UDC 004.738:519.85

MOBILE AMBIENT CALCULUS WITHIN INSERTION MODELING SYSTEM

Klionov D.
Kherson state university

This article is focused on the Insertional Modeling System developed by A.A. Letichevsky of the department 100/105 of the Glushkov Institute of Cybernetics, National Academy of Science of Ukraine, Kyiv, Ukraine Insertion Modeling System (IMS)[1] is buit on the Algebraic Programming System (APS) that also was developed by A.A. Letichevsky in 1987. and on the way of implementation of ***ambient calculus*** *– a process calculus devised by Luca Cardelli and Andrew D. Gordon in 1998, and used to describe and theorize about concurrent systems that include mobility.[6] In this article we are going to discuss the constructing of the framework for mobile ambients using the Insertion Modeling System.*

Key words: *process calculi, ambient calculus, insertion modeling****;***

Introduction

In computer science, the **ambient calculus** is a process calculus, used to describe and theorize about concurrent systems that include *mobility*. There are two areas of *mobility*: computation carried out on mobile devices (laptops, personal digital assistants, etc.), and mobile computation concerning executable code that is able to move between devices (applets, agents, etc.). The idea of ambient calculus is to provide a unified framework for modeling both kinds of mobility. It is used to model interactions in concurrent systems by defining the bounded place where the computation can occur as it's fundamental primitive – ambient. The key notion here is *boundary* it defines a computational agent that can be moved in its entirely. The simplest examples are:

a web page (bound by a file)
a virtual address space (bound by address range)
a file system (bound within a physical volume)
a laptop (bound by it's case and data ports)
The key properties of ambients are:
every ambient is defined by a unique name
ambients can be placed inside each other
ambients can be moved as a whole

A crossing of the boundaries i.e. movement of ambients represent computations. There are three basic capabilities (or operations) of ambients:

in m.P instructs the surrounding ambient to enter some sibling ambient *m*, and then proceed as *P*

out m.P instructs the surrounding ambient to exit its parent ambient *m*

open m.P instructs the surrounding ambient to dissolve the boundary of an ambient *m* located at the same level

Ambient calculus are expected to change the whole Mobile programming the same as the Functional approach revolutionized the programming in general.

In order to create a framework for mobile ambients it is vital to choose an appropriate technology that would be able to provide a fundamental toolkit for π-calculus that is the base for ambient calculus. π-calculus in contrast to the λ-calculus is able to describe concurrent computations whose configuration may change during the computation. The system that combines λ-calculus and π-calculus is Insertion Modeling System(IMS) developed by A.A. Letichevsky of the Glushkov Institute of Cybernetics, National Academy of Science of Ukraine, Kyiv, Ukraine. Insertion modeling is the technology of system design founded on the theory of interaction of agents and environments. It is based on process algebra and is intended for the unification of different models of interaction and computation (such as CCS, CSP, π- calculus, mobile ambients etc.).

The purpose

The purpose of this paper is to define the notion of implementation of the basic concepts of ambient calculus within the Insertion Modeling system.

First let me give a basic overview to the Ambient calculus. The main notion of ambient is boundary, as it already was said before. The most interesting property here is an existence of a boundary around an ambient. It is introduced in order to determine what is inside or outside of an ambient, if we are about to move computations we must be able to define it's boundary uniquely.

Process mobility is actually represented by crossing of their boundaries not as a communication between processes. Security features are represented by ability or inability to cross a certain boundary.

Each ambient has a following structure:

– Every ambient is defined by name. It is used to control an access to inner part of ambient (that what is "inside" of the boundaries). The name is used to extract capabilities that are bound to it, for passing around, and creating a new ambients
– Every ambient has a set of local agents (also known as threads or processes) that represent computations running directly within an ambient.
– Every ambient has a set of subambients.

Mobile ambient calculus's syntax is defined in following table. The main syntactic categories are processes (that combine both ambients and agents that execute actions) and capabilities. (table1)

P,Q ::=	Processes
$(\nu n)P$	Restriction
0	Inactivity
P|Q	Composition
!P	Replication
M[P]	Ambient
M.P	Capability action
(x).P	Input action
<M>	Async output action
M::=	Capabilities
x	Variable
n	Name
in M	Can enter into M
out M	Can exit out of M
open M	Can open M
ε	Null
M.M'	path

Table 1. Ambient calculus syntax.

Abbreviations:

$(\nu n_1 ... \nu_m)P \equiv (\nu n_1)...(\nu n_m)P$

$n[] \equiv n[\mathbf{0}]$

$M \equiv M.\mathbf{0}$ (where appropriate)

The first four process primitives (restriction, inactivity, composition, replication) are commonly presented in process caluli. Ti these we add ambients, an execution of capabilities M.P and primitives for performing communications (input\output action). Next these primitives will be discussed in details. The semantics is introduced informally. A reduction relation P→ Q describes evolution of process P into new process Q.

Restriction
The restriction operator:

$(\nu n)P$

creates a new (unique) name *n* within a scope *P*. The new name can be used to name ambients and to operate on ambients by name.

Inaction
The process:

0

is the process that does nothing. It does not reduce.

Parallel
Parallel execution is denoted by a binary operator that is commutative and associative:

$P|Q$

It obeys the rule:

$P\rightarrow Q \Rightarrow P|R \rightarrow Q|R$

This rule directly covers reduction on the left branch; reduction on the right branch is obtained by commutativity.

Replication
Replication is a technically convenient way of representing iteration and recursion. The process:

$!P$

denotes the unbounded replication of the process *P*. That is, !*P* can produce as many parallel replicas of *P* as needed, and is equivalent to *P* | !*P*. There are no reduction rules for !*P*.

Ambients
An ambient is written:

$n[P]$

where *n* is the name of the ambient, and *P* is the process running inside the ambient. In *n*[*P*], it is understood that *P* is actively running, and that *P* can be the parallel composition of several processes. We emphasize that *P* is running even when the surrounding ambient is moving. We express the fact that *P* is running by a rule that says that any reduction of *P* becomes a reduction of *n*[*P*]:

$P\rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$

In general, an ambient exhibits a tree structure induced by the nesting of ambient brackets. Each node of this tree structure may contain a collection of (non-ambient) processes running in parallel, in addition to subambients. We say that these processes are running in the ambient, in contrast to the ones running in subambients. The general shape of an ambient is, therefore:

$n[P1 \mid ... \mid Pp \mid m1[...] \mid ... \mid mq[...]]$

Nothing prevents the existence of two or more ambients with the same name, either nested or at the same level. Once a name is created, it can be used to name multiple ambients. Moreover, !*n*[*P*] generates multiple ambients with the same name. This way, for example, one can easily model the replication of services.

Ambient I/O
The simplest communication mechanism that we can imagine is local anonymous communication within an ambient (ambient I/O, for short):

(x).P input action

<M> async output action

An output action releases a capability (possibly a name) into the local ether of the surrounding ambient. An input action captures a capability from the local ether and binds it to a variable within a scope. We have the reduction:

$(x).P|\langle M\rangle \rightarrow P\ \{ x \leftarrow M\}$

This local communication mechanism fits well with the ambient intuitions. In particular, long-range communication, like long-range movement, should not happen automatically because messages may have to cross firewalls.

Actions and Capabilities

Operations that change the hierarchical structure of ambients are sensitive. Hence these operations are restricted by *capabilities*. Thanks to capabilities, an ambient can allow other ambients to perform certain operations without having to reveal its true name. With the communication primitives, capabilities can be transmitted as values.
The process:

M.P

executes an action regulated by the capability M, and then continues as the process P. The process P does not start running until the action is executed. For each kind of capability M there is a specific rule for reducing $M.\ P$.

Entry Capability

An entry capability, *in m*, can be used in the action:
in m. P
which instructs the ambient surrounding *in m*. P to enter a sibling ambient named m. If no sibling m can be found, the operation blocks until a time when such a sibling exists. If more than one m sibling exists, any one of them can be chosen. The reduction rule is:

$n[in\ m.\ P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$

If successful, this reduction transforms a sibling n of an ambient m into a child of m. After the execution, the process *in m*. P continues with P, and both P and Q find themselves at a lower level in the tree of ambients.

Exit Capability

An exit capability, *out m*, can be used in the action:

out m.P

which instructs the ambient surrounding *out m*. P to exit its parent ambient named m. If the parent is not named m, the operation blocks until a time when such a parent exists. The reduction rule is:

$m[n[out\ m.\ P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$

If successful, this reduction transforms a child n of an ambient m into a sibling of m. After the execution, the process *in m*. P continues with P, and both P and Q find themselves at a higher level in the tree of ambients.

Open Capability

An opening capability, *open m*, can be used in the action:
open m. P
This action provides a way of dissolving the boundary of an ambient named m located at the same level as *open*, according to the rule:
$open\ m.\ P \mid m[Q] \rightarrow P \mid Q$
If no ambient m can be found, the operation blocks until a time when such an ambient exists. If more than one ambient m exists, any one of them can be chosen.
An *open* operation may be upsetting to both P and Q above. From the point of view of P, there is no telling in general what Q might do when unleashed. From the point of view of Q, its environment is being ripped open. Still, this operation is relatively well-behaved because: (1) the dissolution is initiated by the agent *open m*. P, so that the appearance of Q at the same level as P is not totally unexpected; (2) *open m* is a capability that is given out by m, so $m[Q]$ cannot be dissolved if it does not wish to be.

This is a general overview to the untyped (monadic) mobile ambient calculus that is able to express standard computational constructions such as channel-based communication, functions and agents. There is of course a special type system for ambient calculus that simply wraps the untyped computations with type information.

In order to perform Ambient calculus within Insertion Modeling system I believe it is quite enough to perform next implementations :
first all we must make the Model Driver of the system to understand the notation of Ambient as a bounded place around some abstract process.

M[P] should be interpreted as an ambient surrounding the process P.
The process here is understood as the highest abstraction level of entire Ambient Calculus, that represents any kind of computations also including ambients and other processes.
New Action Language
Action language defined within one of the main functions of Insertion Modeling – *unfold_rs* will have to include not only the simple actions .P but also three capability actions.

in m.P
out m.P
open m.P

The capabilities are bound to ambients, so the capability actions will be executed only on the ambient level. The input action is always shown as dissent action

(x).P

But I believe that it should be interpreted as one more capability action but for a constant ambient placed inside every other ambient that needs some input data.
The same thing is for the async. output action.
<M>
The only difference that there is no action to proceed after the execution of this capability also bound to the constant ambient I've spoke before.
This constant ambient (*Interactor)* needs no other capability actions as there is no way his own boundaries can be changed.
Insertion function
The second main function of the insertion Modeling System is ins – the insertion function itself, is about to define the simple action an variables of the Ambient Calculus System

Conclusions

Ambient calculus are believed to perform a new way for the whole mobile programming in general. Their algebra based upon the π-calculus represents a Turing complete algebra for defining both ways of mobility. The best tool for building a general framework for Ambient Calculus is the Insertion Modeling System - the technology of system design founded on the theory of interaction of agents and environments. This theory has been developed in. It is based on process algebra and is intended for the unification of different models of interaction and computation (such as CCS, CSP, π-calculus, mobile ambients etc.).

BIBLIOGRAPHIC REFERENCES

1. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.
2. A.Letichevsky. Algebra of behavior transformations and its applications, in V.B.Kudryavtsev and I.G.Rosenberg eds. Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry – Vol. 207, pp. 241-272, Springer 2005.
3. S. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert. Leveraging UML to Deliver Correct Telecom Applications. In L. Lavagno, G. Martin, and B.Selic, editors. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam, 2003.
4. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675.
5. J. Kapitonova, A. Letichevsky, V. Volkov, and T. Weigert. Validation of Embedded Systems. In R. Zurawski, editor. The Embedded Systems Handbook. CRC Press, Miami, 2005.
6. Abadi, M. and A.D. Gordon, **A calculus for cryptographic protocols: the spi calculus**. *Proc.of the Fourth ACM Conference on Computer and Communications Security,* 36-47*,* 1997.
7. Amadio, R.M., **An asynchronous model of locality, failure, and process mobility**. *Proc. COORDINATION 97*, Lecture Notes in Computer Science 1282, Springer Verlag. 1997.
8. Berry, G. and G. Boudol, **The chemical abstract machine**. *Theoretical Computer Science* **96**(1), 217-248, 1992.

9. Boudol, G., **Asynchrony and the p-calculus.** *Technical Report 1702, INRIA, Sophia-Antipolis,* 1992.
10. Cardelli, L., **A language with distributed scope**. *Computing Systems,* **8**(1), 27-59. MIT Press. 1995.
11. Cardelli, L., and A.D. Gordon, **Types for mobile ambients**. *Proc. 26th Annual ACM Symposiumon Principles of Programming Languages*, 79-92. 1999.
12. Carriero, N. and D. Gelernter, **Linda in context**. *Communications of the ACM*, **32**(4), 444-458, 1989.