Concurrent Object-Oriented Programming

Arnaud Bailly, Bertrand Meyer and Volkan Arslan

What is a Task?

- It is an entity with resources:
  - memory (so-called address space), and
  - processing time.
- A task may have a **priority**.
- A task may **execute** an **activity**.
- A task may be **idle** (waiting).
- A task may **terminate**.
- The **degree of parallelism** of a system at a given time is the number of **active** tasks executing on that system.

Special case: Synchronous Tasks

- **Synchronous tasks**:
  - The same program is executed on all CPUs
  - Progression is in **lock-step**.
  - Tasks are **always** active.
  - Typical of SIMD for Data-Parallelism (§).

Actions, Objects and Tasks

- **Tasks** execute **actions on behalf of objects**.
- Assume a program counter \( PC \in \mathbb{N} \)
- Define a partial function \( a : \text{task} \times \mathbb{N} \rightarrow (\text{object} \times \text{action}) \)
- \( a \) is total on \( \mathbb{N} \) and can be:
  - many-to-one (\( n \geq 1 \) tasks for 1 object)
  - one-to-one (1 task for 1 object)
  - one-to-many (\( 3 \) passive objects)

We shall only consider the **asynchronous** task model!
The Nature of \( \oplus \) and \( \triangle \) objects

- An object can be **passive**:
  - either not supported by any task, or
  - supported by a task but no autonomous behavior.
- **Data structure** or service repository.
- It is **re-active**, acting only when called.
- An object can be **active**:
  - Has (one or many) task(s) of its own.
  - Method for the object’s body (live routine).
  - **live** terminates: task stops (passivation?).
  - Active, it may or may not provide services.
- Many mixed approaches (Eiffel, ProActive, ...).

The nature of (Inter-)actions

- **Message passing**:
  - explicit send/receive primitives.
- Remote routine invocation:
  - transparent remote invocation,
  - no receive statement necessary.
- Synchronous communication:
  - the client is blocked until the supplier finishes.
- Asynchronous communication:
  - a call never blocks the client.

Relevant issues

**How to create tasks?**

**How to make tasks active?**

**How to coordinate tasks?**

Passive Objects without Task (1)

- **Essentially \( \oplus \)**
- Routine invocations execute in client’s task.

Passive Objects without Task (2)

- Allow intra-object concurrency.
- Use synchronous interaction.
- Clients can communicate through side-effects.
- Concerns orthogonal to task creation.
- Objects can be garbage-collected.
- \( \triangle \) Exception: implicit mobility like in Oz.

Passive objects with a Task

- **Essentially \( \triangle \)**
- One or many tasks.
- Accessed through routine invocation.
- Routines execute in the supplier’s task(s).
- Can easily simulate active objects.
- Examples: SCOOP, Actors...
- Garbage collection is problematic (shared task).
Active objects

- Upon creation, a task is idle!
- Creating an active object is:
  - create a passive object and a task,
  - "start" the object (autonomous routine).
- In Java: t = new Thread(); t.start;
- Starting is automated: live routine (POOL).
- An active object raises the degree of parallelism.
- Use message passing or method invocation.

Example live routine in POOL

```java
CLASS Queue
METHOD enq (item : T) BEGIN cell ! put (rear, item); front := (rear+1) MOD size END enq
METHOD deq () : T BEGIN RESULT cell ! get (front) ; %% early return front := (front +1) MOD size END deq
BODY DO IF empty THEN ANSWER (enq)
ELSIF full THEN ANSWER (deq)
ELSE ANSWER ANY FI OD
```

Other Possibility in CEiffel

- Communication based on method invocation.
- One can place autonomy annotations.
- Annotated method step is repeated forever.

```java
class MOving
creation INT
feature ()
  ... step is ->>
  DO end position.set...
  end
end = MOving
```

How to Create Activated Tasks?

- Through object creation (Java, SCOPP, POOL):
  - Types Thread, SEPARATE, Task...
  - It may be necessary to start the object.
- Through routine invocation (CEiffel, Java/RMI):
  - each routine invocation creates a new task,
  - may keep a limit on the number of active tasks,
  - may allow exclusive executions of routines.
- Through external, specific instructions:
  - fork and join,
  - cobegin-coend.

How to Activate Idle Tasks?

- Relevant for inactive tasks.
- Passive objects with a task.
- Answer: communicate with them.
- Importance of the communication mechanisms.

Communication (1)

- Using synchronous interaction
**Communication (2)**
- Using asynchronous interaction

**Synchrony versus Asynchrony**
- Synchrony coded with asynchrony:
  - send and return message for every interaction.
  - Can be quite heavy!
- Asynchrony coded with synchrony:
  - intermediate passive Queue between objects.
  - Synchrony tends to be too strong (inefficient).
  - Asynchrony can be cumbersome.
  - Allow the two of them?

**Early Return Optimization**
- Using synchronous interaction

**Future Variables**
- Synchronization-by-need in ABCL/1, EiffelV, Oz...

**Other Issues in Communication**
- active objects have different address spaces!
- Upon call s.f(x), transmit x by value or by name?
- Call-by-value: sending values away (RMI).
- Call-by-name: sending references away (Obliq).
- Object mobility:
  - is weak if the object loses (parts of) its state when moving,
  - is strong if the object moves keeps its state.
- What about moving active objects?

**Migration in DOWL [Achauer93]**
- Objects (called type modules) are passive, with a task.
- Call-by-name evaluation.
- Strong mobility.
- Each class can be fixed, mobile, or replicated.
- Each object has attributes location and fixed_at.
- Class attributes and routine forms can be declared attached, visit, or move.
- The graph of attached objects is always preserved.
- It is transitive, but not symmetric.
Example Using DOWL Primitives

```plaintext
class A
    attached b: B;
    c: C;
    // this s: h curr b j c
    g
    var a: D;
    d := f;// f m v s m d l y h a s k f t h i s
d s: 1;// x c u d h a s k f t h
    c e b // b m v s c e a s k
    c f x d = w y r k // c s t x d w y r k
    c b a r t // t h i s a d b m v w y r k
d f f m v s b a c k b r g a s k

class C
    b tr: move #: A
    bar: move b: B
```

Further Issues: Meta-Classes

- Meta-classes are **classes**:
  - one can use them as in `x := new MyClass`
- Meta-classes are **objects**:
  - they exist at run-time,
  - they have a state,
  - One can execute: if `(MyClass.count() > 5)` ...
- Problems in Distributed Smalltalk:
  - an object needs a copy of its class to execute.
  - replicating stateful (meta-)classes is expensive.
  - solution: keep (meta-)classes stateless.

To Follow...

- Coordinating objects, first act.