



Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan



Lecture 13: SCOOP: Advanced techniques - Inheritance and Agents



SCOOP concurrency model

- Inheritance and polymorphism
- Agents



Inheritance and polymorphism

```
class A
feature
  x: separate X
  y: Y
end A
```

```
class B inherit A redefine x, y end
feature
  x: X
  y: separate Y
end B
```

```
-- in class C
r(x: separate X)
  require
    x.some_requirement
  do
    x.f
  ensure
    x.some_guarantee
end
```

```
s(y: Y)
  require
    y.some_requirement
  do
    y.f    -- y is a traitor
  ensure
    y.some_guarantee
end
```

```
a: A
b: B
...
a := b    -- Polymorphic assignment
r(a.x)    -- Valid
s(a.y)    -- Problematic
```



Inheritance and polymorphism

```
class A
feature
  r (x: separate X)
  require
    x.some_requirement
  do
    x.f
  ensure
    x.some_guarantee
  end
s (x: X)
  require
    x.some_requirement
  do
    x.f
  ensure
    x.some_guarantee
  end
end

-- in class C
a: A
my_sep_x: separate X
a := b
a.r (my_sep_x)
a.s (my_x)

b: B
my_x, my_y: X
-- Polymorphic assignment
-- Problematic
-- Valid
```

```
class B inherit A redefine r, s end
feature
  r (x: X)
  do
    x.f -- traitor
    my_y := x -- traitor
  end
s (x: separate X)
  do
    x.f
  end
end
```



The redefinition rules for the individual components of a type are now clarified:

- **Class types** may be redefined **covariantly** both in result types and argument types but the redefinition of a formal argument forces it to become **detachable**. For example, assuming that Y conforms to X , an argument $x: X$ may be redefined into $x: ?Y$ but not $x: Y$. An attribute may be redefined from $my_x: X$ into $my_x: Y$.
- **Detachable tags** may be redefined from $?$ to $!$ in result types. They may be changed from $!$ to $?$ in argument types, provided that no call on the redefined argument occurs in the original postcondition.
- **Processor tags** may be redefined from T to something more specific in result types, and from more specific to T in argument types.



- What are agents?
- Do we need special rules to handle them in SCOOP?
 - We hate special rules, don't we?
- What agents can do for us
 - Convenience
 - Full asynchrony



What are agents?

- An agent represents an operation ready to be called

$x: X$

my_operation: ROUTINE [X, TUPLE]

my_operation := agent x.f

...

my_operation.call ([]) -- Just like calling *x.f*

- Agents can be created by one object, passed to another one, and called by the latter

y.r (my_operation) -- *y* will call agent later on



What are agents?

- Arguments can be **closed** (fixed) or **open**

```
my_operation := agent io.put_string ("Hello World!")  
my_operation.call ([]) -- no arguments necessary;  
                        -- use empty tuple
```

```
my_operation := agent io.put_string (?)  
my_operation.call (["Hello World!"]) -- 1 argument
```

- Based on generic classes

```
ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE]
```

```
PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]
```

```
FUNCTION [BASE_TYPE, OPEN_ARGS -> TUPLE, RESULT_TYPE]
```



Use of agents

- Object-oriented wrappers for operations
 - > strongly-typed function pointers (C++)
 - ~ .NET delegates
- Used in event-driven programming
 - Subscribe an action to an event type
 - The action is executed when event occurs
- Loose coupling of software components
 - Model - View - Controller
- Replace several patterns
 - Observer
 - Visitor
 - . . .



Problematic agents

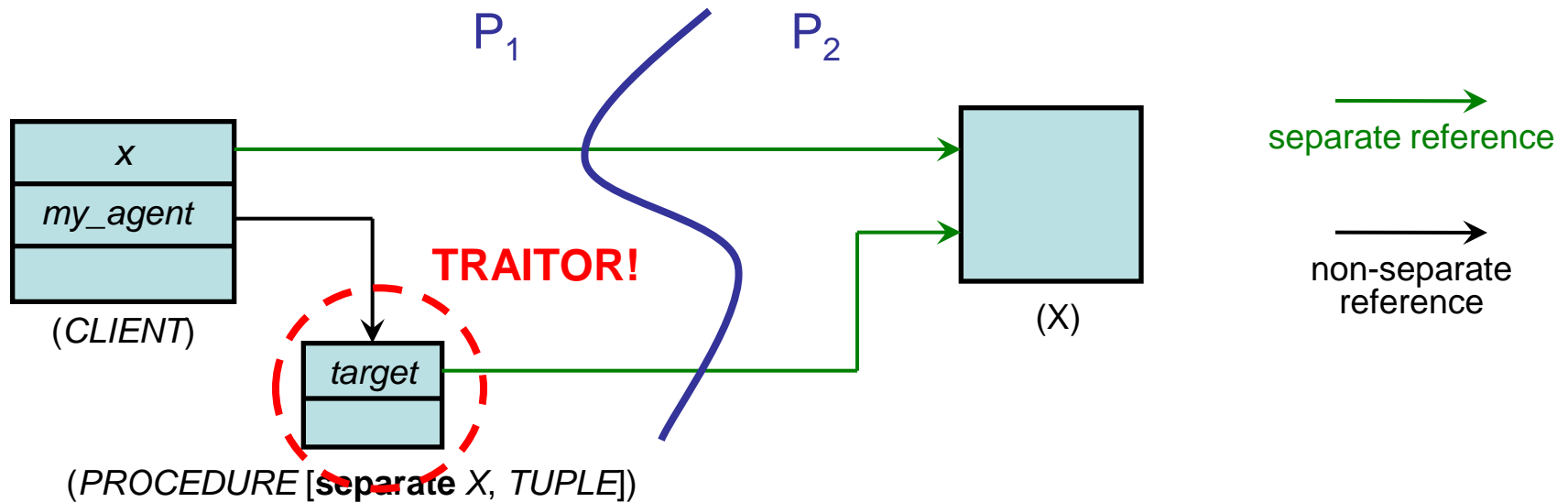
my_agent: PROCEDURE [**separate** ANY, TUPLE]

x: **separate** X

...

my_agent := **agent** *x.f*

my_agent.call ([]) -- Like *x.f* without locking *x*!





Let's make the agent separate!

```
my_agent: separate PROCEDURE [X, TUPLE]
```

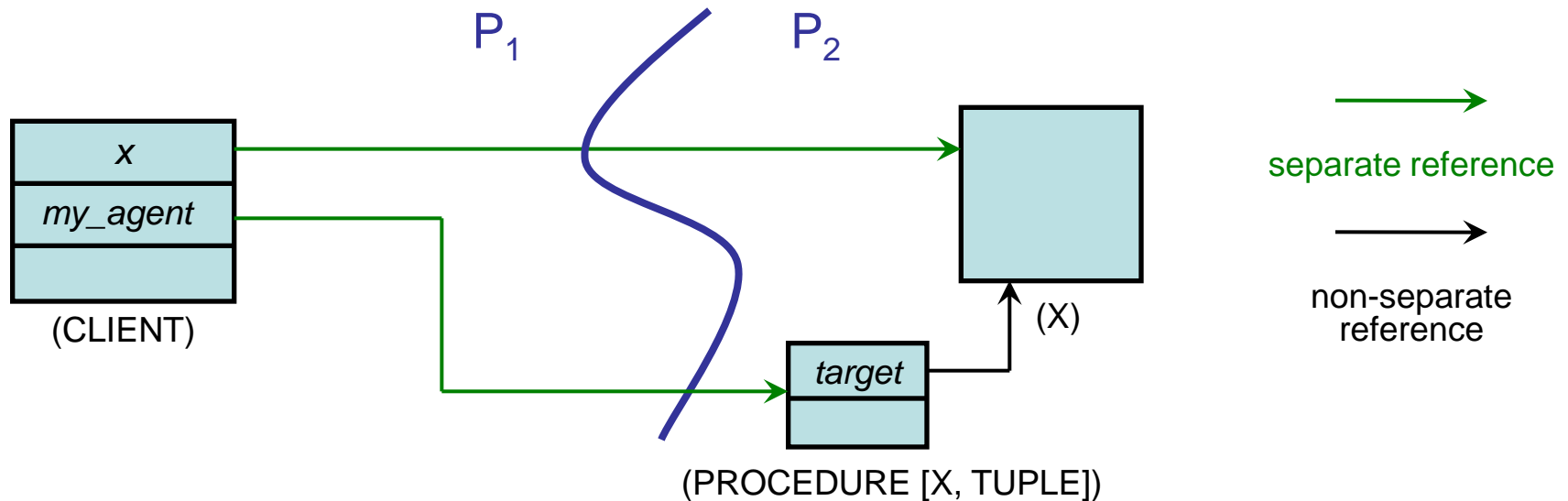
```
x: separate X
```

```
...
```

```
my_agent := agent x.f
```

```
-- agent x.f handled by x's processor
```

```
my_agent.call ([])      -- Invalid call!
```





Separate agents

- Agent built on a separate call becomes itself **separate**
 - It is handled by the same processor as its **target**

```
my_agent: separate PROCEDURE [X, TUPLE]  
x: separate X
```

...

```
my_agent := agent x.f
```

```
call (my_agent)
```

```
call (an_agent: separate PROCEDURE [ANY, TUPLE])  
  do  
    an_agent.call ([])  -- Valid separate call  
  end
```

- No special rules for separate agents +
- Agents pass processors' boundaries just as other objects do +



1st benefit: convenience

- Without agents, enclosing routines are necessary for every separate call

```
my_x: separate X
```

```
r (my_x)      -- x.f
```

```
s (my_x)      -- x.g (5, "Hello")
```

```
...
```

```
r (x: separate X)
```

```
  do
```

```
    x.f
```

```
  end
```

```
s (x: separate X)
```

```
  do
```

```
    x.g (5, "Hello")
```

```
  end
```

- With agents, we can write universal enclosing routine

```
call (agent my_x.f)
```

```
call (agent my_x.g (5, "Hello"))
```

```
call (an_agent: separate PROCEDURE [ANY, TUPLE])
```

```
  -- Universal enclosing routine.
```

```
  do
```

```
    an_agent.call ([])
```

```
  en
```



2nd benefit: full asynchrony

- Without agents, full asynchrony cannot be achieved

```
my_x, my_y: separate X
```

```
...
```

```
r (my_x)      -- Blocking call
```

```
do_local_stuff
```

```
...
```

```
r (x: separate X)
```

```
do
```

```
  x.f  -- Asynchronous
```

```
end
```

- With agents, it's easy-peasy

```
asynch (agent my_x.f) -- Non-blocking call
```

```
do_local_stuff
```

```
...
```

```
asynch (an_agent: ?separate PROCEDURE [ANY, TUPLE])
```

```
  -- Call an_agent asynchronously.
```

```
do
```

```
  ...
```

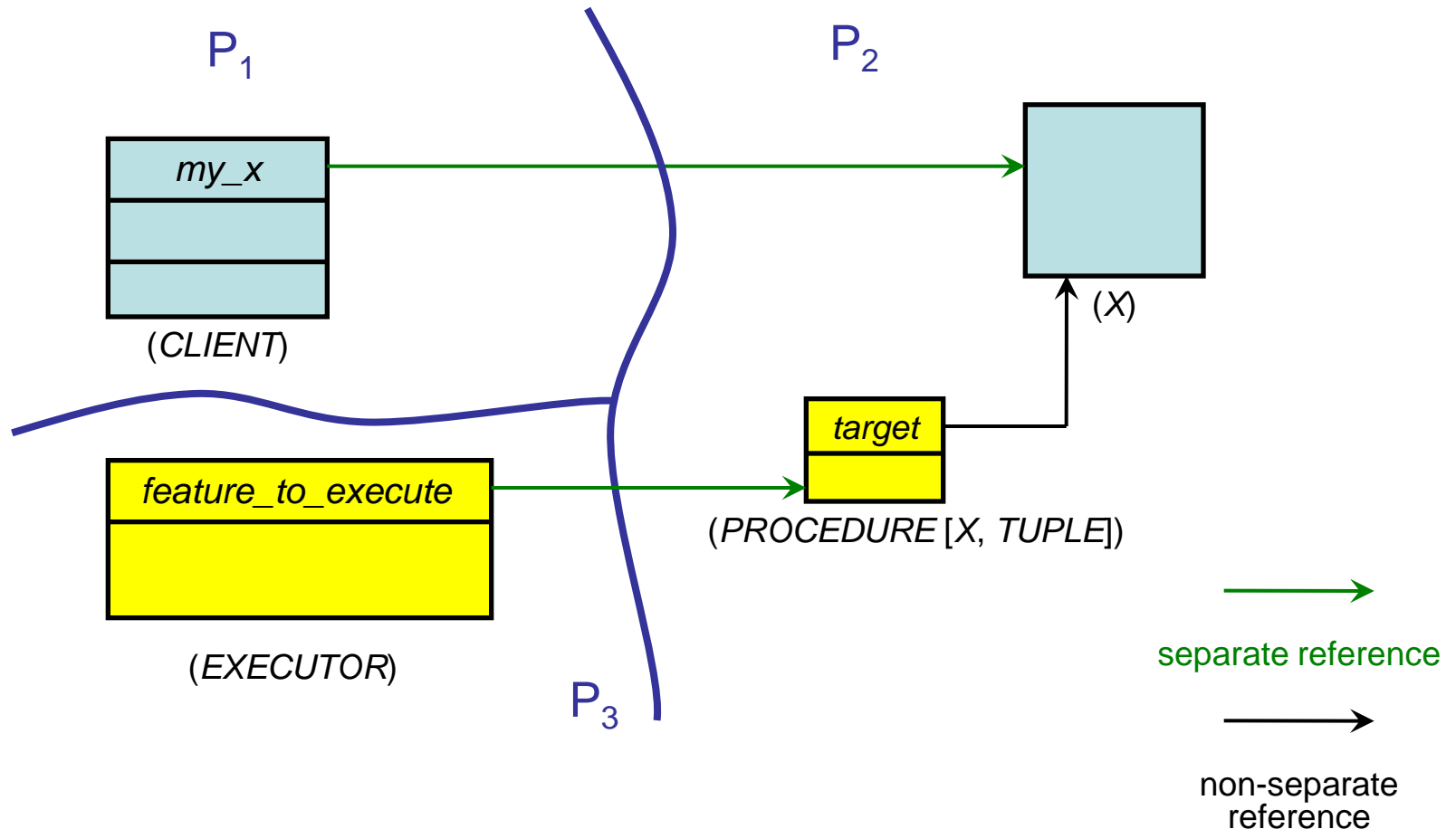
```
end
```



How to achieve full asynchrony

→ *asynch* (**agent** *my_x.f*)

→ *do_local_stuff*





Separate executors

- Feature *asynch* implemented in class *CONCURRENCY*

asynch (**agent** *my_x.f*)

```
asynch (an_agent: ?separate PROCEDURE [ANY, TUPLE])
```

```
-- Call `an_agent` asynchronously.
```

```
-- Note that `an_agent` is not locked.
```

```
local
```

```
  executor: separate EXECUTOR
```

```
do
```

```
  create executor.make (an_agent)
```

```
  launch (executor)
```

```
end
```

- Asynchronous calls on non-separate targets (including **Current**)

asynch (**agent** *f*)

```
-- Call Current.f asynchronously.
```

```
-- It will be executed when current processor becomes idle.
```