Using capabilities for strict runtime invariant checking

Isaac Oscar Gariano, Victoria University of Wellington, New Zealand
Marco Servetto, Victoria University of Wellington, New Zealand
Alex Potanin, The Australian National University, Australia

Reasoning about OO programs

- Method based
  - Precondition
    - something that holds
      before we call a method
  - Postcondition
    - something that holds
      when the method completes
- Type/Class based
  - Representation Invariant
    - something that holds … always?

Invariants

Representation invariants
Class invariants
Object invariants
Type refinements

predicates on the state of an object
and its reachable object graph (ROG)

- They can be presented as documentation, checked as part of static verification, or, as we do in our approach, monitored for violations using runtime verification.
- In our system a class specifies its invariant by defining a method called invariant() that returns a Boolean.
- Invariant holds == invariant() method would return true.

Invariants

Easier in Purely functional setting:

- the programmer only needs to write the code for the invariant check itself, then the runtime needs to call this code each time a value/object is created.

Harder in an impure setting, like most OO languages:

- operations on data structures are often implemented as complex sequences of mutations, where the invariant is temporarily broken.
- To support this behaviour, most invariant protocols present in the literature allow invariants to be broken and observed broken.

Existing invariant protocols

- How do they allow invariants
  to be broken and observed broken?

  The most common idea: Visible state semantics

  ~= invariants can be broken when a (public?) method on the object is active
  (that is, currently executing)

  ~= invariants are just extra pre/post conditions

- The cost:

  Without global knowledge of the current state of the stack trace,
  it is hard to know if any object at any time is in a broken state

  → not very useful for modular reasoning

```
class Range{//Pseudocode example
  private field min;
  private field max;

  method invariant(){ return min < max; }

  method set(min, max){
    if( min >= max ){ throw new Error(...); }
    this.min = min;
    //Here 'this' could be broken
    this.max = max;//Here we fix it
  }
}
```

Good case

- All good, range objects are only seen when their invariant holds.
- We check if the two new min/max values are valid, then we do the two updates atomically
  (parallelism can still break stuff, but it is out of scope for this talk)

Bad case

```
class Range{//Pseudocode example
  private field min;
  private field max;

  method invariant(){ return min < max; }

  method set(min, max){
    if( min >= max ){ throw new Error(...); }
    this.min = min;
    Do.stuff(this);//Do.stuff can now see a broken range
    this.max = max;
  }
}
```

- Now `Do.stuff` and all the code called by `Do.stuff` can see a broken range. This is not desirable.
- We propose a much stricter invariant protocol: at all times, the invariant of every object involved in execution must hold.

Our strict invariant protocol

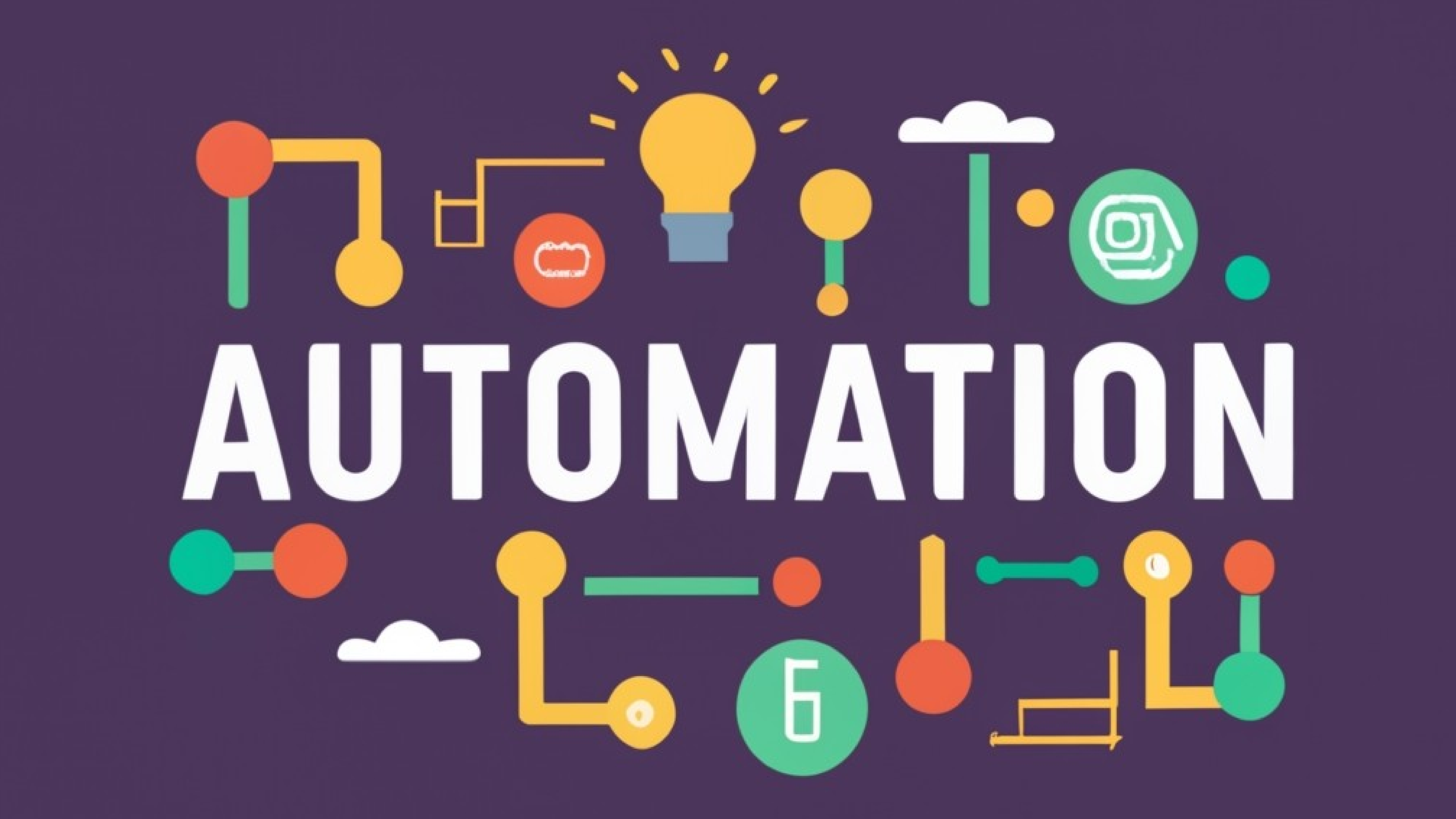
- At all times, the invariant of every object involved in execution holds.
- Thus objects can be broken only when the object is not (currently) involved in execution.
- An object is **involved in execution** when it is in the ROG of any of the objects mentioned in the method call, field access, or field update that is about to be reduced

- Our strict protocol supports easier reasoning: an object can never be observed broken.
- We argue that it is not overly restrictive, and we show many examples supporting our statement.

How to do Range?

```
  class BoxRange{//no invariant in BoxRange
   field min;
   field max;
   BoxRange(min, max){ this.set(min, max); }
   method set(min, max){
    if(min >= max){ throw new Error(...); }
    this.min = min;
    this.max = max;
  }
}
class Range{
  private field box; //box contains a BoxRange
  Range(min, max){ this.box = new BoxRange(min, max); }
  method invariant(){ return this.box.min < this.box.max; }
  method set(min, max){ this.box.set(min, max); }
}
```

- Easy if we accept a little more indirection.
- The idea is that the outer `Range` object is not in the ROG of any object involved in the call to '`this.box.set(min, max)`'


AUTOMATION

Can languages ensure that we follow our approach correctly? Can they inject the checks for us?

```
capsule, mut, imm, read //Many type systems now support those

rep //We add support for ‘rep’ fields
```

- **Type systems can do it!**
- There are now at least 3 different programming languages supporting reference and object capabilities strong enough to encode the restrictions we need:
  L42, Pony and an Internal Microsoft language (Gordon et al)
- Any language with those safety features can support our work.
- Not enough time to explain reference and object capabilities
  - Experiments show that they are not too restrictive or invasive (whole sections of code can basically opt out)

EXAMPLES

```
class Person {
  read method imm Bool invariant() { return !name.isEmpty(); }
  private imm String name;
  read method imm String name() { return this.name; }
  mut method imm Void name(imm String name) {
    this.name = name; // check after field update
    if (!this.invariant()){ throw new Error(...); }
  }
  Person(imm String name) {
    this.name = name; // check at end of constructor
    if (!this.invariant()){ throw new Error(...); }
   }
}
```

Person with valid name

- Full code: explicit checks and explicit ‘imm’ capabilities

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;
  read method String name() { return this.name; }
  mut method Void name(String name) {
    this.name = name;
  }
  Person(String name) {
    this.name = name;
   }
}
```

Person with valid name

- Simplified code:
  checks inserted by the language
  and implicit ‘imm’ capabilities

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;
  read method String name() { return this.name; }
  mut method Void name(String name) {
    this.name = name;
  }
  Person(String name) { this.name = name; }
}
```

What could go wrong here?

This code is accepted by our approach.

We want this code to correctly enforce the invariant.

We want to make sure that no other code can sneakily break it!

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;
  read method String name() { return this.name; }
  mut method Void name(String name) {
    this.name = name;
  }
  Person(String name) { this.name = name; }
}
```

What could go wrong here?


```
class EvilString extends String {//INVALID EXAMPLE
  @Override read method Bool isEmpty() {

    return new Random().nextBool();

  }}
...
method mut Person createPersons(String name) {
// we can not be sure that name is not an EvilString
mut Person schrodinger = new Person(name); // exception here?
assert schrodinger.invariant(); // will this fail?
```

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;
  read method String name() { return this.name; }
  mut method Void name(String name) {
    this.name = name;
  }
  Person(String name) { this.name = name; }
}
```

What could go wrong here?


```
class EvilString extends String {//INVALID EXAMPLE
  @Override read method Bool isEmpty() {
    //Creates a new object capability out of thin air
    return new Random().nextBool();//We restrict the creation of objects
                                   //with system dependent / non deterministic behavior
  }}
...
method mut Person createPersons(String name) {
// we can not be sure that name is not an EvilString
mut Person schrodinger = new Person(name); // exception here?
assert schrodinger.invariant(); // will this fail?
```

```
class Person {
   read method Bool invariant() { return !name.isEmpty(); }
   private String name;
   read method String name() { return this.name; }
   mut method Void name(String name) {
      this.name = name;
   }
   Person(String name) { this.name = name; }
}
```

What could go wrong here?

```
class MagicCounter {//INVALID EXAMPLE
   Int counter = 0;
   method Int incr(){ return unsafe{ counter++ }; } //using a backdoor
}
class NastyS extends String {..
   MagicCounter c = new MagicCounter(0);//can be 'imm' since it is 'unsafe'
   @Override read method Bool isEmpty(){ return this.c.incr() != 2; }
}
...
NastyS name   = new NastyS(); //the type system believes name's ROG is immutable
Person person = new Person(name); // person is valid, counter=1
name.incr();           // counter == 2, person is now broken
person.invariant();   // returns false, counter == 3
person.invariant();   // returns false, counter == 4
```

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;
  read method String name() { return this.name; }
  mut method Void name(String name) {
    this.name = name;
  }
  Person(String name) { this.name = name; }
}
```

What could go wrong here?


```
mut Person bob = new Person("Bob");//INVALID EXAMPLE
//Catch and ignore invariant failure:
try { bob.name(""); } catch (Error t) { }// bob mutated
assert bob.invariant(); // fails!
```

```
try { mut Person bob = new Person("Bob"); bob.name(""); }//Allowed
catch (Error t) { }
```

Invariants on mutable state

Invariants on mutable state

Case 1: The invariant relies on immutable state (String name)

→ we can only affect it by a field update.

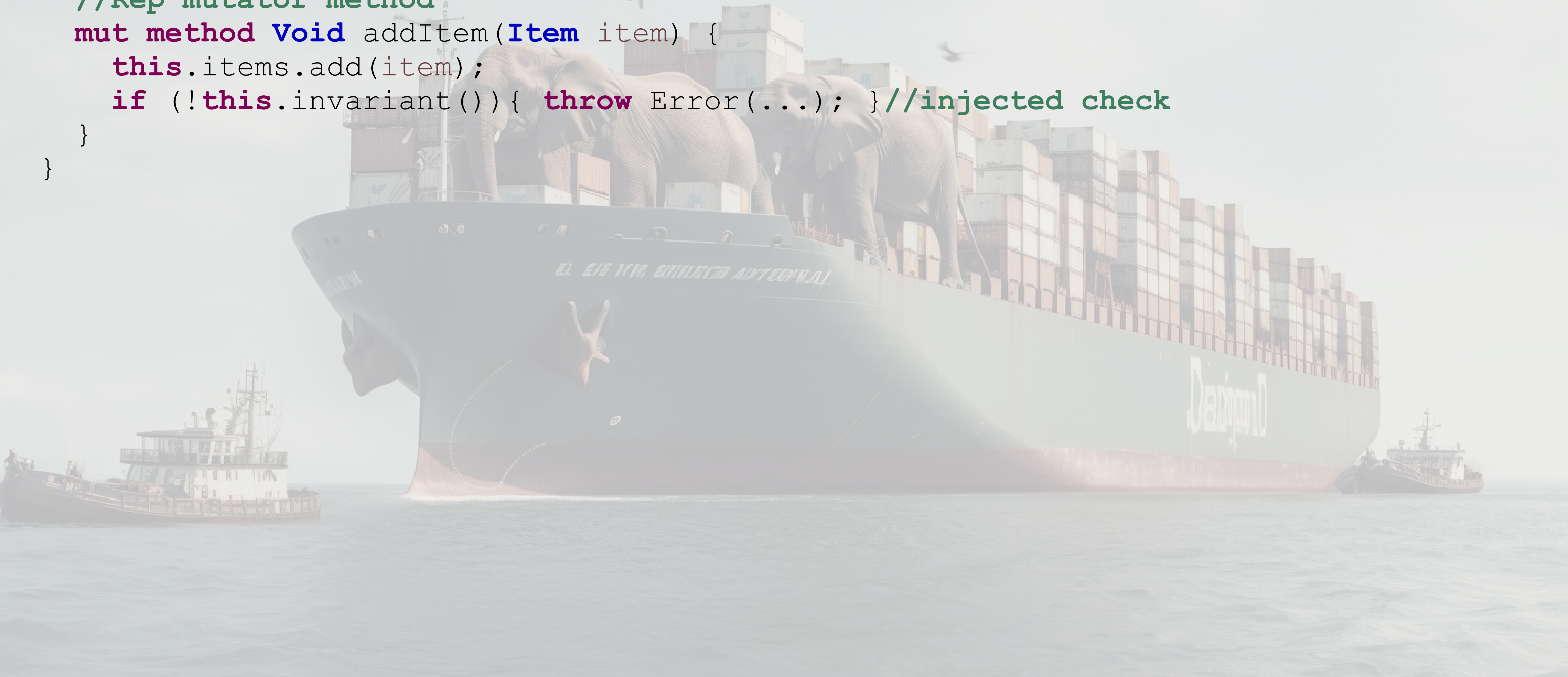
Case 2: The invariant relies on mutable state

→ we can also affect it by a mut method accessing the fields

Invariants on mutable state

```
class ShippingList {
  rep Items items;
  read method Bool invariant(){ return this.items.weight() <= 300; }
  ShippingList(capsule Items items) { this.items = items; }

  //Rep mutator method
  mut method Void addItem(Item item) {
    this.items.add(item);
    if (!this.invariant()){ throw Error(...); }//injected check
  }
}
```



Invariants on mutable state

```
class ShippingList {
  rep Items items;
  read method Bool invariant(){ return this.items.weight() <= 300; }
  ShippingList(capsule Items items) { this.items = items; }

  //Rep mutator method
  mut method Void addItem(Item item) {//1 no mut/read parameters
    this.items.add(item);//2 only one use of 'this'
    if (!this.invariant()){ throw Error(...); }//injected check
  } //3 no mut return type
}
```

- Rep field:
  - can only be initialized/updated with a capsule value
  - can be accessed as ‘read’
  - can only be accessed as ‘mut’ within a rep mutator method
- Rep mutator method:
  - ‘mut’ method accessing a rep field as ‘mut’
  - No mut/read parameters, only 1 use of this, no mut return type

Soundness proof (overview)

- An object is **valid** when calling its invariant() method would deterministically produce ‘true’ in a finite number of steps: this means it does not evaluate to another value, fail to terminate, or produce an error.
- An object is **involved in execution** if it is a receiver or a parameter in the current execution step, or in the ROG of an object involved in execution.
- **If a program is well typed, in any execution step any object involved in execution is valid.**
- This also implies that when the execution is in the body of a method:
  - all objects reachable from temporary results, visible local variables/parameters and returned values are valid.

Performance advantages

- Different runtime verification strategies may run the invariant checks a different amount of times.
- We compared the performance of our approach on a number of examples.
- We discovered that runtime verification tools based on visible state semantic can run exponentially more invariant checks!
- On a realistic problem, we recorded:
  - Language D: invariant() method called 52, 734, 053 times
  - Language Eiffel: invariant() method called 14, 816, 207 times
  - Our approach: invariant() method called 77 times
  - Language Spec#: invariant() method called 77 times

Simplicity advantages

- Different verification systems may be easier or harder to use, and may have different level of expressivity.

- Our main point of comparison was Spec#. Spec# annotations were 10 times more verbose, and in my opinion orders of magnitude harder to understand and use.

Expressivity advantages

- Different verification systems may be easier or harder to use, and may have different level of expressivity.

- Expressivity: we discovered many situations that could be handled by our approach and not by Spec# or similar tools; in the same way we discovered many situations that can not be handled by our approach but can be handled by Spec#
- A problem (Graph/Node invariants) that is considered one of the hardest problems in verifying OO invariants is easily solved using our approach

Concluding

- Reference and Object capabilities have been used to prove correct/unobservable parallelism.
- We have proven that they can be used to ensure representation invariants.
- We have work in progress to show that they can be used for correct caching and automatic cache invalidation too.

- **Any Question?**