How banks can maintain stability

Carlo A. Furia

Chalmers University of Technology

bugcounting.net

Class-invariant based reasoning with semantic collaboration

Reasoning about OO

I'll present a framework for reasoning about the functional correctness of object-oriented programs based on class (object) invariants.

Reasoning about OO

Methodology: semantic collaboration

– includes an ownership scheme

Implementation: AutoProof verifier

– for simplicity, I will also use “AutoProof” to refer to the methodology

Reference language: Eiffel

– but practically everything applicable to Java/JML and similar OO languages

Main features of the framework

- Targets idiomatic OO structures (OO patterns)
- Flexible (semantic)
- Reasonably concise (defaults)
- Applicable to realistic implementations (data structure library)
- Sequential programs only

AutoProof in a nutshell

AutoProof is an auto-active verifier for Eiffel

- Prover for functional properties
- All-out support of object-oriented idiomatic structures (e.g. patterns)
  - Based on class invariants

Nadia Polikarpova

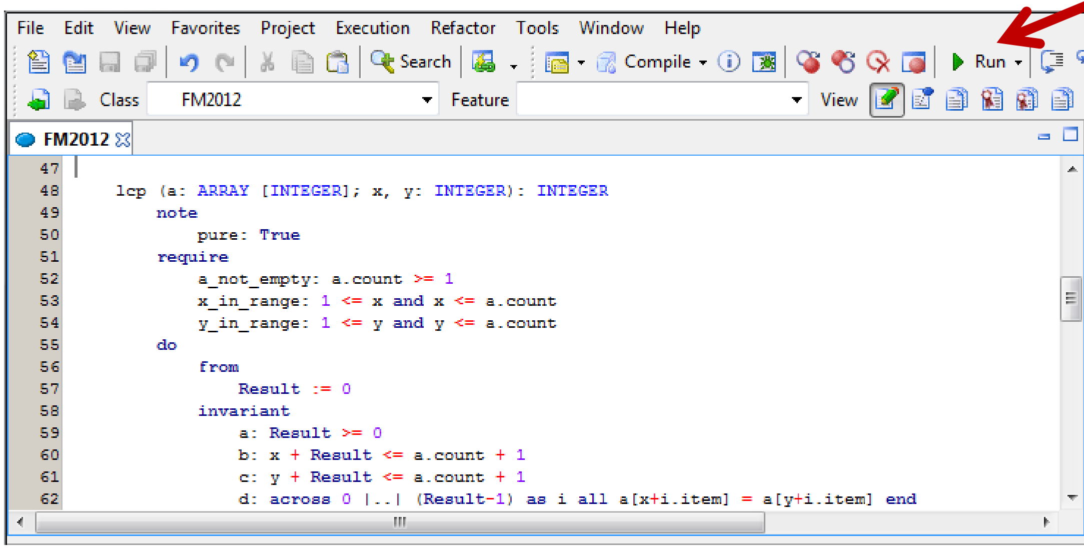
Julian Tschannen

Bertrand Meyer

Auto-active user/tool interaction

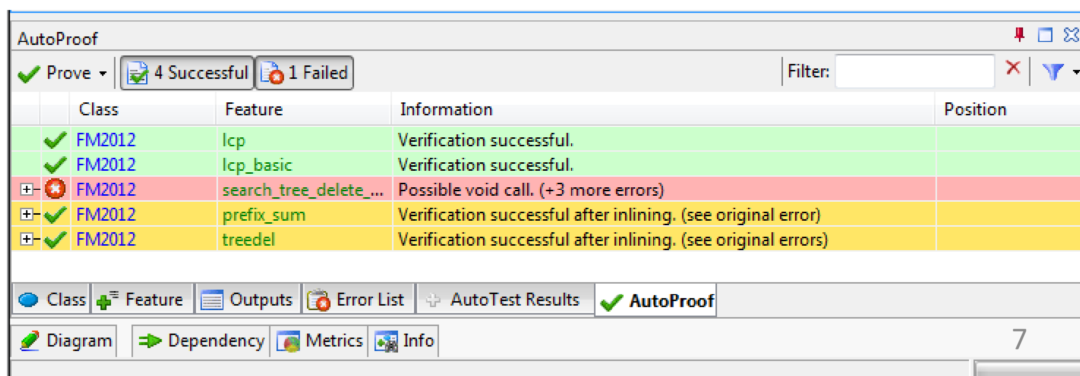
1. Code + Annotations

2. Push button

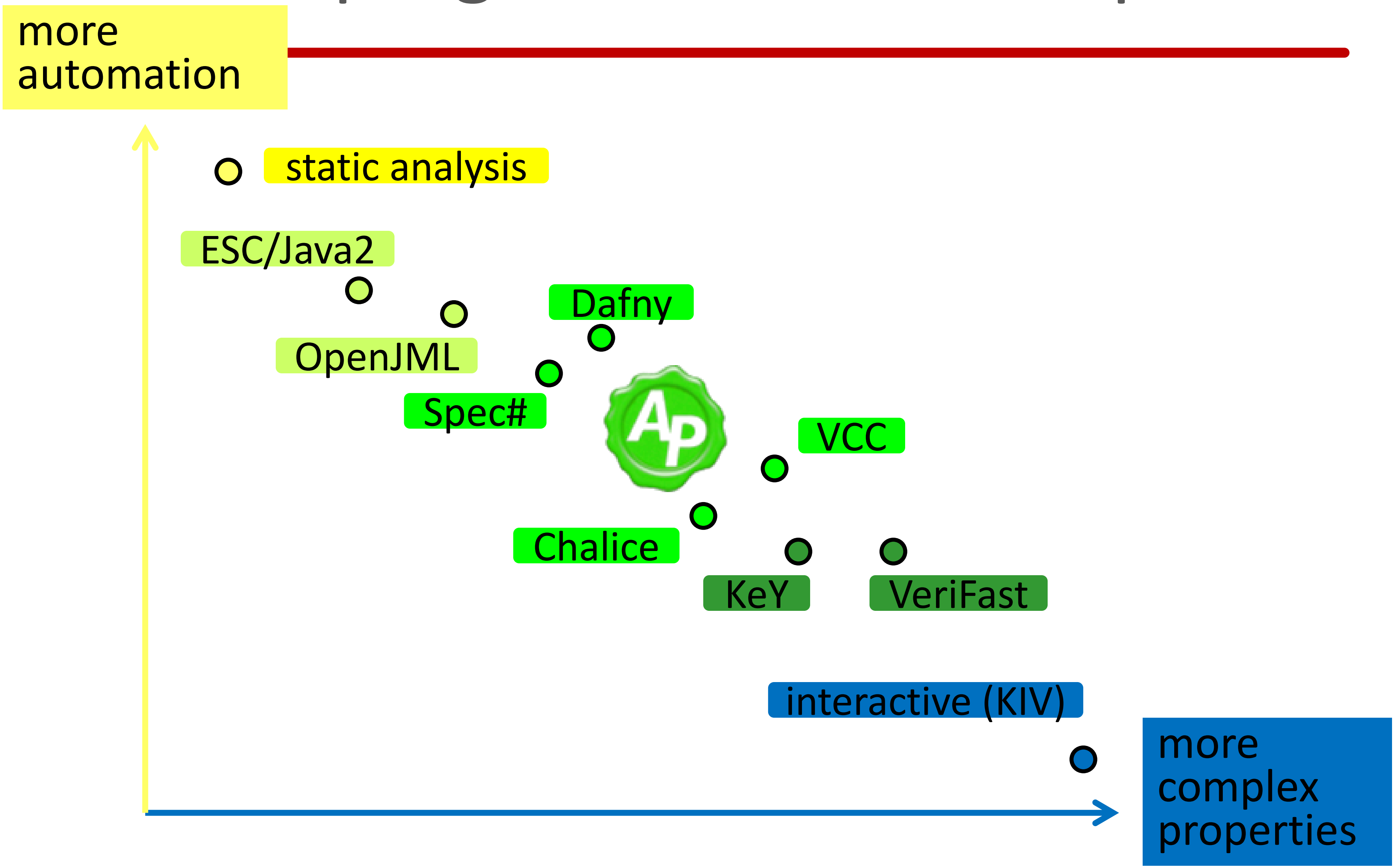


3. Verification outcome

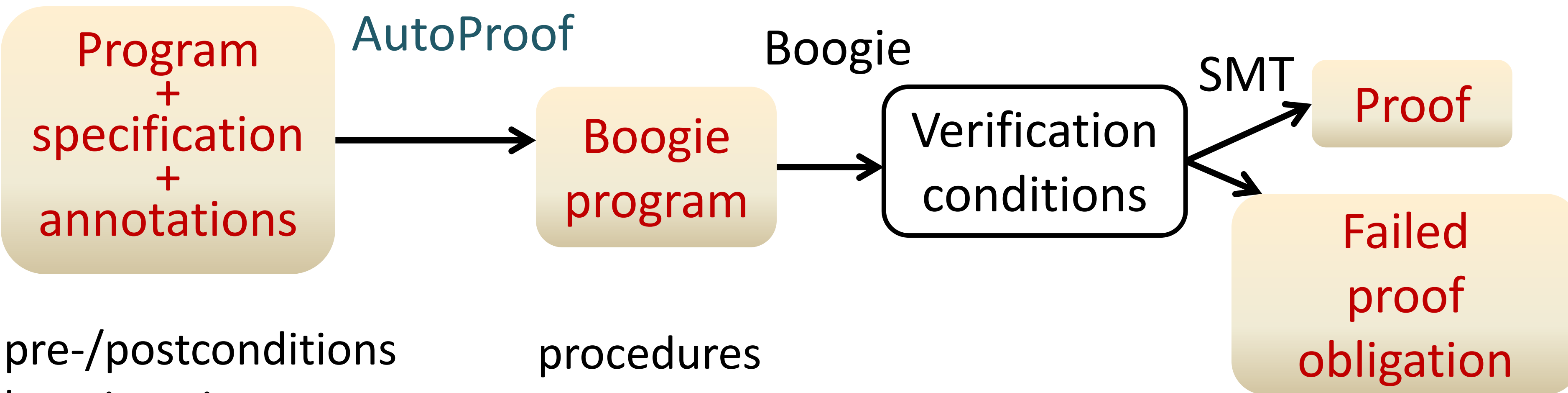
4. Correct/Revise



Sound program verifiers compared



How AutoProof works



pre-/postconditions
loop invariants
intermediate assertions
class invariants
frame specification
object dependencies

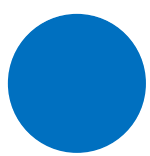
procedures
axioms
specification functions
memory model
background theory
triggers

Reasoning with class invariants

Class invariants are a natural way to reason about object-oriented programs:

invariant = consistency of objects

ACCOUNT



```
invariant
    balance >= 0
```

Demo: AutoProof warmup

AutoProof verifies a basic version of the bank ACCOUNT class

```
deposit (amount: INTEGER)

withdraw (amount: INTEGER)
```

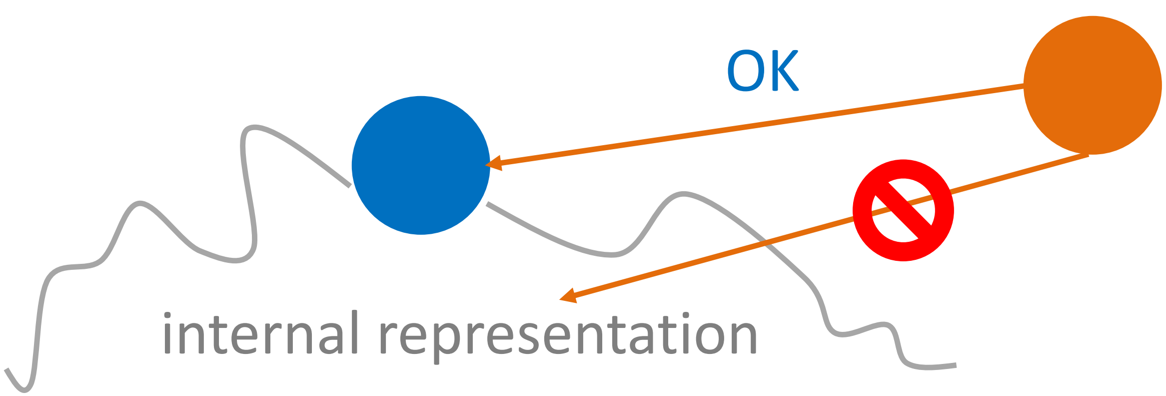
Follow this demo at:
http://comcom.csail.mit.edu/e4pubs/#demo-key
(Tab account_warmup.e)

Stability of invariant reasoning

Invariant-based reasoning should ensure stability:

stability = an operation can affect
an object's invariant only if
it modifies the object explicitly

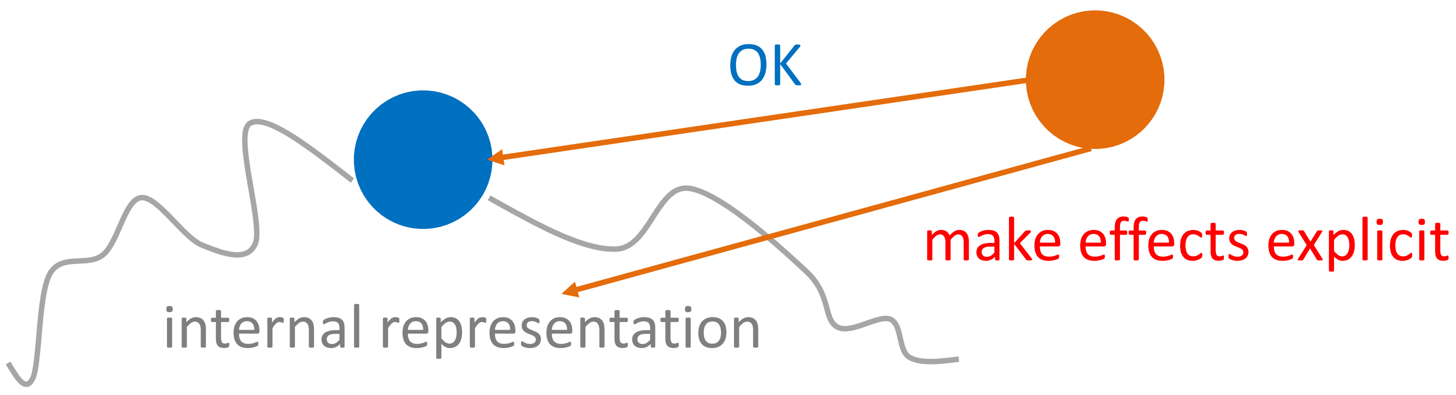
With stability, no one can invalidate an object behind its back!



Stability and encapsulation

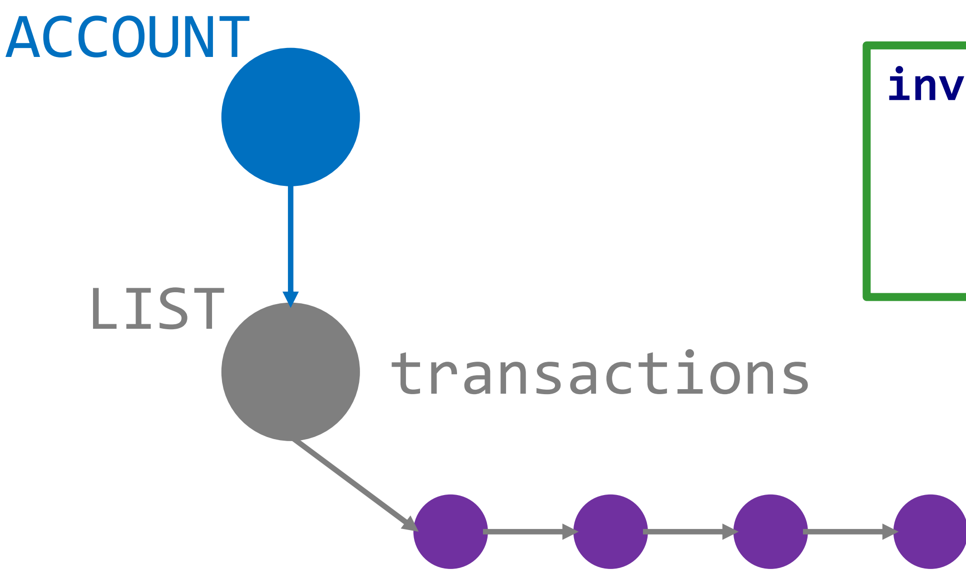
Invariant-based reasoning with stability:

- enforces encapsulation/information hiding
- simplifies client reasoning
- retains flexibility



Multi-object structures

Object-oriented programs involve multiple objects (duh!), whose consistency is often mutually dependent

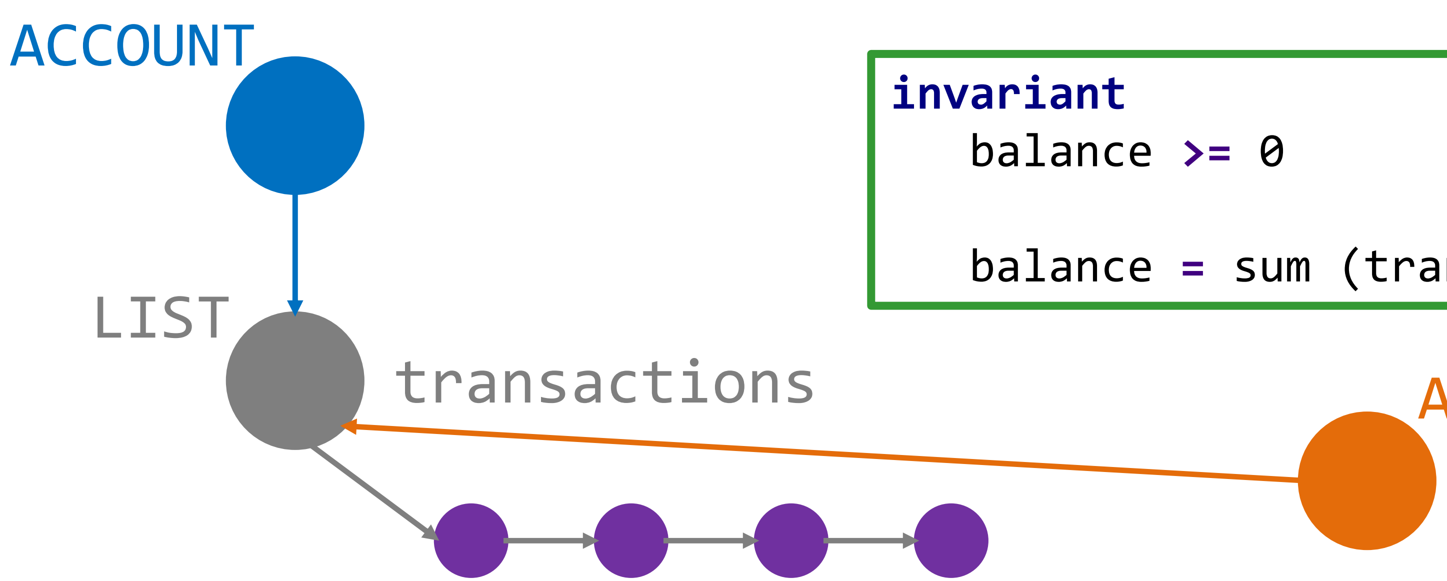



```
invariant
    balance >= 0

    balance = sum (transactions)
```

Consistency of multi-object structures

Mutually dependent object structures require extra care to enforce, and reason about, consistency (cmp. encapsulation)

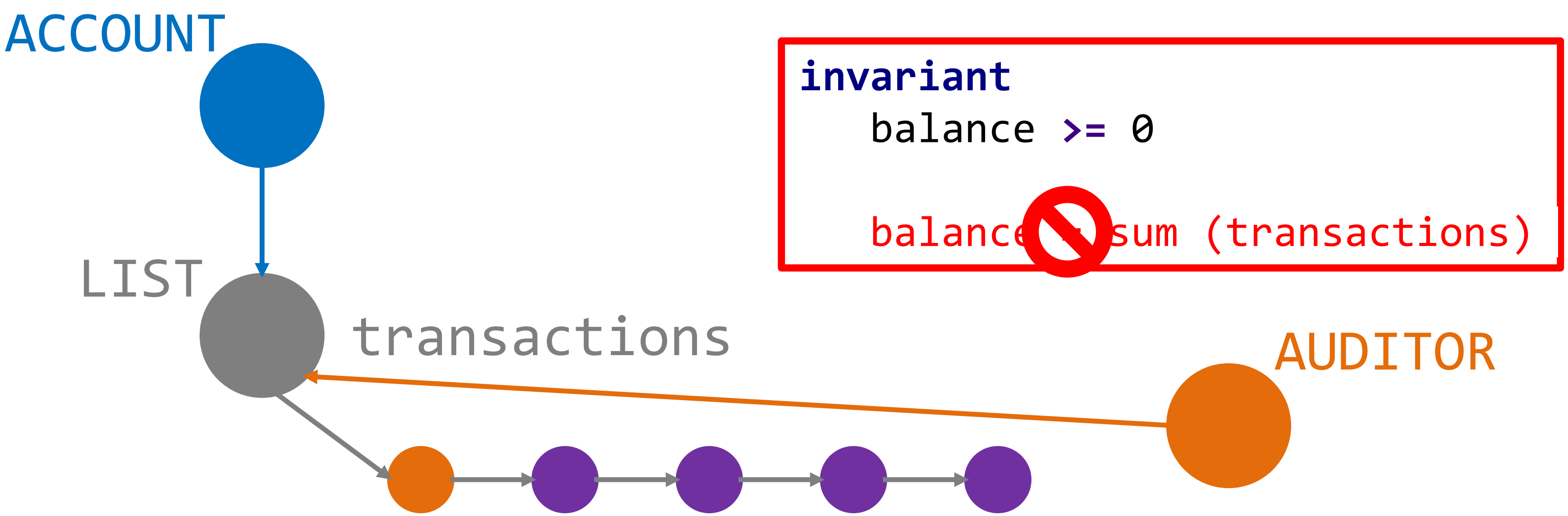



```
invariant
   balance >= 0

   balance = sum (transactions)
```

Consistency of multi-object structures

Mutually dependent object structures require extra care to enforce, and reason about, consistency (cmp. encapsulation)



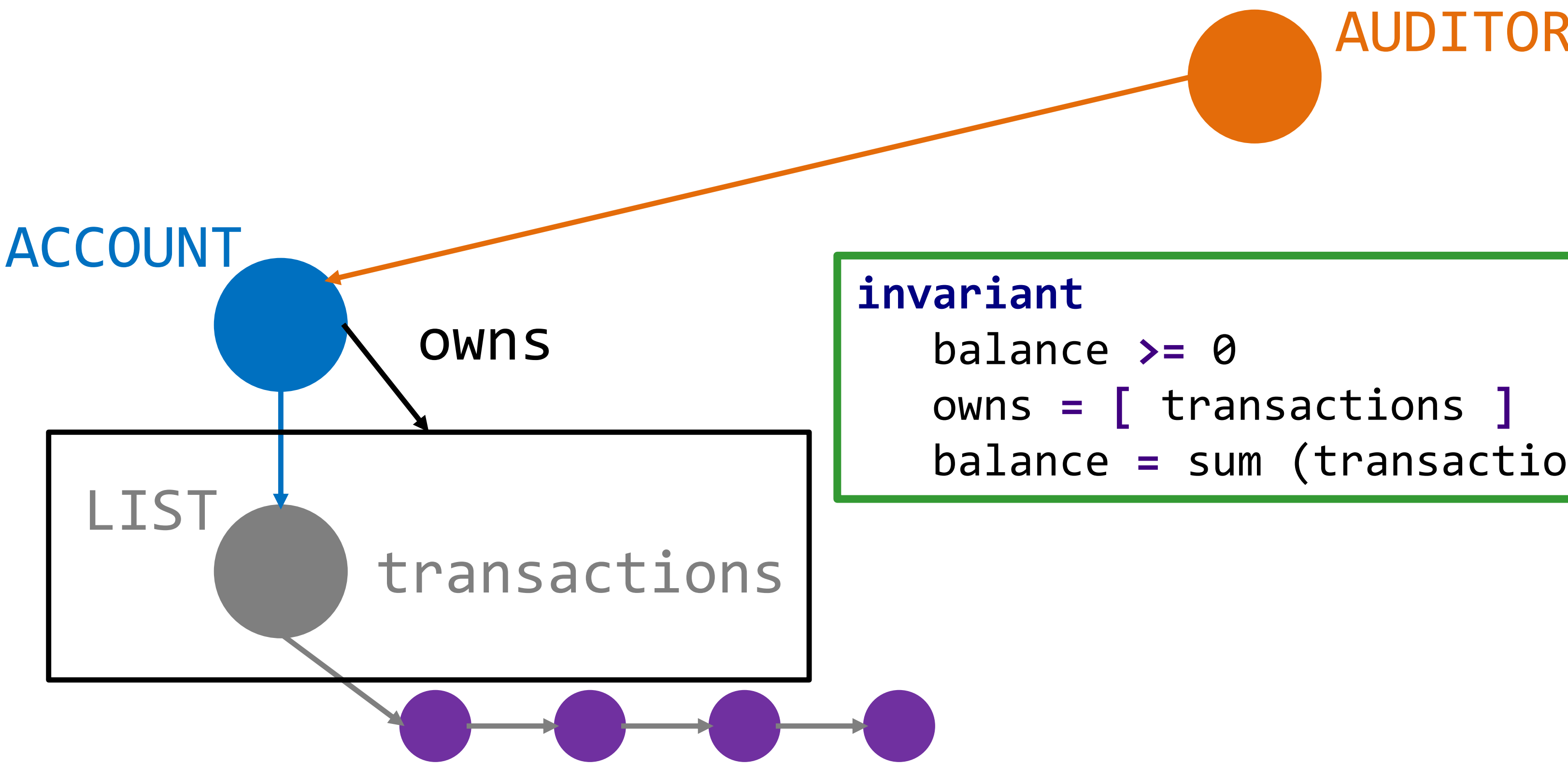
Open and closed objects

When (at which program points) must class invariants hold? To provide flexibility, objects in AutoProof can be open or closed

	CLOSED	OPEN
Object:	consistent	inconsistent
State:	stable	transient
Invariant:	holds	may not hold

Ownership

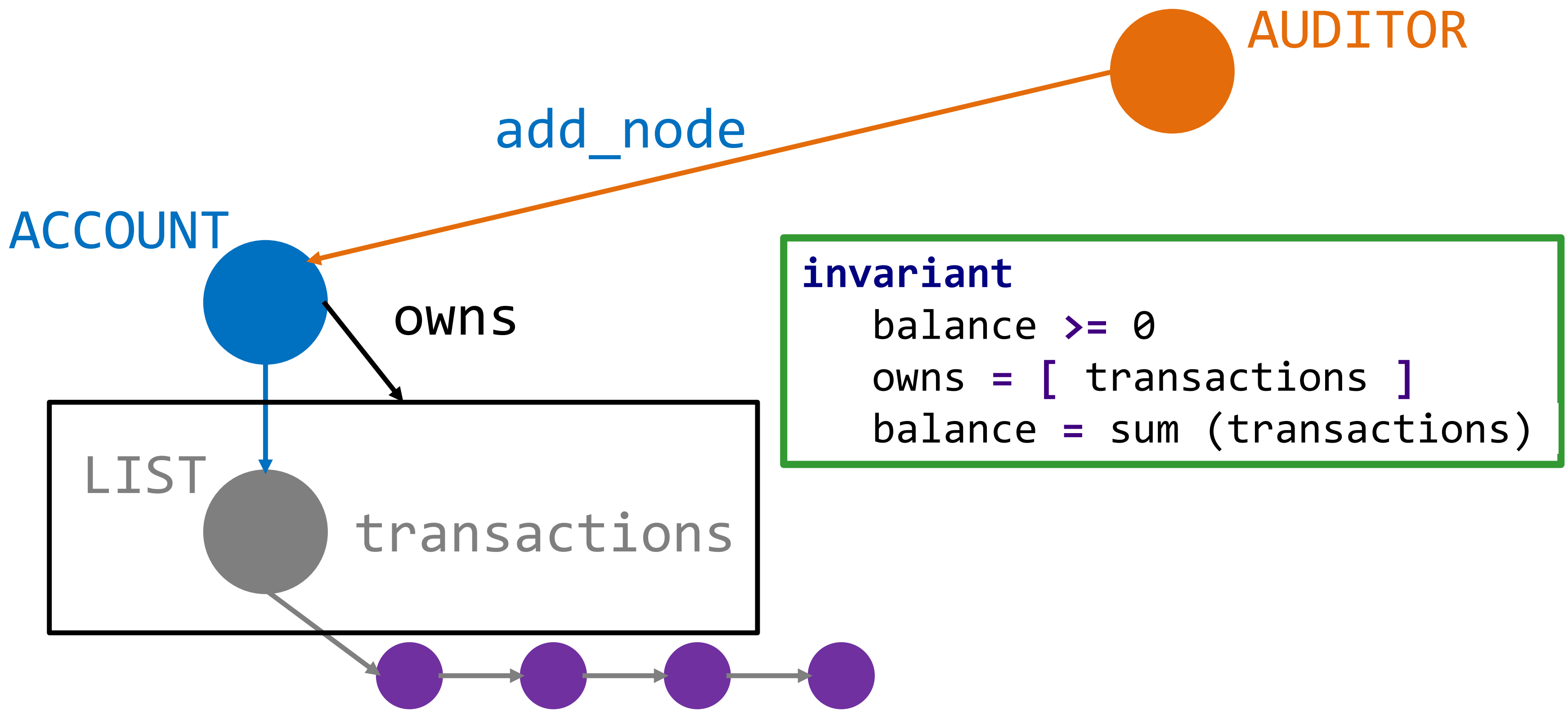
For hierarchical object structures, AutoProof offers an ownership protocol




```
invariant
    balance >= 0
    owns = [ transactions ]
    balance = sum (transactions)
```

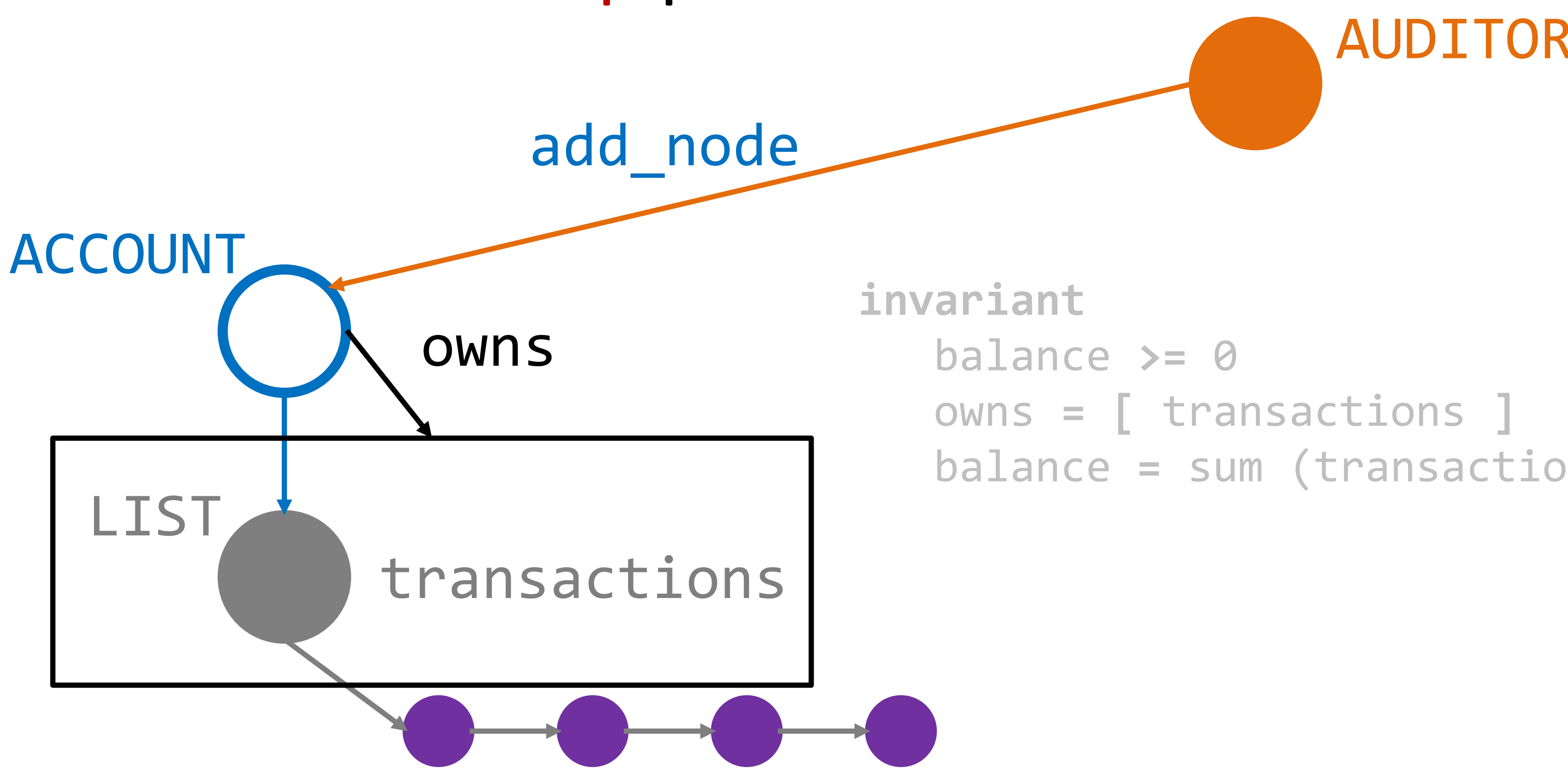
Ownership

For hierarchical object structures, AutoProof offers an ownership protocol



Ownership

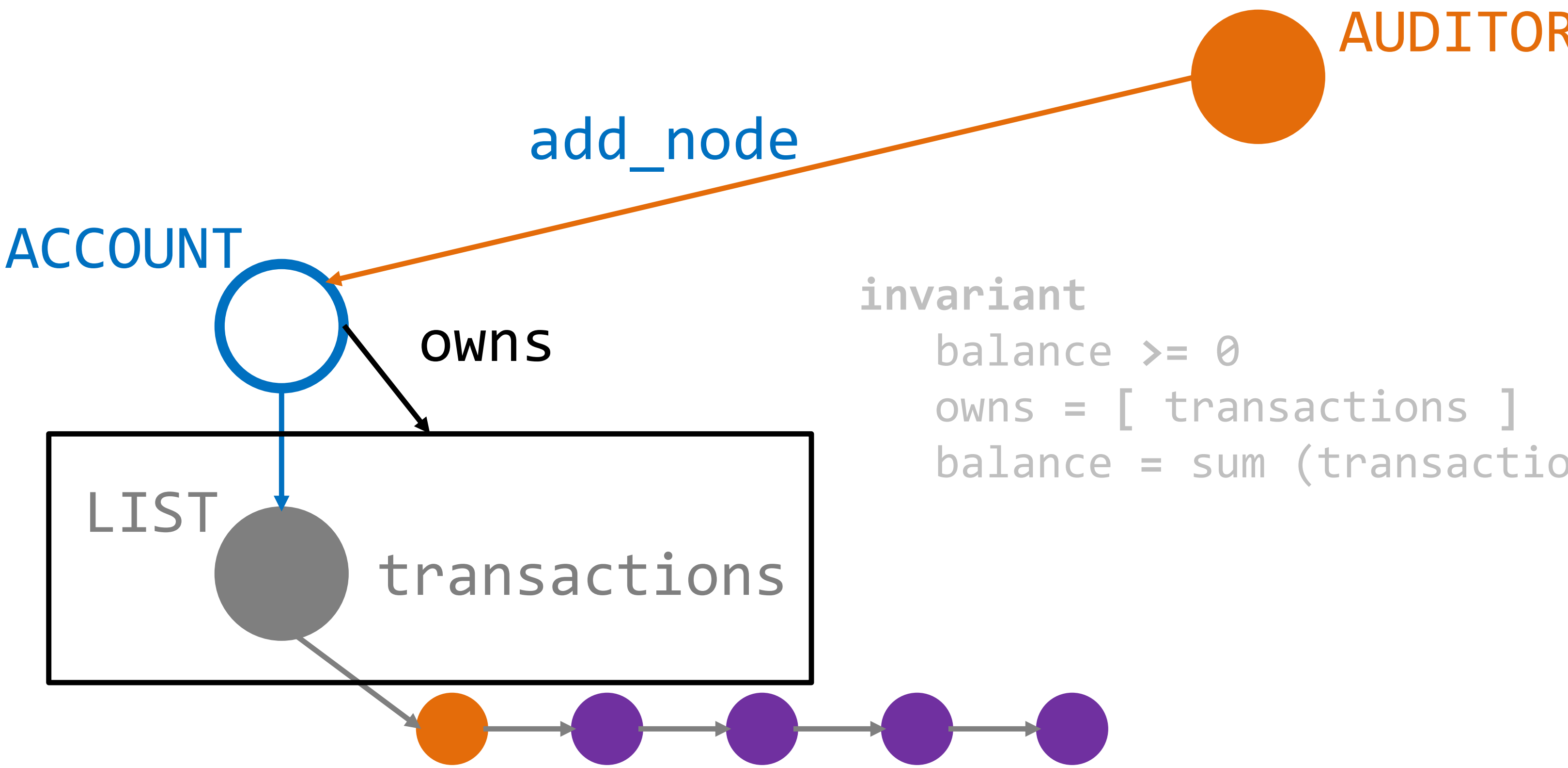
For hierarchical object structures, AutoProof offers an ownership protocol




```
invariant
   balance >= 0
   owns = [ transactions ]
   balance = sum (transactions)
```

Ownership

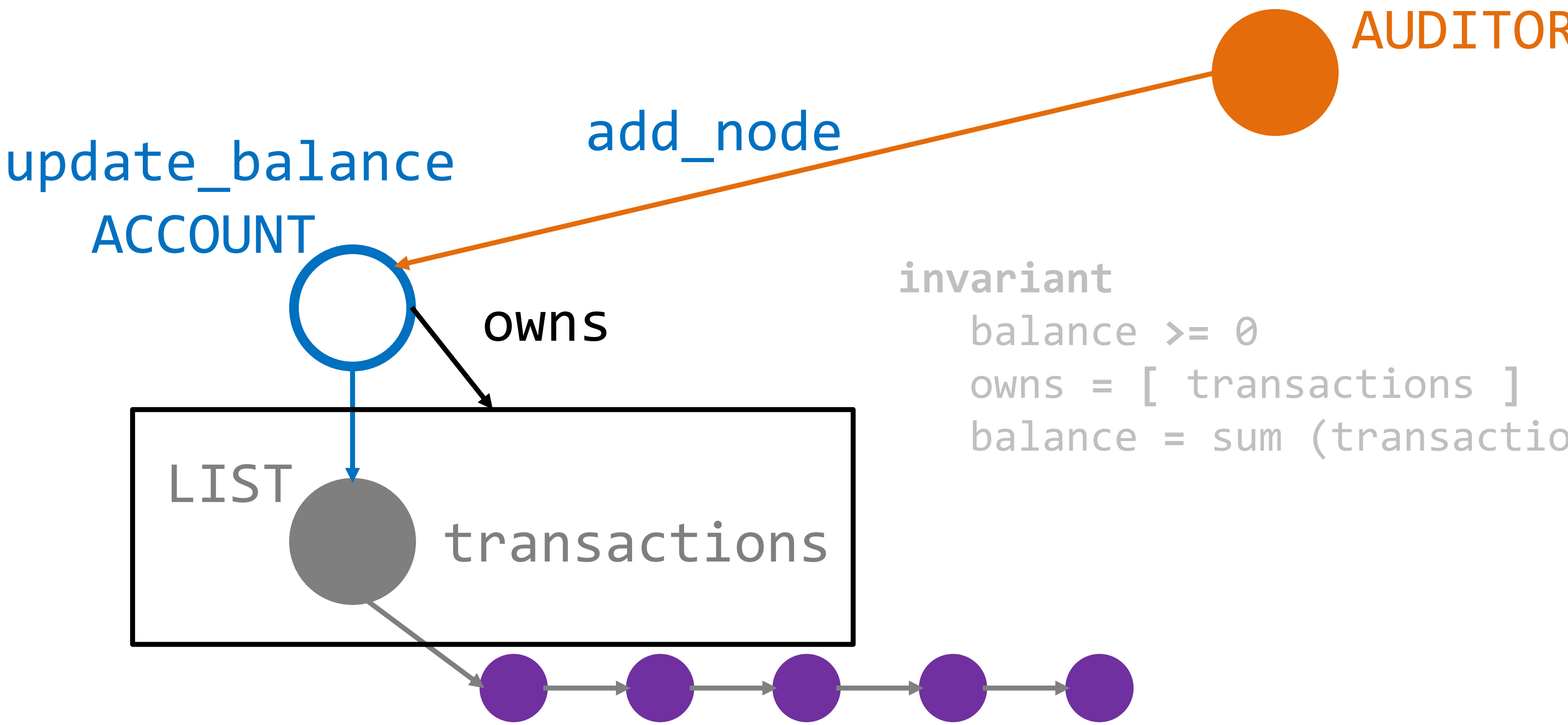
For hierarchical object structures, AutoProof offers an ownership protocol




```
invariant
   balance >= 0
   owns = [ transactions ]
   balance = sum (transactions)
```

Ownership

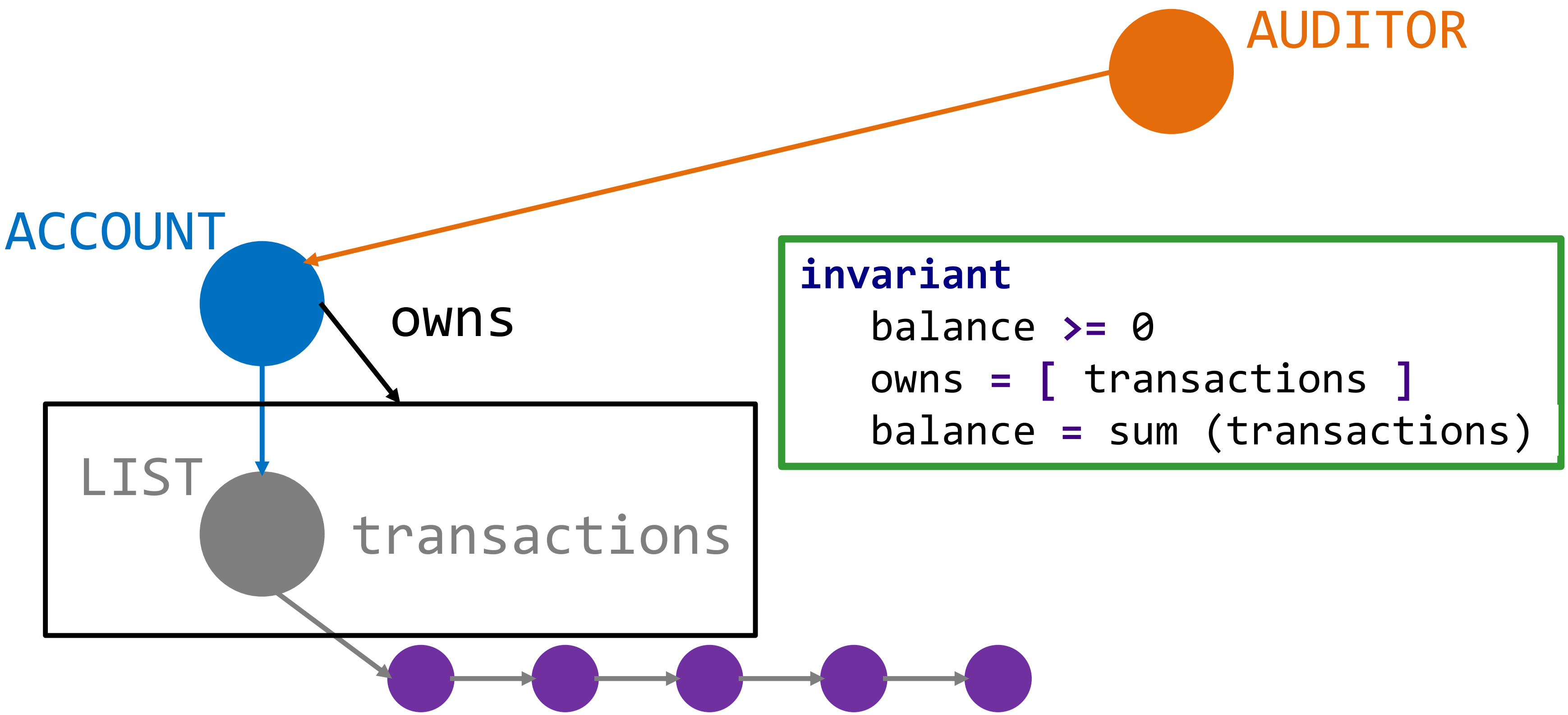
For hierarchical object structures, AutoProof offers an ownership protocol




```
invariant
    balance >= 0
    owns = [ transactions ]
    balance = sum (transactions)
```

Ownership

For hierarchical object structures, AutoProof offers an ownership protocol



Demo: ownership in AutoProof

AutoProof verifies `deposit` and `withdraw` in ACCOUNT with an owned list of transactions

```
transactions: SIMPLE_LIST [INTEGER]
  -- History of transactions:
  --   positive integer = deposited amount
  --   negative integer = withdrawn amount
  --   latest transactions in back of list
```

Follow this demo at:

http://comcom.csail.mit.edu/e4pubs/#demo-key

(Tab account_ownership.e)

Wrapping and unwrapping

Combination on ownership and invariants:

Wrapped object = closed and not owned

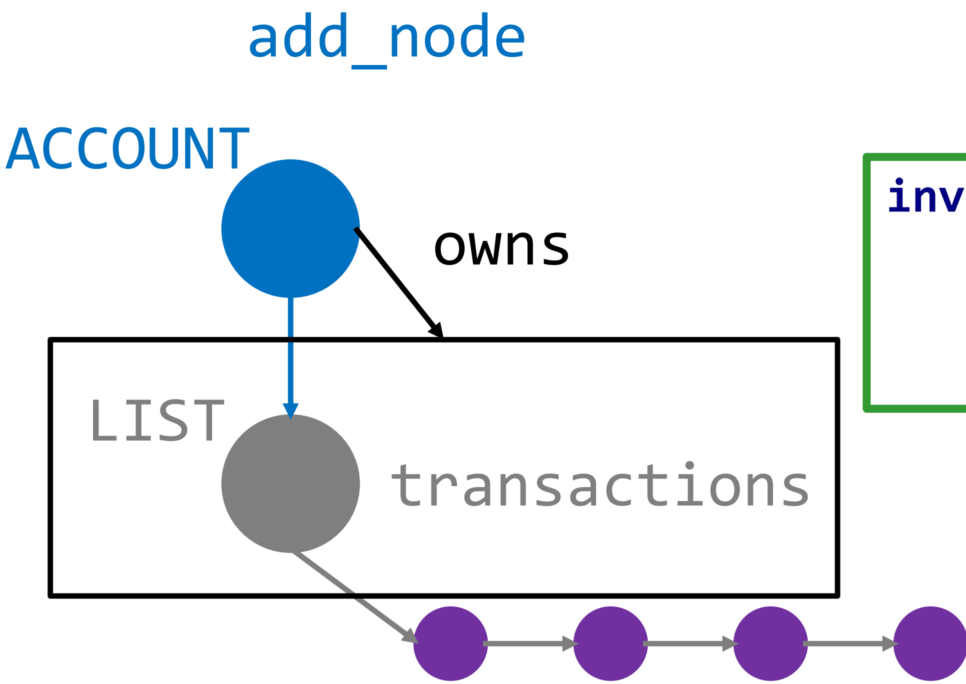
Unwrapped object = open (or owned)

	WRAPPED	UNWRAPPED
Invariant:	holds	may not hold
Clients:	any object	within owner
Modifications:	modify after unwrapping	wrap after modifying

Wrapping and unwrapping

Typical modification pattern:

unwrap, modify, wrap (check consistency)

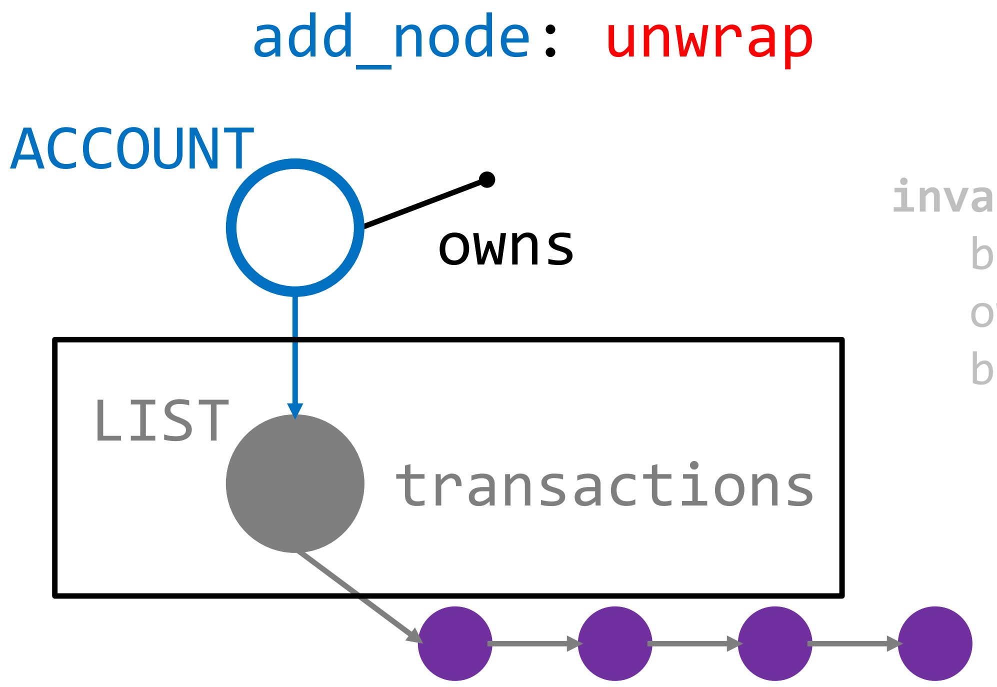



```
invariant
    balance >= 0
    owns = [ transactions ]
    balance = sum (transactions)
```

Wrapping and unwrapping

Typical modification pattern:

unwrap, modify, wrap (check consistency)



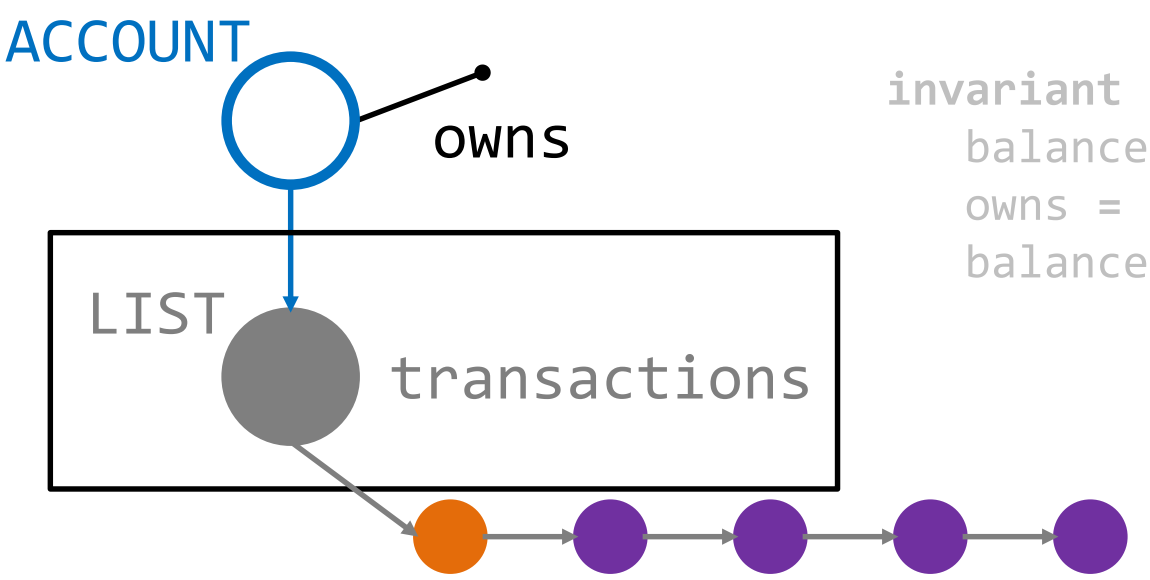

```
invariant
   balance >= 0
   owns = [ transactions ]
   balance = sum (transactions)
```

Wrapping and unwrapping

Typical modification pattern:

unwrap, modify, wrap (check consistency)

```
add_node: unwrap; modify
```



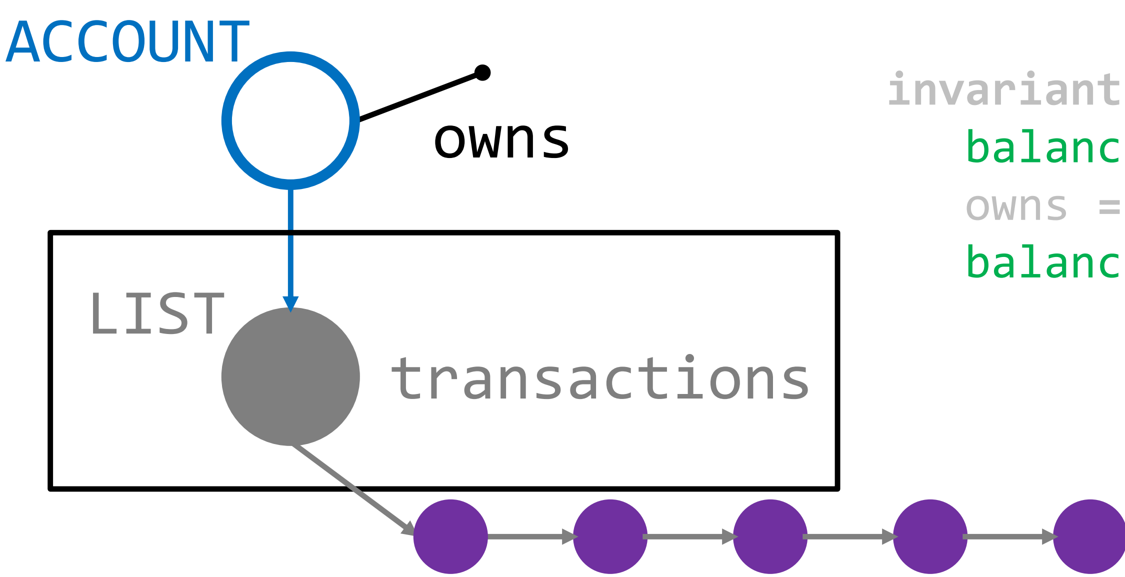

```
invariant
   balance >= 0
   owns = [ transactions ]
   balance = sum (transactions)
```

Wrapping and unwrapping

Typical modification pattern:

unwrap, modify, wrap (check consistency)

```
add_node: unwrap; modify; wrap (check)
```



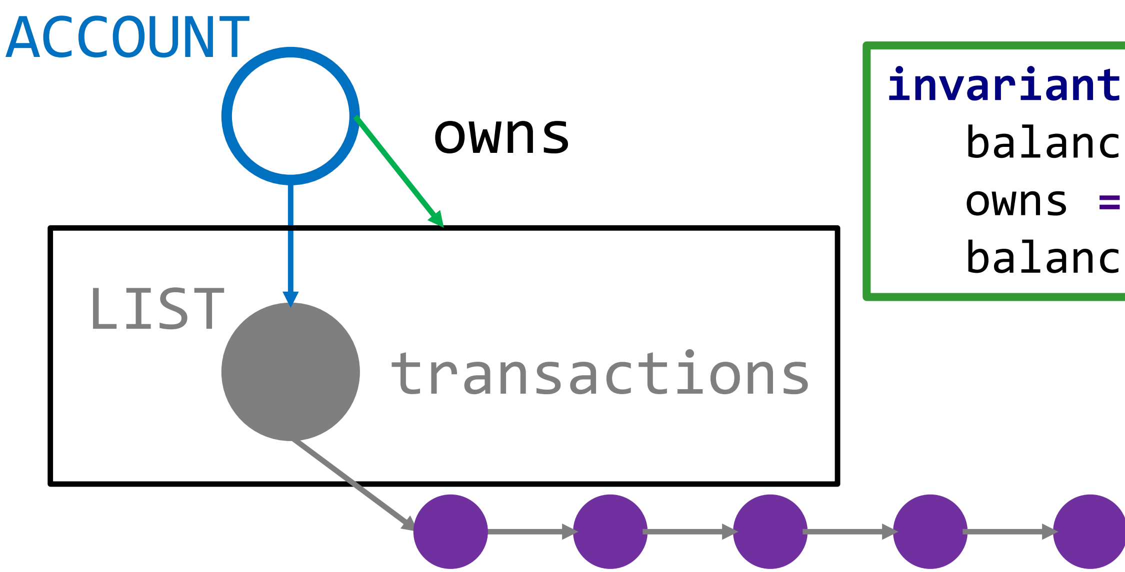

```
invariant
   balance >= 0
   owns = [ transactions ]
   balance = sum (transactions)
```

Wrapping and unwrapping

Typical modification pattern:

unwrap, modify, wrap (check consistency)

`add_node: unwrap; modify; wrap (check)`




```
invariant
    balance >= 0
    owns = [ transactions ]
    balance = sum (transactions)
```

Demo: ownership preserves stability

Ownership achieves stability when leaking references to the internal `transactions` list in ACCOUNT

```
leak_transactions: SIMPLE_LIST [INTEGER]

leak_transactions_unsafe: SIMPLE_LIST [INTEGER]
```

Follow this demo at:

http://comcom.csail.mit.edu/e4pubs/#demo-key

(Tabs account_ownership.e and auditor.e)

Semantic collaboration

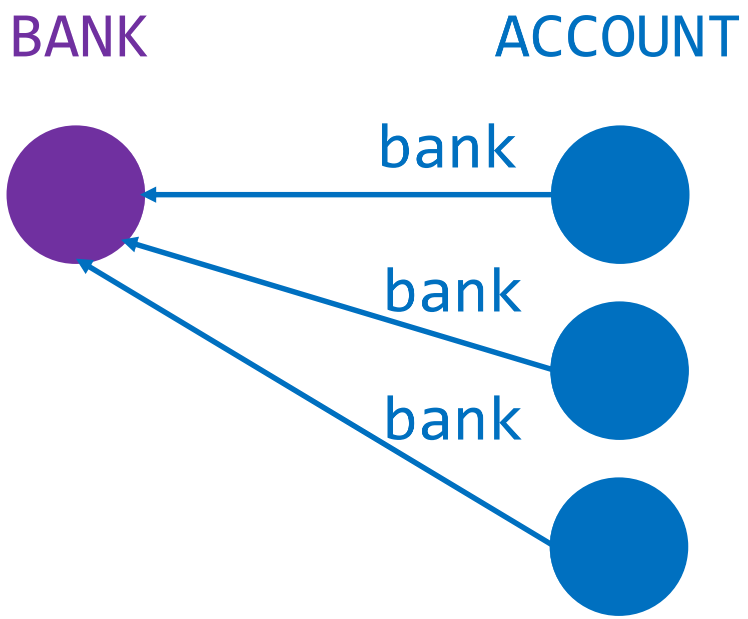
For collaborative object structures, AutoProof offers a novel protocol: semantic collaboration


```
invariant

   interest_rate = bank.rate
```

Semantic collaboration

For collaborative object structures, AutoProof offers a novel protocol: semantic collaboration

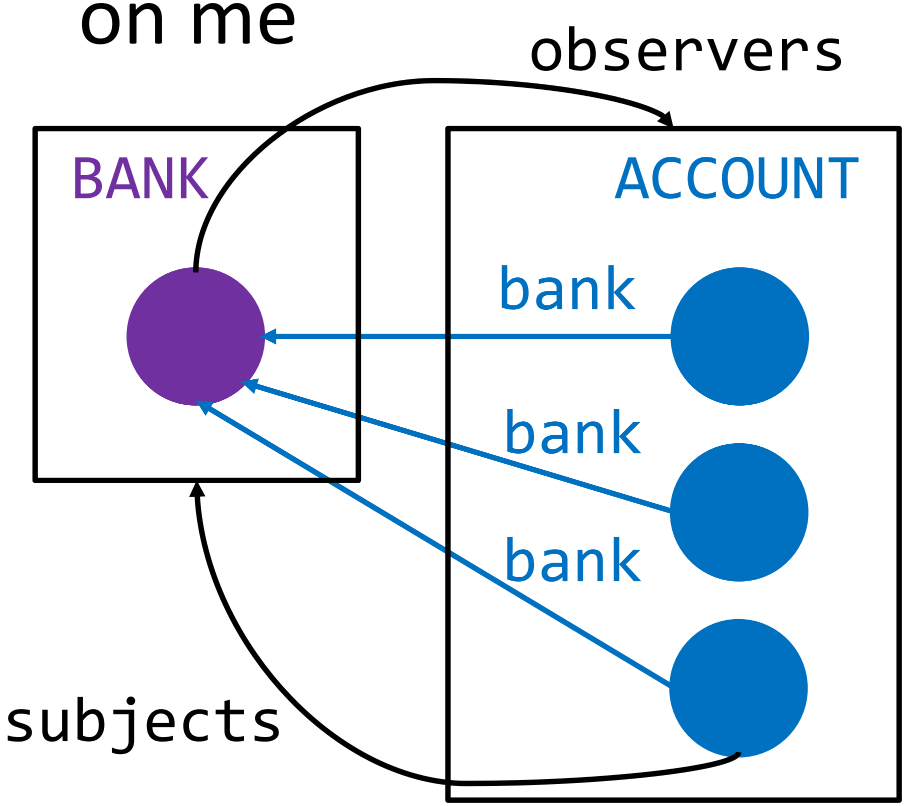



```
invariant

   interest_rate = bank.rate
```

Semantic collaboration

- Subjects = objects my consistency depends on
- Observers = objects whose consistency depends on me



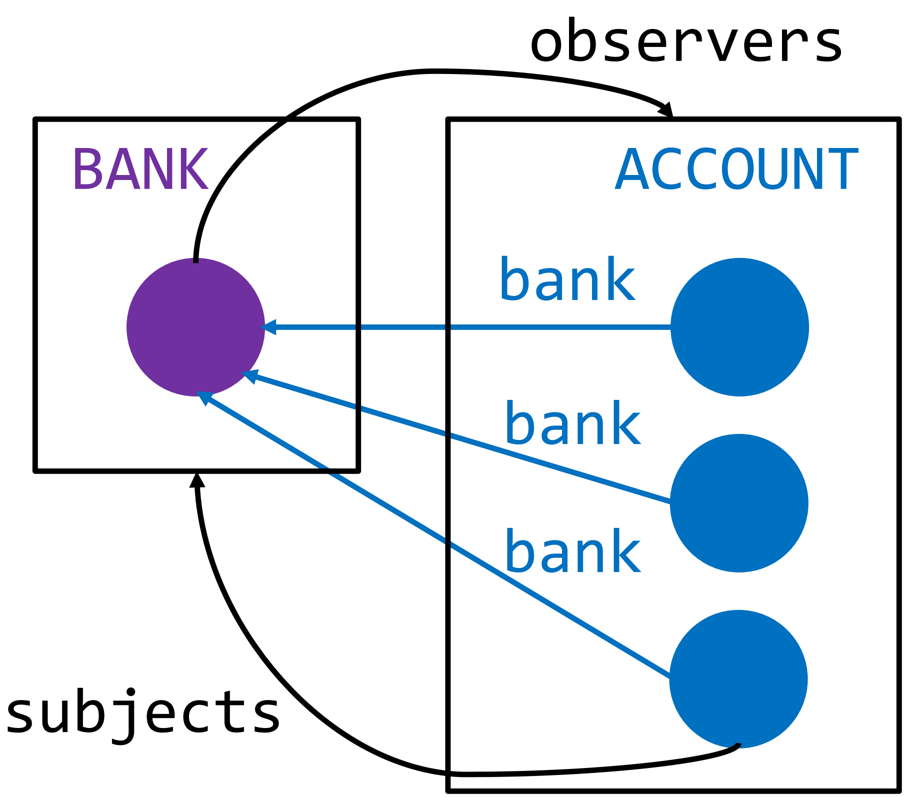

```
invariant
    subjects = [ bank ]
    Current in bank.observers
        -- Implicit in AutoProof

    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

update

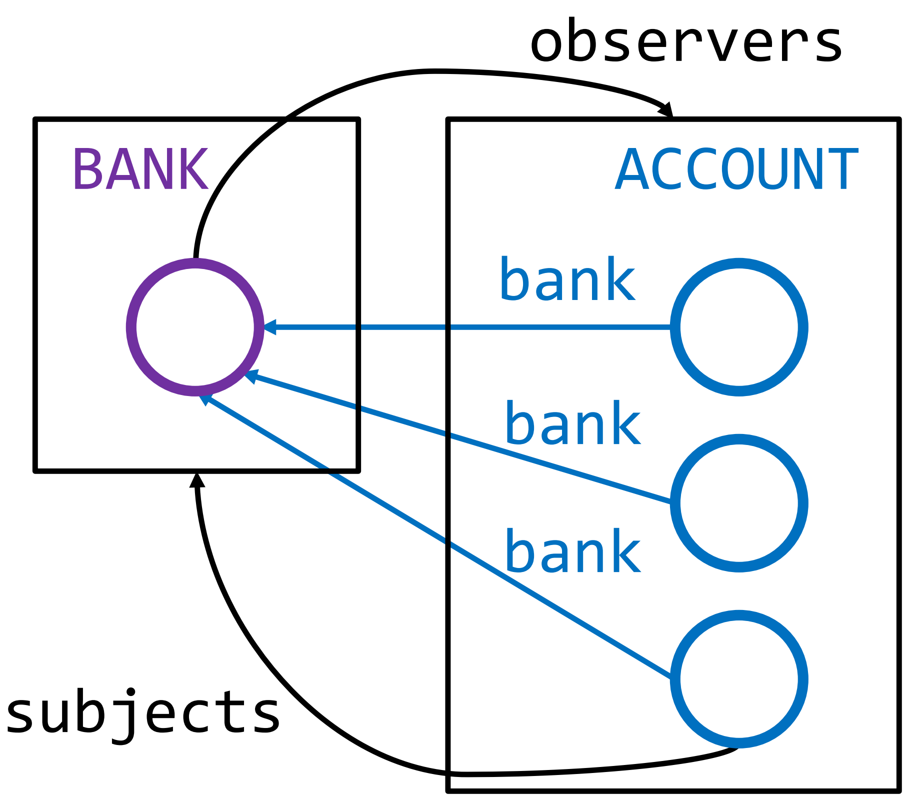



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

```
update: open bank, observers
```

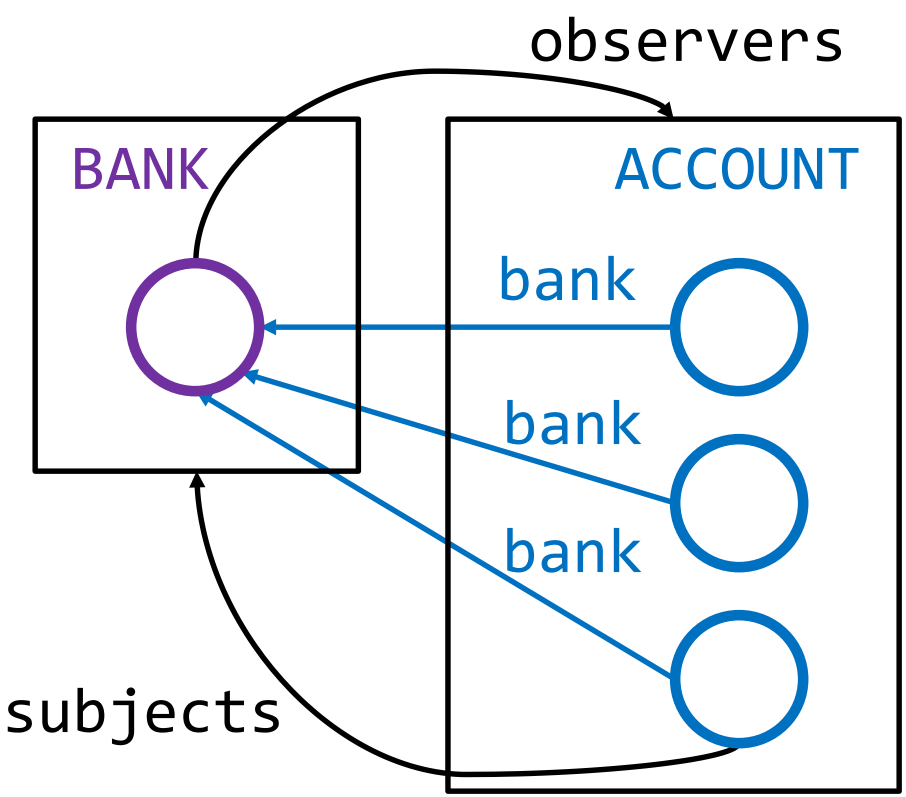



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

```
update: set rate
```

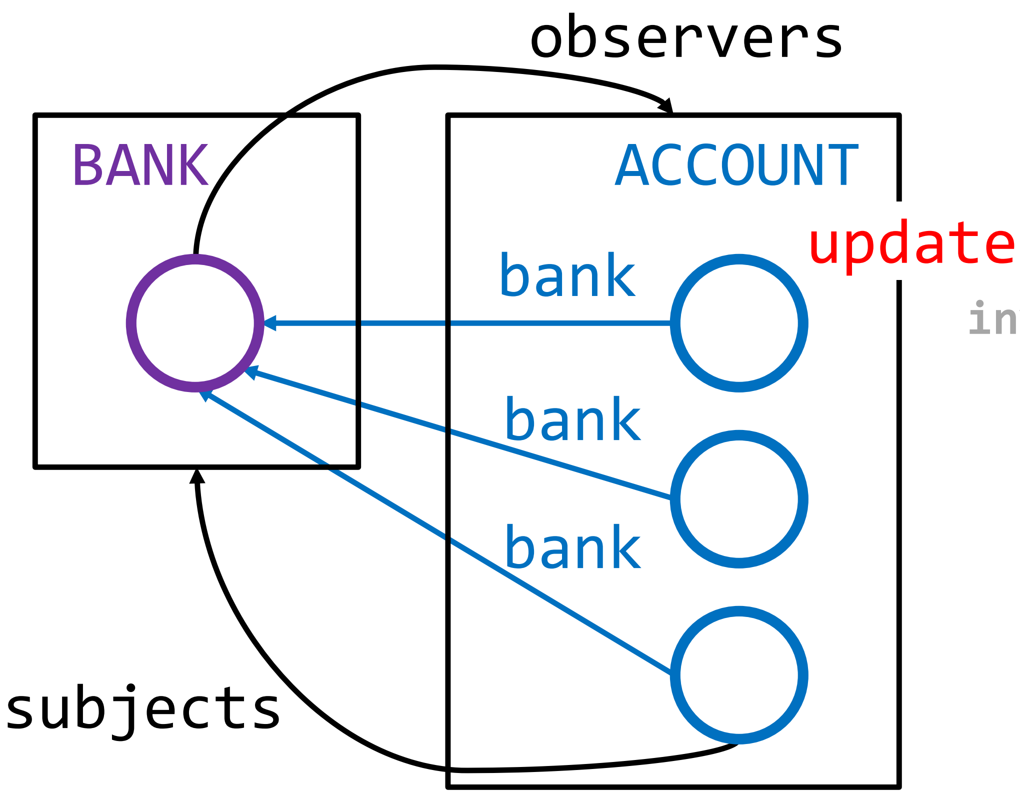



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

```
update: set rate, notify all accounts
```

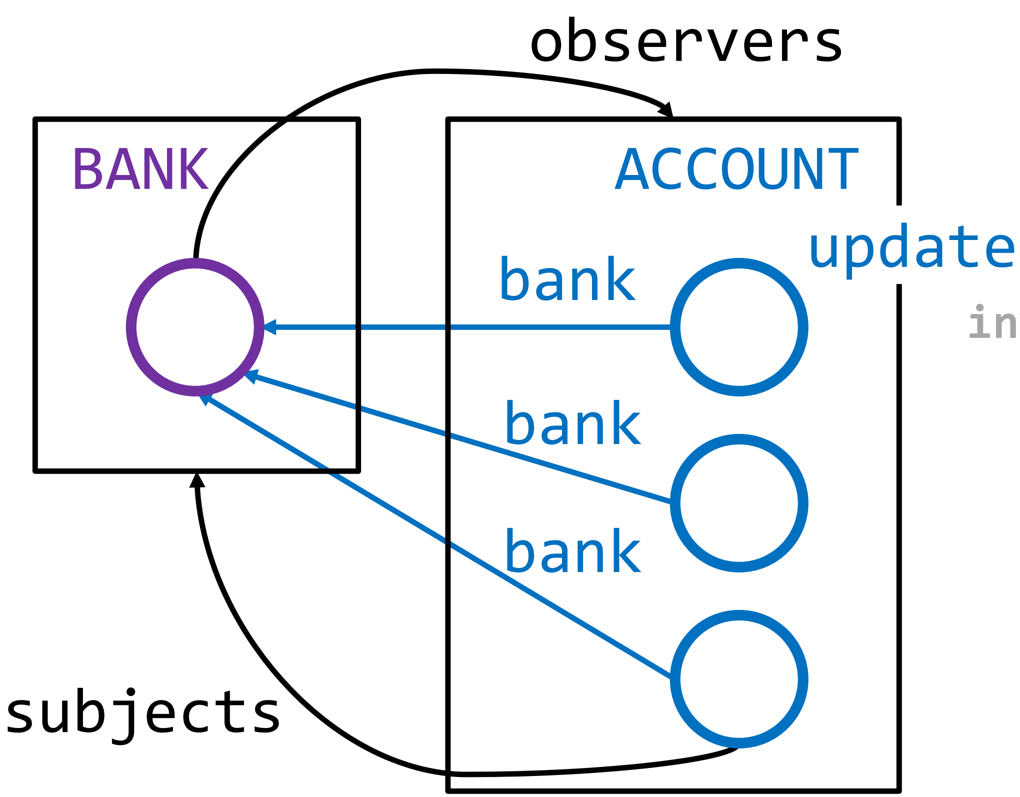



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

```
update: set rate, notify all accounts
```

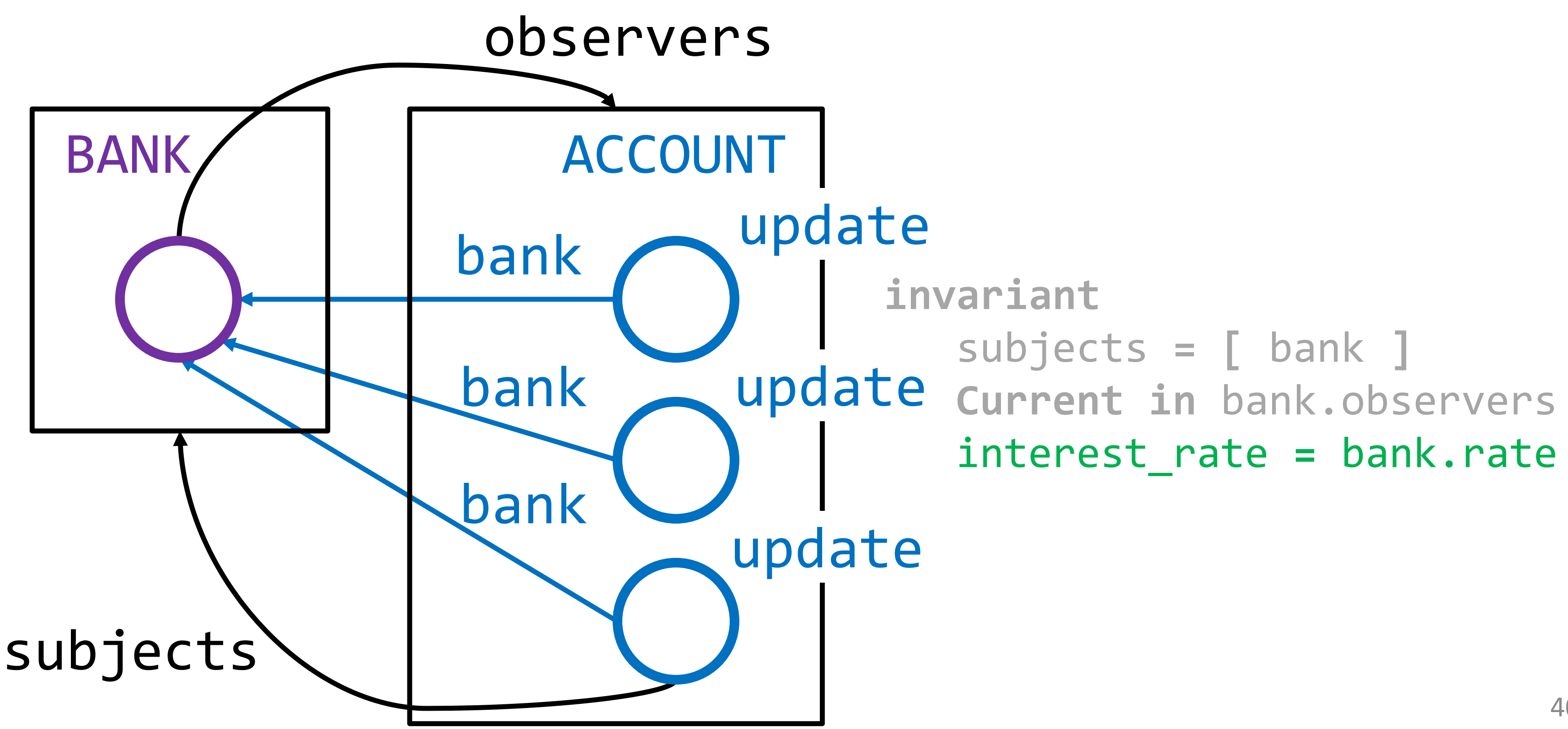



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

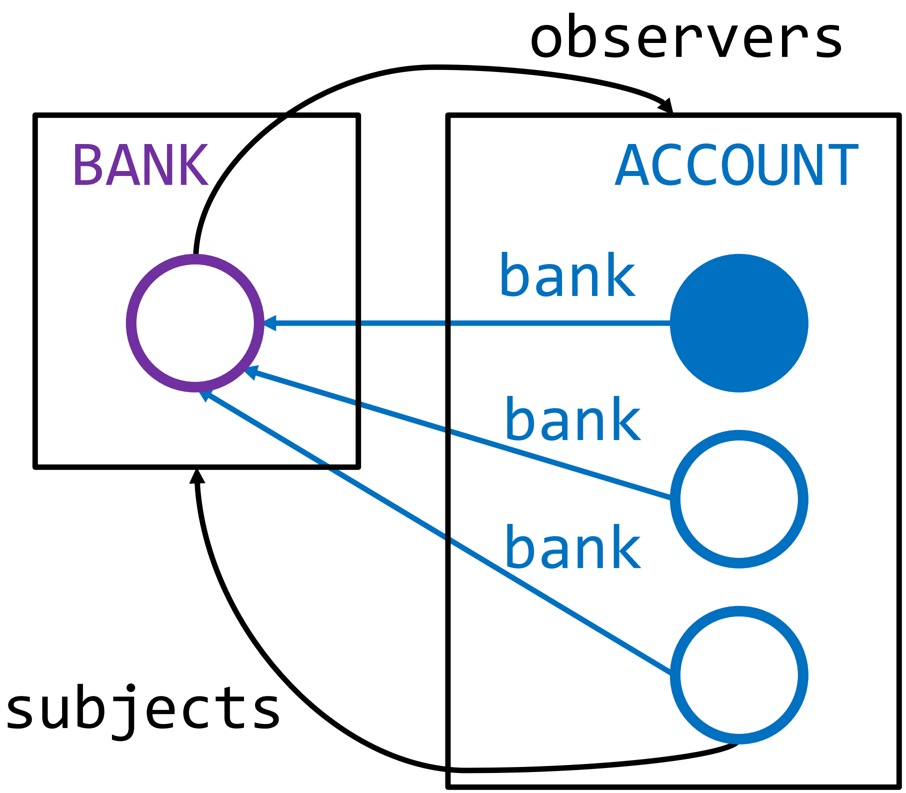
`update: set rate, notify all accounts`



Semantic collaboration

The bank changes the rate (and notifies accounts)

`update: wrap bank, all observers (check)`

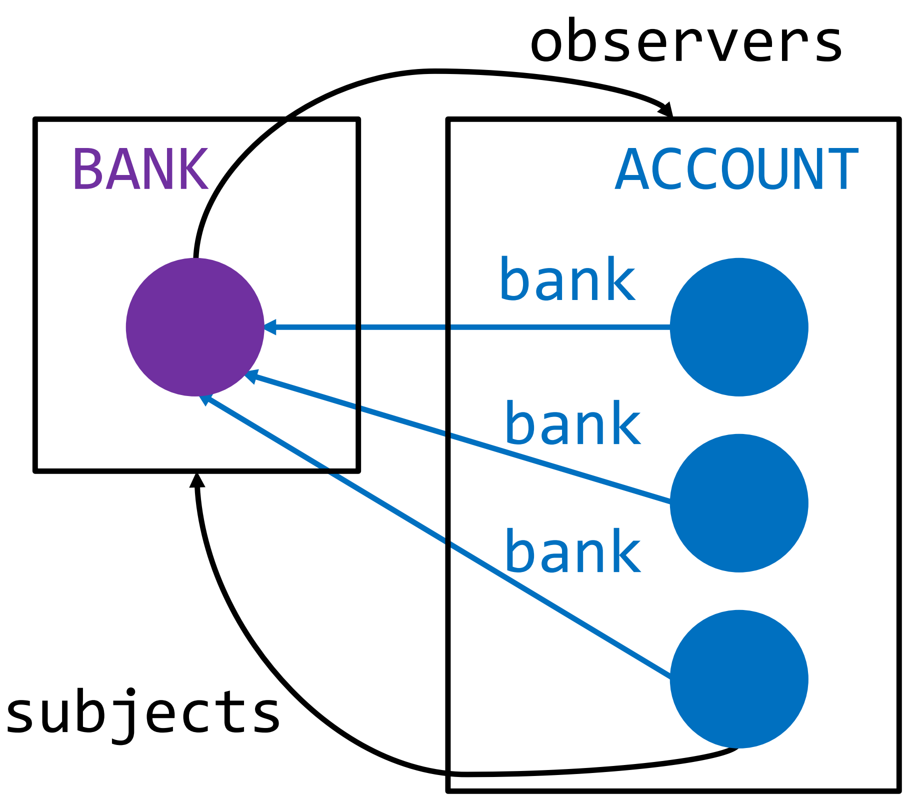



```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Semantic collaboration

The bank changes the rate (and notifies accounts)

`update: open, modify, wrap (check)`




```
invariant
    subjects = [ bank ]
    Current in bank.observers
    interest_rate = bank.rate
```

Demo: collaboration in AutoProof

AutoProof verifies `update_rate` in ACCOUNT and `change_master_rate` in BANK based on semantic collaboration features

```
subjects_definition: subjects = [ bank ]
consistent_rate: interest_rate = bank.master_rate
```

Follow this demo at:

http://comcom.csail.mit.edu/e4pubs/#demo-key

(Tabs account_collaboration.e and bank.e)

Wrapping and unwrapping

In hierarchical structures there is one typical modification pattern:

unwrap, modify, wrap (check consistency)

In collaborative structures, there is more flexibility:

- unwrap, modify, wrap
- unwrap, modify, leave open (invalidate)
- share responsibility for restoring consistency between subjects and observers

Data structures

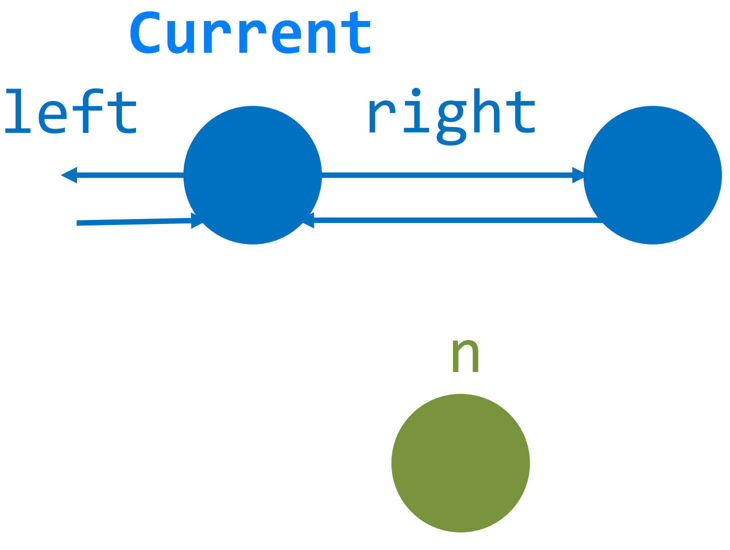
The features of semantic collaboration work well to reason about data structure implementations.

Data structures: doubly-linked list

As an example, let's outline node insertion in a doubly-linked list:

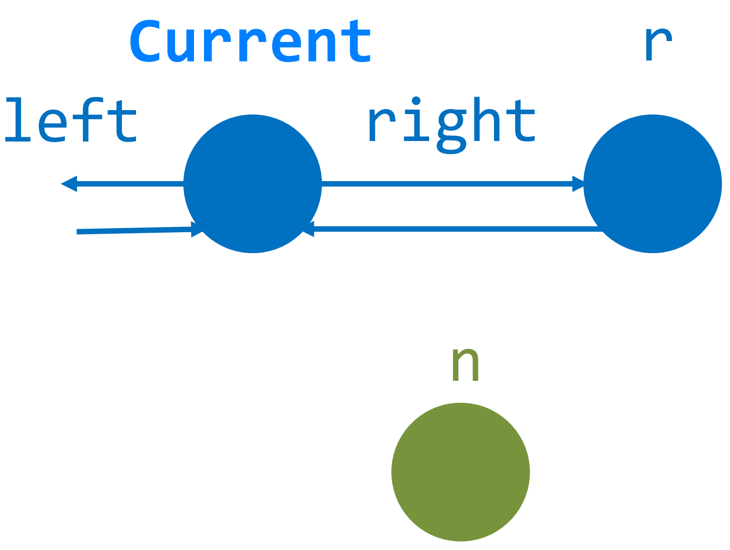
- A singly linked list is hierarchical: the head controls access to the whole list.
- A (circular) doubly-linked list is collaborative: every node depends on its neighbors, and they depend on it

Insert node n to right of Current



Insert node n to right of `Current`

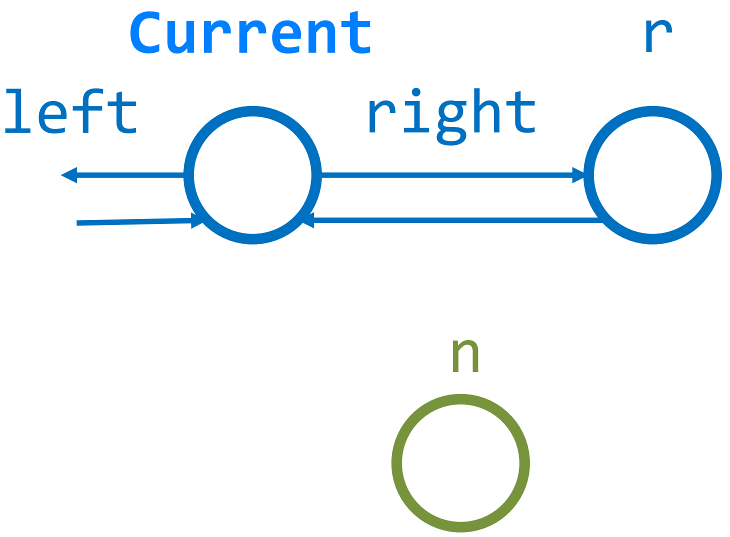
```
var r := right
```



Insert node n to right of **Current**

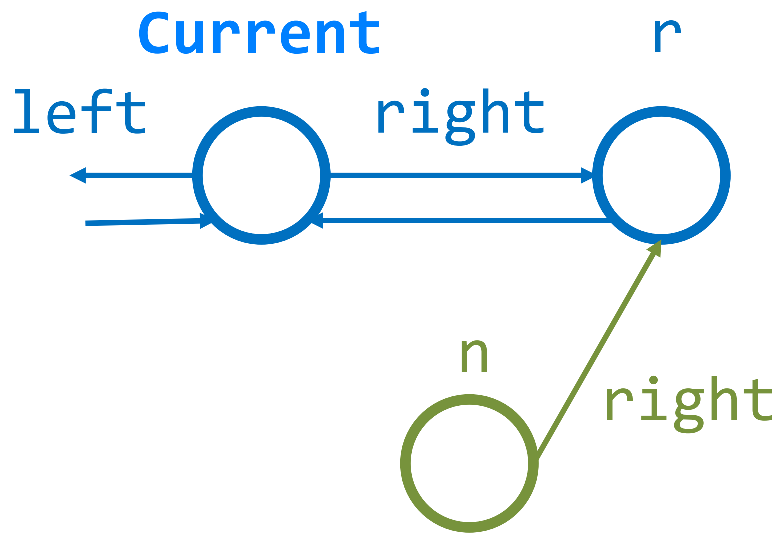
```
var r := right

unwrap Current, r, n
```



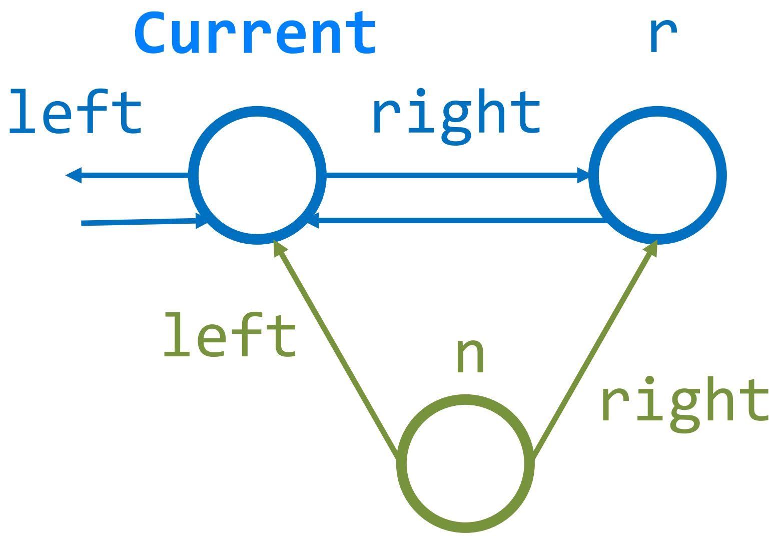
Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
```



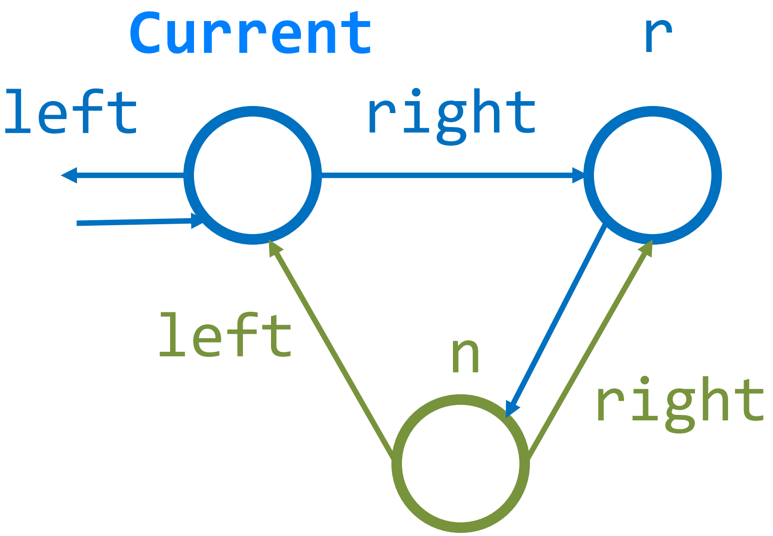
Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
```



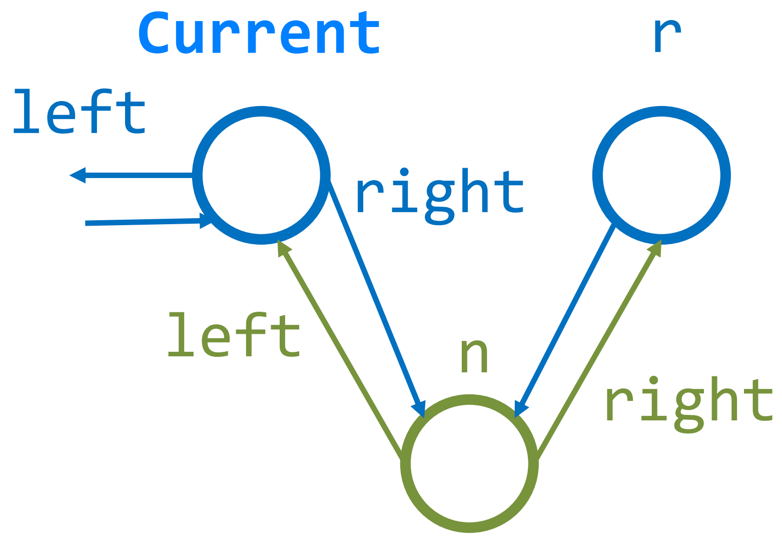
Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
r.left := n
```



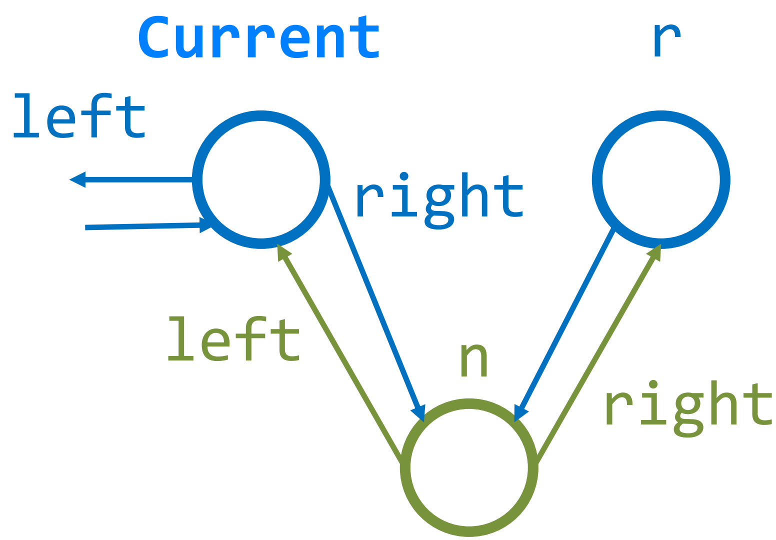
Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
r.left := n
right := n
```



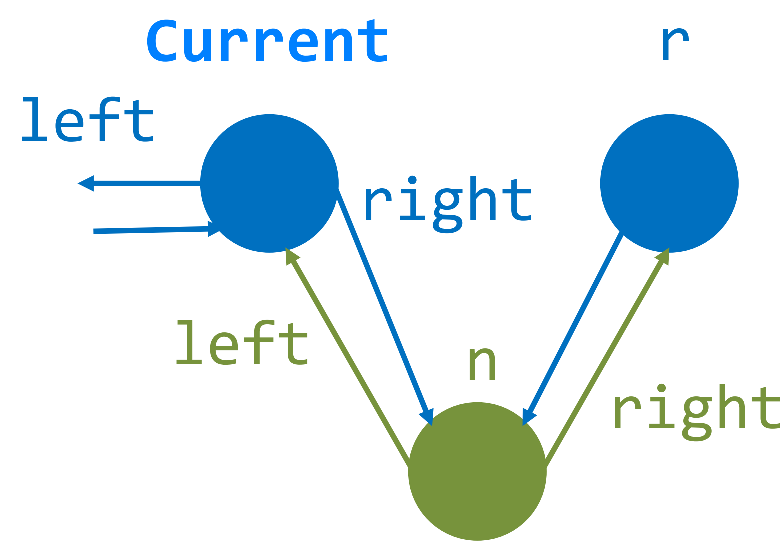
Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
r.left := n
right := n
n.subjects, n.observers := [r, Current]
subjects, observers := [left, n]
r.subjects, r.observers := [n, r.right]
```

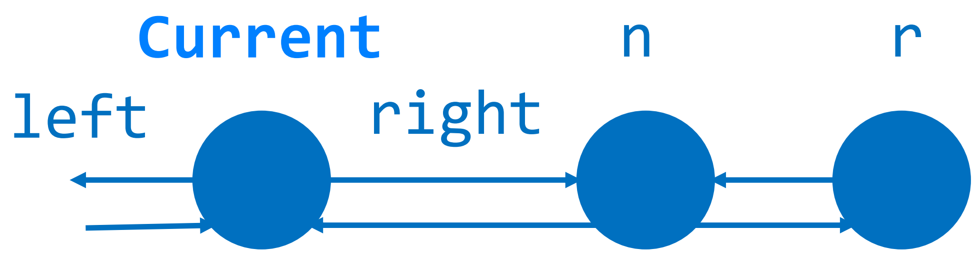


Insert node n to right of Current

```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
r.left := n
right := n
n.subjects, n.observers := [r, Current]
subjects, observers := [left, n]
r.subjects, r.observers := [n, r.right]
wrap Current, r, n
```



Insert node n to right of Current




```
var r := right
unwrap Current, r, n
n.right := r
n.left := Current
r.left := n
right := n
n.subjects, n.observers := [r, Current]
subjects, observers := [left, n]
r.subjects, r.observers := [n, r.right]
wrap Current, r, n
```

Attribute update guards

Who’s responsible for checking that an update to an attribute satisfies the invariant?

```
a: A guard: g(a’, o)
```

- every observer `o` of `Current` that satisfies the guard `g` is responsible for checking that updating `Current`’s attribute `a` to the value `a’` does not violate the invariant of `o`

Update guards in doubly-linked list

```
right: NODE guard: o /= right
```

When changing the value of attribute `right`:

- the `left` node checks that its invariant is not violated by changing `right` in the current node
  - the `left` node's invariant does not depend on `Current.right` (it remains wrapped)
- the current node checks that `right`'s invariant is not violated by changing `Current.right`
  - the `right` node is open when changing `Current.right` (invariant vacuously holds)
  - actual check performed when wrapping `right`

Demo: doubly-linked list

AutoProof verifies class NODE, representing the generic node of a doubly-linked list

```
insert_right (n: NODE)
    -- Insert n to the right of Current.
```

Follow this demo at:

http://comcom.csail.mit.edu/e4pubs/#demo-key

(Tab node.e)

Proving realistic implementations

Semantic collaboration is part of a verification framework with features suitable to reason about realistic implementations:

- model-based specifications
  - completeness
- extensible specification types and MML library
- (abstract) framing with inheritance
- modular verification with inheritance
  - nonvariant, covariant methods
- finely-tuned encoding in AutoProof

AutoProof on realistic software

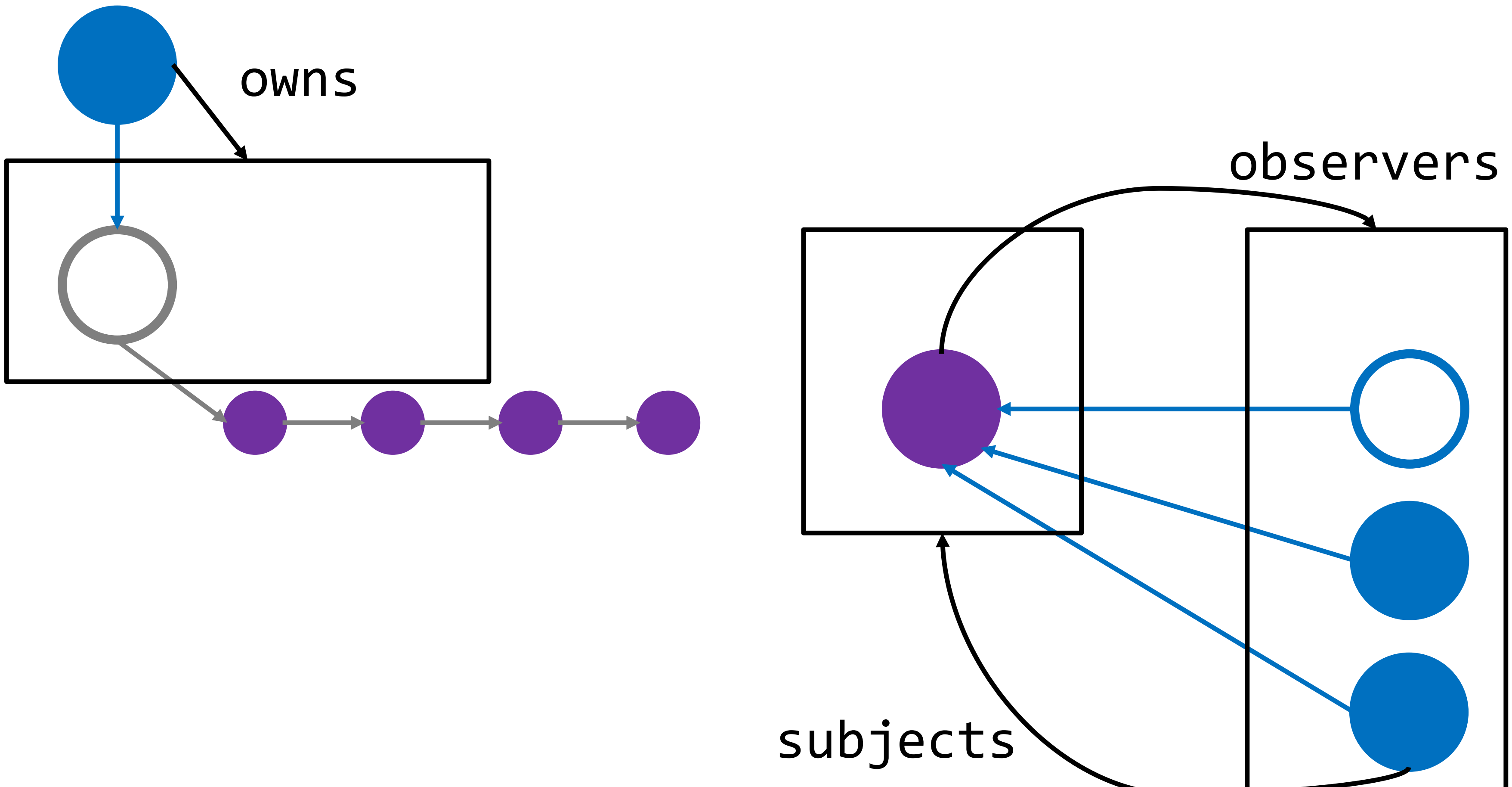
Verification benchmarks:

# programs	LOC	SPEC/CODE	Verification time
25	4400	Lines: 1.0 Tokens: 1.9	Total: 3.4 min Longest method: 12 sec Average method: < 1 sec

EiffelBase2 – a realistic container library:

# classes	LOC	SPEC/CODE	Verification time
46	8400	Lines: 1.4 Tokens: 2.7	Total: 7.2 min Longest method: 12 sec Average method: < 1 sec

Class-invariant based reasoning with semantic collaboration



[VSTTE ‘13, FM ‘14, TACAS ‘15, FM ‘15, STTT ‘16]