Verification of Object-Oriented Programs with Invariants

Best Group

Alexander Borsboom, Andrew Hughson, Andrew Luey, Sam Metson, Tony Young

Serious Agenda

1. Overview
2. Invariant Validation
3. Component Invariants
4. Ownership Semantics
5. Subclass Invariants
6. Applicability
7. Limitations
8. Conclusions

Object Invariant Overview

- A way to ensure objects are *correct*.
- `invariant` — condition of an object that can be relied upon to be true during execution of a program, or during some portion of it.
- A program is erroneous if it ever reaches a false `assert`.
- Pre/Post conditions.
  - Predicates which are true on entry/exit of methods on an object.

Example with Pre/Post Conditions

```
class T {
    private x, y : int;

    public constructor T() ensures 0 ≤ x < y; {
        x := 0; y := 1;
    }

    public method M() requires 0 ≤ x < y;
                      modifies x, y;
                      ensures 0 ≤ x < y; {
        assert y − x ≥ 0;
        x := x + 3; y := 4 * y;
    }
}
```

Better Invariants

- Pre/Post conditions don't ensure validity while in a method - bad.
- Pre/Post conditions expose internal implementation - bad.
- Better - expose only valid/invalid state.
- States = { *Valid*, *Invalid* }
    - *Valid* — invariants are checked and true.
    - *Invalid* — invariants are not necessarily true.
- Invariants restricted to object fields.
    - Start simple, then extend to the concepts of *components* and *subclasses*.

Invariant Validation

- Fields are read only when object is valid.
- `pack` and `unpack` allow writes to fields.

`pack o`

1. assert `o.state` is *Invalid*
2. assert invariants are true on `o`
3. set `o.state` to *Valid*

`unpack o`

1. assert `o.state` is *Valid*
2. set `o.state` to *Invalid*

Example with Object Invariants

```
class T {
    private x, y : int;
    invariant 0 ≤ x < y;
    public constructor T() ensures st = Valid; {
        x := 0; y := 1;
        pack this;
    }

    public method M() requires st = Valid; modifies x, y; {
        assert y − x ≥ 0;
        unpack this;
        x := x + 3; y := 4 * y;
        pack this;
    }
}
```

Component Invariants

- *component* — a field with a complex type.
- A *component* of an object has fields which can be mentioned in the object invariant.
- How do we ensure components stay valid?
  - States = { *Valid*, *Invalid*, *Committed* }
- *Committed* means object is valid and has an owner.
- *Committed* objects are read only and can only be un-committed by their owner.
- Components are declared with `rep` modifier.

Component Invariants

`pack` o

1. assert `o.state` is *Invalid*
2. assert invariants are true on `o`
3. for each component `p`, check `p.state` is *Valid*
4. for each component `p`, set `p.state` to *Committed*
5. set `o.state` to *Valid*

`unpack` o

1. assert `o.state` is *Valid*
2. set `o.state` to *Invalid*
3. for each component `p`, set `p.state` to *Valid*

- new step

Component Invariants (pack)

```
public class Unicycle {
    public rep Wheel wheel;
    public rep Seat seat;
    invariant seat ≠ null ∧ wheel ≠ null;
}
```

1. Checks `Unicycle` invariant is true.
2. Checks if `Wheel` and `Seat` states are *Valid*.
3. Sets state of `rep` (component) fields to *Committed*.
4. Sets `this` state to *Valid*.

Component Invariants (unpack)

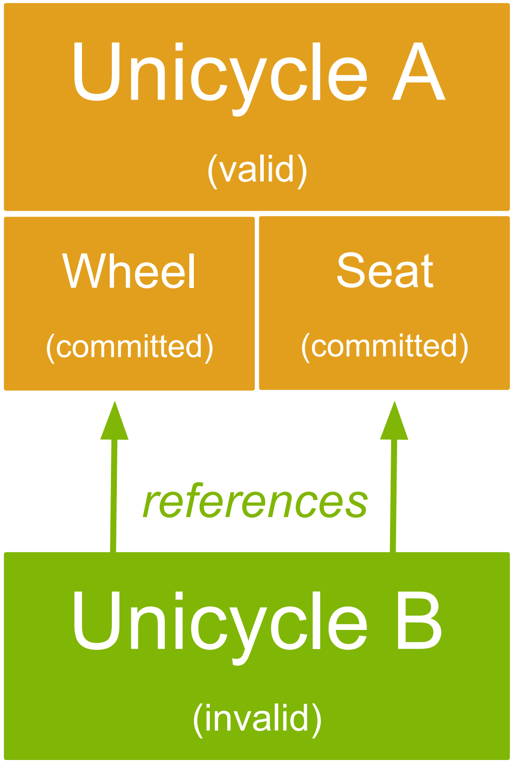
```
public class Unicycle {
    public rep Wheel wheel;
    public rep Seat seat;
    invariant seat ≠ null ∧ wheel ≠ null;
}
```

1. Checks `Unicycle` state is *Valid*.
2. Sets `Unicycle` state to *Invalid*.
3. Un-commits `Wheel` and `Seat` (sets their states to *Valid*).

Ownership Semantics

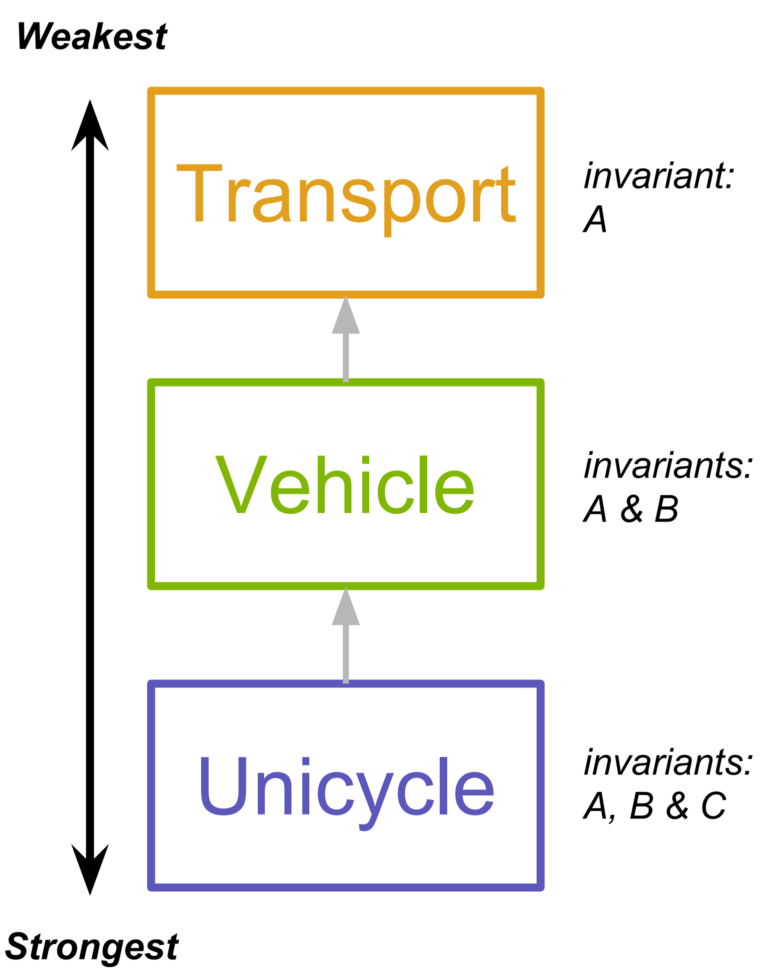
1. One valid "owner" can modify the components (rep).
2. Multiple invalid objects can have read references.

"Ownership" means that components are unique and other objects with different invariants cannot modify them.



Subclass Invariants

- `boolean committed`
- `inv`
  - The most derived class in inheritance hierarchy that `this` conforms to.
- Unpack to superclass, pack back down.
- When *Committed* is true, an object can't be written to.



Subclass Invariants

`pack o as T`

1. assert `o.inv` is set to the supertype of `T`
2. assert invariants are true on o
3. for each component p, check `p.inv` is `type(p)`
4. for each component p, set `p.committed` to `true`
5. set `o.inv` to `T`

`unpack o from T`

1. assert `o.inv` is `T`
2. set `o.inv` to supertype of `T`
3. for each component p, set `p.commited` to `false`

Applicability to *HazGas*

- An overarching System class which owns Room components.
- Invariants can be used to ensure Rooms are venting when the upper threshold is exceeded.
- The System class can have an invariant on alarming depending on Room components.
- Subclass invariants not required.
- Our implementation models with processes, so these methods are not applicable.

Limitations

- Usefulness limited for uncertainty — invariants may not always hold at runtime.
  - However, very useful for fixed-state systems.
- Can’t strengthen invariants safely, e.g.:

```
invariant x ≤ 10; /* 1 */
invariant x ≤ 5;  /* 2 */
```

Safe assignment doesn’t always exist from invariant 1 to 2; runtime checking needs to be enforced (invariant 2 is stronger than invariant 1).

Related Work

- Design by contract assertions exist in Eiffel and D at runtime only.
- Stronger type systems exist for compile-time verification:
  - Type-level programming: Haskell, Scala
  - Dependent typing: Idris
  - Theorem proving: Coq, Agda

Conclusions

- Useful for model checking, as models have a limited state space.
- May be cumbersome for large scale programs.
- *pack*/*unpack* pairs are used so we can have intermediate invalid states.
- Not completely verifiable at compile-time; some checks need to be inserted for runtime.

Questions?