Conditional Concurrency Combinators

Paweł T. Wojciechowski
Poznań University of Technology

Beijing, 13 June 2012

Motivations

Using low-level synchronization primitives is notoriously difficult and error-prone.

Higher-level constructs were proposed; they can help to write correct code, e.g. *transactional memory* avoids lock-induced deadlocks.

Do programmers need more control on synchronization policies?

Do we need a diversity of concurrency constructs?

If 'yes', then we need to develop new concurrency models (or **calculi**) to understand the foundations.

This talk (work in progress)

We designed a calculus of declarative synchronization which allows:

- a program and synchronization to be defined separately
- a global synchronization policy to be locally revoked
- sync policies to be declared for classes, objects, and expressions

Our design abstracts from any concrete implementation.

The actual implementation might use only a subset of the calculus for efficiency or usability.

Simple policy declaration

Serializability using Java-like `synchronized`:

```
public class SyncCounter {
  private int c = 0;
  public synchronized void increment() { c++; }
  public synchronized void decrement() { c--; }
  public synchronized int value() { return c; }
}
```

Alternatively, we could use locks.

Simple policy declaration

Below the same program using concurrency combinators:

```
class SyncCounter {
  c = 0
  increment() {c := c+1}
  decrement() {c := c-1}
  value() {c}
  sync SyncCounter.ANY isol [ANY]
}
```

No visible gain here.

Conditional synchronization policy

A shared buffer using `synchronized`:

```
public synchronized int get() {
  int result;
  while (items == 0)
    wait ();
  items --;
  result = buffer[items];
  notifyAll ();
  return result;
}
```

Conditional synchronization policy

A shared buffer using `atomic`:

```
public int get() {
  atomic (items > 0) {
    items --;
    return buffer[items];
  }
}
```

e.g., based on *transactional memory* [Harris, Fraiser, OOPSLA '03].

TM systems typically implement either:

- *strong atomicity*: atomicity guaranteed between transactions and non-transactional code (safe but inefficient), or
- *global weak atomicity*: atomicity guaranteed among only transactions (not safe in general).

Conditional synchronization policy

... or using `atomic` and `retry`:

```
public int get() {
  atomic {
    if (items > 0)
      items --;
      return buffer[items];
    else
      retry
  }
}
```

e.g., TM in Haskell [Harris *et al.*, PPoPP '05].

Optimistic TM systems restrict the use of I/O operations in `atomic` due to implicit (or explicit as above) rollback.

Strong and weak atomicity

A shared buffer using *concurrency combinators*:

```
class P {
  int get() {
    sync (items > 0) P.get isol [ X ] in
      items := items - 1;
      buffer[items]
  }
}
```

We declare atomicity **with respect to** X using `[` X `]`.

- if X=`ANY` then `get` is strongly atomic
- if $X \neq$`ANY` then weakly atomic for code $\neq X$

We give the **choice** to the programmers who may know better what they need (hopefully).

Policy revocation

Let's assume that `get` is strongly atomic, i.e. `P.get isol [ANY]`.
If required, this global policy can be locally weakened.

E.g., we can *revoke* atomicity of `get` in expression `e` with respect to method `dirty_read` (any other code isn't affected):

```
sync
  P.get !isol [Q.dirty_read]
in
  e
```

X `!`s `[`Y`]` revokes any valid synchronization constraint s declared for X with respect to Y.

Syntactic sugar and equations

Let p be either s or $!\,s$. We use syntactic sugar:

- $X\ p\ Y$ for $X\ p\ [Y]\ \wedge\ Y\ p\ [X]$
- $X\ p$ `self` for $X\ p\ X$

Some equations:

- $X\ p\ Y \equiv Y\ p\ X$
- $X\ p\ [Y] \neq Y\ p\ [X]\ (X \neq Y)$
- $X\ p\ [X] \equiv X\ p\ X$

Complex policy declaration

Let's declare the Readers-Writers synchronization policy for all objects of class RW:

```
class RW {
  v = 0
  read () = { v }
  write (x:Int) = {v := x}

  sync RW.write isol RW.read ∧ RW.write isol RW.write
}
```

A lock-based implementation would be less intuitive and not easily customized. A higher-level RW library would be OK (but less control).

Policy refinement locally

Suppose we want to locally customize RW synchronization, e.g., allow concurrent writes and reads in expression e.

Below is the code using concurrency combinators:

```
let o = new RW in
  sync
     o.write !isol o.read
  in
    e
```

The calculus of concurrency combinators

A call-by-value λ-calculus extended with classes and objects:

Variables	x, y, z, o	$\in Var$		
Comb. arg. names	A, B	$\in Lab$		
Class names	P, Q	$\in Lab$		
Field names	f			
Method names	m			
Selector names	n	$\in Sel$	$::=$	$f \mid m$
Types	t		$::=$	$P \mid \texttt{Unit} \mid \texttt{Boolean} \mid \overline{t} \to t'$
Combinator args	X, Y		$::=$	$e \mid e.\texttt{ANY} \mid P.n \mid P.\texttt{ANY} \mid \texttt{ANY} \mid X \oplus Y \mid A$
Combinators	a, b, c		$::=$	$X\ \texttt{isol}\ [Y] \mid X \bowtie [Y] \mid X\ !\texttt{isol}\ [Y] \mid X\ !\bowtie\ [Y] \mid$ $a \wedge b \mid \texttt{if}\ e\ \texttt{then}\ a\ \texttt{else}\ b$
Funct. abstractions	F		$::=$	$\overline{x} : \overline{t} = \{e\}$
Methods	M		$::=$	$t\ m\ F$
Classes	K	$\in Class$	$::=$	$\texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k,\ M_1, \ldots, M_n\} \mid$ $\texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k,\ M_1, \ldots, M_n,\ e_s\}$
Values	v, w	$\in Val$	$::=$	$() \mid \texttt{new}\ P \mid \textbf{true} \mid \textbf{false} \mid F$
Expressions	e	$\in Exp$	$::=$	$x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e := e \mid \texttt{fork}\ e \mid e_s$
Sync. expressions	e_s	$\in Exp_s$	$::=$	$\texttt{let}\ A \leftarrow X\ \texttt{in}\ e \mid \texttt{sync}\ (e)a\ \texttt{in}\ e$

Nested atomic sections

Barrier synchronization implemented using `atomic`:

```
void barrier() {
  atomic { count++; }
  atomic(count == NUMTHREADS) {
    /* Barrier reached */
  }
}
```

```
atomic {
  ... barrier(); ...
}
```

Closed-nesting? => safety guaranteed but **deadlock**!

Open-nesting? => more flexible but not safe

Nested atomic blocks

The same program using our language:

```
class P {
  barrier() = {  /* e.g., code as before */ }
  m() = {
      ...
      barrier()
      ...
  }

  sync P.m isol [ANY] ∧ P.m !isol [P.barrier]
}
```

If any invariants protected by `atomic` do not depend on variable `count`, they (most likely) are not invalidated by policy revocation.

Nested atomic blocks

The same program using our language:

```
class P {
  barrier() = {  /* e.g., code as before */ }
  m() = {
      ...
      barrier()
      ...
  }

  sync P.m isol [ANY] ∧ P.m !isol [P.barrier]
}
```

If any invariants protected by `atomic` do not depend on variable `count`, they (most likely) are not invalidated by policy revocation.

We would like to be able to verify this statically (safety).

Barrier synchronization combinator

We can use a *barrier combinator* $\bowtie$ to declare barrier synchronization on any variable (below we use a variable `barrier`):

```
sync o.barrier ⋈ self in
  e
```

where `e` spawns threads, each one executing:

```
if (o.barrier =< NUMTHREADS) then
  o.barrier := o.barrier + 1;   /* block on a write /*
  e1
else
  sync o.barrier ! ⋈ self in
  o.barrier := 0;
  e2
```

Conclusions and future work

Synchronization policy:

- can be declared separately from the main code
- can be declared with respect to classes, objects, and expressions
- can be locally revoked, e.g., to avoid deadlock

Future work:

A type system for safe local revocation of synchronization policy.