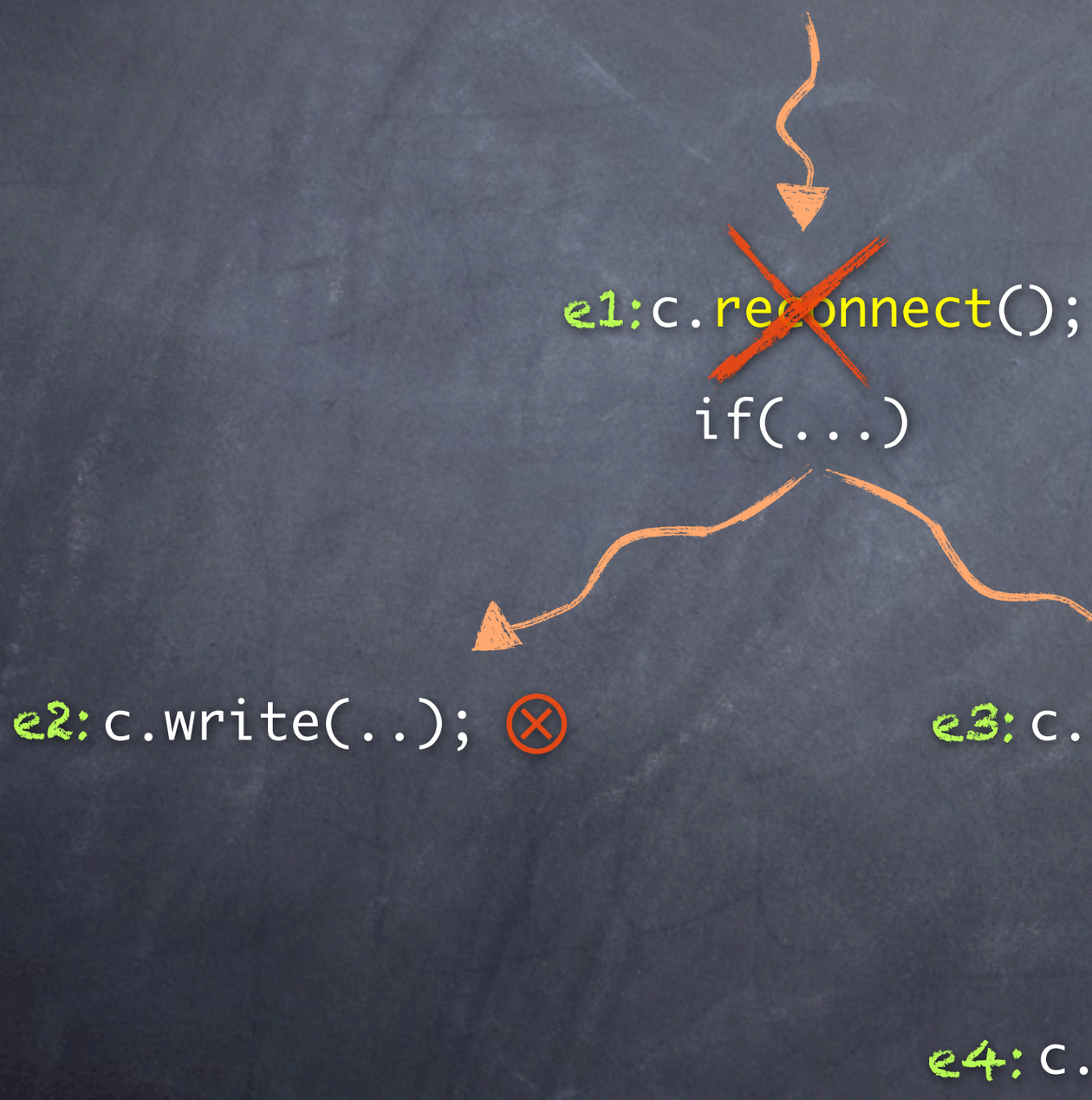
Inverse case: match-preventing events



advice "reconnect" must execute at e1 if omitting "reconnect" at e1 may change the events at which a DSM reaches an error state

14

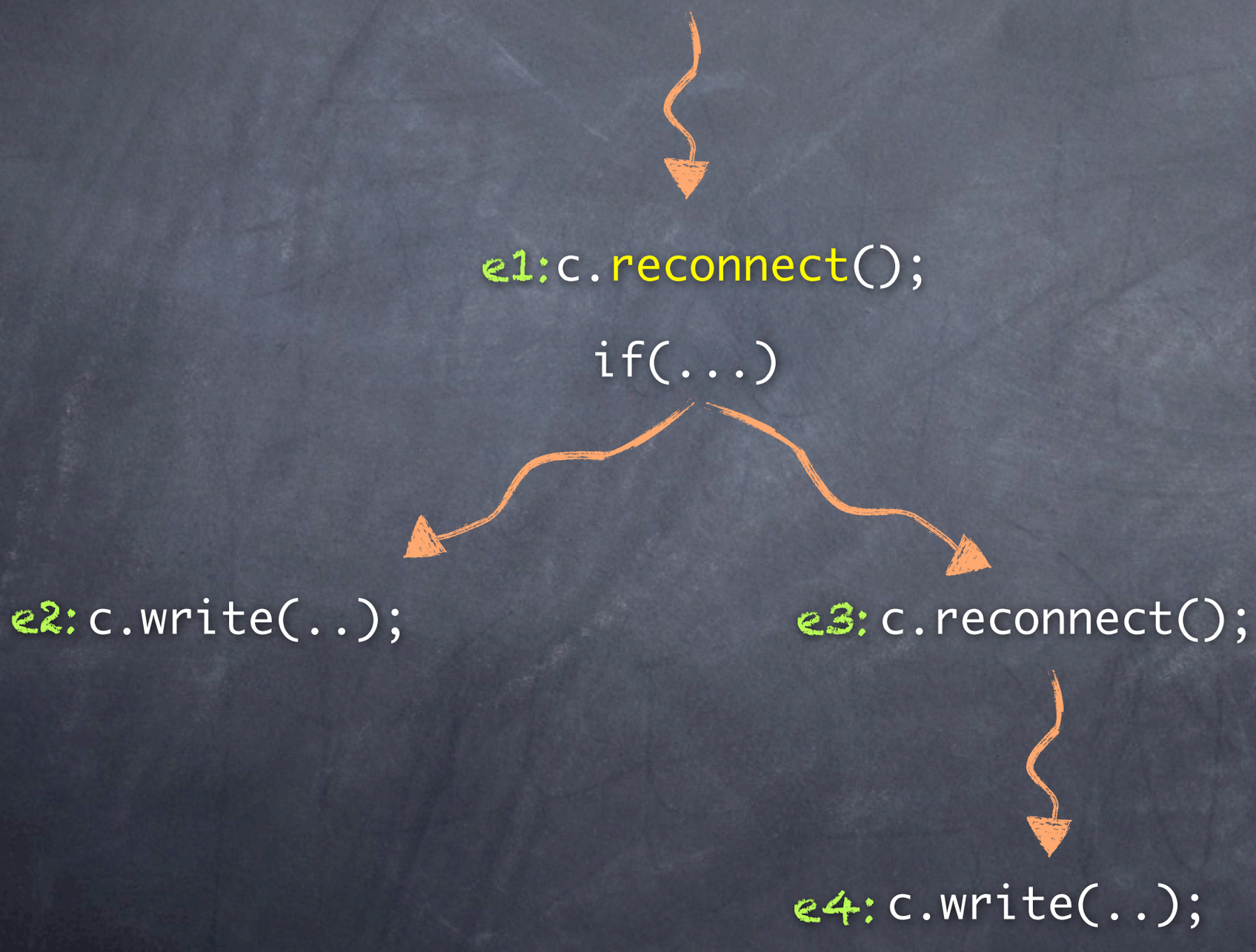
Inverse case: match-preventing events



advice "reconnect" must execute at e1 if omitting "reconnect" at e1 may change the events at which a DSM reaches an error state

14

Inverse case: match-preventing events



advice "reconnect" must execute at e1 if omitting "reconnect" at e1 may change the events at which a DSM reaches an error state

14

Variable bindings matter

```
c.close();
```

```
e1:c.reconnect();

if(...)
```

```
e2: c2.write(..);
```

```
e3:c.reconnect();
```

```
e4:c.write(..);
```


15

Variable bindings matter

```
c.close();
```

```
e1:c.reconnect();

if(...)
```

```
e3: c.reconnect();
```

```
e4: c.write(..);
```


15

Variable bindings matter

```
c.close();
```

```
e1:c.reconnect();
```

```
if(...)
```

```
e3: c.reconnect();
```

```
e4: c.write(..);
```


15

AspectJ matching function:

$$match: \quad \mathcal{A} \times \mathcal{J} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}.$$

Dependent advice allow family of possible optimized matching functions:

$$stateMatch: \quad \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$stateMatch(a, \hat{t}, i) :=$$
$$\text{let } \beta = match(a, e) \text{ in}$$
$$\begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in groundTraces(\hat{t}) \text{ such that } necessaryShadow(a, t, i) \\ \bot & \text{otherwise} \end{cases}$$

Optimization goal: return ⊥ whenever possible but β whenever necessary

16

AspectJ matching function:

$$match: \quad \mathcal{A} \times \mathcal{J} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}.$$

Dependent advice allow family of possible optimized matching functions:

$$stateMatch: \quad \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$stateMatch(a, \hat{t}, i) :=$$

$$\text{let } \beta = match(a, e) \text{ in}$$

$$\begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in groundTraces(\hat{t}) \text{ such that } necessaryShadow(a, t, i) \\ \bot & \text{otherwise} \end{cases}$$

Optimization goal: return ⊥ whenever possible but β whenever necessary

16

$$stateMatch : \quad \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$stateMatch(a, \hat{t}, i) :=$$

$$\text{let } \beta = match(a, e) \text{ in}$$

$$\begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in groundTraces(\hat{t}) \text{ such that } necessaryShadow(a, t, i) \\ \bot & \text{otherwise} \end{cases}$$

Any **sound** optimization must satisfy:

$$\forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+.\ \forall i \leq n \in \mathbb{N}.$$

$$matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_i t_{i+1} \ldots t_n) \neq matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n)$$

$$\implies necessaryShadow(t_i, t, i)$$

17

More on the semantics

CLARA also supports Collaborative Runtime Verification, which distributes instrumentation overhead among multiple users; and ranking heuristics, which aid programmers in inspecting remaining instrumentation manually [13] [2, Ch. 6 & 7]. Space limitations preclude us from discussing ranking and Collaborative Runtime Verification here.

CLARA is freely available as free software at `http://bodden.de/clara/`, along with extensive documentation, the first author's dissertation [2], which describes CLARA in detail, and benchmarks and benchmark results.

We next describe the syntax and semantics of Dependency State Machines, the key abstraction of CLARA. This abstraction allows CLARA to decouple runtime monitor implementations from static analyses.

3 Syntax and Semantics of Dependency State Machines

Dependency State Machines extend the AspectJ language to include semantic information about relationships between different pieces of advice. Runtime verification tools which generate AspectJ aspects can use this extension to produce augmented aspects. CLARA can reason about the augmented aspects to prove that programs never violate monitored properties or to generate optimized code.

3.1 Syntax

Our extensions modify the AspectJ grammar in two ways: they add syntax for defining Dependent Advice [14] and Dependency State Machines. The idea of Dependent Advice is that pieces of monitoring advice are often inter-dependent in the sense that the execution of one piece of advice only has an effect when executing before or after another piece of advice, on the same objects. Dependency State Machines allow programmers to make these dependencies explicit so that static analyses can exploit them. Our explanations below refer to the ConnectionClosed example in Figure 2.

The `dependent` modifier flags advice to CLARA for potential optimization; such advice may be omitted from program locations at which it provably has no effect on the state of the runtime monitor. Dependent advice must be named. Lines 4, 7 and 10 all define dependent advice.

The Dependency State Machines extension enables users to specify state machines which relate different pieces of dependent advice. Dependency State Machine declarations define state machines by including a list of edges between states and an alphabet; each edge is labelled with a member of the alphabet. CLARA infers the set of states from the declared edges. Line 16 declares the state machine's alphabet: `{disconn, write, reconn}`. Every symbol in the alphabet references dependent advice from the same aspect. Lines 17–19 enumerate, for each state, a (potentially empty) list of outgoing transitions. An entry `"s1: t -> s2"` means "there exists a `t`-transition from `s1` to `s2`". Users can also mark states as `initial` or `final` (error states). Final states denote states

This Paper

+ upcoming Journal Paper

VERIFYING FINITE-STATE PROPERTIES
OF LARGE-SCALE PROGRAMS

by

Eric Bodden

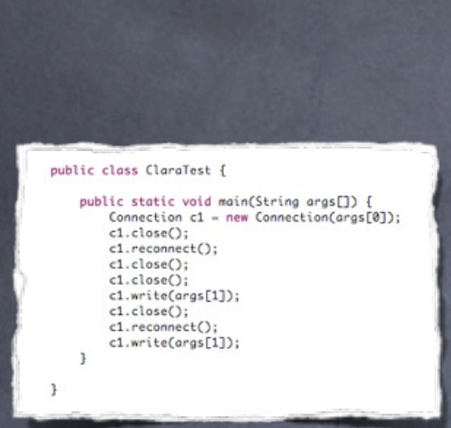
School of Computer Science
McGill University, Montréal

June 2009

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Thesis:
proves that analyses
obey "necessaryShadow"

The Clara Framework



"no write after close"

JavaMOP, abc, ...

after(): call(...)

abc

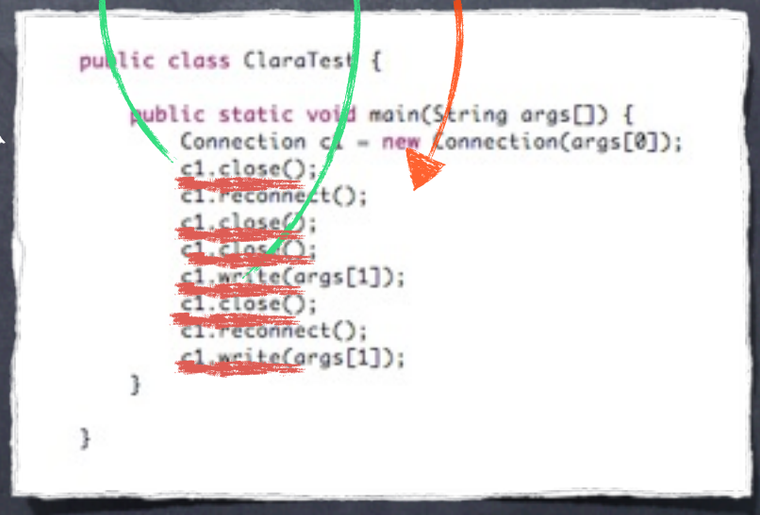
compile & weave

Quick Check

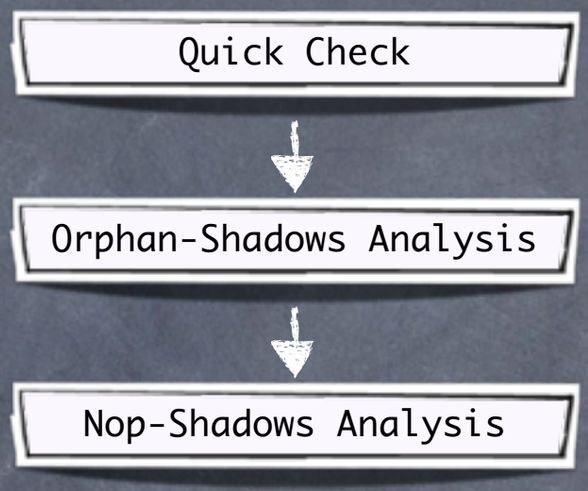
Clara

Orphan-Shadows Analysis

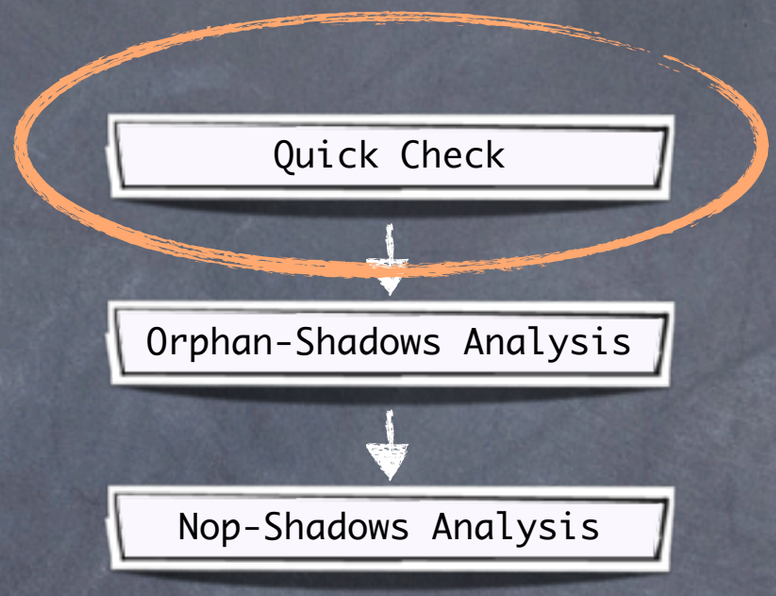
Nop-Shadows Analysis



19



20

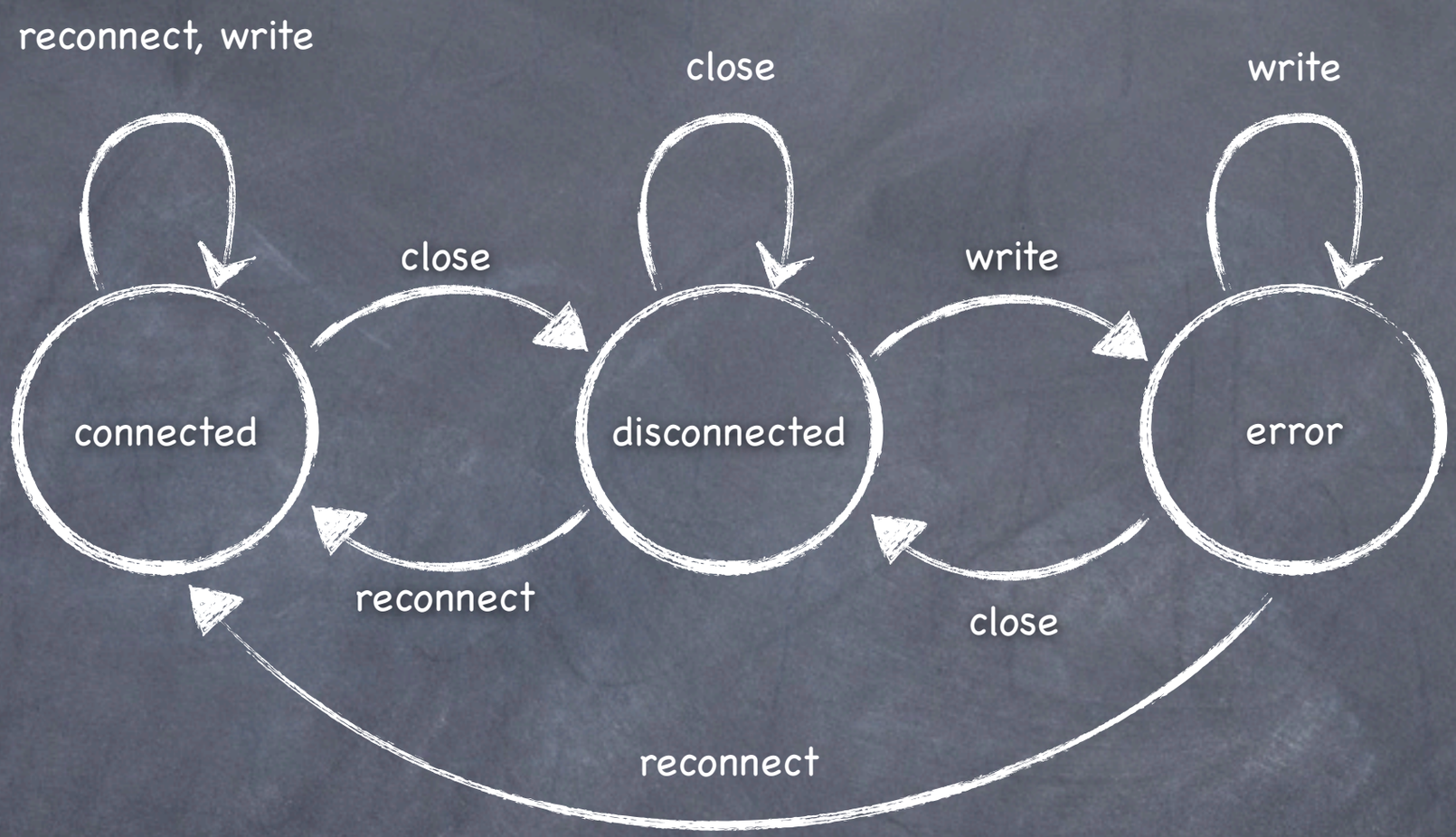


20

Quick Check

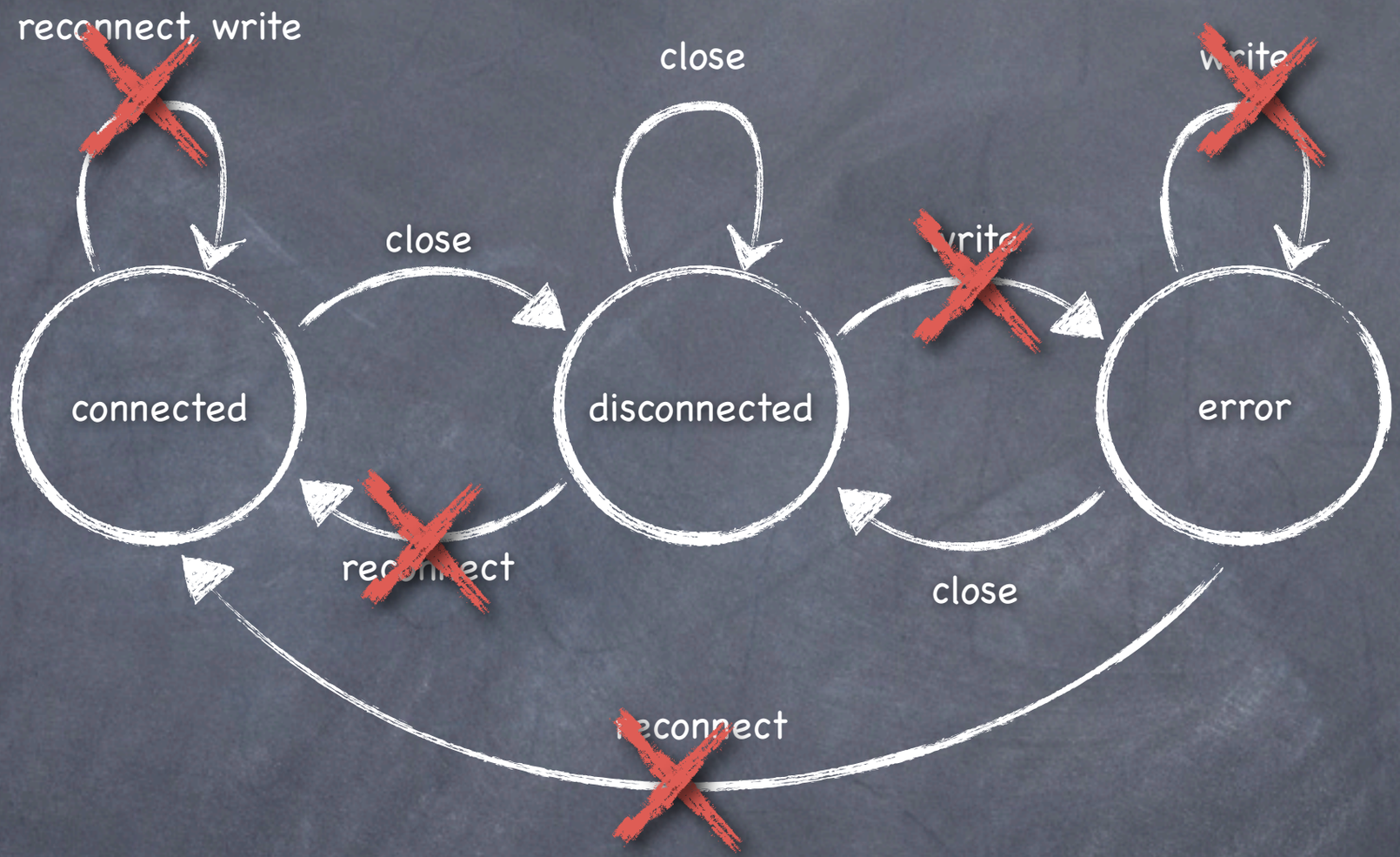
```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Quick Check



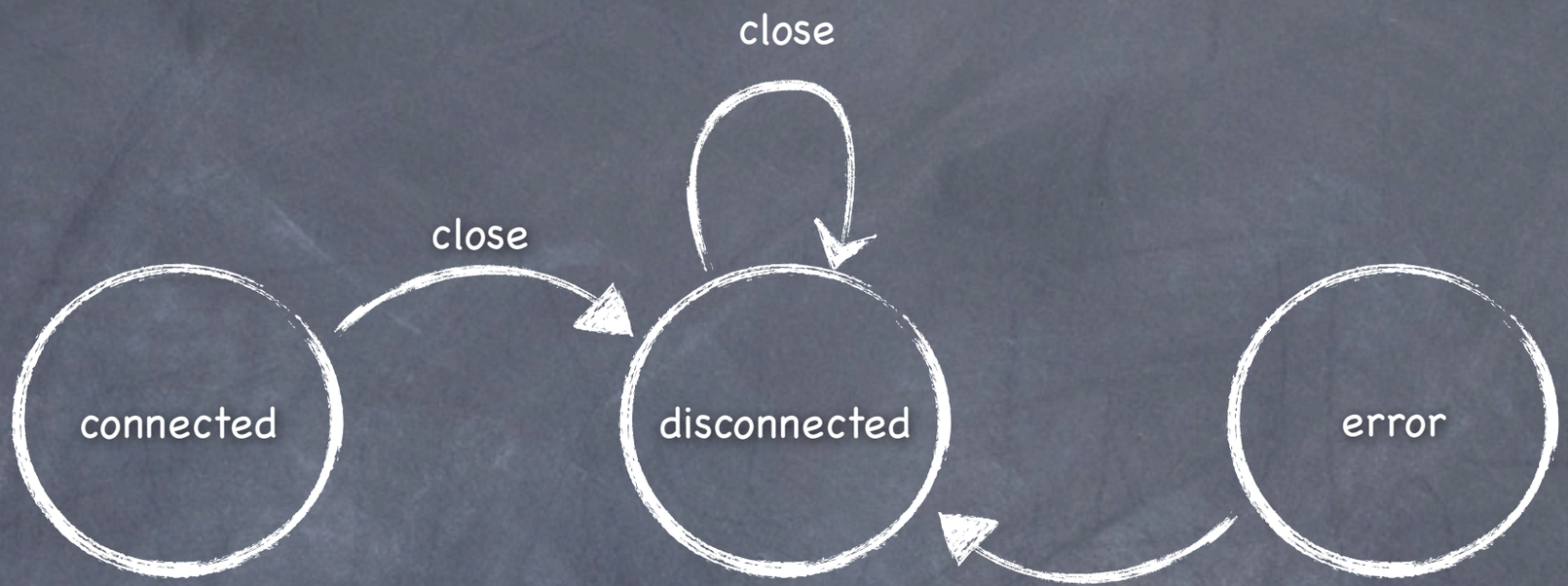

```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Quick Check



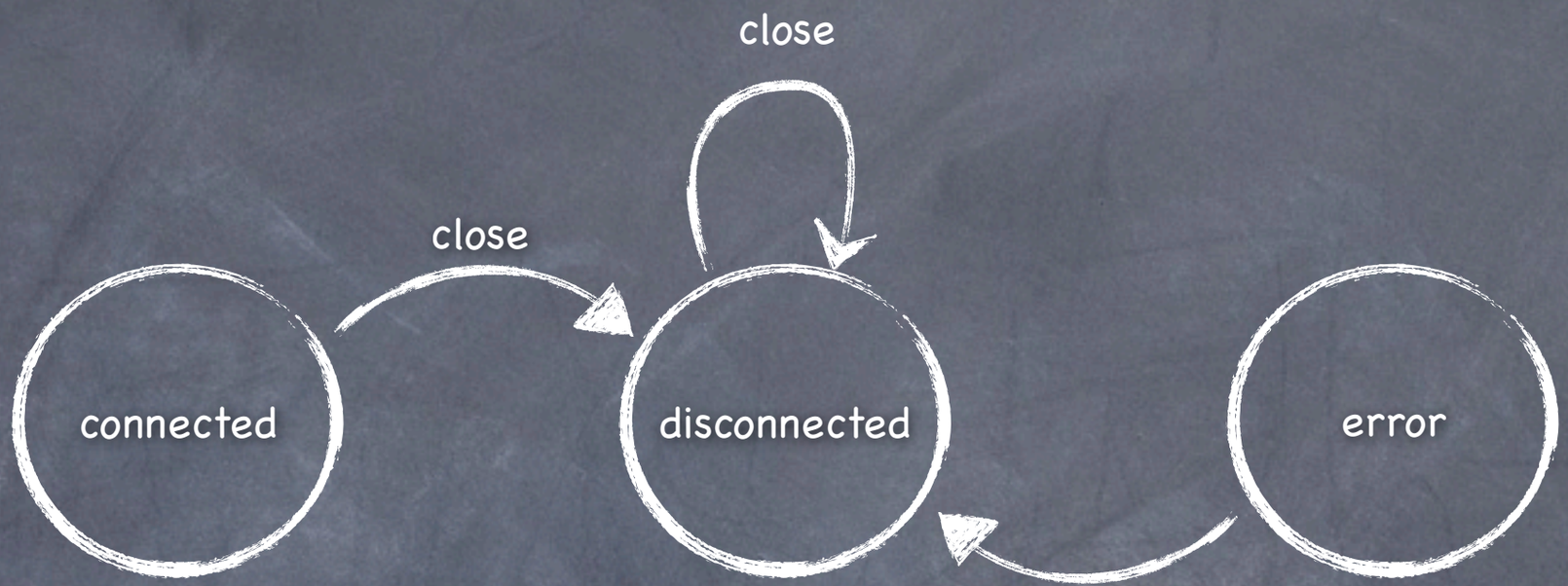

```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Quick Check



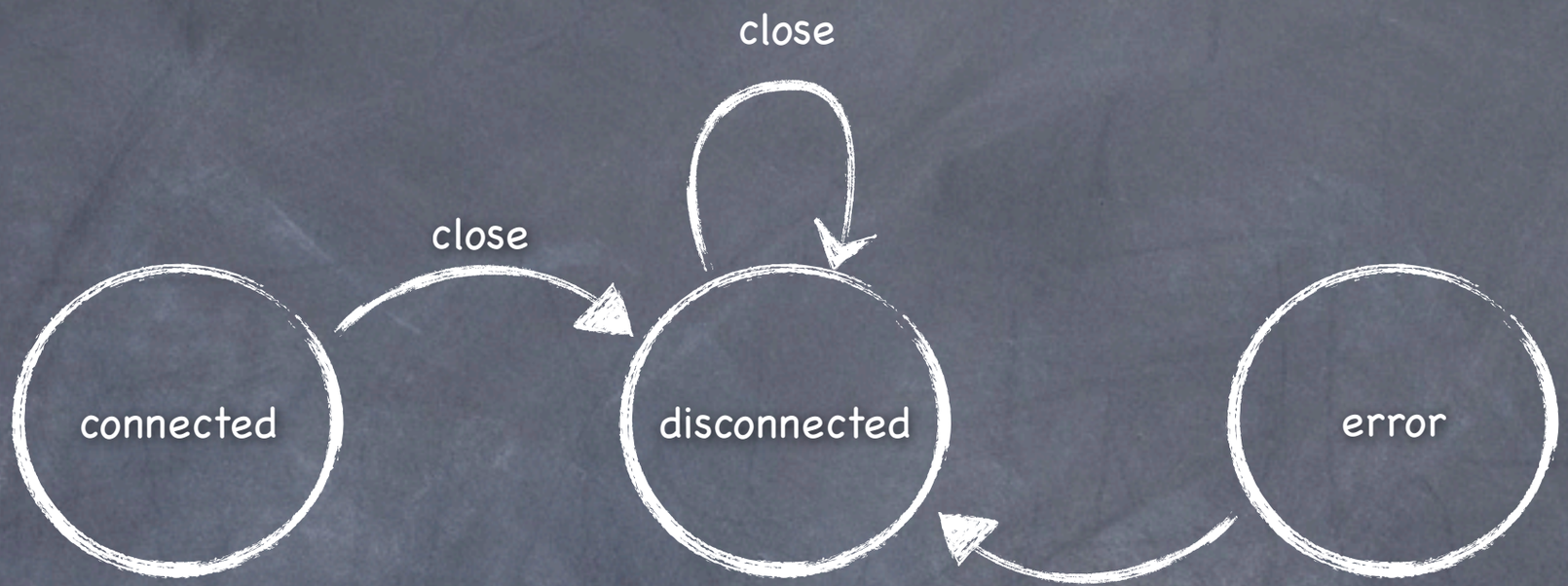

```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Quick Check




```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Quick Check




```
public static void main(String[] args) {
    Connection c = new Connection();
    while(c.hasMoreData()) {
        System.err.println(c.read());
    }
    c.close();
}
```

Proving QuickCheck sound

$$necessaryTransitionQC(a, t, i) := a \in symbolsThatNeedMonitoring.$$

symbolsThatNeedMonitoring:
- ∅ if QuickCheck succeeds
- Σ otherwise

$$\forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+.\ \forall i \leq n \in \mathbb{N}.$$
$$matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_i t_{i+1} \ldots t_n) \neq matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n)$$
$$\implies necessaryShadow(t_i, t, i)$$

22

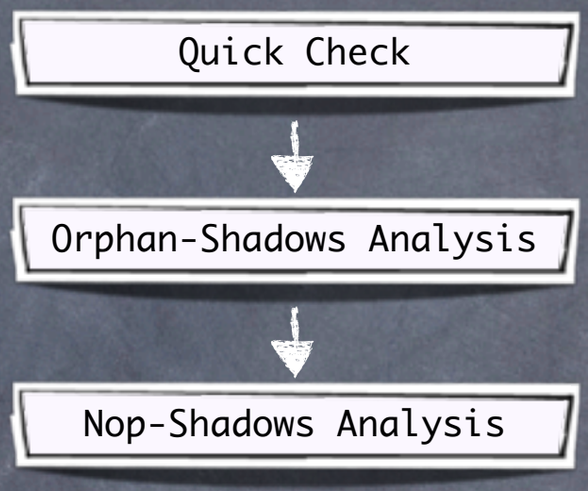
Proving QuickCheck sound

$$necessaryTransitionQC(a, t, i) := a \in symbolsThatNeedMonitoring.$$

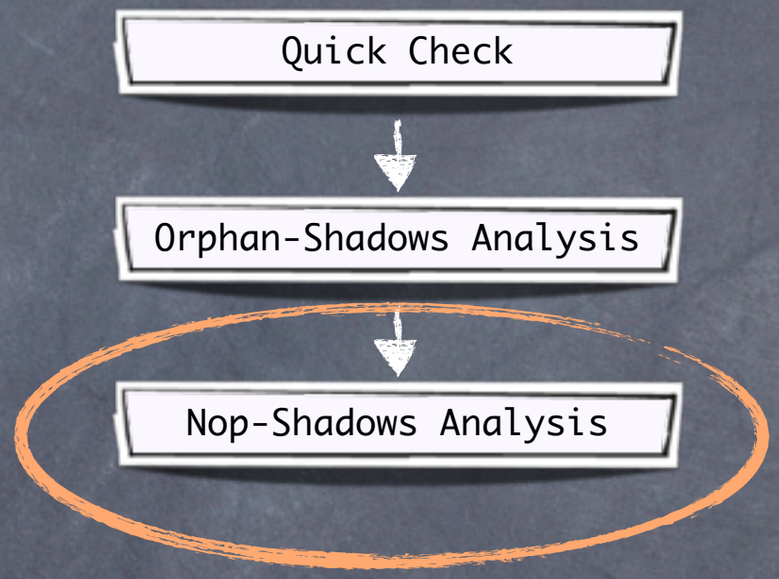
symbolsThatNeedMonitoring:
- ∅ if QuickCheck succeeds
- Σ otherwise

$$\forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+.\ \forall i \leq n \in \mathbb{N}.$$
$$matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_i t_{i+1} \ldots t_n) \neq matches_{\mathcal{L}(\mathcal{M})}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n)$$
$$\Longrightarrow a \in symbolsThatNeedMonitoring$$

22



23

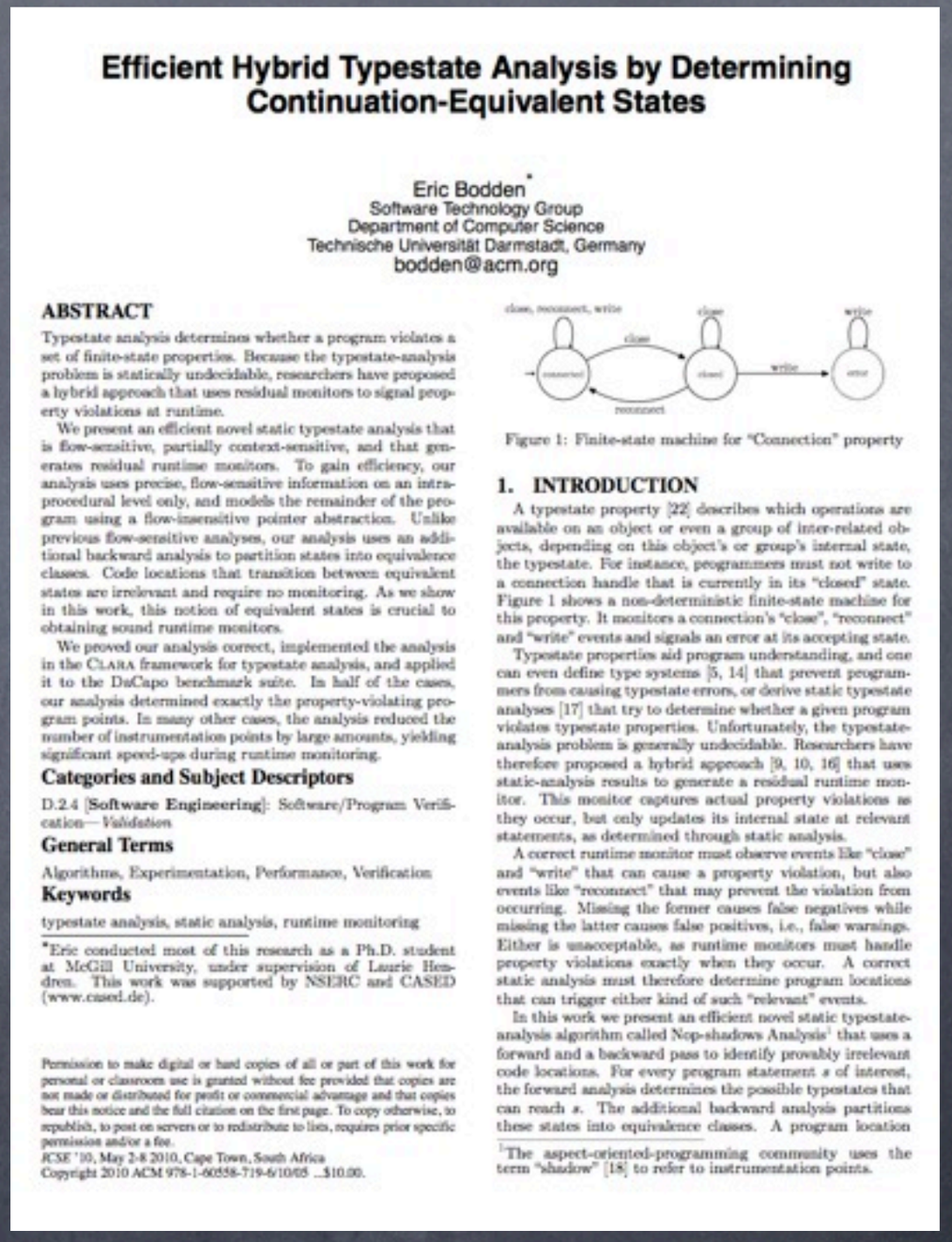


23

More details...

Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States (Eric Bodden)

In ICSE '10: International Conference on Software Engineering, pages 5–14, ACM, 2010.



Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States

Eric Bodden*
Software Technology Group
Department of Computer Science
Technische Universität Darmstadt, Germany
bodden@acm.org

ABSTRACT

Typestate analysis determines whether a program violates a set of finite-state properties. Because the typestate-analysis problem is statically undecidable, researchers have proposed a hybrid approach that uses residual monitors to signal property violations at runtime.

We present an efficient novel static typestate analysis that is flow-sensitive, partially context-sensitive, and that generates residual runtime monitors. To gain efficiency, our analysis uses precise, flow-sensitive information on an intra-procedural level only, and models the remainder of the program using a flow-insensitive pointer abstraction. Unlike previous flow-sensitive analyses, our analysis uses an additional backward analysis to partition states into equivalence classes. Code locations that transition between equivalent states are irrelevant and require no monitoring. As we show in this work, this notion of equivalent states is crucial to obtaining sound runtime monitors.

We proved our analysis correct, implemented the analysis in the CLARA framework for typestate analysis, and applied it to the DaCapo benchmark suite. In half of the cases, our analysis determined exactly the property-violating program points. In many other cases, the analysis reduced the number of instrumentation points by large amounts, yielding significant speed-ups during runtime monitoring.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Algorithms, Experimentation, Performance, Verification

Keywords

typestate analysis, static analysis, runtime monitoring

*Eric conducted most of this research as a Ph.D. student at McGill University, under supervision of Laurie Hendren. This work was supported by NSERC and CASED (www.cased.de).

ICSE '10, May 2-8 2010, Cape Town, South Africa

Figure 1: Finite-state machine for "Connection" property

1. INTRODUCTION

A typestate property [22] describes which operations are available on an object or even a group of inter-related objects, depending on this object's or group's internal state, the typestate. For instance, programmers must not write to a connection handle that is currently in its "closed" state. Figure 1 shows a non-deterministic finite-state machine for this property. It monitors a connection's "close", "reconnect" and "write" events and signals an error at its accepting state.

Typestate properties aid program understanding, and one can even define type systems [5, 14] that prevent programmers from causing typestate errors, or derive static typestate analyses [17] that try to determine whether a given program violates typestate properties. Unfortunately, the typestate-analysis problem is generally undecidable. Researchers have therefore proposed a hybrid approach [9, 10, 16] that uses static-analysis results to generate a residual runtime monitor. This monitor captures actual property violations as they occur, but only updates its internal state at relevant statements, as determined through static analysis.

A correct runtime monitor must observe events like "close" and "write" that can cause a property violation, but also events like "reconnect" that may prevent the violation from occurring. Missing the former causes false negatives while missing the latter causes false positives, i.e., false warnings. Either is unacceptable, as runtime monitors must handle property violations exactly when they occur. A correct static analysis must therefore determine program locations that can trigger either kind of such "relevant" events.

In this work we present an efficient novel static typestate-analysis algorithm called Nop-shadows Analysis[1] that uses a forward and a backward pass to identify provably irrelevant code locations. For every program statement s of interest, the forward analysis determines the possible typestates that can reach s. The additional backward analysis partitions these states into equivalence classes. A program location

[1]The aspect-oriented-programming community uses the term "shadow" [18] to refer to instrumentation points.

24

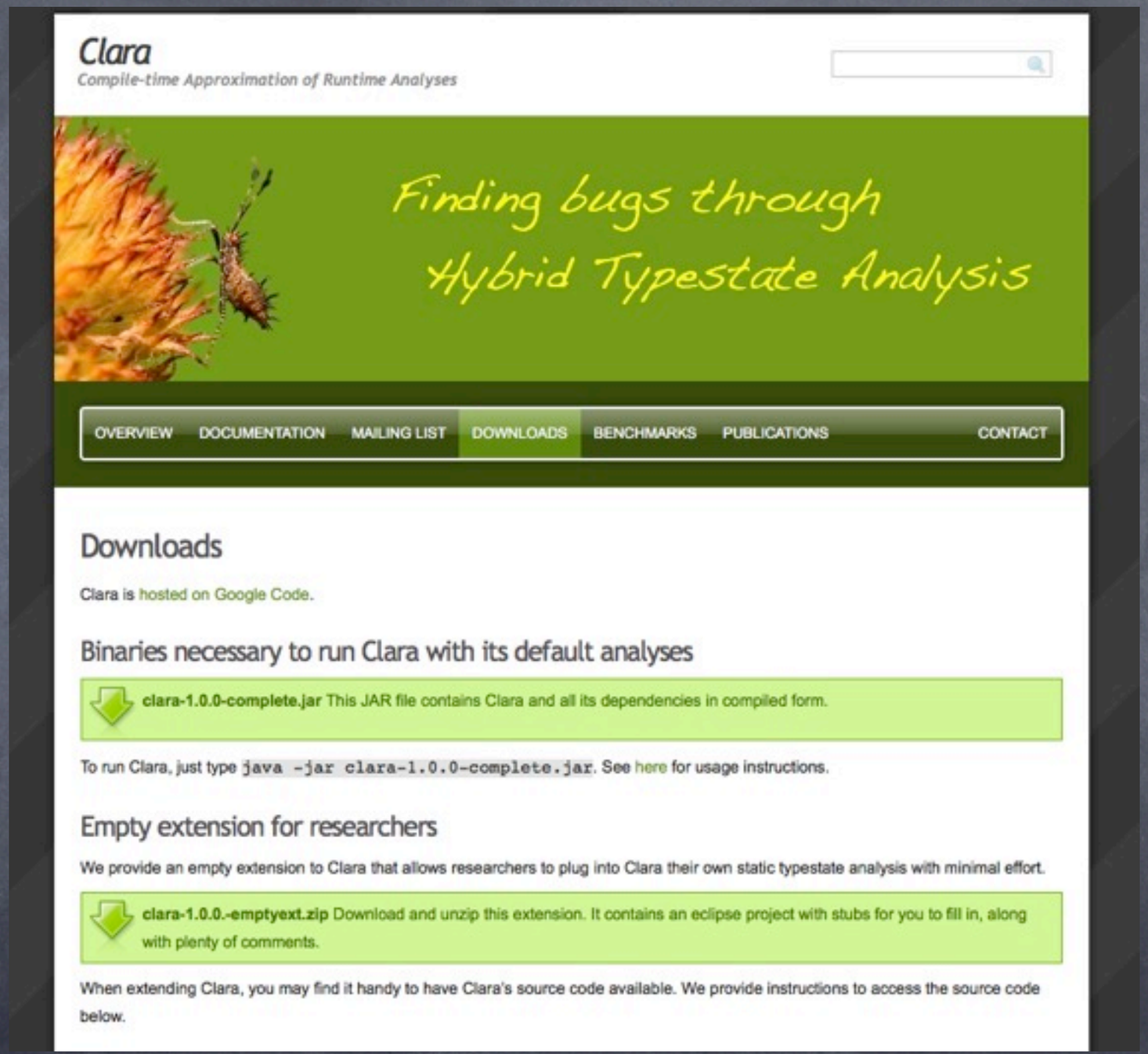
Implementation

- Clara is an extension to the AspectBench Compiler www.aspectbench.org
- Builds on Soot Program-Analysis Framework www.sable.mcgill.ca/soot/
- Is extensible: may implement your own static analysis
- Can partially evaluate any* AspectJ-based runtime monitor

*DSM annotation required

25

Extending Clara



http://www.bodden.de/clara/downloads/

26

Overall success

	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

27

Why so effective?

- Very precise abstractions:
  - Resolve aliasing using three different alias analyses (some context sensitive, others flow sensitive)
  - Analysis is path sensitive
- Program properties:
  - Most objects only accessed in few methods
  - Most programs are mostly correct!

28

Related and previous work

Runtime Verification

AOP / AspectJ

Static Analysis

29

Related and previous work

Runtime Verification

Tracecuts (Walker & Viggers, FSE 04)

PTQL (Goldsmith et al., OOPSLA 05)

PQL (Martin et al., OOPSLA 05)

Larva (Colombo et al., SEFM 09)

M2Aspects (Krüger et al., SCESM 06)

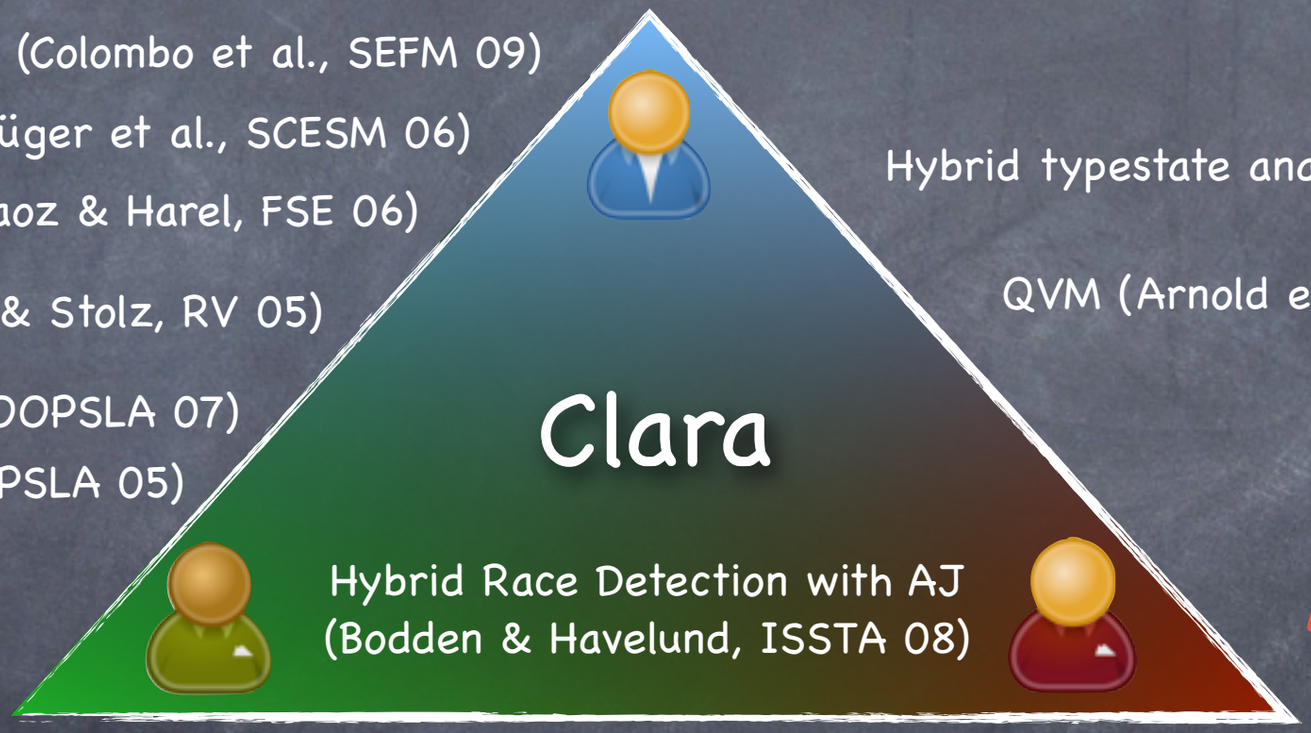
S2A (Maoz & Harel, FSE 06)

J-LO (Bodden & Stolz, RV 05)

JavaMOP (Chen et al., OOPSLA 07)

Tracematches (Allan et al., OOPSLA 05)

AOP / AspectJ



Hybrid typestate analysis (Dwyer et al., ASE 07)

QVM (Arnold et al., OOPSLA 08)

Static Analysis

Static Typestate Analysis (Fink et al., ISSTA 06)

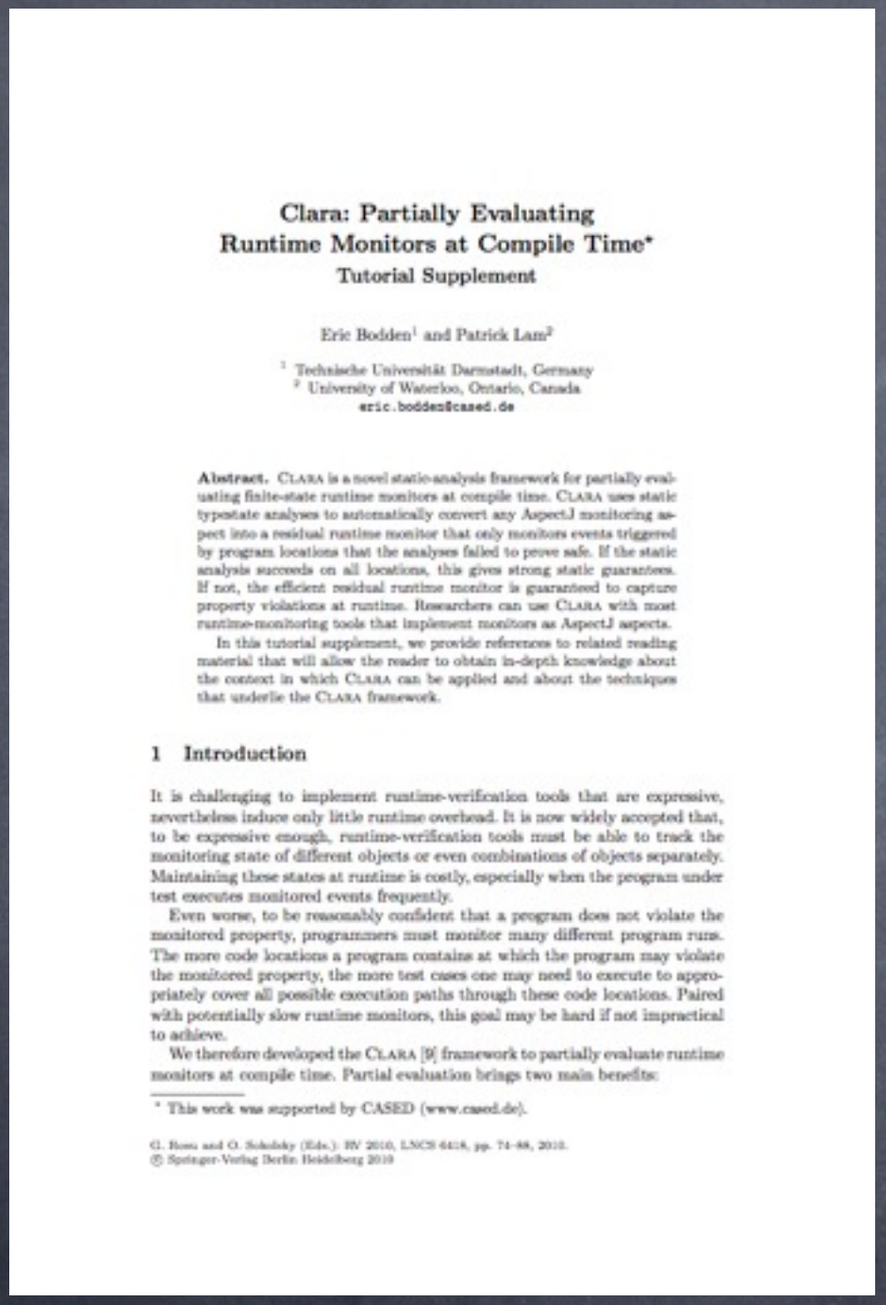
SCoPE (Aotani & Masuhara, AOSD 07)

Optimizing cflow (Avgustinov et al., PLDI 05)

Statically optimizing tracematches
(Naeem & Lhotak, OOPSLA 08; Bodden et al. ECOOP 07, FSE 08)

29

Related and previous work



Clara: Partially Evaluating Runtime Monitors at Compile Time*
Tutorial Supplement

Eric Bodden[1] and Patrick Lam[2]

[1] Technische Universität Darmstadt, Germany
[2] University of Waterloo, Ontario, Canada
eric.bodden@cased.de

Abstract. CLARA is a novel static-analysis framework for partially evaluating finite-state runtime monitors at compile time. CLARA uses static typestate analyses to automatically convert any AspectJ monitoring aspect into a residual runtime monitor that only monitors events triggered by program locations that the analyses failed to prove safe. If the static analysis succeeds on all locations, this gives strong static guarantees. If not, the efficient residual runtime monitor is guaranteed to capture property violations at runtime. Researchers can use CLARA with most runtime-monitoring tools that implement monitors as AspectJ aspects.

In this tutorial supplement, we provide references to related reading material that will allow the reader to obtain in-depth knowledge about the context in which CLARA can be applied and about the techniques that underlie the CLARA framework.

1 Introduction

It is challenging to implement runtime-verification tools that are expressive, nevertheless induce only little runtime overhead. It is now widely accepted that, to be expressive enough, runtime-verification tools must be able to track the monitoring state of different objects or even combinations of objects separately. Maintaining these states at runtime is costly, especially when the program under test executes monitored events frequently.

Even worse, to be reasonably confident that a program does not violate the monitored property, programmers must monitor many different program runs. The more code locations a program contains at which the program may violate the monitored property, the more test cases one may need to execute to appropriately cover all possible execution paths through these code locations. Paired with potentially slow runtime monitors, this goal may be hard if not impractical to achieve.

We therefore developed the CLARA [9] framework to partially evaluate runtime monitors at compile time. Partial evaluation brings two main benefits:

* This work was supported by CASED (www.cased.de).

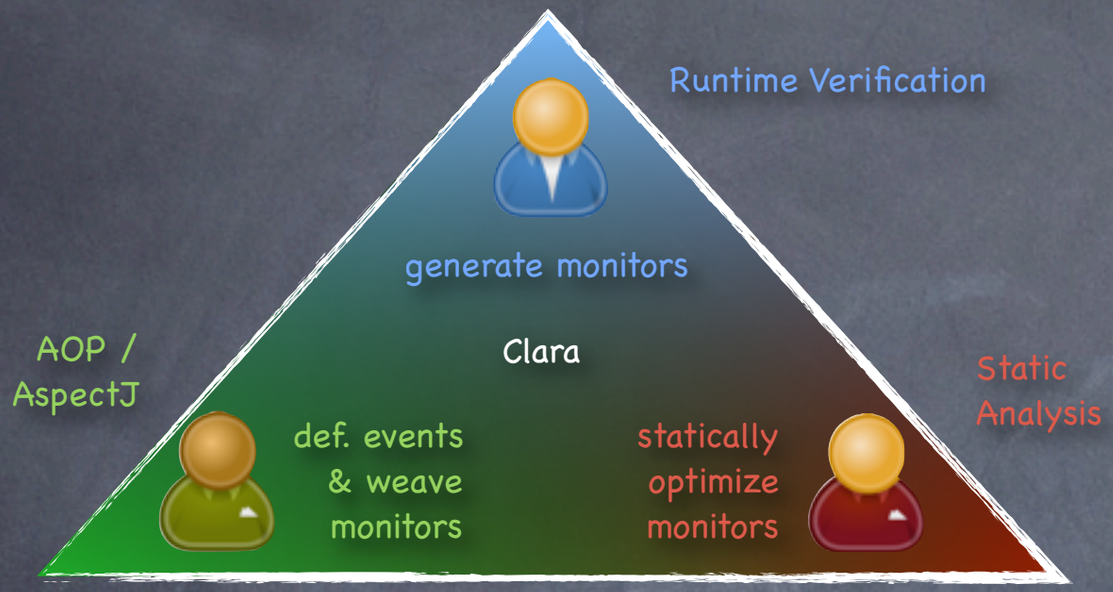
G. Rosu and O. Sokolsky (Eds.): RV 2010, LNCS 6418, pp. 74–88, 2010.

More:

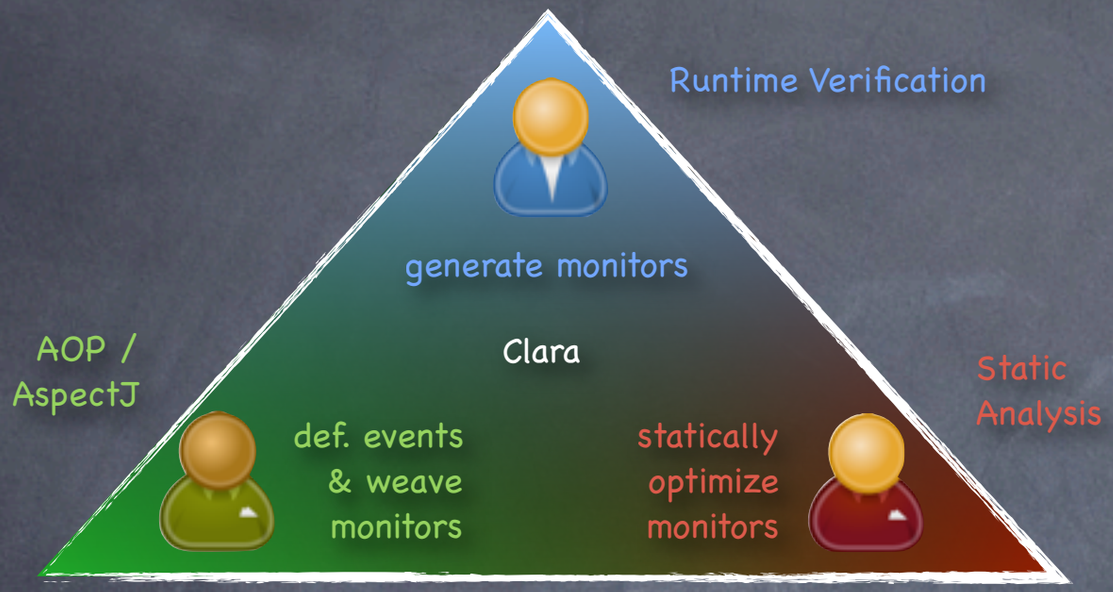
Tutorial paper, pages 183–197

30

31

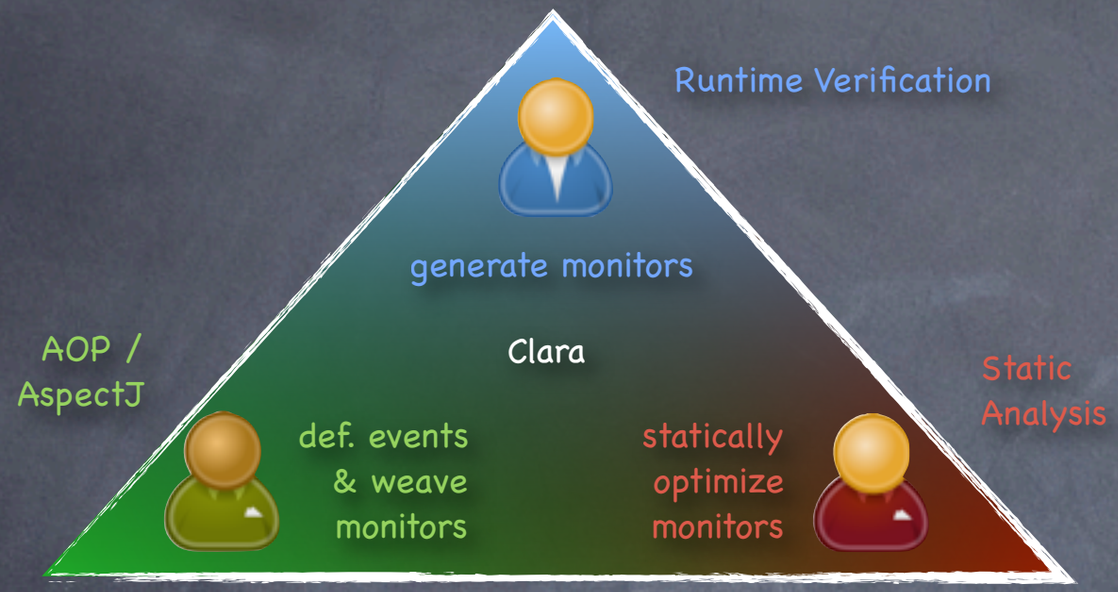

Runtime Verification
generate monitors
Clara
AOP /
AspectJ
def. events
& weave
monitors
statically
optimize
monitors
Static
Analysis

31

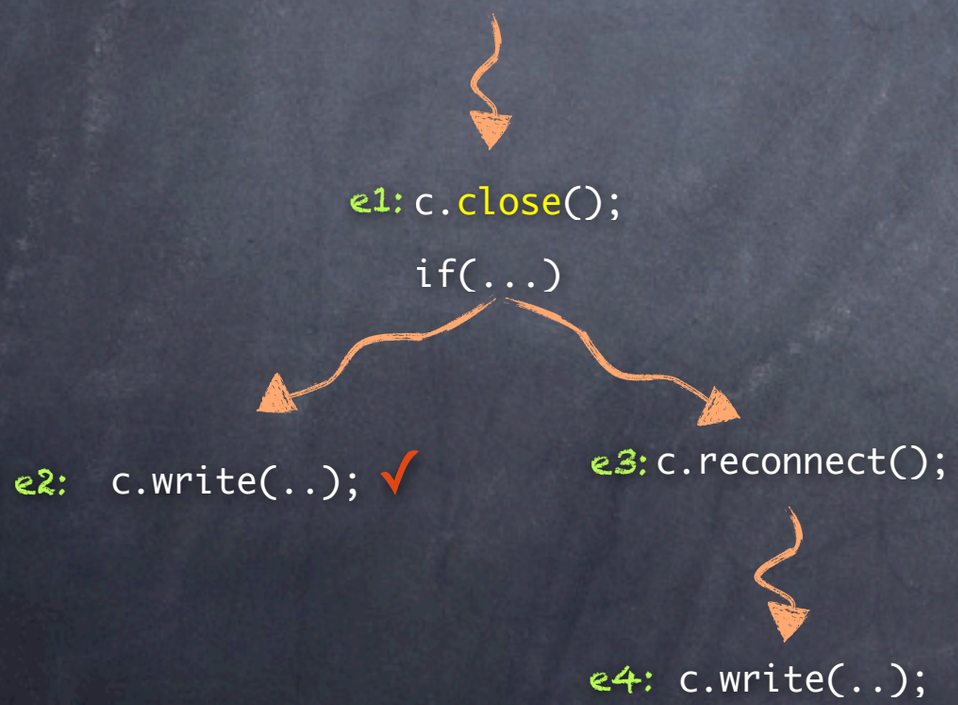



```
dependency{
    disconnect, write, reconnect;
    initial  connected: disconnect -> connected,
                        write -> connected,
                        reconnect -> connected,
                        disconnect -> disconnected;
                    disconnect: disconnect -> disconnected,
                        write -> error;
    final       error: write -> error;
}
```

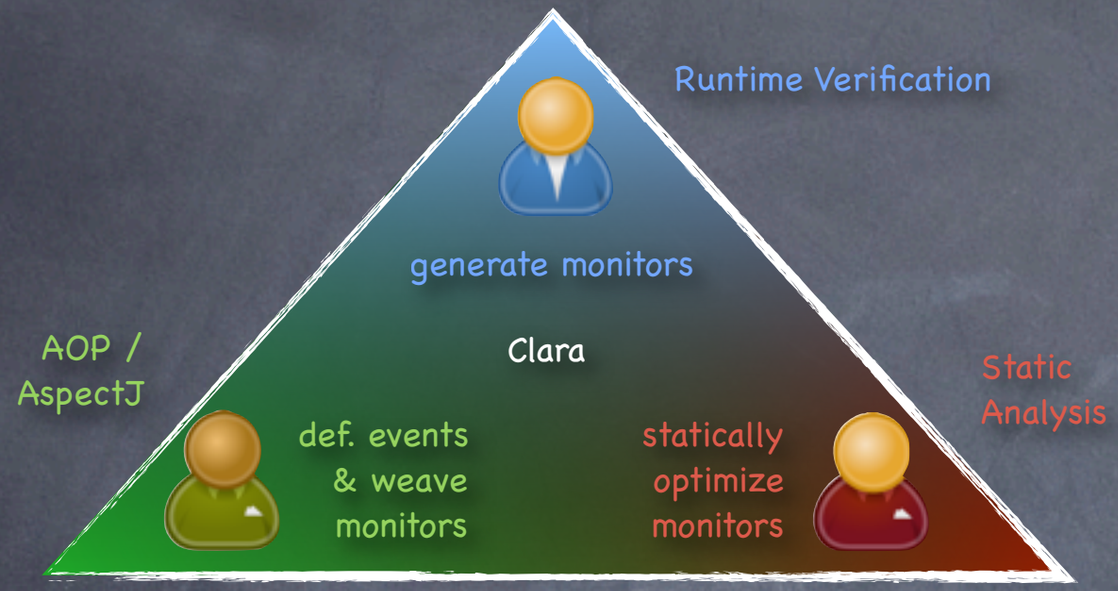

31



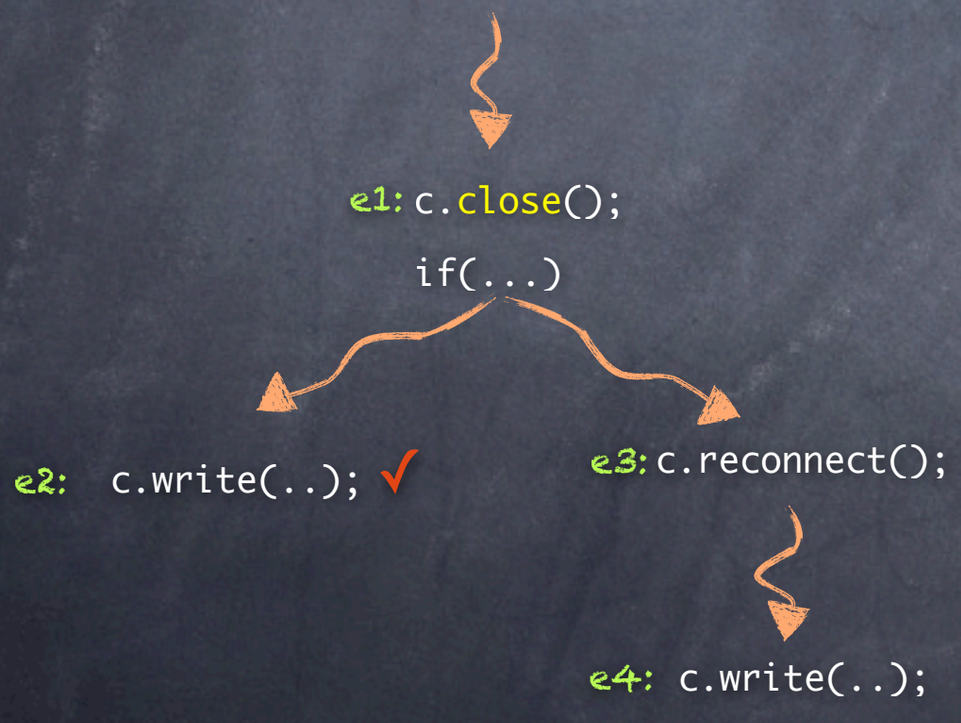

```
dependency{
    disconnect, write, reconnect;
    initial  connected: disconnect -> connected,
                        write -> connected,
                        reconnect -> connected,
                        disconnect -> disconnected;
                    disconnect: disconnect -> disconnected,
                        write -> error;
    final       error: write -> error;
}
```



31



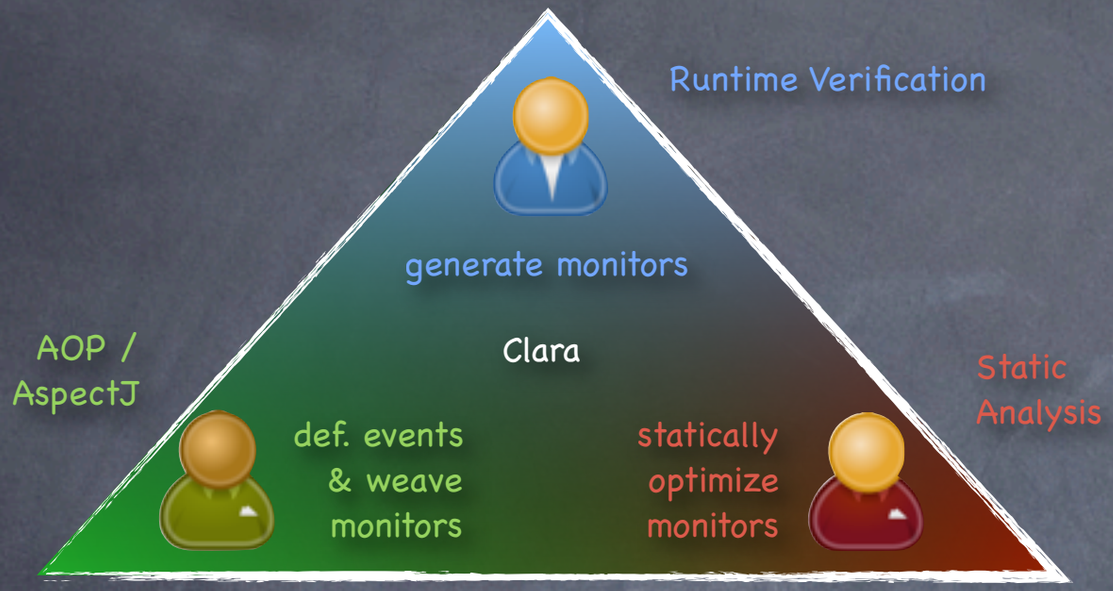

```
dependency{
    disconnect, write, reconnect;
    initial  connected: disconnect -> connected,
                        write -> connected,
                        reconnect -> connected,
                        disconnect -> disconnected;
                    disconnect: disconnect -> disconnected,
                        write -> error;
    final       error: write -> error;
}
```



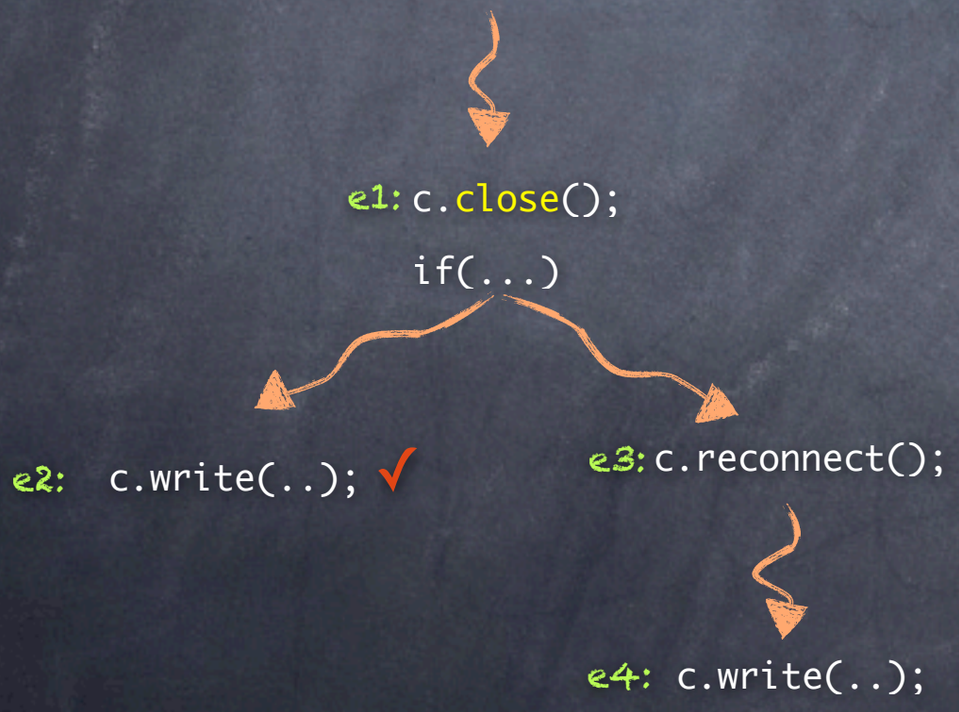
	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

31

www.bodden.de/clara/




```
dependency{
    disconnect, write, reconnect;
    initial  connected: disconnect -> connected,
                        write -> connected,
                        reconnect -> connected,
                        disconnect -> disconnected;
                disconnect: disconnect -> disconnected,
                        write -> error;
    final       error: write -> error;
}
```



	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

31

32