An Implementation of Ambients in JoCaml *

(position paper for MOS'99)

Cédric Fournet
fournet@microsoft.com
Microsoft Research

Alan Schmitt
Alan.Schmitt@inria.fr
INRIA Rocquencourt

April 1999

1 Two notions of mobile computation

We relate two models of distributed mobile programming, the distributed join-calculus [6, 5] and the ambient calculus [3]. Technically, these two models have a lot in common. They consist of name-passing process calculi in the spirit of the π-calculus. They make explicit the spatial structure of the computation by distributing processes over a tree of nested locations (a.k.a. ambients) that stand for both machines and agents. They provide some mechanism to rearrange the location tree as part of the computation, thereby describing agent migration. Still, these two models address different aspects of wide-area distributed computations, and thus yield different interpretations of locality.

In the join-calculus, *locality is transparent*; it may affect performances, and even behavior in case of partial failure, but otherwise it does not affect communication and migration capabilities. Much as in a distributed object system, locations can communicate with one another as soon as they have access to names of their interfaces, independently of their respective positions in the location tree. To retain a realistic programming model, interaction between locations is restricted to the asynchronous sending of messages or sub-locations. Thus, location transparency can be implemented by some silent routing mechanism toward the machine that host the receiving location. Interaction is local, in the sense that synchronization always occurs within a single location, independently of its sub-locations. Overall, locations in the join-calculus are adequate for high-level programming with asynchronous messages and agents. Several distributed implementations are available, e.g., JoCaml [8].

In the ambient calculus, *locality and control are tightly connected*; each ambient acts as an opaque box, and interactions can occur only between processes that are in adjacent ambients. The routing of a process from one ambient to another is kept explicit; to accomplish the migration, the moving ambient must be aware of the path in the ambient tree; if an intermediate ambient decides to block the migration, or if the path evolves during the migration, ambients may get stuck or lost. Interaction is local, in the sense that any reduction involves processes separated by at most one ambient boundary, but the synchronization between these processes is rather complex. Overall, ambients are good at expressing administrative domains, highly dynamic environments, and controlled migration [3]. The ambient calculus

*This work is partly supported by the RNRT project MARVEL 98S0347

has been implemented in Java [1]. This centralized implementation grabs a lock for every ambient potentially involved in a reduction. While the ambient calculus typically describes wide-area networks, a distributed implementation is not obvious. For instance, a reduction may atomically involve several ambients located on different machines, and thus require some distributed synchronization scheme.

To investigate the relation between ambients and locations, we describe an implementation of the ambient calculus into the distributed join-calculus. Our implementation makes explicit the key synchronizations required to implement the operational semantics of ambients in a message-passing style. Taking advantage of the existing JoCaml distributed platform, our programming exercise in mobile computing also provides a lightweight, distributed implementation of ambients.

Our implementation proceeds in two stages; first a formal translation turns ambient expressions into processes of a distributed join-calculus (extended with non-linear join-patterns and with sets), then this translation gets refined into an actual JoCaml program. The syntax and the chemical semantics for the distributed join calculus and the ambient calculus are omitted; they can be found in [6] and in [3], respectively.

2 From ambients to the distributed join-calculus

Our translation maps every ambient to a location, and is compositional, so the source tree of nested ambients and the resulting tree of nested locations initially coincide. In order to emulate the dynamic properties of ambient contexts, the translation makes explicit the synchronization mechanisms of ambients. Once this is done, the correspondence between nested ambients and nested locations is easily maintained: since migration do not depend on localization in the join-calculus, it suffices to interleave some join-calculus *go* primitives with the translation of control.

To give an idea of our translation style, we focus on a simple fragment of the ambient calculus, given by the grammar

$$P ::= a[P] \quad | \quad \texttt{in}\, a.P \quad | \quad \texttt{out}\, a.P \quad | \quad P \mid P' \quad | \quad \texttt{0}$$

The translation appears in Figure 1. It assumes that the process being translated is placed within a nameless top-level ambient. It essentially relies on the creation of new ambient managers using the (meta) function *Ambient*. An ambient is mapped to a location containing the definitions of join-patterns for the names that represent the interface of this ambient. These names are packaged in different manners:

The interface e represents an ambient as seen from its sub-ambients; it is a record that contains fields *loc*, *amb*, *in*, and *out*. *loc* is the name of the location that hosts the ambient; *amb* is a name used for messages representing subambients (of the form $e.amb(i, a, e, r)$ where i is an identifier, a a name, e its own interface, and r a continuation channel for relocation purposes); *in* and *out* are names for messages representing migration requests, of the form $e.in(i, a, \kappa)$ and $e.out(i, a, \kappa)$ to match migration requests $\texttt{in}\, a$ and $\texttt{out}\, a$ issued by sub-ambients with identifier i, and with continuation κ for their guarded process.

The interface f represents an ambient environment for translating processes. It is a record that contains fields e, *in*, and *out*. e is the interface to the enclosing ambient; *in* and *out* are names for subjective migration requests for the ambient, of the form $in(a, \kappa)$ and $out(a, \kappa)$ where a is the target ambient name and κ is the continuation channel for their guarded process.

Internally, the name s carries the current state $s(i, e, l)$ of the ambient, that is, an identifier i, the interface to its parent e, and a log of running actions l; the log

$$
\begin{array}{rcl}
[\![\, a[\,P\,]\,]\!]_f & = & \mathit{Ambient}(a, f.e, P) \\
[\![\mathtt{in}\ a.P]\!]_f & = & \mathtt{def}\ \kappa() \triangleright [\![P]\!]_f\ \mathtt{in}\ f.in(a,\kappa) \\
[\![\mathtt{out}\ a.P]\!]_f & = & \mathtt{def}\ \kappa() \triangleright [\![P]\!]_f\ \mathtt{in}\ f.out(a,\kappa) \\
[\![P \mid Q]\!]_f & = & [\![P]\!]_f \mid [\![Q]\!]_f \\
[\![\mathtt{0}]\!]_f & = & \mathtt{0}
\end{array}
$$

where $\mathit{Ambient}(a, e_0, P)$ is

$$
\begin{array}{l}
\mathtt{loc}\ \mathit{here}\ \mathtt{init} \\
\mathtt{def}\ s(i,e,l) \mid in(b,\kappa)\ \triangleright s(i,e,l \cup \{\mathit{In}\ b\ \kappa\}) \mid e.in(i,b,\kappa) \\
\mathtt{and}\ s(i,e,l) \mid out(b,\kappa) \triangleright s(i,e,l \cup \{\mathit{Out}\ b\ \kappa\}) \mid e.out(i,b,\kappa) \\
\mathtt{and}\ s(i,e,l) \mid amb(j,b,e_b,r_b) \mid amb(k,c,e_c,r_c) \mid sub_{in}(k,b,\kappa) \\
\quad \triangleright s(i,e,l) \mid amb(j,b,e_b,r_b) \mid r_c(e_b,\kappa) \\
\mathtt{and}\ s(i,e,l) \mid amb(j,b,e_b,r_b) \mid sub_{out}(j,a,\kappa) \triangleright s(i,e,l) \mid r_b(e,\kappa) \\
\mathtt{and}\ s(i,e,l) \mid reloc(e',\kappa) \\
\quad \triangleright go(e'.here); \mathtt{let}\ i' = uid()\ \mathtt{in}\ \mathtt{let}\ l' = \{\mathit{Act}\ d\ \kappa' \in l \mid \kappa' \neq \kappa\}\ \mathtt{in} \\
\qquad s(i',e',\emptyset) \mid \kappa() \\
\qquad \mid e'.amb(i',a,\{loc = here, amb = amb, in = sub_{in}, out = sub_{out}\}, reloc) \\
\qquad \mid \mathit{Flush}(l', in, out)\ \ \mathtt{in} \\
\quad \mathtt{let}\ i = uid()\ \mathtt{in}\ s(i,e_0,\emptyset) \\
\mid \mathtt{let}\ e = \{loc = here, amb = amb, in = sub_{in}, out = sub_{out}\}\ \mathtt{in}\ e_0.amb(i,a,e,reloc) \\
\mid \mathtt{let}\ f = \{e = e, in = in, out = out\}\ \mathtt{in}\ [\![P]\!]_f\ \ \mathtt{end\_loc}
\end{array}
$$

Figure 1: Translation from in/out ambient processes to the distributed join-calculus

is a set of entries of the form $\mathit{In}\ b\ \kappa$ or $\mathit{Out}\ b\ \kappa$. Finally, the name *reloc* is used for executing the migration: $reloc(e, \kappa)$ triggers the relocation of the current ambient towards a new parent ambient with interface e.

Next, we describe the dynamics of an action "$\mathtt{in}\ b$" within an ambient named a, e.g., in the translation of $c[\ a[\,\mathtt{in}\ b.Q\ ]\ \mid\ b[\,0\,]\,]$. (The dynamics of "$\mathtt{out}\ b$" is similar but simpler.) First, a continuation κ is allocated for Q, and a message $in(b, \kappa)$ is issued in a. Using the first rule of a's *Ambient*, the request is added to a's log as an entry $\mathit{In}\ b\ \kappa$ and the request is forwarded to the enclosing ambient c described by the interface e. One level up, the resulting message $sub_{in}(i, b, \kappa)$ may be consumed using the third rule of c's *Ambient*. The rule also requires that both the ambient that issued the request and another ambient with name b be actually present (both ambients are represented by *amb* messages). Due to concurrent actions, there is in general no guarantee that the ambient that issued the request is still present, so we check its presence by consuming an *amb* message for this ambient with the same identifier k as in the request. Once this message has been consumed, other actions competing for the same message are thus blocked. One level down, a message is sent to the *reloc* channel of a's *Ambient*, causing a join-calculus migration toward the location of the new enclosing ambient b, triggering the continuation κ, and re-emitting updated messages on s and *amb*. At this stage, l' is the log of other actions that were preempted by the chosen action. The process $\mathit{Flush}(l', in, out)$—whose definition has been omitted—emits a fresh message $in(d, \kappa)$ or $out(d, \kappa)$ in this ambient for every entry $\mathit{In}\ d\ \kappa$ or $\mathit{Out}\ d\ \kappa$ that appears in l'. Since a fresh identifier is picked at every move, stale requests cannot be mistaken for fresh ones, even if the ambient comes back. (These requests are discarded in the actual implementation.)

Dealing with other ambient constructs Ambient processes may create local names ($\nu a.P$), communicate values within every ambient ($\langle v \rangle$ and $(v).P$), and be replicated ($!P$). These constructs are easily implemented: the scope extrusion mechanism is the same in both calculi, while pi-calculus channels and replication are derived constructs in the join-calculus [5].

Ambient processes may also dynamically merge ambients ($\texttt{open}\ a.P$), and this action is harder to implement. Since locations boundaries never disappear, we lose the one-to-one mapping from ambients to locations, and we must distinguish two states for each *Ambient* location: either the ambient is still running, or it has been opened and henceforth messages sent to its interface are forwarded to the ambient that performed the open.

Implementation in JoCaml The implementation closely follows the formal translation for the whole ambient calculus. It yields a compact code—less than 500 lines for the interpreter, less than 40k in bytecode for the object files—and benefits from the overall efficiency of JoCaml. A preliminary version of the source code is available electronically [7].

Ongoing work We are studying the correctness of the full translation with regards to the operational semantics of the two calculi. Independently, the semantics of Ambients must be refined to reflect localization information, in the case several machines are involved. Another intriguing aspect is the translation of ambient types [4, 2].

Acknowledgments. This work benefits from discussions with Luca Cardelli, Fabrice Le Fessant, Jean-Jacques Lévy, and Luc Maranget.

References

[1] L. Cardelli. Ambit, 1997. Available from `http://www.luca.demon.co.uk/Ambit/Ambit.html`.

[2] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, LNCS, 1999.

[3] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, volume 1378 of *LNCS*, pages 140–155, 1998.

[4] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM, Jan. 1999.

[5] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998. INRIA, TU-0556.

[6] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*, pages 406–421, Aug. 1996.

[7] C. Fournet and A. Schmitt. An implementation of Ambients in JoCAML, preliminary version. Available from `http://join.inria.fr/ambients.html`, 1999.

[8] F. Le Fessant. The JoCAML system prototype. Software and documentation available from `http://join.inria.fr/jocaml`, 1998.