

Construction and Verification of Software

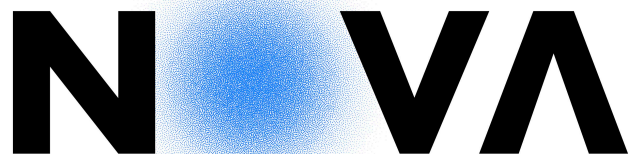
2021 - 2022

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 7 - Separation Logic

Bernardo Toninho (htoninho@fct.unl.pt)

based on previous editions by **João Seco** and **Luís Caires**



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Outline

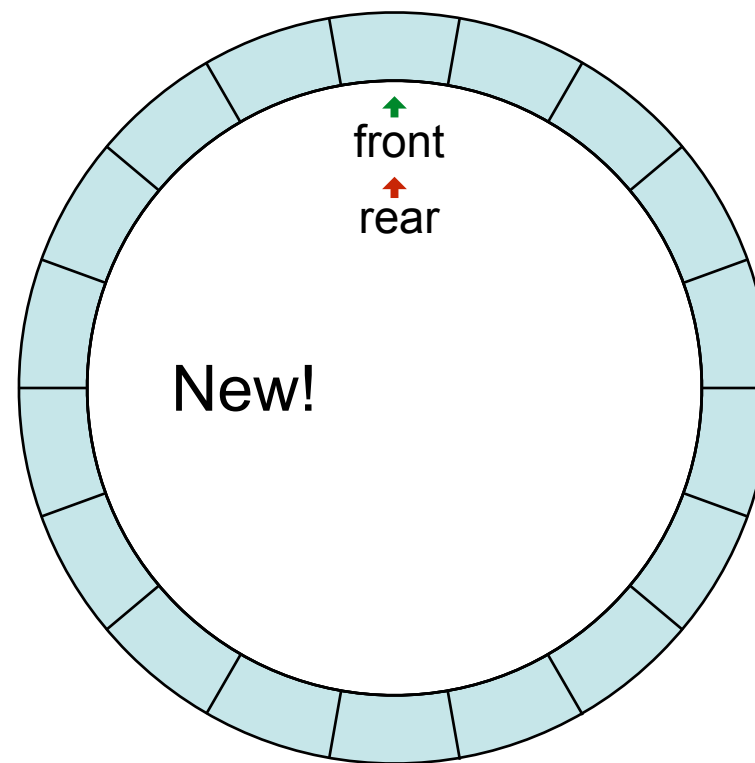
- Typestates (Example from last lecture!)
- Interference and aliasing (Overview)
- Separation Logic
- Separation Logic in Verifast
- An ADT by Example

Part I

Typestates (Example)

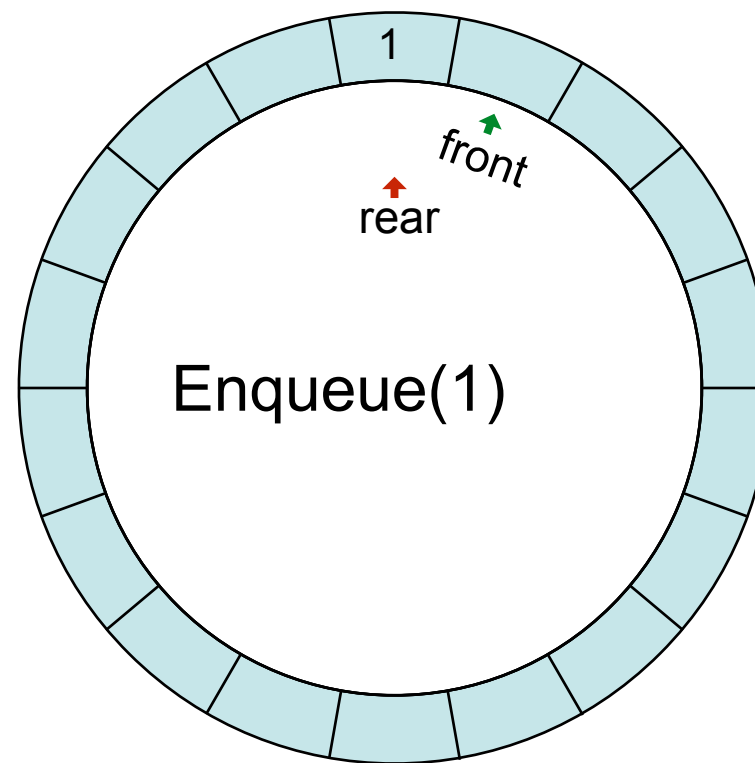
TypeStates - Queue

- An implementation using a circular buffer...



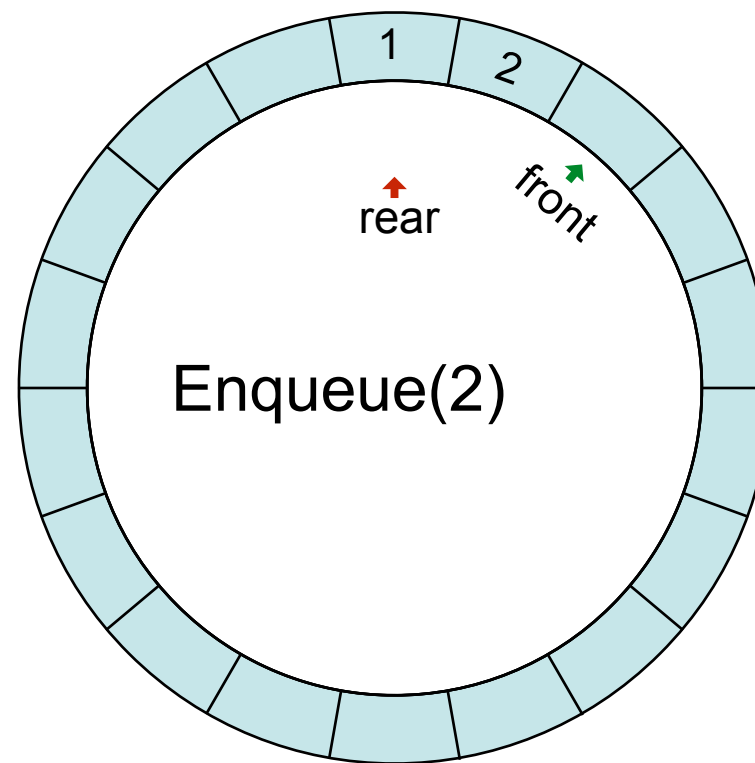
TypeStates - Queue

- An implementation using a circular buffer...



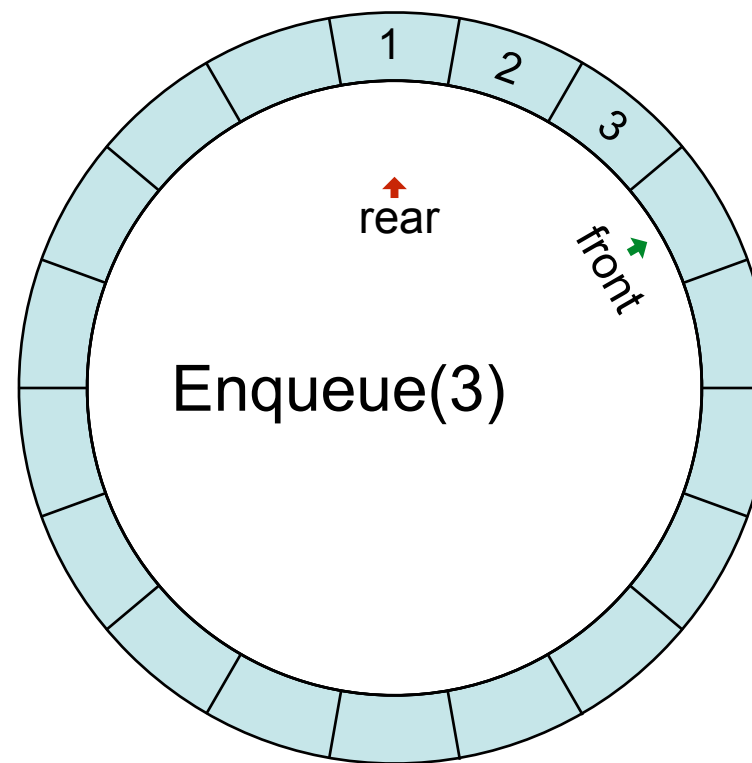
TypeStates - Queue

- An implementation using a circular buffer...



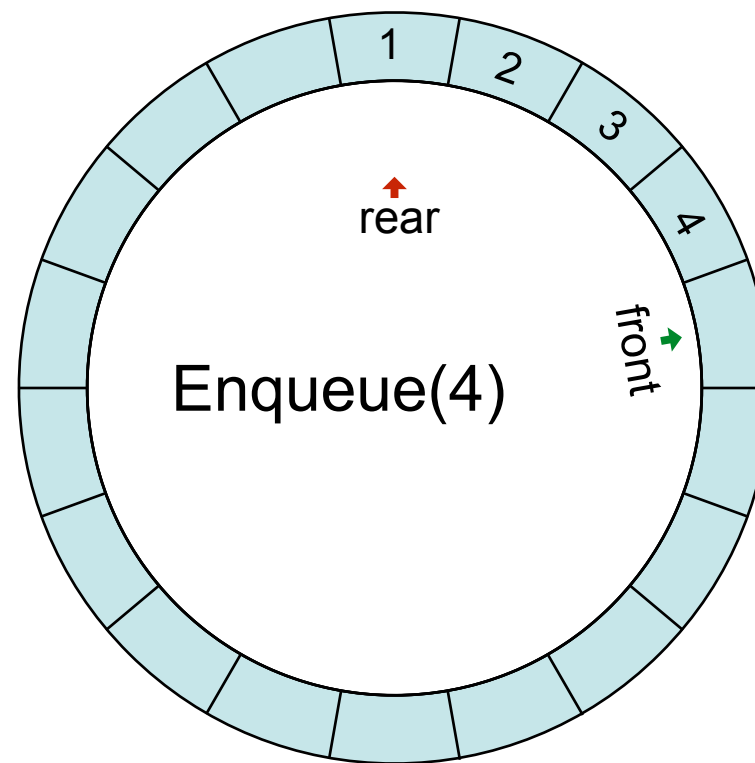
TypeStates - Queue

- An implementation using a circular buffer...



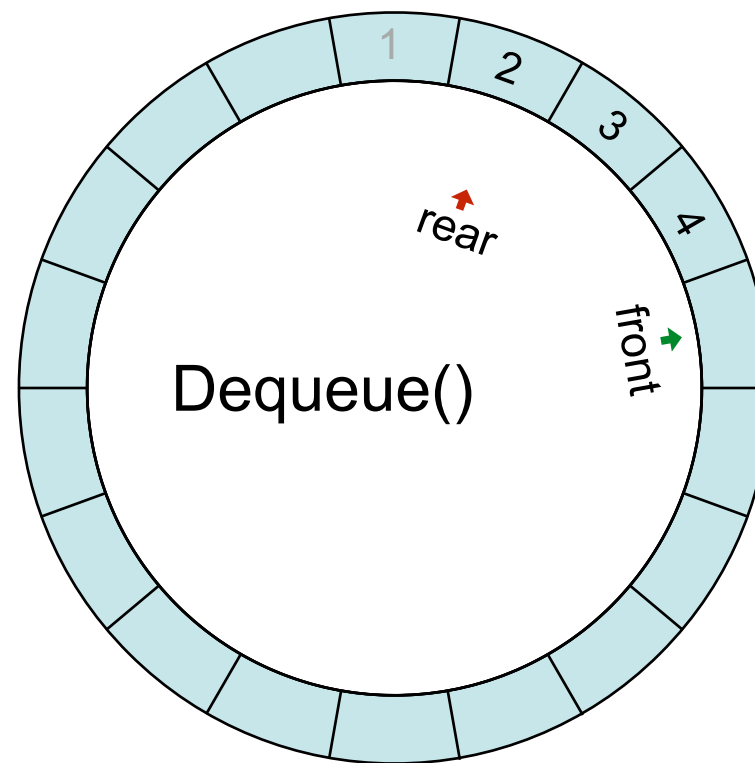
TypeStates - Queue

- An implementation using a circular buffer...



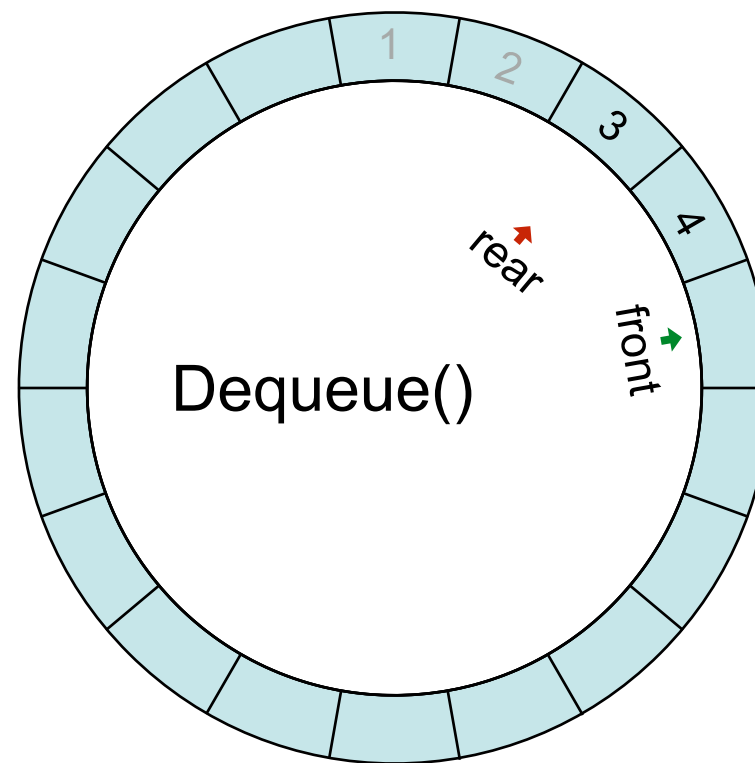
TypeStates - Queue

- An implementation using a circular buffer...



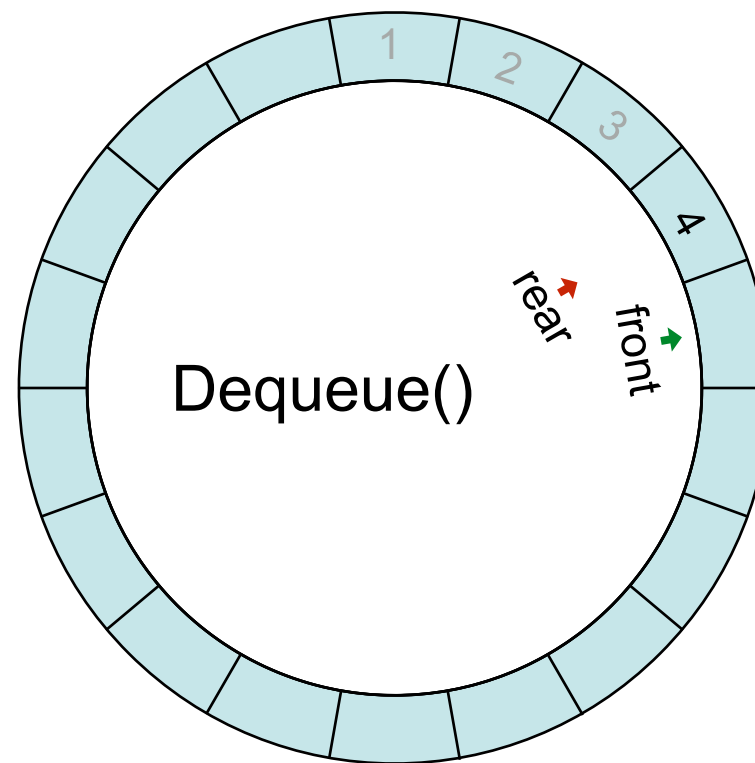
TypeStates - Queue

- An implementation using a circular buffer...



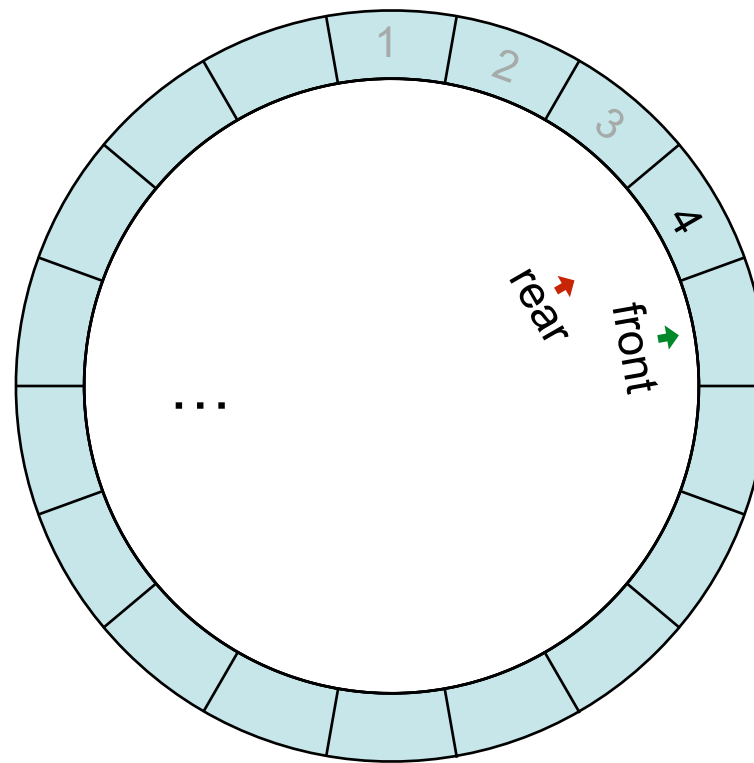
TypeStates - Queue

- An implementation using a circular buffer...



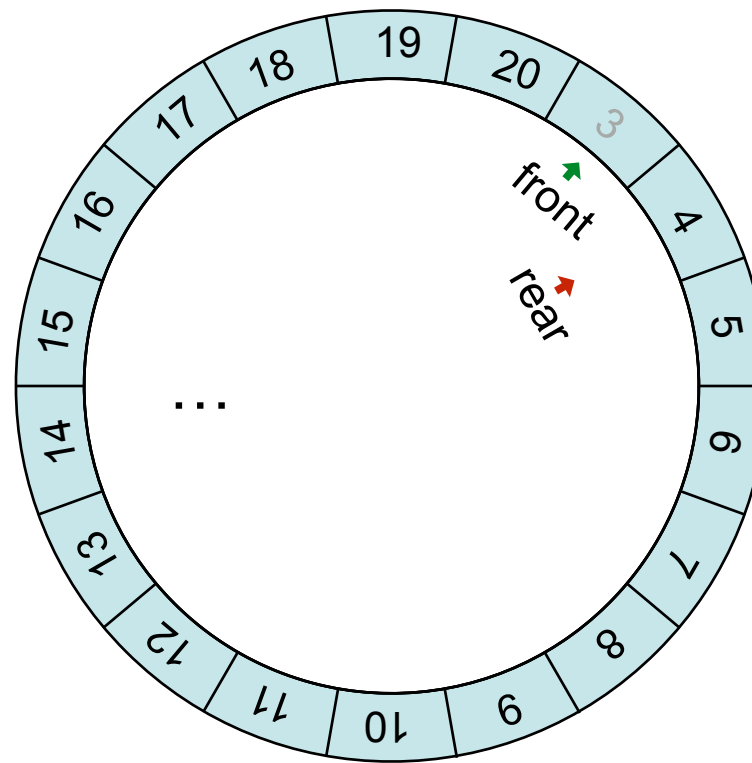
TypeStates - Queue

- An implementation using a circular buffer...



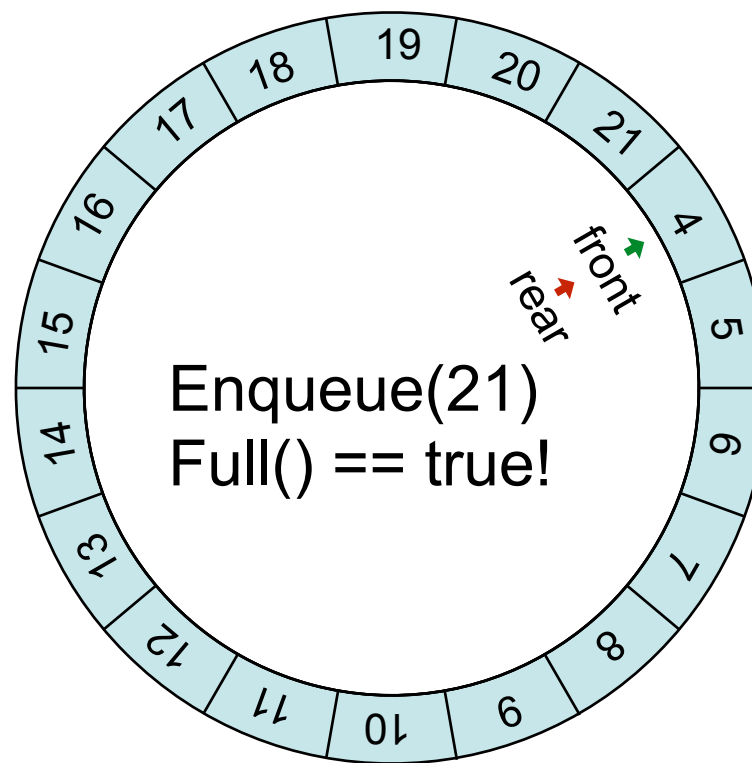
TypeStates - Queue

- An implementation using a circular buffer...



TypeStates - Queue

- An implementation using a circular buffer...



TypeStates - Queue

- An implementation using a circular buffer...

```
class {autocontracts} Queue {  
  // Representation type  
  var a:array<int>;  
  var front: int;  
  var rear: int;  
  var numberOfElements: int;  
  ...  
  
  constructor(N:int)  
    requires 0 < N  
  {  
    a := new int[N];  
    front, rear := 0 , 0;  
    Repr := {this};  
    numberOfElements := 0;  
  }  
  ...  
}
```


TypeStates - Queue

- What's wrong with it? a ReplInv is necessary to maintain front and rear within bounds...

```
class { :autocontracts } Queue {  
  
  ...  
  method Enqueue(V:int)  
  {  
    a[front] := V;  
    front := (front + 1)%a.Length;  
    numberOfElements := numberOfElements + 1;  
  }  
  
  method Dequeue() returns (V:int)  
  {  
    V := a[rear];  
    rear := (rear + 1)%a.Length;  
    numberOfElements := numberOfElements - 1;  
  }  
}
```


TypeStates - Queue

```
class {:autocontracts} Queue {  
  // Representation type  
  var a:array<int>;  
  var front: int;  
  var rear: int;  
  var numberOfElements: int;  
  
  // Representation invariant  
  predicate RepInv()  
    reads Repr  
  { 0 <= front < a.Length && 0 <= rear < a.Length }  
  
  constructor(N:int)  
    requires 0 < N  
    ensures RepInv()  
  {  
    a := new int[N];  
    front, rear := 0 , 0;  
    Repr := {this}  
    numberOfElements := 0;  
  }  
  ...  
}
```

TypeStates - Queue

```
class {:autocontracts} Queue {  
  ...  
  method Enqueue(V:int)  
    requires RepInv()  
    ensures RepInv()  
  {  
    a[front] := V;  
    front := (front + 1)%a.Length;  
    numberOfElements := numberOfElements + 1;  
  }  
  
  method Dequeue() returns (V:int)  
    requires RepInv()  
    ensures RepInv()  
  {  
    V := a[rear];  
    rear := (rear + 1)%a.Length;  
    numberOfElements := numberOfElements - 1;  
  }  
}
```

TypeStates - Queue

- Not enough... No runtime errors but the correct behaviour is not yet ensured...
wrong values may be returned,
valid elements maybe overwritten

```
method Main()  
{  
    var q:Queue := new Queue(4);  
    var r:int;  
  
    q.Enqueue(1);  
    r := q.Dequeue();  
    r := q.Dequeue();    ????  
    q.Enqueue(2);  
    q.Enqueue(3);  
    q.Enqueue(4);  
    q.Enqueue(4);    ???  
    q.Enqueue(4);  
    q.Enqueue(5);  
}
```

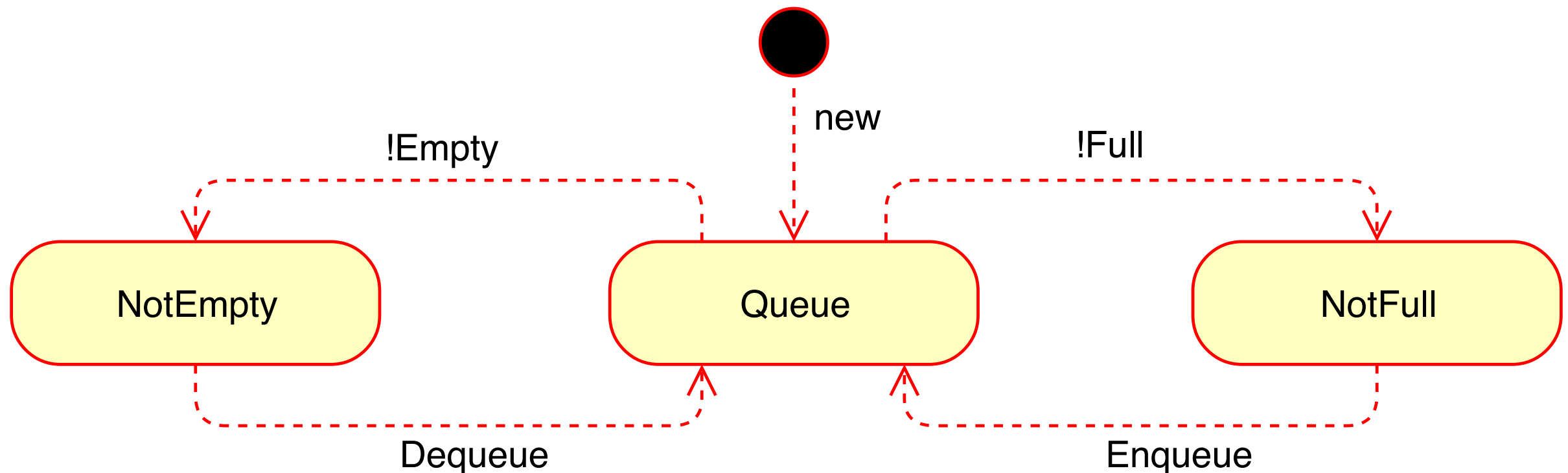
TypeStates - Queue

- RepInv must be refined to ensure that we stay inside the domain of valid queues...

```
predicate RepInv()  
  reads this, Repr  
{  
  0 <= front < a.Length &&  
  0 <= rear < a.Length &&  
  if front == rear then  
    numberOfElements == 0 ||  
    numberOfElements == a.Length  
  else  
    numberOfElements ==  
      if front > rear  
      then front - rear  
      else front - rear + a.Length  
}
```

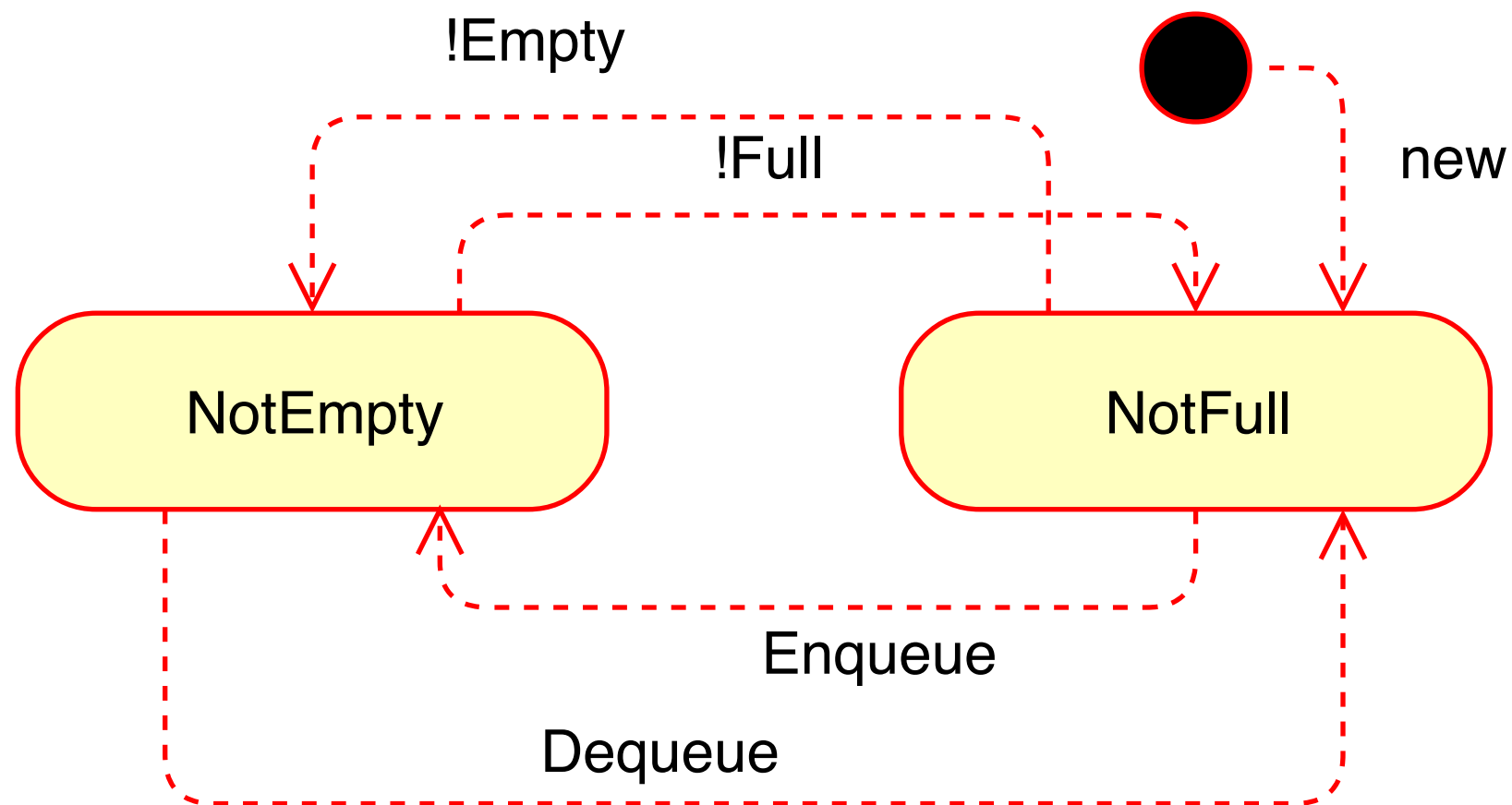
TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states... Obtained by dynamic testing operations.



TypeStates - Queue

- Enqueue and Dequeue Operations are only valid in certain states... Obtained by dynamic testing operations.



TypeStates - Queue

```
class {:autocontracts} Queue {  
  ...  
  predicate NotFull()  
    reads this  
  { RepInv() && numberOfElements < a.Length }  
  
  predicate NotEmpty()  
    reads this  
  { RepInv() && numberOfElements > 0 }  
  
  constructor(N:int)  
    requires 0 < N  
    ensures NotFull()  
  { ... }  
  
  method Enqueue(V:int)  
    requires NotFull()  
    ensures NotEmpty()  
  { ... }  
  
  method Dequeue() returns (V:int)  
    requires NotEmpty()  
    ensures NotFull()  
  { ... }
```

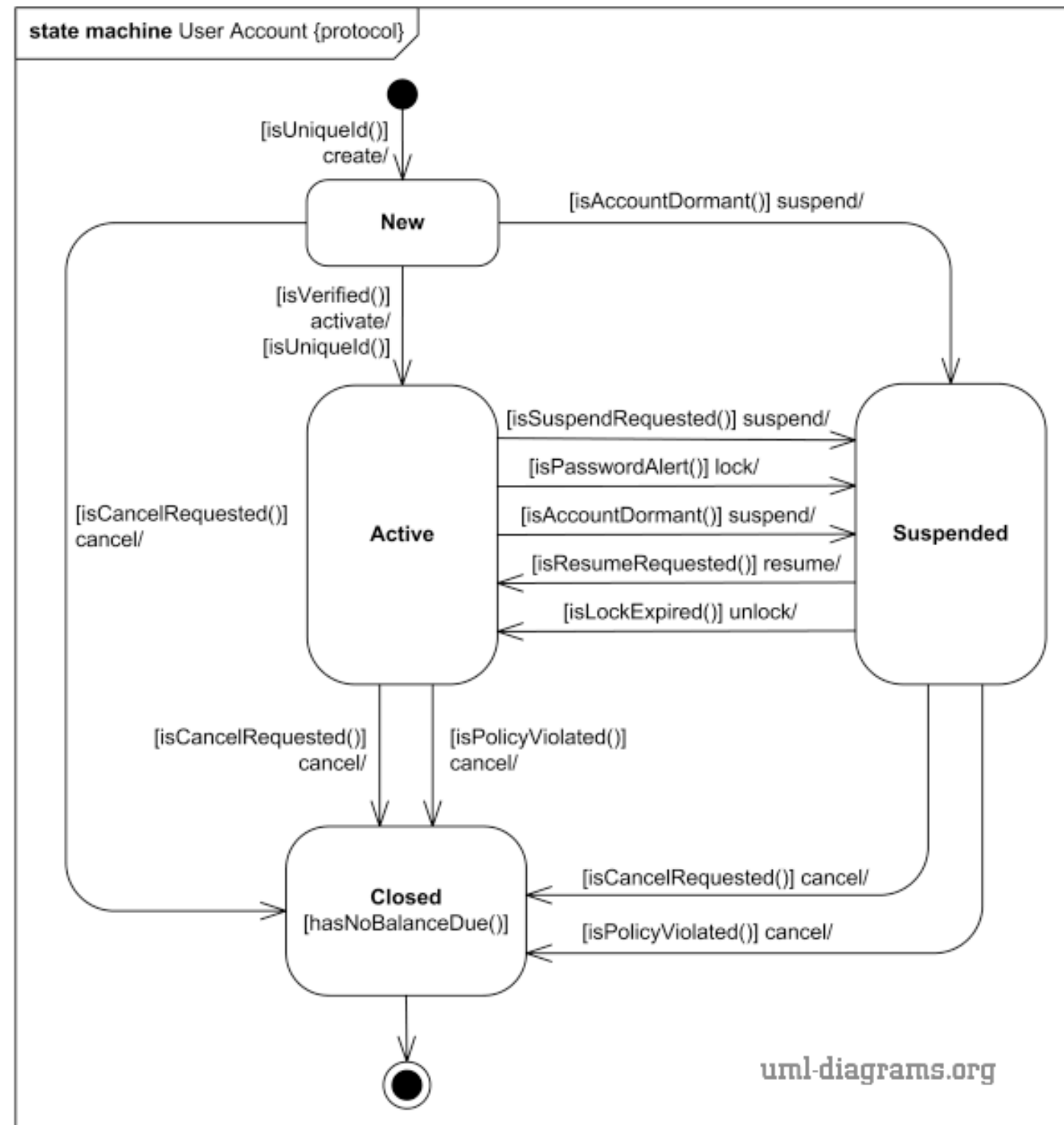
```
method Main()  
{  
  var q:Queue := new Queue(4);  
  var r:int;  
  
  q.Enqueue(1);  
  r := q.Dequeue();  
  r := q.Dequeue();  
  q.Enqueue(2);  
  q.Enqueue(3);  
  q.Enqueue(4);  
  q.Enqueue(5);  
}
```


TypeStates - Queue

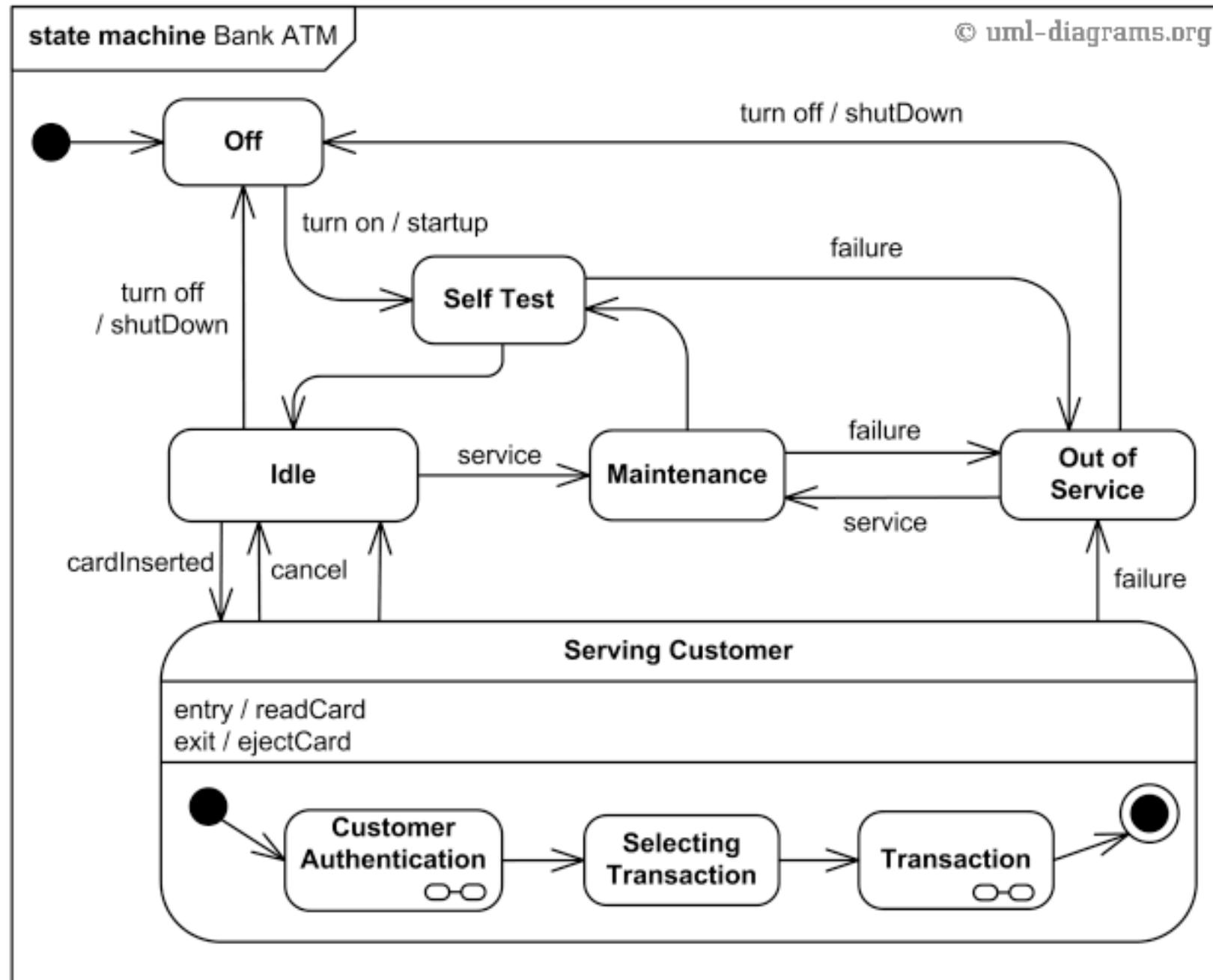
- Dynamic Tests ensure the proper state for a given operation...

```
method Main()  
{  
    var q:Queue := new Queue(4);  
    var r:int;  
  
    q.Enqueue(1);  
    r := q.Dequeue();  
    if !q.Empty()  
    { r := q.Dequeue(); q.Enqueue(2); }  
    if !q.Full() { q.Enqueue(3); }  
    if !q.Full() { q.Enqueue(4); r := q.Dequeue(); }  
    if !q.Full() { q.Enqueue(5); }  
}
```


TypeStates - UserAccount in a store



TypeStates - ATM



Part II

Interference and Aliasing

Account ADT in Java

- Consider the previous implementation of the ADT for a Bank Account, using Java

```
class Account {  
    int balance;  
  
    Account() { }  
  
    int getBalance() { return balance; }  
  
    void deposit(int amount) { balance += amount; }  
  
    void withdraw(int amount) { balance -= amount; }  
  
    void transferTo(Account target, int amount) {  
        withdraw(amount);  
        target.deposit(amount);  
    }  
}
```

Account ADT (Java)

```
class Account {
    int balance; // RepInv() = balance >= 0;

    Account() { } // ensures RepInv()

    int getBalance() { return balance; }

    void deposit(int amount) { balance += amount; } // requires v >= 0;

    void withdraw(int amount) { balance -= amount; } // requires v >= 0 && v <= amount;

    void transferTo(Account target, int amount) { // requires ??
        withdraw(amount);
        target.deposit(amount);
    }

    public static void main(String[] args) {
        Account a = new Account();
        a.deposit(1000);
        int ba = a.getBalance();
        assert ba == 1000; // Is it possible to prove this ??
    }
}
```

Account ADT (Java)

```
class Account {
    int balance; // RepInv() = balance >= 0;

    Account() { } // ensures RepInv()

    int getBalance() { return balance; }

    void deposit(int amount) { balance += amount; } // requires v >= 0;

    void withdraw(int amount) { balance -= amount; } // requires v >= 0 && v <= amount;

    void transferTo(Account target, int amount) { // requires ??
        withdraw(amount);
        target.deposit(amount);
    }

    public static void main(String[] args) {
        Account a = new Account();
        Account b = new Account();
        b.deposit(1000);
        int ba = a.getBalance();
        assert ba == 0; // Is it possible to prove this ??
    }
}
```

Account ADT (Java)

```
class Account {
    int balance; // RepInv() = balance >= 0;

    Account() { } // ensures RepInv()

    int getBalance() { return balance; }

    void deposit(int amount) { balance += amount; } // requires v >= 0;

    void withdraw(int amount) { balance -= amount; } // requires v >= 0 && v <= amount;

    void transferTo(Account target, int amount) { // requires ??
        withdraw(amount);
        target.deposit(amount);
    }

    public static void main(String[] args) {
        Account b = new Account();
        Account a = new Account();
        a.deposit(1000);
        a.transferTo(a, 500);
        int ba = a.getBalance();
        assert ba == 1000; // ?????
    }
}
```


Account ADT (Java)

- Consider the following code fragment and Hoare triple, intended to transfer v from $a1$ to $a2$

```
{  $v > 0$  &&  $a1.getBalance() \geq v$  }  
{  
    int v1;  
    v1 = a1.getBalance();  
    if (v1  $\geq v$ ) {  
        a1.withdraw(v);  
        a2.deposit(v);  
    }  
}  
{ $a1.getBalance() < old(a1.getBalance())$  }
```

- Is this Hoare triple valid?

Account ADT (Java)

- Consider the following code fragment and Hoare triple, intended to transfer v from $a1$ to $a2$

```
{  $v > 0$  &&  $a1.getBalance() \geq v$  }  
{  
    int v1;  
    v1 = a1.getBalance();  
    if (v1  $\geq v$ ) {  
        a1.withdraw(v);  
        a2.deposit(v);  
    }  
}  
{ $a1.getBalance() < old(a1.getBalance())$  }
```

- Only if **a1** and **a2** refer to different account objects!
- If they are aliases, the Hoare triple is invalid.

Account ADT (Java)

- Tracking aliasing is **challenging**, e.g.,

```
static void safeTransfer(Account a1, Account a2, int v)
{
    int v1;
    v1 = a1.getBalance();
    if (v1 >= v) {
        a1.withdraw(v);
        a2.deposit(v);
    }
}

static void main (String args[] )
{
    Account b1 = new Account();
    Account b2 = new Account();
    b1.deposit(10);
    safeTransfer(b1,b2,2);
    safeTransfer(b1,b1,2);
}
```

- Effect of test depends on whether a1 and a2 are aliases

Aliasing and Interference

- Two programming language expressions are **aliases** if they refer to the same memory location
- Aliasing occurs in any programming language with pointers (e.g., C, C++) or references (Java, C#)
- **Two** program fragments **interfere** if the execution of **one** may change the effect of the **other**
- **Interference** is particularly important in the context of concurrent programs (we will see more on this later)
- **Interference** and aliasing is hard to detect syntactically, and hard to reason about semantically.

Hoare Logic is unsound for aliasing

- Recall the basic Hoare Logic rule:

$$\{A[E/x]\} x := E \{A\}$$

- The soundness of this reasoning principle is rooted on the fact that no other variable aliases x . We have:

$$\{y \geq 0 \wedge x \geq 0\} x := -1 \{y \geq 0 \wedge x < 0\}$$

- But, if y and x are aliases, we would have, e.g.

$$\{y \geq 0 \wedge x \geq 0\} x := -1 \{y < 0 \wedge x < 0\}$$

Hoare Logic is unsound for aliasing

- In our Account ADT example:

$\{ x.balance() == K \ \&\& \ y.balance() > 0 \}$

$x.withdraw(K)$

$\{ x.balance() == 0 \ \&\& \ y.balance() > 0 \}$

- Again, if y and x are aliases, we would have, e.g.

$\{ x.balance() == K \ \&\& \ y.balance() > 0 \}$

$x.withdraw(K)$

$\{ x.balance() == 0 \ \&\& \ y.balance() == 0 \}$

- To reason about programs with interference (aliasing or concurrency) a different approach is needed.

Aliasing in Dafny (&others)

- Dafny and others are conservative with relation to aliasing by not assuming that objects from the same class are different.

```
method M(a:C, b:C)
  requires a != b // This precondition is essential
  modifies a
{
  a.f := 2+b.f;
  assert b.f == old(b.f);
}
```

- Framing conditions (reads, modifies) help to manage knowledge about objects.

Aliasing in Dafny (&others)

- Dafny and others are conservative with relation to aliasing by not assuming that objects from the same class are different.

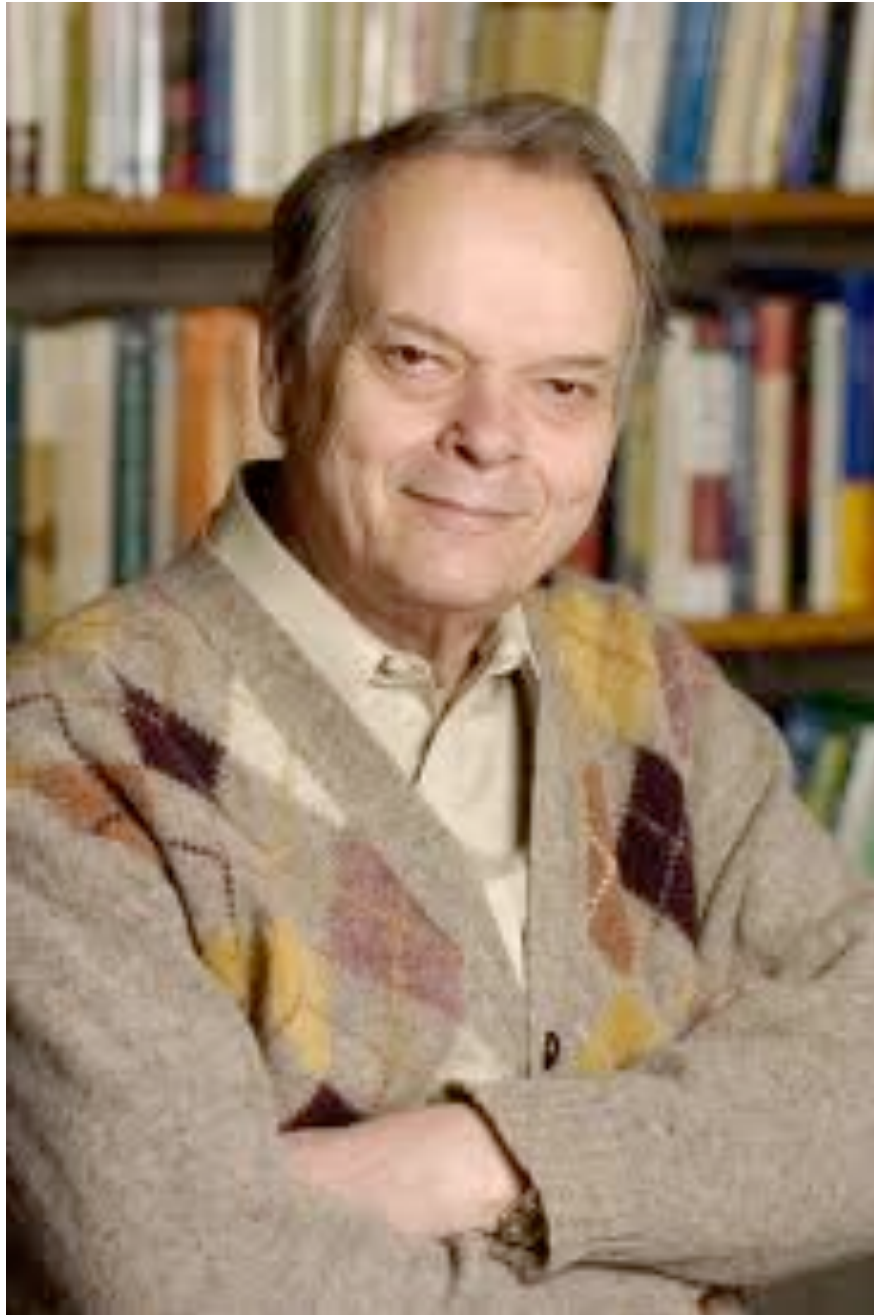
```
method N() {  
    var a := new C();  
    var b := new C();  
  
    M(a,b);  
    assert b.f == 0; // Cannot be proven...  
}
```

- Framing conditions (reads, modifies) help to manage knowledge about objects.
- Other approaches include ownership hierarchy (Spec#), Ownership and data groups (JML), and **separation logic** (up next) which uses small footprint reasoning.

Part III

Separation Logic

Separation Logic



John C. Reynolds

<https://www.cs.cmu.edu/~jcr/>



Peter O'Hearn

Syntactic control of interference

Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

SYNTACTIC CONTROL OF INTERFERENCE

John C. Reynolds
School of Computer and Information Science
Syracuse University

ABSTRACT In programming languages which permit both assignment and procedures, distinct identifiers can represent data structures which share storage or procedures with interfering side effects. In addition to being a direct source of programming errors, this phenomenon, which we call interference can impact type structure and parallelism. We show how to eliminate these difficulties by imposing syntactic restrictions, without prohibiting the kind of constructive interference which occurs with higher-order procedures or SIMULA classes. The basic idea is to prohibit interference between identifiers, but to permit interference among components of collections named by single identifiers.

The Problem

It has long been known that a variety of anomalies can arise when a programming language combines assignment with a sufficiently powerful procedure mechanism. The simplest and best-understood case is aliasing or sharing between variables, but there are also subtler phenomena of the kind known vaguely as "interfering side effects".

tantamount to calling n by value. But while this solution is adequate for simple variables, it can become impractical for arrays. For example, the procedure

```
procedure transpose(X, Y); real array X, Y;  
  for i := 1 until 50 do  
    for j := 1 until 50 do  
      Y(i, j) := X(j, i)
```

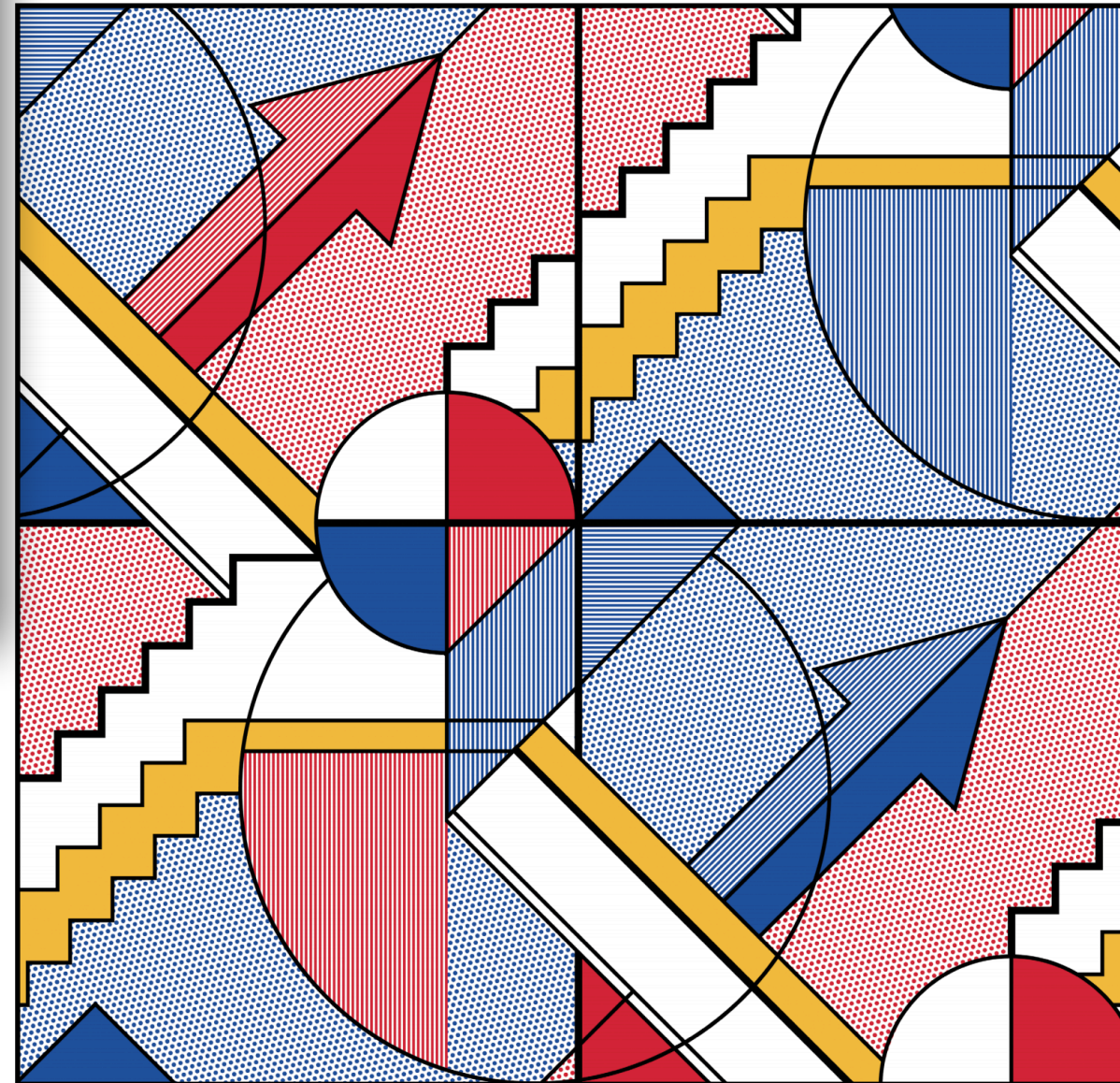

Separation Logic

DOI:10.1145/3211968

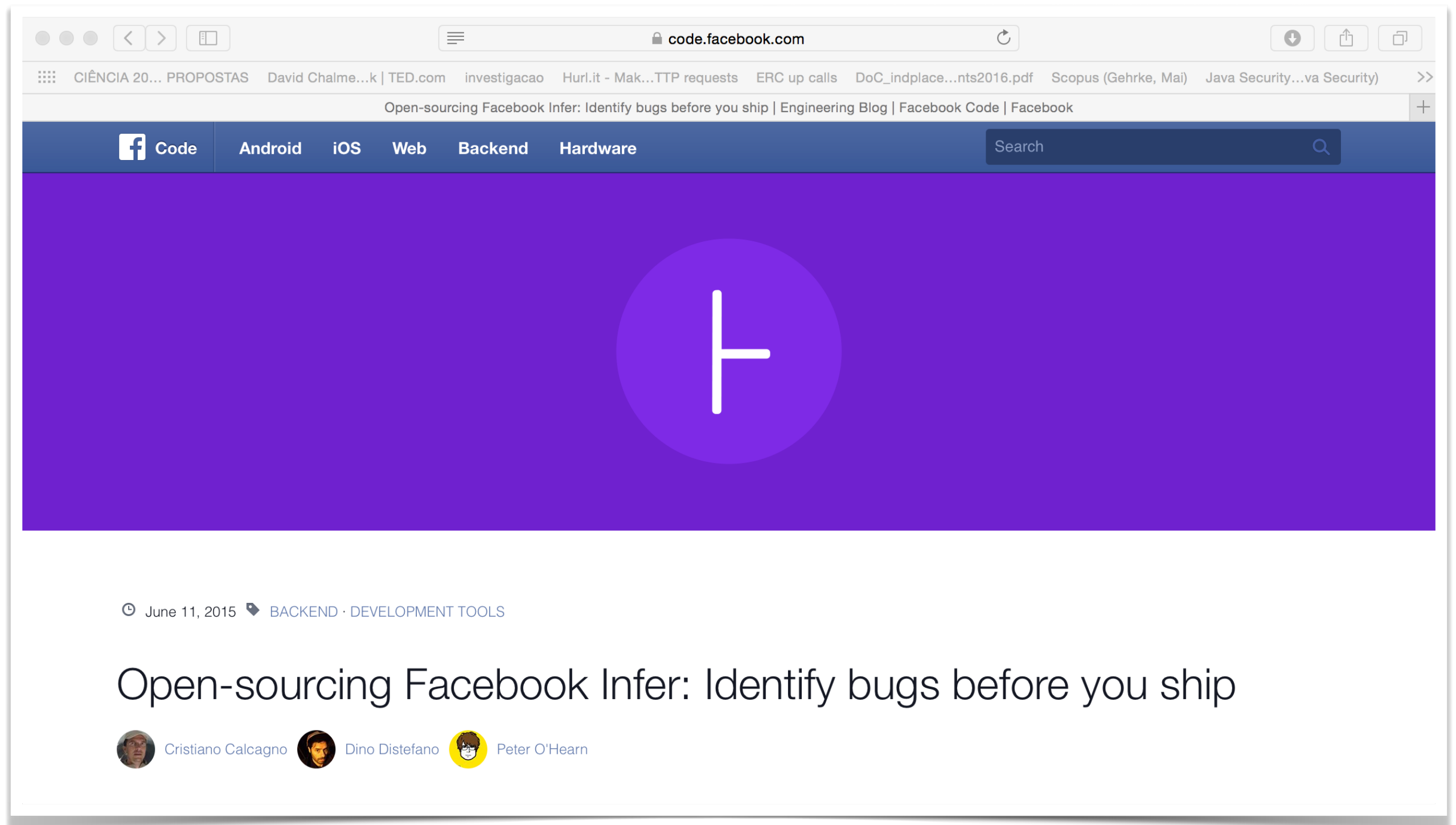
Separation logic is a key development in formal reasoning about programs, opening up new lines of attack on longstanding problems.

BY PETER O'HEARN

Separation Logic



Separation Logic @ facebook



Separation Logic

- An axiomatisation of language operations on pointers
- Allowing the verification of access of unrelated parts of the code to the shared memory locations
- Deals with aliasing by providing a precise shape of the dynamically allocated memory.
- Provides a basis for a general theory for modular reasoning about concurrent and sequential programs.

Separation Logic

- Separation logic is based in two key principles:

(1) **Small footprint**

The precondition of a code fragment describes the part of the memory (heap) that the fragment needs to use.

(2) **Implicit framing**

No need to explicitly specify the properties of state that is changed / not changed by the program (modifies)

- It adds to Hoare Logic two key novel primitives

- the separating conjunction operator
- the “precise” memory access assertion

$$A * B$$

$$L \mapsto V$$

Separation Logic (a fragment)

Separation logic assertions used in our CVS course are described by the following grammar:

A	$::=$	Separation Logic Assertions
	$L \mapsto V$	Memory Access
	$ A * A$	Separating Conjunction
	$ \text{emp}$	Empty heap
	$ B$	Boolean condition (pure)
	$ B?A : A$	Conditional

$B ::= B \wedge B \mid B \vee B \mid V = V \mid V \neq V \mid \dots$

$V ::= \dots$ Pure Expressions

$L ::= x.\ell$ Object field

Separation Logic

the **memory access** assertion

$$L \mapsto V$$

Assertion $L \mapsto V$ holds of the “piece” of the state consisting precisely of memory location L holding V

the **empty assertion**

$$\text{emp}$$

Assertion emp holds of the empty heap

the **separating conjunction**

$$A * B$$

Assertion $A * B$ holds of any “piece” of the state that can be **disjointly** decomposed in a “piece” that satisfies A and another piece that satisfies B .

NOTE: if B is “pure” then $A * B \Leftrightarrow A \wedge B$

Digression: The Stack and the Heap

Stack

Stores local variables and method parameters

The so-called call-stack, also stores return addresses

Memory recover discipline: LIFO, release on block exit

Heap

Stores dynamically allocated objects (e.g. new / malloc)

Recover discipline: explicit release or garbage collection

Heap Model

A sequence of mem locations (L) with their contents (V)

Memory contents are value of basic type (e.g. ints) or references (pointers) to memory locations

Separation Logic

Pure assertion (remember this from Dafny)

An assertion that does not depend on the state (e.g., a boolean expression not referring to memory locations).

Precise assertion

An assertion that uniquely specifies some part of the memory heap (that has a **unique footprint**)

Examples:

- No pure assertion is precise
- $L \mapsto V$ is precise, for a unique value V
- $L \mapsto 3 * \text{true}$ is not precise — how many (sub)heaps satisfy this condition?

<https://www.cs.cmu.edu/~jcr/mfps2005.ps>

Examples (SL / HL)

$$\{x \mapsto 2\} [x] := 4 \{x \mapsto 4\}$$

holds in SL

$$\{\} [x] := 4 \{x \mapsto 4\}$$

does not hold in SL

$$\{\} x := 4 \{x = 4\}$$

holds in HL

$$\{x \mapsto 2 * y \mapsto 3\} [x] := y \{x \mapsto 3 * y \mapsto 3\}$$

holds in SL

$$\{x \mapsto V * x \mapsto U\}$$

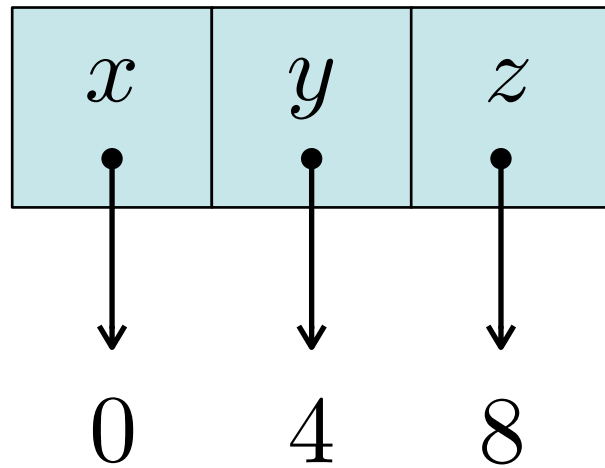
never holds in SL

$$\{x = V \wedge x = U\}$$

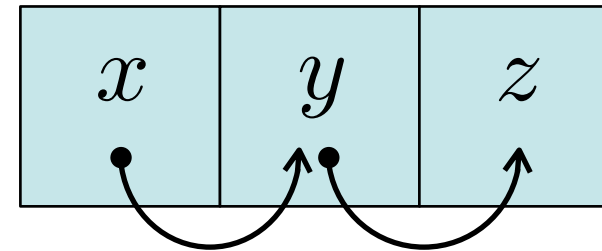
may hold in HL

- Note that in this case $[x] := v$ denotes an assignment of a value to the memory location denoted by x .

Separation Logic



$$\{x \mapsto 0 * y \mapsto 4 * z \mapsto 8\}$$



$$\{x \mapsto y * y \mapsto z * z \mapsto \dots\}$$

Rules of SL — Assignment Rule

- Recall the basic Hoare Logic Rule:

$$\{A[E/x]\} x := E \{A\}$$

- The assignment rule in separation logic is

$$\{x \mapsto V\} x := E \{x \mapsto E\}$$

- Note the small footprint principle, the precondition refers exactly to the part of the memory used by the fragment

Frame Rule (SL)

- A key principle in SL is the **Frame Rule**

$$\frac{\{A\} P \{B\}}{\{A * C\} P \{B * C\}}$$

- This frame rule allows us to preserve info about the “rest of the world”, and **locally reason** about the effects of a program that only manipulates a given piece of the state
- The given piece footprint is specified by precondition A
- There is no need to specify **what is modified** (it is clear from the pre-condition A) neither **what is not modified** (it is framed away, as C above).

Lookup Rule (SL)

- The memory lookup rule in SL is

$$\{L \mapsto V\} \ y := L \ \{L \mapsto V \wedge y = V\}$$

- Here y is a stack variable, not a heap location L

$$\{L \mapsto V\} \ L := L + 1 \ \{L \mapsto V + 1\}$$

$$\{L \mapsto V\} \ y := L$$

$$\{L \mapsto V \wedge y = V\} \ L := y + 1$$

$$\{L \mapsto y + 1 \wedge y = V\}$$

$$\{L \mapsto V + 1\}$$

Part IV

Separation Logic in Verifast

Verifast

Verifast

VeriFast is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic.

Jacobs, Smans, Piessens, 2010-14

```
public void broadcast_message(String message) throws IOException
{
    //@ requires room(this) && message != null;
    //@ ensures room(this);

    //@ open room(this);
    //@ assert foreach(?members0, _);
    List membersList = this.members;
    Iterator iter = membersList.iterator();
    boolean hasNext = iter.hasNext();
    //@ length_nonnegative(members0);
    while (hasNext)
    {
        /*@
        invariant
            foreach<Member>(?members, @member) && iter(iter, membersList, members, ?i)
            && hasNext == (i < length(members)) && 0 <= i && i <= length(members);
        @*/
        {
            Object o = iter.next();
            Member member = (Member)o;
            //@ mem_nth(i, members);
            //@ foreach_remove<Member>(member, members);
            //@ open member(member);
            Writer writer = member.writer;
            writer.write(message);
            writer.write("\r\n");
            writer.flush();
            //@ close member(member);
            //@ foreach_unremove<Member>(member, members);
            hasNext = iter.hasNext();
        }
    }
    //@ iter_dispose(iter);
    //@ close room(this);
}
```



Account ADT (Java + Verifast)

```
public class Account {
```

```
    int balance;
```

```
    /*@  
    predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;  
    @*/
```

```
    public Account()
```

```
    //@ requires true;
```

```
    //@ ensures AccountInv(0);
```

```
{
```

```
    balance = 0;
```

```
}
```

```
...
```

```
}
```

Account ADT (Java + Verifast)

```
public class Account {  
  
    int balance;  
  
    ...  
    void deposit(int v)  
        //@ requires AccountInv(?b) &*& v >= 0;  
        //@ ensures AccountInv(b+v);  
    {  
        balance += v;  
    }  
  
    void withdraw(int v)  
        //@ requires AccountInv(?b) &*& b >= v;  
        //@ ensures AccountInv(b-v);  
    {  
        balance -= v;  
    }  
    ...  
}
```

Account ADT (Java + Verifast)

```
public class Account {  
    int balance;  
  
    ...  
    void deposit(int v)  
        //@ requires AccountInv(?b) &*& v>=0;  
        //@ ensures AccountInv(b+v);  
    {  
        //@ open AccountInv()  
        balance += v;  
        //@ close AccountInv()  
    }  
}
```

- Verifast sometimes requires the programmer to explicitly **open** and **close** predicates, if assertions are not precise
- Not needed here, since `AccountInv(_)` is precise

Account ADT (Java + Verifast)

```
public class Account {  
  
    int balance;  
  
    ...  
    int getBalance()  
    //@ requires AccountInv(?b);  
    //@ ensures AccountInv(b) &* result==b ;  
    {  
        return balance;  
    }  
  
    ...  
}
```

Part V

An Example

Stack ADT (using a linked list)

- Separation logic allows us to precisely define the shape and memory layout (footprint) of arbitrary heap allocated data structures.
- The framing principles allow us to easily express the functionality of heap manipulating operations using separation logic pre and post conditions
- Let's look to a simple example, a stack ADT implemented using a linked list.
- Again, we will use Verifast to annotate Java code.

Stack ADT (using a linked list)

```
/*@
  predicate Node(Node n; Node nxt, int v) = n.next |-> nxt &*& n.val |-> v;
  predicate List(Node n;) = n == null ? emp : Node(n,?h,_) &*& List(h);
@*/

public class Node {
  Node next;
  int val;

  public Node()
  //@ requires true;
  //@ ensures Node(this,null,0);
  {
    next = null;
    val = 0;
  }
}
...
}
```


Stack ADT (using a linked list)

- The predicate `Node(node n; node nxt, int v)` represents a single node object in the heap.

N.B. In Verifast, the “;” separates the input parameters, from i/o parameters, which may be queried using “?”.

- The predicate `List(node n;)` represents the shape and layout of properly formed linked list in the heap.

N.B. `List(node n;)` is recursively defined. A list is either empty (no memory) or contains an allocated head node linked to a possibly empty list.

- N.B. Note that our definition forbids cycles in the list.

Stack ADT (using a linked list)

```
/*@
  predicate Node(Node n;Node nxt,int v) = n.next |-> nxt &*& n.val |-> v;
  predicate List(Node n;) = n == null ? emp : Node(n,?h,_) &*& List(h);
@*/
public class Node {
  Node next;
  int val;

  public void setnext(node n)
  //@ requires Node(this,_,?v);
  //@ ensures Node(this,n,v);
  {
    next = n;
  }
  public void setval(int v)
  //@ requires Node(this,?n, _);
  //@ ensures Node(this,n, v);
  {
    val = v;
  }
  ...
}
```

Stack ADT (using a linked list)

```
/*@
    predicate Node(Node n;Node nxt,int v) = n.next |-> nxt &*& n.val |-> v;
    predicate List(Node n;) = n == null ? emp : Node(n,?h,_) &*& List(h);
@*/
public class Node {
    Node next;
    int val;

    public Node getNext()
    //@ requires Node(this,?n,?v);
    //@ ensures Node(this,n,v) &*& result == n;
    {
        return next;
    }
    public int getval()
    //@ requires Node(this,?n,?v);
    //@ ensures Node(this,n,v) &*& result == v;;
    {
        return val;
    }
}
```

Stack ADT (using a linked list)

```
/*@
    predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
@*/

public class Stack {

    Node head;

    public Stack()
    //@ requires true;
    //@ ensures StackInv(this);
    {
        head = null;
    }

    ...

}
```

Stack ADT (using a linked list)

```
/*@
    predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
@*/

public class Stack {

    Node head;

    public void push(int v)
    //@ requires StackInv(this);
    //@ ensures StackInv(this);
    {
        node n = new Node();
        n.setval(v);
        n.setnext(head);
        head = n;
    }

    ...

}
```

Stack ADT (using a linked list)

```
/*@
  predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
@*/

public class Stack {

    Node head;

    public void push_buggy(int v)
    //@ requires StackInv(this);
    //@ ensures StackInv(this);
    {
        node n = new Node();
        n.setval(v);
        n.setnext(n);
        head = n;
    }

    ...

}
```

This code will not check in Verifast. **Why?**

Stack ADT (using a linked list)

```
/*@
    predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);

@*/

public class Stack {

    Node head;

    public int pop()
    //@ requires StackInv(this);
    //@ ensures StackInv(this);
    {
        if(head != null) {
            int v = head.getval();
            head = head.getnext();
            return v;
        }
        return -1; // ??? how to deal with partiality ?
    }
}
```


Stack ADT (using a linked list)

```
/*@
    predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
@*/

class StackEmptyE extends Exception {}

public class Stack {
    Node head;

    public int pop_maybe()
        throws StackEmptyE //@ ensures StackInv(this);
        //@ requires StackInv(this);
        //@ ensures StackInv(this);
    {
        if(head!=null) {
            int v = head.getval();
            head = head.getnext();
            return v;
        } else throw new StackEmptyE();
    }
}
```


Exceptions in method contracts

- Java method contracts need to deal with exceptional behaviour, so Verifast allows postconditions to be specified at every exception thrown
- In the previous example, the `pop_maybe` method is partial and throws `StackEmptyE` if the stack is empty.
- Instead of making a method partial, we may try to define an appropriate precondition.
- To make `pop` total we may add a special pre-condition expressing the typestate “non-empty-stack”.
- This will require client code to be able to dynamically check the ADT state (e.g. via an `isEmpty` method).

Stack ADT (using a linked list)

```
/*@
  predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
  predicate NonEmptyStack(Stack s;) = s.head |-> ?h &* & h != null &* & List(h);
@*/

public class Stack {

    Node head;

    public int pop()
    //@ requires NonEmptyStack(this);
    //@ ensures StackInv(this);
    {
        int v = head.getval();
        head = head.getnext();
        return v;
    }

    ...
}
```

Stack ADT (using a linked list)

```
/*@
  predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
  predicate NonEmptyStack(Stack s;) = s.head |-> ?h &* & h != null &* & List(h);
@*/

public class Stack {
    Node head;

    public boolean isEmpty()
    //@ requires StackInv(this);
    //@ ensures (result?StackInv(this):NonEmptyStack(this));
    {
        return head == null;
    }
    ...
}
```

Stack ADT (using a linked list)

```
/*@
  predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
  predicate NonEmptyStack(Stack s;) = s.head |-> ?h &* & h != null &* & List(h);
@*/

public class Stack {

    Node head;

    public void push(int v)
    //@ requires StackInv(this);
    //@ ensures NonEmptyStack(this);
    {
        node n = new node();
        n.setval(v);
        n.setnext(head);
        head = n;
    }
    ...
}
```

Abstract States and Substates

- In general, it is very useful to introduce abstract substates representing particular cases of the general representation invariant of an ADT.
- All abstract substates logically imply the most general representation invariant, for instance, in our examples, for every Stack s , the following implication is valid:

$$\text{NonEmptyStack}(s) \Rightarrow \text{StackInv}(s)$$

- Often these implications are not automatically derived by Verifast, but the programmer may give an hint, using **open** and **close** annotations (next slide).

Stack ADT (using a linked list)

```
/*@
  predicate StackInv(Stack s;) = s.head |-> ?h &* & List(h);
  predicate NonEmptyStack(Stack s;) = s.head |-> ?h &* & h != null &* & List(h);
@*/

static void main()
//@ requires true;
//@ ensures true;
{
    stack s = new Stack();
    s.push(1);
    if (! s.isEmpty()) {
        s.pop();
    }
    s.push(2);
    //@ open NonEmptyStack(s);
    s.push(3);
    s.pop();
}
```