Mobile Ambient Synchronization

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
<www.research.digital.com/SRC/personal/Luca_Cardelli/home.html>

1 Introduction

This note describes a non-distributed implementation of the basic operations of the ambient calculus [1]. The implementation uses standard shared-memory concurrent programming technology (threads, mutexes, conditions), in the form provided by Java [2]. The presentation is self-contained, but previous familiarity with the ambient calculus is useful for motivation and intuitions.

An ambient, in the sense in which we are going to use this word, has the following main characteristics:

- An ambient is a *bounded* placed where computation happens. The interesting property here is the existence of a boundary around an ambient. If we want to move computations easily we must be able to determine what should move; a boundary determines what is inside and what is outside an ambient.
- An ambient is something that can be nested within other ambients, creating a hierarchy of ambients.
- An ambient is something that can be moved as a whole, along with all its subambients.

More precisely, we will investigate ambients that have the following structure:

- Each ambient has a name, which must be exhibited when entering or exiting the ambient.
- Each ambient has a collection of local agents (a.k.a. threads, processes, etc.). These are the computations that run directly within the ambient and, in a sense, control the ambient. For example, each agent can concurrently instruct the ambient to move in a different direction.
- Each ambient has a collection of subambients. Each subambient has its own name, agents, subambients, etc.

2 Ambients

We review the syntax and semantics of the ambient calculus.

2.1 Syntax

For the purpose of this note, we consider the following (sub-)calculus of ambients:

Ambient Terms

n, m, etc.	ambient names
P,Q ::=	terms
$n[P]$	ambient with name n and content P
0	termination
$P \mid Q$	parallel execution
enter n. P	enter an ambient n, then do P
exit n. P	exit an ambient n, then do P

Syntactic conventions:

parentheses (P) can be used for grouping.

enter n. P|*Q* is read as (*enter n. P*)|*Q*; similarly for *exit*.

Common abbreviations:

$n[] \triangleq n[0]$

enter n $\triangleq$ *enter n.* 0

exit n $\triangleq$ *exit n.* 0

2.2 Structural equivalence relation

Terms of the syntax are identified up to the following equivalence:

(Struct Refl)	$P \equiv P$
(Struct Symm)	$P \equiv Q \Rightarrow Q \equiv P$
(Struct Trans)	$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$
(Struct Par Zero)	$P \mid 0 \equiv P$
(Struct Par Comm)	$P \mid Q \equiv Q \mid P$
(Struct Par Assoc)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
(Struct Amb)	$P \equiv Q \Rightarrow n[P] \equiv n[Q]$
(Struct Par)	$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$
(Struct Enter)	$P \equiv Q \Rightarrow$ *enter n.* $P \equiv$ *enter n.* Q
(Struct Exit)	$P \equiv Q \Rightarrow$ *exit n.* $P \equiv$ *exit n.* Q

N.B:

$n[P] \mid n[Q] \not\equiv n[P \mid Q]$

2.3 Reduction relation

The operational behavior of the ambient calculus is captured by a non-deterministic reduction relation between terms. If a term has no reduction, we say it is *blocked*.

One-step reduction is the smallest binary relation on terms, $\longrightarrow$, generated by the following rules:

(Red Enter)	$n[\mathit{enter}\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$
(Red Exit)	$m[n[\mathit{exit}\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$
(Red Amb)	$P \longrightarrow Q \ \Rightarrow\ n[P] \longrightarrow n[Q]$
(Red Par)	$P \longrightarrow Q \ \Rightarrow\ P \mid R \longrightarrow Q \mid R$
(Red Struct)	$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \ \Rightarrow\ P' \longrightarrow Q'$

The interesting reductions are (Red Enter) and (Red Exit); the rest are congruences and structural equivalences.

N.B.:

$0 \not\longrightarrow$

$P \longrightarrow Q \ \not\Rightarrow\ \mathit{enter}\ n.\ P \longrightarrow \mathit{enter}\ n.\ Q$

$P \longrightarrow Q \ \not\Rightarrow\ \mathit{exit}\ n.\ P \longrightarrow \mathit{exit}\ n.\ Q$

The reduction relation, $\longrightarrow^*$, is the reflexive and transitive closure of $\longrightarrow$.

3 Examples

The following example shows a race between moves in different ambients.

$n[\mathit{enter}\ m.\ \mathit{exit}\ m] \mid m[\mathit{enter}\ n.\ \mathit{exit}\ n]$

It produces, e.g.:

Ambient n: Entered m
Ambient n: Exited m
Ambient m: Entered n
Ambient m: Exited n
Result: n[] | m[]

Another interesting race is given by an ambient that can go either up or down:

$n[\ m[\mathit{enter}\ p.\ \mathit{exit}\ p \mid \mathit{exit}\ n.\ \mathit{enter}\ n] \mid p[]]$

It can produce either:

Ambient m: Entered p
Ambient m: Exited p
Ambient m: Exited n
Ambient m: Entered n
Result: n[p[] | m[]]]

Or:

Ambient m: Exited n
Ambient m: Entered n
Ambient m: Entered p
Ambient m: Exited p
Result: n[p[] | m[]]

Note how an *enter* agent blocks waiting for the right sibling to appear, possibly forever, and an *exit* agent blocks waiting for the right parent to materialize, possibly forever.

4 Implementation

4.1 The problem

An ambient has the general shape:

$$n[P_1 \mid ... \mid P_p \mid m_1[...] \mid m_q[...]] \qquad \text{for } p,q \geq 0$$

where (w.l.o.g.) each P_i is an agent starting with *enter* or *exit*.

The problem
We want to find a (nondeterministic) implementation of the reduction relation $\rightarrow^*$, such that each P_i in an ambient is executed by a concurrent thread (and so on recursively in the subambients $m_j[...]$).
Desirable properties of the implementation are:
Liveness: if $P \rightarrow Q$ then the implementation must reduce P.
Soundness: if the implementation reduces P to Q, then we must have $P \rightarrow^* Q$.
Completeness: if $P \rightarrow^* Q$, then the implementation must be able (however unlikely) to reduce P to some $Q' \equiv Q$.

Completeness is a very strong requirement; it may be unachievable in a practical implementation. However, the abstract algorithm on which the implementation is based should be able to satisfy it.

4.2 An algorithm (informal description)

The nested ambients form a tree structure. (During transitions, though, a forest structure temporarily develops.) The *enter* and *exit* operations modify this tree structure. Each node in the tree has a lock/condition. Whenever node A is an ancestor of node B, the lock for A is acquired before the lock for B. An operation holds at most two locks at a time. A parent-child relationship can be created or deleted only when both nodes are locked.

A movement operations is initiated by a thread in a child ambient and is synchronized on (the lock of) the parent ambient. (E.g.: *parent*[*child*[*thread* | ...] | ...].)

For any movement operation, a child thread first races upwards to find the parent (without any locks held to avoid trivial deadlocks) and then locks the parent to begin the operation.

After the parent lock is established, the child thread makes sure the child is still a child (it might have moved just before the parent was locked). If it is no longer a child, the thread backs out and tries again to lock its (new) parent.

When the parent lock is acquired, and the child is still a child, the child thread checks to see if the movement operation can fire. That is, on *exit n*, the parent name must be *n* and a grandparent must exist; on *enter n*, there must exist a sibling named *n* (these conditions cannot change while the parent lock is held). If movement cannot fire, the thread goes into a wait for a signal on the parent node indicating that something has changed and that movement might possibly fire.

When the child thread is so signalled, the conditions for the operation may be right. However, the child thread cannot just continue from where it left off; it reacts by going all the way back to the beginning and starting again looking for its parent, acquiring its lock, etc. This is because the parent may have changed while the child was sleeping, by the action of some other thread, and the node it was waiting on may no longer be its parent.

Finally, if the parent is locked, if the child is still a child, and if the operation can fire, then something interesting happens depending on the operation.

Enter: we have *parent*[*child*[*enter n.P*| ...] | *n*[Q] | ...]. (A) remove phase: the lock on the *child* ambient is acquired, the parent-child link is removed, the parent and child nodes are signalled, and both locks are released; (B) insert phase: the locks on *child* (the root of a disconnected tree) and then on *n* are acquired; a parent-child link is established between *n* and *child*, both nodes are signalled, and both locks are released.

Exit: we have *parent*[*child*[*exit parent.P*| ...] | ...]. (A) remove phase: the lock on the *child* ambient is acquired, the parent-child link is removed, the parent and child nodes are signalled, and both locks are released; (B) insert phase: the locks on *child* and then *grandpa* (the parent of *parent*) are acquired; a parent-child link is established between *grandpa* and *child*, both nodes are signalled, and both locks are released.

N.B.: Between the remove and insert phases we have a forest situation with all the locks released. (The *child* lock could be held through the transition, but the future parent lock has never been acquired.) The insertion of a given child node into a given parent node, however, has been decided and will be carried out at some opportune time. The effect is that an ambient may temporarily be “in transit”, neither here nor there, chasing its future parent while this may be moving. Since there is no way to notice the absence of an ambient, this implementation detail has no observable effect.

4.3 An algorithm (transition diagrams)

The algorithm is described pictorially as a sequence of state transitions. Each operation has a description of the form:

The state transitions proceed left-to-right, except where indicated by a right-to-left arrow looping back to a previous state.

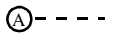
The operations work on the tree structure of ambients. The state of a node in this structure is indicated as follows:

a node in the tree of ambients

the thread of interest is waiting for a condition on this node

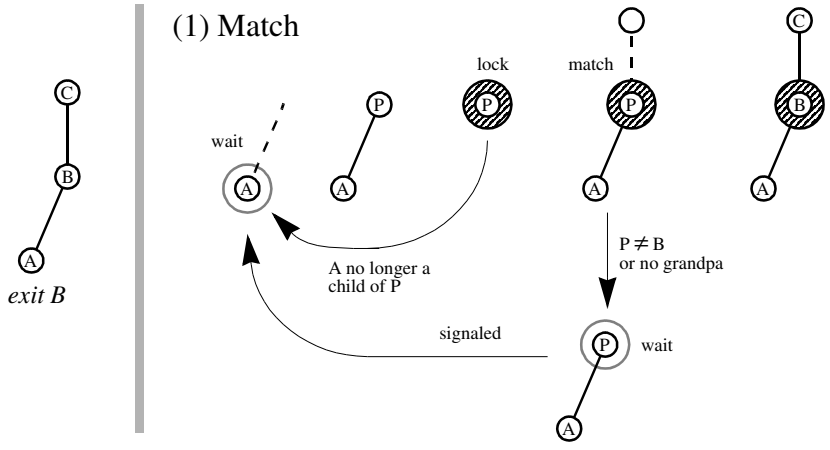
the thread of interest has locked this node

the thread of interest has signaled a condition on node

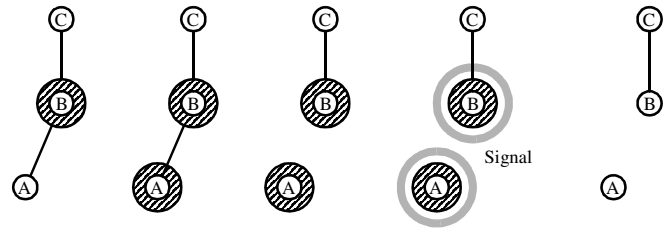


the thread of interest is looking for an appropriate neighbor of this node

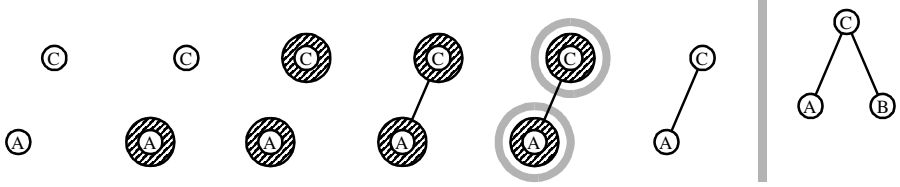
4.3.1 A thread of A executes exit



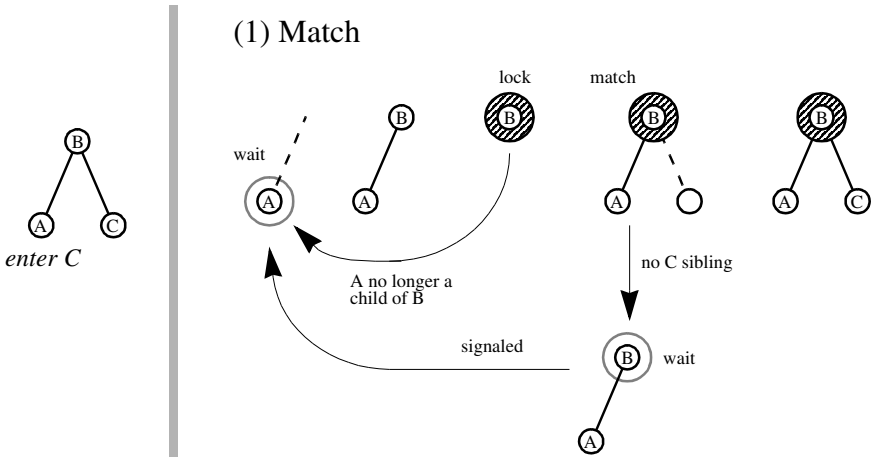
(2) Remove



(3) Insert



4.3.2 A thread of A executes enter



4.4 Java code

This section provides the essential Java code for the implementation of the algorithm. The class implementing ambients is called `Ambient`. It relies on a class `Name` (for the ambient names *n*), a class for entry capabilities, `InCap` (for the *enter n* syntax), and a class for exit capabilities, `OutCap` (for the *exit n* syntax).

The latter three classes are not shown. A name is represented as a triple of keys (that is, random numbers), of which the first is the entry key and the second is the exit key. (The third key prevents reconstructing a name from the first two.) A capability holds a single key. An entry capability is obtained from a name by copying the first key. An exit capability is obtained from a name by copying the second key. The only interesting method between names and capabilities is the `matches` method of names. A name matches an entry capability if the key of the capability is the first key of the name. A name matches an output capability if the key of the capability is the second key of the name.

```
package Ambit;

import java.lang.Thread;
import java.util.Vector;
import java.util.Enumeration;

public class Ambient {
// An ambient.

  // === Instance Variables === //

  private Name name; // The current name of the ambient.
  private Ambient parent; // The ambient surrounding this, or null.
  private Vector ownAmbients; // The set of active subambients.
  private Vector ownAgents; // The set of agents in this ambient.

  // === Constructors === //

  public Ambient(Name name) {
    this.parent = null;
    this.name = name;
    this.ownAmbients = new Vector();
    this.ownAgents = new Vector();
  }

  // === Name, Parent, Children === //

  public synchronized Name getName() {
  // The name of this ambient.
    return name;
  }

  private synchronized void setParent(Ambient newParent) {
  // Set the parent of this ambient.
    parent = newParent;
  }

  private synchronized void nullifyParent() {
  // Null the parent of this ambient.
    parent = null;
    notifyAll(); // anybody who might be waiting here
  }
```

```
private synchronized Ambient waitForParent() {
// Get the parent of this ambient (blocks on null parent).
  while (parent == null){
    try {wait();} catch (InterruptedException ex) {}
  }
  return parent;
}

// === InsertChild === //

private synchronized void insertInto(Ambient newParent) {
  parent = newParent;
  newParent.insertChild(this);
  notifyAll(); // wake up agents waiting for a non-null parent
}

private synchronized void insertChild(Ambient ambient) {
  ownAmbients.addElement(ambient);
  notifyAll(); // wake up siblings waiting for new child
}

// === RemoveChild === //

private void removeChild(Ambient ambient) {
  ownAmbients.removeElement(ambient);
  ambient.nullifyParent();
  notifyAll(); // wake up waiting agents so they can go find new parent
}

// === Moving out === //

public void moveOut(OutCap parentCap) throws AmbitException {
// Move this ambient above parent.
  Ambient newParent;
  do {
    newParent = waitForParent().movingOut(this, parentCap);
    if (newParent != null) { break; }
  } while (true);
  insertInto(newParent);
}

private synchronized Ambient movingOut(Ambient ambient, OutCap outCap) {
  Ambient myParent = null;
  if (ownAmbients.contains(ambient)) { // arbitrates race
    myParent = findMatchingParent(outCap);
    if (myParent != null) {
      removeChild(ambient);
    } else {
      try {wait();} catch (InterruptedException ex) {}
    }
  }
  return myParent;
}

private Ambient findMatchingParent(OutCap outCap) {
  if ((parent != null) && getName().matches(outCap)) {
    return parent;
  } else {
    return null;
  }
}

// === Moving in === //

public void moveIn(InCap receiverCap) throws AmbitException {
// Move this ambient inside receivingAmbient; they are currently siblings.
  Ambient newParent;
```

```
    do {
      newParent = waitForParent().movingIn(this, receiverCap);
      if (newParent != null) { break; }
    } while (true);
    insertInto(newParent);
  }

  private synchronized Ambient movingIn(Ambient ambient, InCap inCap) {
    Ambient myChild = null;
    if (ownAmbients.contains(ambient)) { // arbitrates race
      myChild = findMatchingChild(ambient, inCap);
      if (myChild != null) {
        removeChild(ambient);
      } else {
        try {wait();} catch (InterruptedException ex) {}
      }
    }
    return myChild;
  }

  private Ambient findMatchingChild(Ambient movingAmbient, InCap inCap) {
    Enumeration enum = ownAmbients.elements();
    while (enum.hasMoreElements()) {
      Ambient ambient = (Ambient)enum.nextElement();
      if ((ambient != movingAmbient) && (ambient.getName().matches(inCap))) {
        return ambient;
      }
    }
    return null;
  }

}
```

References

[1] Cardelli, L. and A.D. Gordon, **Mobile ambients**. *(To appear.)* 1997.

[2] Lea, D., **Concurrent programming in Java: design principles and patterns**. Addison-Wesley. 1996.

[3] C. Schlatter Ellis, **Concurrent search and insertion in 2-3 trees**. *Acta Informatica* **14**, 63-86. 1980.