Modular Verification in Object-oriented Programming

Arnd Poetzsch-Heffter

University of Kaiserslautern

DEN.
Baltic Sea
North Sea
Neuendorfer bei Wilster
Kiel Canal
Kiel
Rostock
Lübeck
Hamburg
Bremerhaven
Emden
Bremen
BERLIN
POL.
Hannover
NETH.
Magdeburg
Duisburg
Essen
Elbe
Leipzig
Düsseldorf
Kassel
Dresden
Cologne
Bonn
BEL.
Frankfurt am Main
Wiesbaden
Rhine
CZECH REPUBLIC
Mannheim
LUX.
Nürnberg
Karlsruhe
Kaiserslautern
Stuttgart
Danube
FRANCE
Munich
AUSTRIA
Zugspitze
0 50 100 km
0 50 100 mi
SWITZ.
LIECH.

Overview:

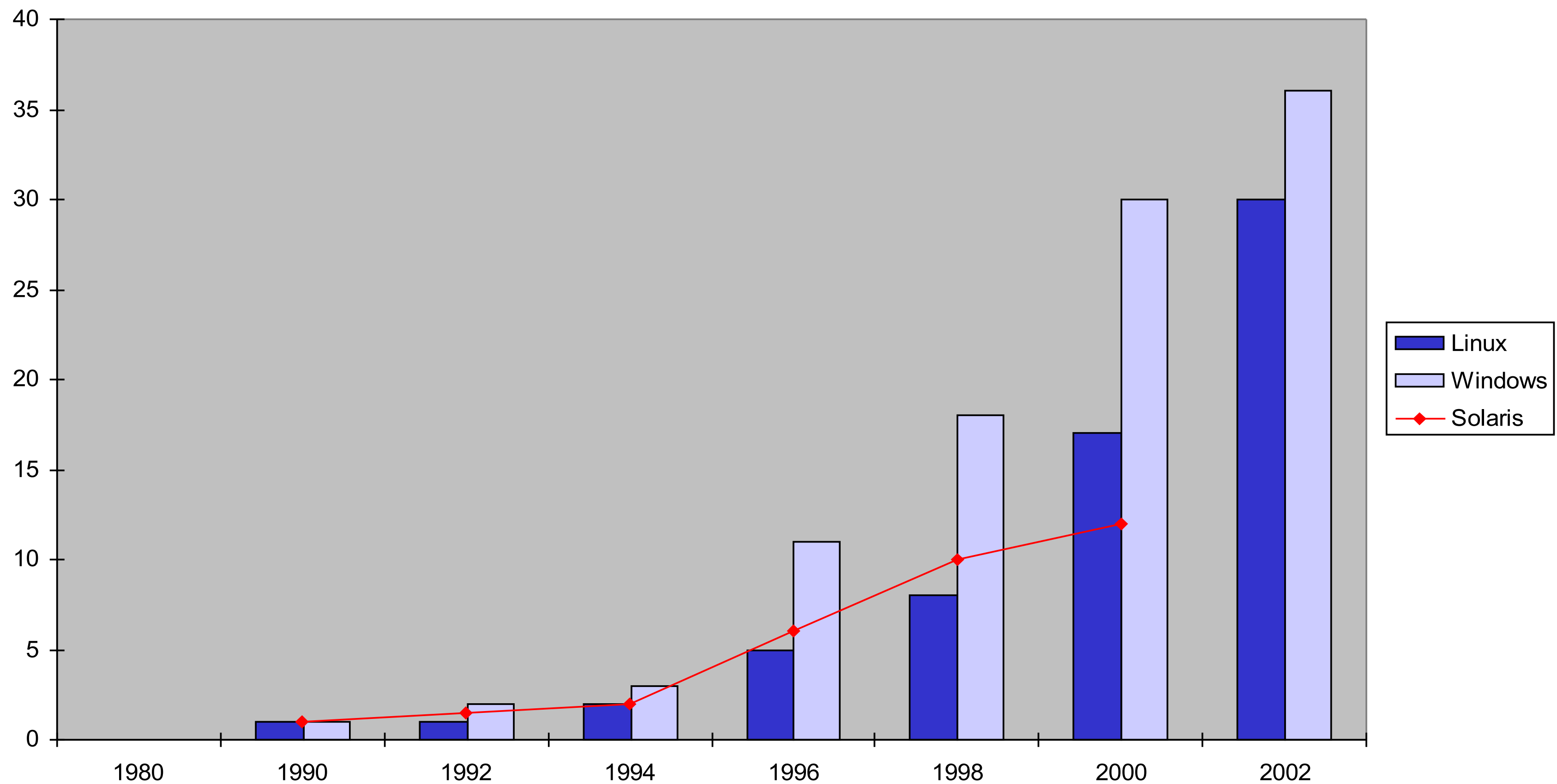
- Introduction
- Modular Verification of Classes with Spec#
- Modular Behavioural Specification (& Verification)
- Concluding Remarks

Slides partially adopted from

Mike Barnett, Manuel Fähndrich, Rustan Leino,
Peter Müller, and Wolfram Schulte

MANY THANKS TO THEM

1. Introduction



Goals of static and dynamic verification:

1. Detection of programming errors:
   - No IndexOutOfBounds-, NullPointer-, Cast-, DivisionByZero-exception
   - *Language-based* properties: no need for specification

2. Guaranteed/checked *program-specific* properties:
   - Specification of properties: need for spec. language
   - Support of behavioral abstraction

```
class Simple {
  int a, b;

  public Simple(int ap, int bp )
  {  a = ap;  b = bp; }

  public int Foo( int x )
  {
     int tmp =  x / (b-a);
     a += tmp;  b+= tmp;
     return b-a;
  }
}
```

How can we prove it modularily?

No division by zero!

```
class Simple {
  int a, b;
  invariant a < b;
  public Simple(int ap, int bp )
     requires ap < bp;
     ensures a==ap && b==bp;
  {  a = ap;  b = bp; }

  public int Foo( int x )
     ensures result > 0;
 {
     ...  // see above
  }}
```

program-specific specification

When is a specification "sufficient" for modular verification?

```
interface Simple {
  model int c;
  public Simple(int ap, int bp )
    requires  ap < bp;
    ensures c == bp-ap ;

 public int Foo( int x )
    ensures result == c;
}
```

behavioral spec
at boundary

precisely

What is the *boundary*
of an object or
of class/module?

Goals of modularity:

2.Verification of modules/libraries without knowing the application context:

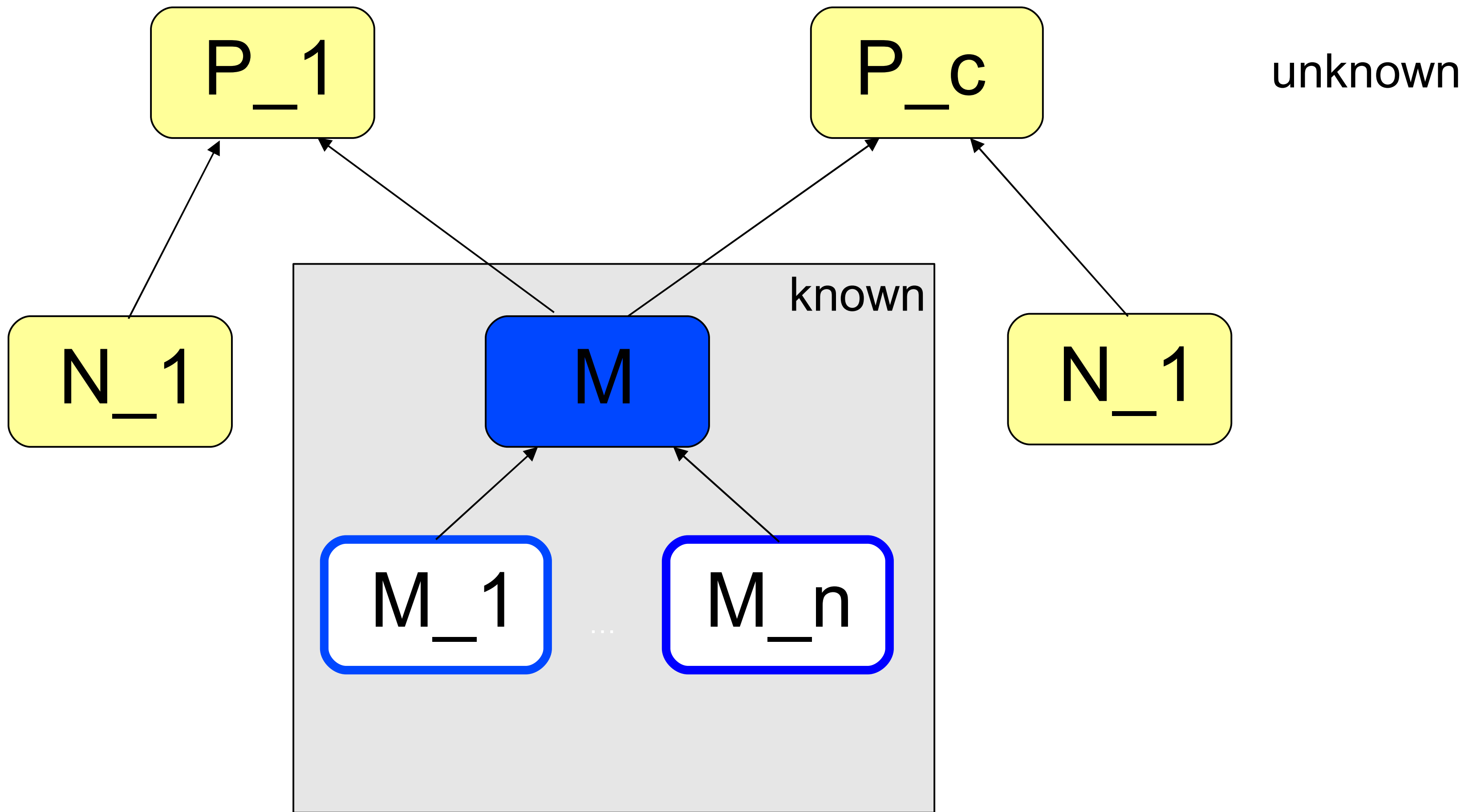
- Which application contexts are admissible?

2. Module contracts:

- Providers can modify implementation
- Users need not to know implementation

3. Scalability through compositionality:

- To verify M use only the contracts of modules used by M



... at verification time

Modular verification technique:

a) Technique(s) to verify program modules

•Proof of *modular soundness*:

All proofs done on program modules remain valid in all admissible contexts.

Modularity depends on sophisticated interplay of:

- Programming language semantics
- Specification language and technique
- Programming logic

Modular Verification Technique: A Gedankenexperiment

- Programming language:
  - Class-based language w/o inheritance and subtyping
  - No reference types for fields
  - No recursive methods, sequential statements/expressions

- Specification language and technique:
  - Pre-, postconditions, object invariants
  - Assertions: boolean expressions over fields and params

- Programming logic:
  - Wp-calculus for method bodies
  - Preconditions & invariants may be assumed in prestates
  - Postconditions & invariants have to be proven for poststates
  - Pre-/postconditions may be used to verify calls

```
class Simple {
  int a, b;
  invariant a < b;
  public Simple(int ap, int bp)
      requires ap < bp;
  {  a = ap;  b = bp; }

  public int Foo( int x )  {
    int tmp =  x / (b-a);
    a += tmp;  b+= tmp;
    return b-a;
  }
  ...
```

```
...
  public void Woo( C cp ) {
     a = 0;
     cp.Doo( this );
     b = a + 1;
  }
}
```

```
class C {
   invariant ... ;
  public void Doo(Simple s)
  {  ...  }
}
```

At least four problems:

1. Fields are not encapsulated

```
Simple s = new Simple(0,1);
s.a =1;
```

2. Callbacks are possible on objects with violated invariants

```
public void Doo(Simple s)
{  s.Foo(1);  }
```

(problems continued:)

3. How do we prove that invariants hold at call sites

```
public void Woo( C cp ) {
   a = 0;
   cp.Doo( this );
   ...
}
```

4. No framing: Modifications are not specified

   No knowledge about effect of Doo to its parameter

   ➔ Cannot establish invariant of Simple

Questions/Problems/Challenges:

1. Unit of modularity / boundary?
2. Does boundary encapsulate static or dynamic entities?
3. Callbacks, assumptions about invariants?
4. Hiding and framing?
5. How to handle object structures?
6. Subtyping and message dispatch
7. Inheritance and extended state

Design space for verification frameworks:

- static vs. dynamic
- modular vs. non-modular
- relation of programming and specification language
- properties/programs of interest
- automatic vs. interactive verification

2. Modular Verification of Classes with Spec#/Boogie

Main design decisions/focus for Spec#/Boogie:

- modular static verification on class level
- support for dynamic checking
- tight integration of programming and specification
- goals of the approach:
  elimination of programming errors
  implementation-related properties
- automatic verification based on user annotations

Section overview:

2. Introductory Remarks on Spec#/Boogie
3. Modular verification of objects
4. Multi-object invariants and ownership

- Subtyping, inheritance, and extended state
- Remarks on further aspects

Spec#/Boogie has been developed at Microsoft Research (Redmond) under the lead of K. Rustan M. Leino.

2.1 Introductory Remarks on Spec#/Boogie

The Spec# Programming System:

Spec# programming language extends C# with:
- non-null types,
- checked exceptions and throws clauses,
- method contracts and object invariants.

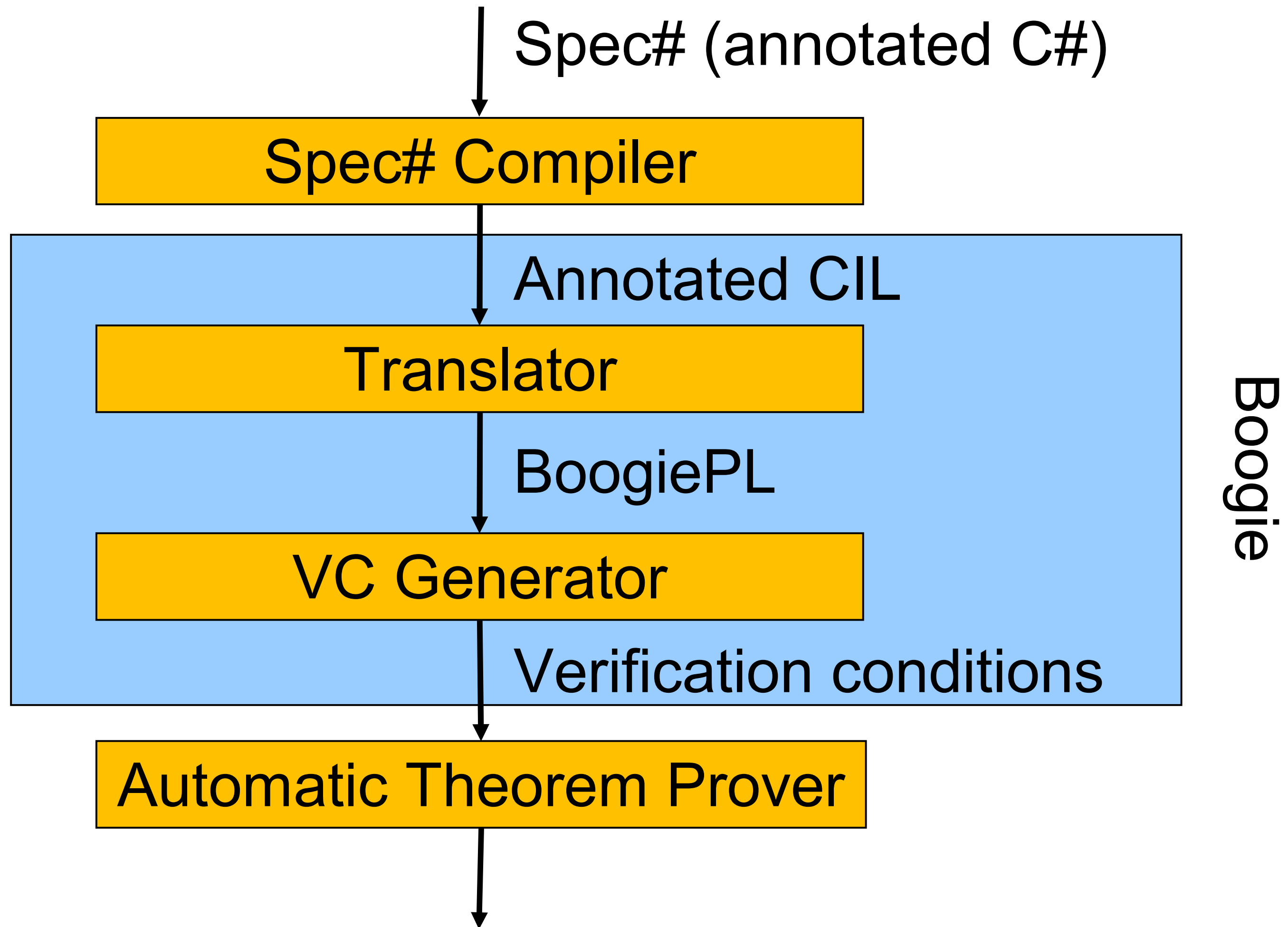
Spec# compiler:
- statically enforces non-null types
- emits run-time checks for method contracts and invariants
- records the contracts as metadata for downstream tools

Spec# static program verifier Boogie:
- generates logical verification conditions
- uses automatic theorem prover

Classic verification example: Insertion sort

```
class ArraySort { // Insertion Sort Method by R. Monahan & R. Leino / APH
  public static void sortArray( int[]! a )
    modifies a[*];
    ensures forall{int j in (1:a.Length);(a[j-1] <= a[j])};
  {
    int t, k=1;
    if (a.Length > 0) {
        while(k < a.Length)
          invariant 1 <= k && k <= a.Length;
          invariant forall { int j in (1:k), int i in (0:j); (a[i] <= a[j]) };
       {
          for( t = k; t>0 && a[t-1]>a[t]; t-- )
             invariant k < a.Length;
             invariant 0<=t && t<=k;
             invariant forall { int j in (1:k+1), int i in (0:j); j==t || a[i] <= a[j] };
          { int temp; temp = a[t]; a[t] = a[t-1]; a[t-1] = temp; }
          k++;
    } } }
```

Spec# Tool Architecture:



Goals of the Spec# Project:

- Experiment with programming logic,
  i.e., the generation of verification conditions
- Experiment with programming methodology,
  i.e., which constructs allow for simpler reasoning
- Build a componentized, state-of-the-art verifier
- Apply it to real code bases

2.2 Modular Verification of Objects

Modular verification needs a notion of *consistency* for

- objects
- object structures

➔ Formulate consistency by *object invariants*

Reasons:

- Hiding: Consistency might depend on private information
- Modularity: Assumptions on objects/classes out of scope are needed for verification

Hiding: Consistency might depend on private information

```
class C {
    private int a, z;
    public void M( )
        requires a ≠ 0;
    { z := 100 / a; }
}
```

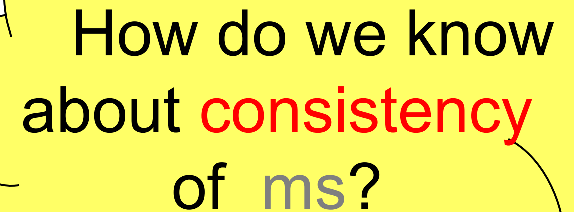
```
class C {
    private int a, z;
    invariant a ≠ 0;
    public void M( )
    { z := 100 / a; }
}
```

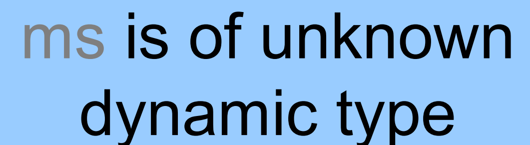
Field a not allowed in public requires clause

Modularity: Assumptions on objects/classes out of scope are needed for verification

```
public class Broker
{
   public void Call( IService! ms ) {
      ms.Do();
   }
}

public class IService
{
  virtual public int Do()
   modifies this.0;
  { return 0; }
}
```





```
public class Service : IService
{
    string s;
    invariant  s != null;

    public Service( string! s )
    { this.s = s; }

    override public int Do()
    {  return s.Length;   }
}
```

Depends on invariant

… now that we have invariants:

- What is their meaning/when should they hold?
- What are they allowed to depend on?

Spec# approach to consistency and invariants:

- Invariants define what should hold in consistent states
- Programmer defines when objects should be consistent
- A *consistency discipline* :
  - yields assumptions about objects out of scope
  - avoids callback problems

… in a setting w/o owners and w/o inheritance:

Object o is consistent / valid

- when the constructor has finished
- when o is not exposed / mutable.

Central modularity invariant:

$$(\forall o \bullet \; o.\mathit{IsExposed} \vee \mathit{Inv}(o))$$

Object is peer consistent

- if it and all its peers are consistent.

Exposing objects:

```
class Counter{
    int c;
    bool even;
    invariant  0 <= c;
    invariant  even <==> c % 2 == 0;

    public Counter()
    {       c= 0;
            even = true;
    }

    public void Inc ()
      modifies c,even;
      ensures c == old(c)+1;
    {     expose (this) {
              c++;
              even = !even ;
          }
}   }
```

The invariant may be broken in the constructor

The invariant must be established & checked after construction

The object invariant may be broken within an expose block

Consistency provide assumptions on unknown objects:

Per default method and results are peer consistent.

```
public class Broker
{
   [NoDefaultContract]
   public void Call( IService! ms )
      requires  this.IsPeerConsistent
             &&  ms.IsPeerConsistent;
      ensures  this.IsPeerConsistent;
  {
     ms.Do();
   }
}
```

```
public class IService
{
  [NoDefaultContract]
  virtual public int Do()
    requires  this.IsPeerConsistent;
    modifies  this.0;
    {
      return 0;
    }
}
```

ms is peer consistent, thus its invariant holds

Callbacks and exposed objects:

```
class  Broker
{
   Boolean   locked;
   invariant  ! locked;

  public void CallService( Service! ms )
     modifies locked;
  {
     expose( this ) {
        locked = true;
        ms.Do();
        locked = false;
      }
  }
  public void Foo()
     modifies this.0;
  {
   if( locked ) {
        int[] a = new int[10];   a[20] = a[21];
} } }
```

this exposed ➔

this is not consistent

ms is not peer consistent

Even if only consistency

is required for targets ➔

No callbacks into

exposed objects

Is this enough? - A case study:

```
class class Purse {
    [SpecPublic]  int  amount;
    invariant  0 <= amount;

    public Purse( int amt )
        requires  amt >= 0;
        ensures  amount == amt;
    {  amount = amt; }

    public  int  contains()
       modifies  this.0;
       ensures  result == amount;
    {  return  amount; }
    …
```

```
…

    public  int  take( int amt )
        requires  amt  > 0;
        modifies  amount;
        ensures
          amount >= old(max{0,amount-amt});
     {
        int  rtnamt =
               (amount >= amt ? amt : amount);
        expose( this ) {
            amount  -=  rtnamt;
        }
        return  rtnamt;
     }
}
```

… let‘s use the purse!

```
class Person {
   Purse! purse;
   invariant  purse.amount >= 100;

   public Person() {
   {  purse = new Purse(100); base(); }

   public bool pay( Person! p, int amt )
      requires  amt > 0;
      modifies  p.*,  purse;
   {
     int payedamt = 0;

     if( purse.contains() >= amt+100 ) {
          payedamt = purse.take(amt);
     }
     p.recieve( payedamt );
     return payedamt!=0;
   }

   void recieve( int amt ){ /* ... */ }
}
```

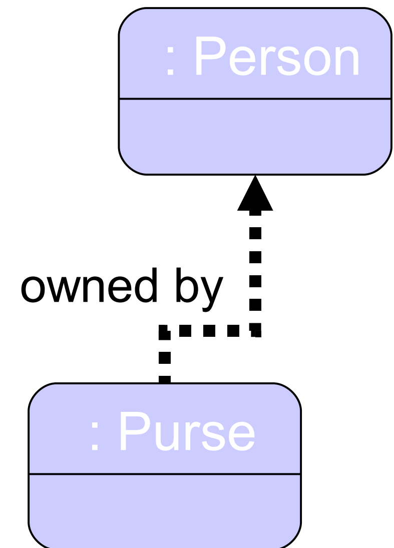
Multi-object invariant

Control of invariant in the presense of aliasing?

Who checks multi-object invariants?

Possible invariant violation

2.3 Multi-Object Invariants and Ownership



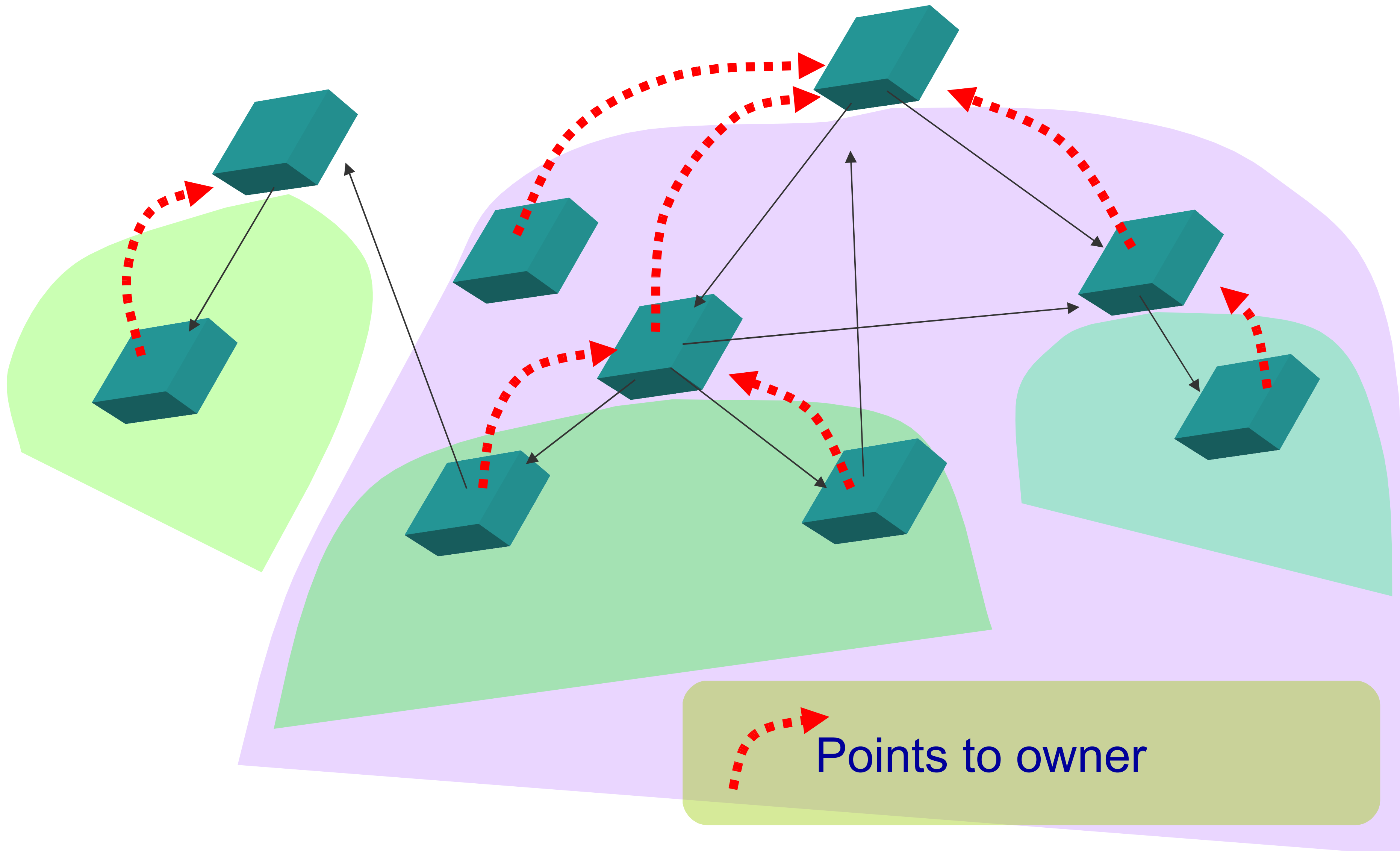
Problem:

Above technique not sufficient for invariants depending on representation objects.

Approach:

- Establish hierarchy (ownership) on objects
- Ownership discipline: When an object is exposed, so are its (transitive) owners
- An invariant of object x may only depend on
  - The fields of x and
  - The fields of objects (transitively) owned by x

Ownership as additional structure in the heap:



Dynamic ownership:

- Each object has a special field, owner, that points to its owner object
- owner is set when the object is created
- **rep** and **peer** declarations lead to implicit invariants

```
class Person {
    [Rep]  Purse! purse;
    [Peer] Person spouse;
    …
}
```

invariant purse.owner = **this**;

invariant spouse ≠ **null** ⇒
    spouse.owner = **this**.owner;

… in a setting with owners and w/o inheritance:

Object o is consistent / valid

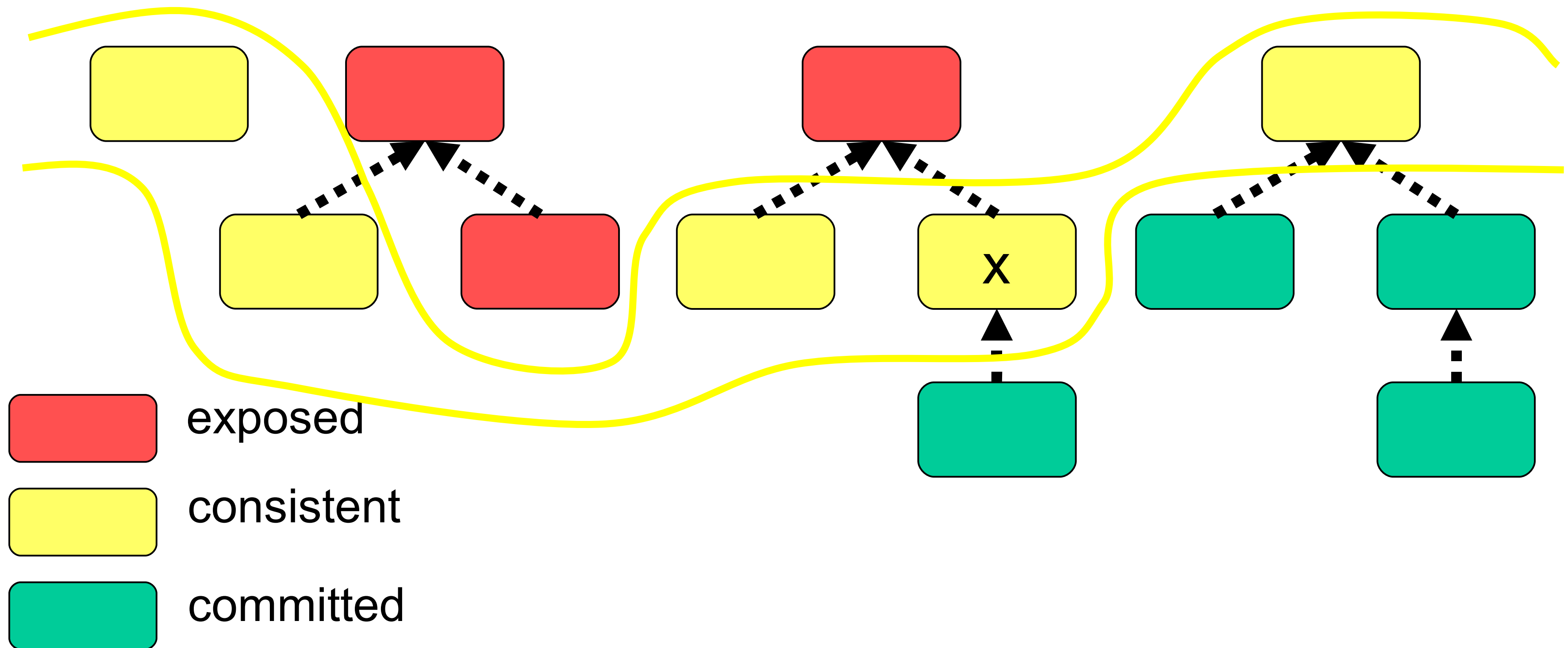
- when the constructor has finished
- when o is not exposed / mutable and
- either has no owner or the owner is exposed.

Object is peer consistent if it and all its peers are consistent.

Object o is committed

- when the constructor has finished
- when o is not exposed / mutable and
- it has an owner and the owner is not exposed.

Enforcing the ownership discipline:



Entering **expose** block for x (unpack) moves area down.

Leaving **expose** block moves area up.

Together: Order enforces ownership discipline!

```
class Person {
   [Rep] Purse! purse;
   invariant purse.amount >= 100;

   public Person() {
   { purse = new Purse(100); base(); }

   public bool pay( Person! p, int amt )
      requires amt > 0;
      modifies p.*, purse;
   {
      int payedamt = 0;
      expose( this ) {
         if( purse.contains() >= amt+100 ) {
            payedamt = purse.take(amt);
      } }
      p.recieve( payedamt );
      return payedamt!=0;
   }

   void recieve( int amt ){ /* ... */ }
}
```

this consistent
purse committed

this exposed
purse consistent

this consistent
purse committed

Check invariant when leaving expose block

Remarks on method framing:

- Ownership is a general means of abstraction
  - Addresses the transitivity problem of modifies clauses
- Allow methods to modify committed objects
- Given

```
class A { [Rep] B b; }
class B { [Rep] C c; }
```

  the method

```
void M( A a )
  modifies a.*;  { … }
```

  is allowed to modify the fields of a.b and a.b.c