A general translation from SCOOP to Promela

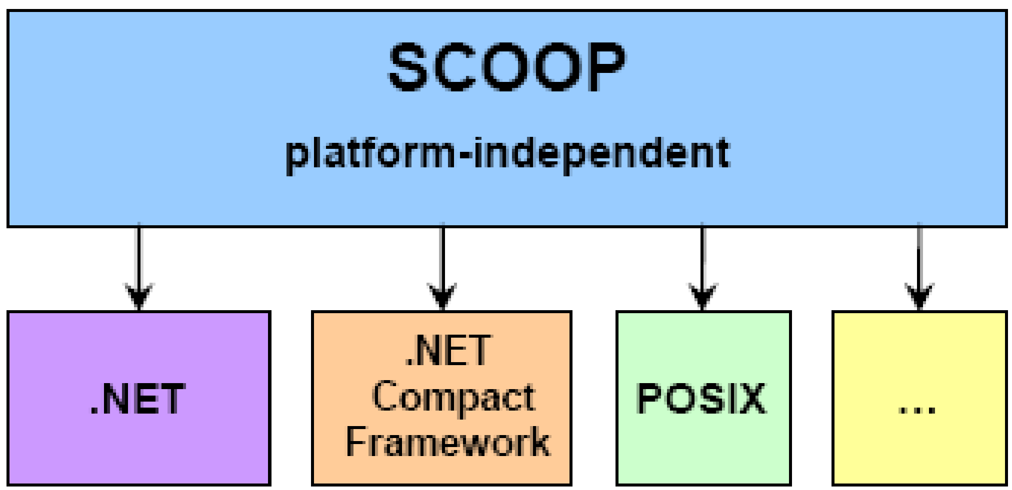
Hai Feng Huang
York University, Canada

SCOOP

- What is it ?
  - Simple Concurrent OO programming
- Main idea
  - Take OO programming as simple and pure form based on the Design by Contract
  - The running of separate objects achieves the concurrency in SCOOP

SCOOP

- Architecture



Separate Objects

- Separate objects
  - Object runs on the different processor
- Separate call is used by means of being wrapped in a routine

```
data: separate DATA
run is
    do
        do_one(data)
    end

do_one(d: separate DATA) is
    local
        test: BOOLEAN
    do
        d.one
        test := d.x = 1
    end
```

Separate Contracts

- Separate preconditions
  - Either a wait condition or a assertion
- Separate postconditions
  - separate postconditions executes asynchronously and individually.
- Invariants
  - The same as the invariants in the sequential context

Objects in SCOOP

- Non-separate objects behave the same as those in the sequential context
- Each separate object runs on the different processor and has a request queue handling separate calls
- Concurrency is achieved by separate calls

a1 || a2 || a3 || …

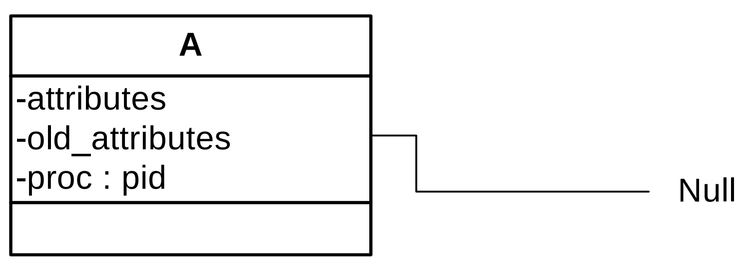
Model the non-separate object

- A class -- SCOOP

  typedef A {attribute, old_attribute, proc} -- Promela

  A a_heap[10]

  - attributes are class attributes
  - old_attributes are the old attributes for class attributes
  - proc points to the address of its corresponding processor



Model the non-separate object

```
-- SCOOP
class A create
    make
feature
  data: separate DATA
  count: INTEGER
  make(d: separate DATA) is
     do
        …
     end
  …
end -- class
```

```
-- Promela
typedef A {
  short data, old_data;
  int count, old_count;
  short proc;
}
```

- Routine is modeled by the *inline* method

Model the non-separate object

- The return result in a query routine is added in *typedef*

e.g.

```
check_data (d: separate DATA): BOOLEAN is
    do
        Result := d.ping and  d.pong
    end

typedef A {
    short data, old_data ;
    bool check_data, old_check_data;
    int count, old_count;
    short proc;
 };
```

Model the separate object

- x: **separate** A -- SCOOP
- Promela representation

typedef A {attribute, old_attribute, proc}

typedef A_ARG {feature1_arg;
feature2_arg …}

typedef A_PROC {
pid proc_id;
pid lock;
chan queue = [10]{feature_id,
A_ARG}}

A a_heap[10]

A_PROC a_procs[10]

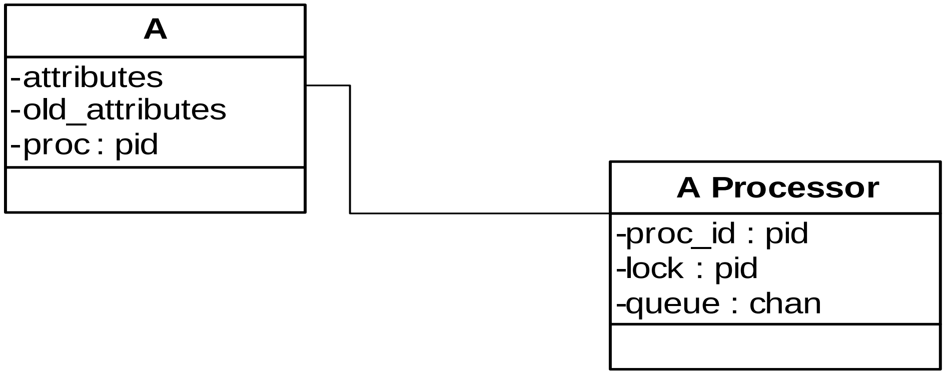
A_ARG a_arg

proctype A_CLASS (**short** ref)

{**do**

:: for each feature in the queue →
inline(routine) method *or* assertion

od; }



Model the separate object

- Each separate object has two definitions
  - One is for the sequential context (typedef A)
  - Another is for the concurrent context (typedef A_PROC)
    - proc_id is the actual processor identifier in Promela
    - lock is the identifier of processor who locks the current processor
    - Channel simulates the request queue in separate object
      - As a tool for two processor's communication
      - Declared as the asynchronized channel

```
chan queue = [10]{ feature_id,  ARG }
```

Parameters

first : the called feature id of the receiver

Second(optional): the parameters passing to the receiver

Model the separate object

- Arguments in routines (typedef A_ARG)
  - The syntax for each element in the type definition :
    "routine name" + "_" + "the argument in the routine"
  - If a routine has more than one arguments, those arguments are defined separately e.g.

Root class :

```
run(a1:separate A; b: separate B)      typedef ROOT_ARG {
  is do                                    short run_a1 ;
     …                                     short run_b1 ;
  end                                  };
```

Model the separate object

- The request queue in separate object is defined as a asynchronous channel
  - Asynchronized channel likes a bound buffer
  - feature call is achieved by send operation e.g.

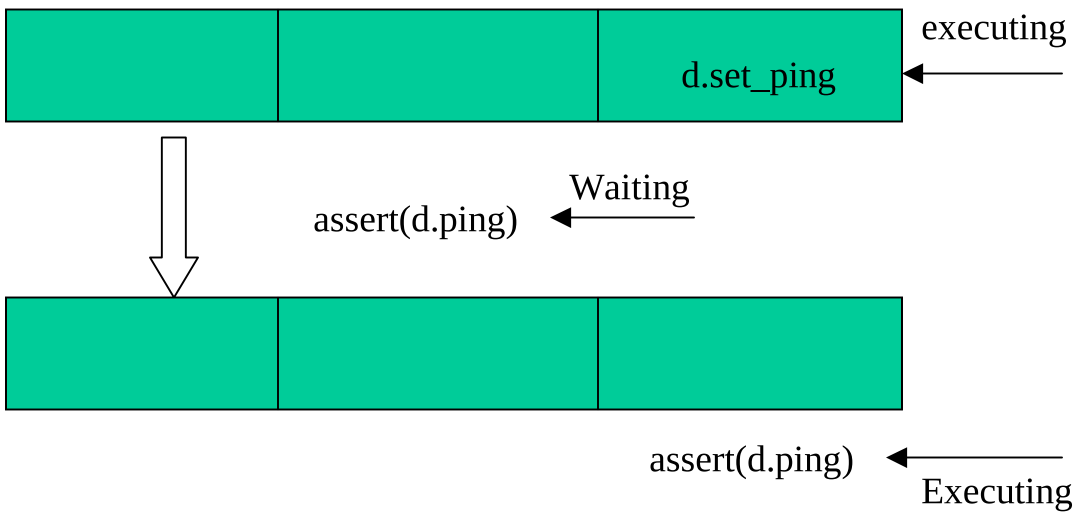
```
d.set_ping
data_procs[data_heap[d].proc].queue ! DATA_set_ping_id
```

  - The receive operation is composed of a poll and receive operations
    - Simulates the dequeue operation
    - Simplifies the separate query call operation
    - e.g.

```
a_procs[a_heap[ref].proc].queue ? <A_run_id, arg> ->
    A_run(ref);
    a_procs[a_heap[ref].proc].queue ? A_run_id, arg;
```

Model the separate object

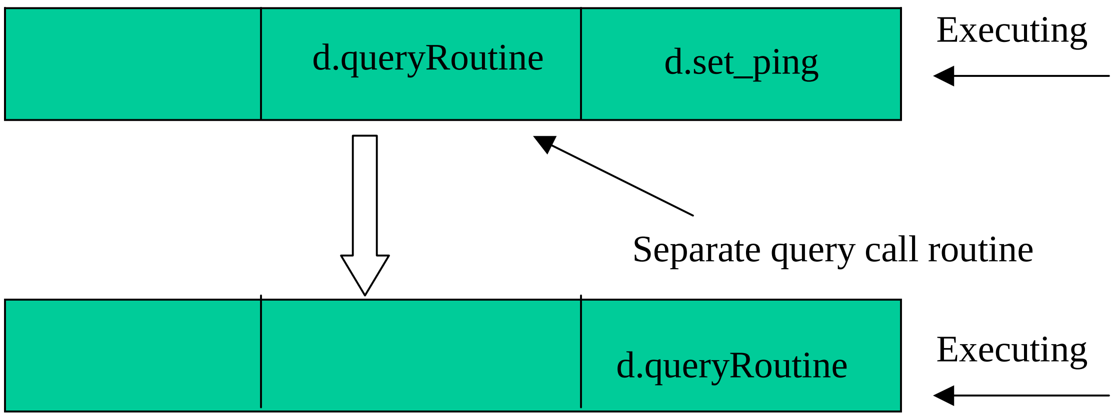
- Separate query call returning attributes
  - Executes when the channel is empty



empty(data_procs[data_heap[d].proc].queue) ->
assert(data_heap[d].ping)

Model the separate object

– Separate query call routines



empty(data_procs[data_heap[d].proc].queue) ->
some variable = data_heap[ref].queryRoutine

Model the separate object

- Both cases in separate query call use unique form:
  - empty(channel) ->

    gets the return result

Examples – models the receiver

- Each separate object has a *proctype* with its heap reference

a is a separate object of type A

```
proctype A_class(short ref) {
  do
  :: a_procs[a_heap[ref].proc].queue ?<A_run_id, arg> ->
           a_run(ref);
           a_procs[a_heap[ref].proc].queue ? A_run_id, arg;
  :: ...
  :: timeout -> break;
  od ;
};
```

Examples – models the separate object routine

-- A class (SCOOP)

```
make(d: separate
    DATA)    is
  do
        …
  end
```

-- A class (Promela)

```
inline A_make(ref, d)
{
   atomic{
      atomic{(data_procs[data_heap[d].proc].lock == 255
        || data_procs[data_heap[d].proc].proc_id ==_pid)}
               ->
       atomic{
          data_procs[data_heap[d].proc].lock = _pid;
          …
       }
     }
 }
```

Examples – models the non-separate object routine

```
-- A class (SCOOP)
make(d: DATA)  is
  do
        …
  end
```

```
-- A class (Promela)
inline A_make(ref, d)
{
  atomic{
              …
       }
 }
```

Examples – models the separate precondition

-- A class (SCOOP)

```
await_ping (d:separate
    DATA) is
  require
      d.ping
  do
      …
  end
```

-- A class (Promela)

```
inline A_make(ref, d)
{
  atomic{
    atomic{(data_procs[data_heap[d].proc].lock == 255
    || data_procs[data_heap[d].proc].proc_id ==_pid)
          && (d >= 0) && data_heap[d].ping} ->
        …
     }
   }
}
```

Examples – models the non-separate precondition

-- A class (SCOOP)

```
await_ping (d: DATA)  is
   require
       d.ping
   do
        …
   end
```

-- A class (Promela)

```
inline A_make(ref, d)
{
   atomic{
      assert (d >= 0 && data_heap[d].ping) ;
          …
       }
     }
 }
```

Examples – models the separate postcondition

-- A class (SCOOP)

```
await_ping (d:separate
                DATA) is
  require
       d.ping
  do
     …
  ensure
       d.ping
  end
```

-- A class (Promela)

```
inline A_make(ref, d)
{
   atomic{
     atomic{(data_procs[data_heap[d].proc].lock == 255
    || data_procs[data_heap[d].proc].proc_id ==_pid)
          && (d >= 0) && data_heap[d].ping} ->
        …

       data_procs[data_heap[d].proc]. queue!
              DATA_assert_ping_id;
       data_procs[data_heap[d].proc]. queue!
              PROCESSOR_unlock_id;
     }
    }
}
```

Examples – models the separate postcondition

```
proctype DATA_class(short ref) {
   do
   ::    ...
   :: data_procs[data_heap[ref].proc].queue ?< DATA_assert_ping_id > ->
           assert(data_heap[ref].ping);
           data_procs[data_heap[ref].proc].queue ? DATA_assert_ping_id
   :: data_procs[data_heap[ref].proc].queue ?PROCESSOR_unlock_id->
          data_procs[data_heap[ref].proc].lock = 255;
  od;
}
```

Examples – models the non-separate postcondition

-- A class (SCOOP)

```
await_ping (d:separate
                DATA) is
  require
      d.ping
  do
     …
  ensure
      1 == 1
  end
```

-- A class (Promela)

```
inline A_make(ref, d)
{
   atomic{
      atomic{(data_procs[data_heap[d].proc].lock == 255
      || data_procs[data_heap[d].proc].proc_id ==_pid)
              && (d >= 0) && data_heap[d].ping} ->
          …

          assert ( 1 == 1);
          data_procs[data_heap[d].proc]. queue !
                PROCESSOR_unlock_id;
       }
     }
}
```

Thank you !

Questions ?