A Calculus for Context-Awareness

Pascal Zimmer

BRICS - University of Aarhus, Denmark
pzimmer@daimi.au.dk

Abstract. In order to answer the challenge of pervasive computing, we propose a new process calculus, whose aim is to describe dynamic systems composed of agents able to move and react differently depending on their location. This *Context-Aware Calculus* features a hierarchical structure similar to mobile ambients, and a generic multi-agent synchronization mechanism, inspired from the join-calculus. After general ideas and introduction, we review the full calculus' syntax and semantics, as well as some motivating examples, study its expressiveness, and show how the notion of computation itself can be made context-dependent.

1 Introduction

The current world of desk personal workstations and laptops is rapidly evolving towards a more ubiquitous and pervasive one, in which computation is performed in multiple, embedded, mobile and often invisible small devices, all interconnected through a wireless network. Those devices can be very different in nature, use very different technologies and notions of computation, but should also be able to interact in a uniform way.

In order to formally describe such systems, we propose a new process calculus based on *context-awareness* [MD01, SAW94]. This concept denotes the ability for agents to react differently depending on their current environment, which is a broad term to include all of their enclosing, surrounding and inner environments. For example, when attending a performance in a concert hall, it would be more appropriate for a cell phone to vibrate instead of beeping; in order to achieve such a behaviour, the cell phone should be aware both of its location and of the concert going on (example borrowed from the introduction of [MD01]).

While many works tried to describe the concept of context-awareness and related issues, only recently did computer scientists attempt to formalize this notion into a well-founded process calculus [RJP04, Hen04]. Our *Context-Aware Calculus* (CAC in short) is one of those proposals and relies on two essential features:

a notion of location: Each location, or *agent*, represents either a physical or logical unit of computation and can run many processes. To be more generic, we will retain the hierarchical structure of locations already used in *mobile ambients* [CG98]. In this model, agents can also contain subagents, and some specific commands that allow them to move around the structure, either inside a sibling agent or outside the enclosing one.

a multi-agent synchronization: This is the essential point for context-awareness in CAC. Agents will not be directly aware of their environment, but will instead inform their environment of their current *capabilities* through the sending of asynchronous *atoms*. It is then the duty of the enclosing environment to provide rules to capture those atoms, possibly from different sources, and perform a global synchronization.
The *join-calculus* [FG96, FGL$^+$96] already proposes such a mechanism, but only on a local level, so that synchronization can only occur between different processes in a single location (in the case of the distributed join-calculus). Instead, we plan to achieve synchronization across agent boundaries.

Outline In the next Section, we start with a general presentation of the ideas behind this calculus based on a few simple examples. In Section 3, we review the full calculus by giving its syntax and semantics. Then, in Section 4, we show how to use this model to encode more complicate systems of agents requiring context-awareness. In Section 5, the expressive power of CAC is demonstrated by encoding π- and λ-calculi into it; moreover, we give an encoding of λ-calculus for which the computation itself is made dependent on the context. Finally, in Section 6, we make a few remarks, about the way CAC might be implemented and the detection of absence.

2 General Presentation and First Examples

In this Section, we will focus on the communication mechanism used in CAC, based on a few examples. Its other capabilities, namely movement and restriction of names, will be reviewed only in the next Section, as they are more conventional.

Single-Agent Synchronization

Let us consider an agent, willing to print some document:

$$P = app()[\ print\langle letter\rangle\ ]$$

Such an agent is represented with a bounded place named *app*, which contains a single process or "*atom*" $print\langle letter\rangle$, representing an asynchronous (and possibly polyadic) output of some value *letter* on a channel *print*. People familiar with the literature on process calculi can think of $app[\ldots]$ as an ambient [CG98], while $print\langle letter\rangle$ has the same meaning as in the join-calculus [FG96]. The empty parentheses in P indicate that this agent has no local definitions.

Now, let us consider agent P running on the following computer a:

$$a(print\langle x\rangle \triangleright send\langle x, laser\_printer\rangle)[\ \ldots\ \mid P \mid\ \ldots\ ]$$

Such a computer is again modelled by a bounded place a, containing P among its processes. This computer also provides a local definition in the form of a rewriting rule or *pattern*:

$$print\langle x\rangle \triangleright send\langle x, laser\_printer\rangle$$

In such a situation, the request of P to print can be accepted, and the document sent to the local laser printer. This is modelled by the following reduction:

$$\begin{aligned}
&a(print\langle x\rangle \triangleright send\langle x, laser_printer\rangle)[\ app()[print\langle letter\rangle]\ ] \\
\rightarrow\ &a(print\langle x\rangle \triangleright send\langle x, laser_printer\rangle)[\ app()[send\langle letter, laser_printer\rangle]\ ]
\end{aligned}$$

Note that the command *send* has replaced command *print* **in the requesting agent**; this will be a general rule. Moreover, note that the command *print* was visible not only in *app*, but also in its enclosing agent a. This is where we depart from the distributed join-calculus [FGL$^+$96], where rewriting can only occur in one single location. This mechanism will also allow us to perform a synchronization between different agents in different sub-locations, as we shall see below.

Suppose now that our agent P has first moved to the following site b:

$$b(print\langle x\rangle \triangleright send\langle x, color\_printer\rangle)[\ \ldots\ \mid P \mid\ \ldots\ ]$$

Here the printing will be done on another printer; in other words we have modelled a form of *context-dependency* for the access to some resources.

If our agent moves to site c:

$$c()[\ \ldots\ \mid P \mid\ \ldots\ ]$$

no printing will ever occur because agent c does not propose this capability. We have modelled some form of *availability* of resources.

We can even define some site d, where it is forbidden to print, and any agent attempting to do so will be reported to the administrator:

$$d(print\langle x\rangle \triangleright mail\langle root, \text{“Access violation''}\rangle)[\ \ldots\ \mid P \mid\ \ldots\ ]$$

Multi-agent Synchronization

Now that we have reviewed the basic structure of terms, let us consider a more complicate example where two different agents a and b synchronize on names x and y:

$$\begin{aligned}
&c(x\langle\rangle \parallel y\langle\rangle \triangleright P \parallel Q)\ [\ a()[x\langle\rangle] \mid b()[y\langle\rangle]\ ] \\
\rightarrow\ &c(x\langle\rangle \parallel y\langle\rangle \triangleright P \parallel Q)\ [\ a()[P] \mid b()[Q]\ ]
\end{aligned}$$

Note that each atom $x\langle\rangle$ and $y\langle\rangle$ is replaced by the corresponding process in the matching rule (in other words, the order of atoms and processes in a rule is significant). Intuitively, the meaning of such a rule

$$x\langle\rangle \parallel y\langle\rangle\ \triangleright\ P \parallel Q$$

is that, whenever we have atoms on x and y active in subagents, the pattern might be triggered, and atoms replaced by P and Q respectively.

In this way, we have designed a form of *context-awareness*, since agents a and b receive the information that there is a corresponding atom somewhere around willing to interact (their atoms would never be consumed if they were alone), while they actually never have to get in touch directly. On the other hand, agent c defines the scope where such a pattern can be activated, provides a local logical unit of computation, and takes care of the synchronization.

In the previous example, no actual value was transmitted along the channels for simplicity. Of course, we may want to output some value as before; the pattern rule should now look like:

$$x\langle z\rangle \mid\mid y\langle t\rangle \ \triangleright\ P \mid\mid Q$$

where z and t are variables that are bound in *both* P and Q. The corresponding reduction step will now look like:

$$\begin{aligned} & c(x\langle z\rangle \mid\mid y\langle t\rangle \triangleright P \mid\mid Q)\ [\ a()[x\langle u\rangle] \mid b()[y\langle v\rangle]\ ] \\ \rightarrow\ & c(x\langle z\rangle \mid\mid y\langle t\rangle \triangleright P \mid\mid Q)\ [\ a()[P\sigma] \mid b()[Q\sigma]\ ] \end{aligned}$$

where $\sigma = \{^u/_z\ ,\ ^v/_t\}$. Note that agents a and b can now exchange the information represented by u and/or v if z appears free in Q and/or t appears free in P.

When multiple reductions are possible, there is a notion of priority. Intuitively, the deepest rule that matches is activated first. For example, in the following process:

$$c(x\langle\rangle \mid\mid y\langle\rangle \triangleright P \mid\mid Q)\ [\ \ a(x\langle\rangle \triangleright R)[x\langle\rangle]\ \ \mid\ \ b()[y\langle\rangle]\ \ ]$$

the rule on x and y cannot be activated, because the pattern on x in a matches first[1]. Of course, when multiple equivalent reductions are possible, as in:

$$a(x\langle z\rangle \triangleright R)[\ x\langle u\rangle \mid x\langle v\rangle\ ]$$

they all have the same priority and fair indeterminism should be used, as in many process calculi.

3 A Review of CAC

3.1 Syntax

We assume an infinite set of *names* $a, b, \ldots, x, y, z, \ldots$. We write $\tilde{x}$ for sequences of names, and $|\tilde{x}|$ for the arity of such a sequence.

The complete syntax of CAC is given in Fig. 1.

[1] Without this notion of priority, the semantics of Section 3 would be much easier to write. However, it would also lead to *grave interferences* in communications between agents, to a less efficient implementation, and to less control on concurrency and the scope of patterns.

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} & \text{nil process} \\
 & | & P \mid P' & \text{parallel composition} \\
 & | & (\nu x)P & \text{restriction of name} \\
 & | & x\langle\tilde{v}\rangle & \text{atom} \\
 & | & a(D)[P] & \text{agent} \\
 & | & def\ D\ in\ P & \text{new definitions} \\
 & | & go(*, P) & \text{movement}
\end{array}
\qquad
\begin{array}{rcll}
* & ::= & \uparrow & \text{move out} \\
 & | & a & \text{move in } a \\
 & & & \\
D & ::= & J_1, \ldots, J_k & \text{definitions}
\end{array}
$$

$$
J \ ::= \ x_1\langle\tilde{y_1}\rangle \ || \ \ldots \ || \ x_n\langle\tilde{y_n}\rangle \ \ \triangleright_{\tilde{z}} \ \ P_1 \ || \ \ldots \ || \ P_n \qquad \text{pattern}
$$

Fig. 1. Full syntax of CAC

The nil process $\mathbf{0}$, parallel composition and restriction operator (νx) have the same meaning as in π-calculus [2] [MPW92]. In $(\nu x)P$, the name x is bound in P and may be α-converted if needed. The definition of the set of *free names* $fn(P)$ of a process P is straightforward and left to the reader. We also write $(\nu\tilde{x})$ for a sequence of restrictions on many names. As in join-calculus, there is no need for replication, since pattern rules may be used to encode replicated servers.

The atom $x\langle\tilde{v}\rangle$ was already presented extensively in the previous Section, and is an asynchronous output on channel x of a tuple of values $\tilde{v}$.

The construction $a(D)[P]$ represents an *agent*, with contents process P and active *definitions* D. The construction $def\ D\ in\ P$ is a process whose definitions D are not yet activated, but will be added to the set of active definitions of the enclosing agent at some point.

Definitions are unordered sets of *rules* $J_1, \ldots, J_k$, and each rule has the following shape:

$$
J ::= \ \ x_1\langle\tilde{y_1}\rangle \ || \ \ldots \ || \ x_n\langle\tilde{y_n}\rangle \ \ \triangleright_{\tilde{z}} \ \ P_1 \ || \ \ldots \ || \ P_n
$$

where the names $\tilde{y_1}, \ldots, \tilde{y_n}, \tilde{z}$ are all distinct, bound in $P_1, \ldots, P_n$, and can be α-converted when needed. The meaning of this rule is to create some join-calculus style pattern-matching rewriting relation: whenever all atoms $x_i\langle\tilde{v_i}\rangle$ are present in the process, variables $\tilde{y_i}$ will be bound to $\tilde{v_i}$, fresh names $\tilde{z}$ will be created (this is required for some examples where we need to create fresh nonces for every activation of a pattern rule), and process P_i will replace atom $x_i\langle\tilde{v_i}\rangle$. Note

[2] One may remark here that we use the same set for names of agents and names of channels (and also for variables). This is an arbitrary choice, as those two sets are unrelated and can in no way interact. Our motivation was to simplify the grammar, since otherwise we should have distinguished different constructs for restrictions on agent/output names, for variables, etc... The only strange consequence is that in:

$$
(\nu a)(\ a(a\langle x\rangle \triangleright P)[\ a\langle v\rangle \mid go(a, Q)\ ]\ )
$$

all four occurrences of a are bound. Such a case will never occur in our examples.

that there must be the same number n of components on both sides of the rewriting rule. Note also that the order of processes is important (i.e. $||$ is *not* commutative), because they correspond one-by-one to atoms.

The primitive $go(*, P)$ represents a command allowing the enclosing agent to perform a move. It can either move outside the parent agent ($go(\uparrow, P)$), or inside a sibling agent with name a ($go(a, P)$). In both cases, the process P is the continuation to be executed after the move has been performed. In order to simplify notations in the examples, we extend the syntax with *paths*, which are sequences of directions to follow:

$$M ::= \varepsilon \mid * .M$$

Then, the primitive go is extended as follows:

$$\begin{aligned}&go(\varepsilon, P) = P\\&go(*.M, P) = go(*, go(M, P))\end{aligned}$$

The two forms of go correspond respectively to the *in* and *out* primitives in mobile ambients. However, as in boxed ambients [BCC01], there is no way to open an agent and reveal its contents. We argue that in our setting this would be an unsafe feature, both for the opened agent and for the opening one. Agents can come from anywhere so we cannot fully trust their contents; it is not difficult to construct a malicious agent that would be able to entrap any agent with *open* behind a restriction. Moreover, we want agents to keep a safe inner computation place, and opening an agent is not required to send messages to the upper level as in mobile ambients, as the communication mechanism of CAC provides a much more general framework to handle such a case. Also, even if no opening of agent can take place, we believe the following relation should be true for any sensible notion of equivalence:

$$(\nu a)a(D)[\mathbf{0}] \;\simeq\; \mathbf{0}$$

such that useless empty agents whose name does not appear anywhere else can be garbage-collected.

3.2 Semantics

The semantics of CAC is defined in chemical style through a reduction relation $\rightarrow$. All its rules are given in Fig. 2, and we will detail them in this Section.

The first two axioms (Mv Out) and (Mv In) define the semantics of the go primitive. They are very similar to the primitives *out* and *in* of mobile ambients. In the first case, an agent named b enclosed in a and with an active process $go(\uparrow, P)$ moves out of a (note that we depart from mobile ambients where we need to provide the name of the parent ambient when going out). In the second case, an agent a moves into its sibling agent b by consuming its active process $go(b, P)$.

The axiom (Def) handles the case when new definitions need to be activated; they are simply added to the set of local definitions of the enclosing agent.

$$\frac{}{a(D)[\ b(D')[\ go(\uparrow, P) \mid Q\ ] \mid R\ ] \quad \rightarrow \quad a(D)[\ R\ ] \mid b(D')[\ P \mid Q\ ]} \text{ (Mv Out)}$$

$$\frac{}{a(D)[\ go(b, P) \mid Q\ ] \mid b(D')[\ R\ ] \quad \rightarrow \quad b(D')[\ a(D)[\ P \mid Q\ ] \mid R\ ]} \text{ (Mv In)}$$

$$\frac{}{a(D)[\ def\ D'\ in\ P \mid Q\ ] \quad \rightarrow \quad a(D, D')[\ P \mid Q\ ]} \text{ (Def)}$$

$$\frac{\begin{array}{ll} Q = \mathbf{C}[\ x_1\langle\tilde{v_1}\rangle, \ldots, x_n\langle\tilde{v_n}\rangle\ ] & Q \text{ reduction-free for } \{x_1, \ldots, x_n\} \\ J = x_1\langle\tilde{y_1}\rangle || \ldots || x_n\langle\tilde{y_n}\rangle \ \triangleright_{\tilde{z}}\ P_1 || \ldots || P_n & |\tilde{v_i}| = |\tilde{y_i}| \text{ for } 1 \leq i \leq n \\ \sigma = \{^{\tilde{v_i}}/_{\tilde{y_i}}\}_{1 \leq i \leq n} & \tilde{z} \cap fn(\mathbf{C}) = \emptyset \end{array}}{a(J, J_1, \ldots, J_k)[\ Q\ ] \quad \rightarrow \quad a(J, J_1, \ldots, J_k)[\ (\nu\tilde{z})\ \mathbf{C}[\ P_1\sigma, \ldots, P_n\sigma\ ]\ ]} \text{ (React)}$$

$$\frac{P \equiv Q \qquad Q \rightarrow Q' \qquad Q' \equiv P'}{P \rightarrow P'} \text{ (Struct)}$$

$$\frac{P \rightarrow Q}{\mathbf{E}[P] \rightarrow \mathbf{E}[Q]} \text{ (Ev Cont)}$$

Fig. 2. Semantics of CAC

The last axiom (React) defines the main communication mechanism of CAC. Its definition requires some care in order to ensure the priority between patterns that we have seen before is respected. First of all, we need to define in which places active atoms can appear in subterms; this is captured by the notion of *contexts* with *holes* [ ], whose definition is given in Fig. 3. Basically, a context $\mathbf{C}$ is a process term with some number of holes that can appear only in active positions (i.e. not in the continuation of *go* and *def* primitives).

We define free and bound names for contexts as for processes. In particular, $bn(\mathbf{C})$ denotes the set of bound names of $\mathbf{C}$ (e.g. x is bound in $(\nu x)\ \mathbf{C}$), and those names can be α-converted.

Note that a context $\mathbf{C}$ can have any number of holes. If $\mathbf{C}$ has exactly n holes, and if the holes have been ordered and numbered from 1 to n (not necessarily in the order in which they appear in $\mathbf{C}$!), then we write $\mathbf{C}[P_1, \ldots, P_n]$ for the context $\mathbf{C}$ where the n holes have been respectively replaced by processes P_i, provided that $fn(P_i) \cap bn(\mathbf{C}) = \emptyset$ for any i, i.e. provided that $\mathbf{C}$ does not capture free names in P_i (this condition can always be satisfied by renaming bound names in $\mathbf{C}$ first). It is not difficult to check that the result is a valid process.

$$
\begin{array}{llcllll}
\mathbf{C} ::= & [\,] & \text{contexts} & \qquad & \mathbf{E} ::= & [\,] & \text{evaluation contexts} \\
| & \mathbf{0} & & & | & \mathbf{E} \mid P & \\
| & \mathbf{C} \mid \mathbf{C}' & & & | & (\nu x)\ \mathbf{E} & \\
| & (\nu x)\ \mathbf{C} & & & | & a(D)[\mathbf{E}] & \\
| & x\langle \tilde{v} \rangle & & & & & \\
| & a(D)[\mathbf{C}] & & & & & \\
| & def\ D\ in\ P & & & & & \\
| & go(*, P) & & & & &
\end{array}
$$

Fig. 3. Contexts and Evaluation Contexts

We can now detail the main reaction rule:

$$a(J, J_1, \ldots, J_k)[\ Q\ ] \quad \rightarrow \quad a(J, J_1, \ldots, J_k)[\ (\nu \tilde{z})\ \mathbf{C}[\ P_1\sigma, \ldots, P_n\sigma\ ]\ ]$$

where $Q = \mathbf{C}[\ x_1\langle \tilde{v_1} \rangle, \ldots, x_n\langle \tilde{v_n} \rangle\ ]$, σ is the substitution $\{^{\tilde{v_i}}/_{\tilde{y_i}}\}_{1 \leq i \leq n}$, and

$$J = x_1\langle \tilde{y_1} \rangle \ ||\ \ldots\ ||\ x_n\langle \tilde{y_n} \rangle \quad \triangleright_{\tilde{z}} \quad P_1 \ ||\ \ldots\ ||\ P_n$$

In other words, if atoms $x_i\langle \tilde{v_i} \rangle$ appear in subprocesses while their names x_i are not bound, and if they match a rewriting rule J, then fresh names $\tilde{z}$ are created, and each atom $x_i\langle \tilde{v_i} \rangle$ is replaced by process P_i, where variable substitution has occurred (we remind that all names $\tilde{y_i}$ must be distinct in J) [3].

We have not detailed yet what "reduction-free" means for Q. Intuitively, it is quite simple: there should not be a possible reduction in Q that would involve only a subset of the atoms $x_1\langle \tilde{v_i} \rangle, \ldots, x_n\langle \tilde{v_n} \rangle$. The precise definition is a bit more tricky, and will require some further technical definitions.

We define $msg(P)$ as the multiset of names for which P has some active atoms. In other words, $msg(x\langle \tilde{v} \rangle) = \{x\}$, and the other cases are defined inductively as for free names (erasing restricted names). Moreover, for a pattern $J = x_1\langle \tilde{y_1} \rangle || \ldots || x_n\langle \tilde{y_n} \rangle \ \triangleright_{\tilde{z}}\ P_1 || \ldots || P_n$, we define the multiset of *pattern names* as $pn(J) = \{x_1, \ldots, x_n\}$.

Finally, we can define the reduction-freedom of some process Q for some multiset S as follows. The base case is when $Q = a(J_1, \ldots, J_k)[P]$. We say that Q is reduction-free for S if the following two conditions are satisfied:

- P is reduction-free for S
- $(pn(J_i) \subseteq msg(P)) \Rightarrow (pn(J_i) \cap S = \emptyset)$ for $1 \leq i \leq k$ (i.e. if some pattern J_i can be triggered at this point, it is completely independent of any pattern on S)

The other cases are trivially defined by induction on Q.

[3] If arities do not match for any of the atom x_i, no reduction can take place. This might be statically checked using a type system with sorts like in π-calculus.

Structural congruence $\equiv$ is defined as usual for the reordering of subterms, and for scope-extrusion. Its complete definition is omitted. It is the least congruence on processes that is a commutative monoid for $(\mathbf{0}, |)$, that can commute patterns in definitions, and that respects the two following scope-extrusion rules:

$$
\begin{array}{lll}
P \mid (\nu x)Q \;\equiv\; (\nu x)(P \mid Q) & & \text{if } x \notin fn(P) \\
a(D)[\ (\nu x)P\ ] \;\equiv\; (\nu x)a(D)[P] & & \text{if } x \neq a \text{ and } x \notin fn(D)
\end{array}
$$

The reduction rule (Struct) captures the fact that such a reordering of terms can take place at any time.

Finally, we define *evaluation contexts* **E** in Fig. 3, together with the corresponding reduction rule (Ev Cont). In other words, reduction can take place anywhere in the term, except in the continuations of go and def.

4 Further Examples

Different implementations in different places In Section 2, we have seen an agent willing to print a document whose final behaviour would depend on the local printing capabilities and authorizations.

In the same way, we can express in CAC that a same "function call" can be implemented differently, depending on what resources/computational power are locally available. It is a trivial observation that neither all machines have the same computing power, nor do they run the same operating system, nor do they have the same libraries available. In such a diverse world, we would however like to be able to access those resources in a uniform way, while keeping the actual implementation context-dependent. This is easy in CAC: for example, the two following locations provide a test function for prime numbers, but the latter uses a precomputed table while the former implements a simpler but slower sieve algorithm:

$$a(prime\langle n\rangle \triangleright do\_sieve\langle n\rangle)[\ldots]$$

$$b(prime\langle n\rangle \triangleright lookup\_prime\_table\langle n\rangle)[\ldots]$$

Remote Procedure Call (RPC) Sometimes, it happens that the requested function cannot be computed locally. Instead an agent is sent somewhere else where the actual computation takes place, and the result is forwarded.

We first define the requesting agent. Since the result will not be available immediately, and since we do not know the exact location of the calling agent, we use a general mechanism of continuation which will be called with the result when available:

$$B = b()[\ (\nu k)\ def\ k\langle result\rangle \triangleright P\ in\ prime\langle n, k\rangle\ ]$$

Using continuation-passing style to get the result of a computation is a quite frequent requirement when dealing with more complex systems, and it is quite standard when programming in π-calculus or join-calculus.

$$A = a(\ prime\langle n,k\rangle \ || \ local\langle\rangle \ \triangleright_{k'} \ k'\langle\rangle \ || \ (Q \mid local\langle\rangle)\ )\ [\ local\langle\rangle \mid B\ ]$$

In the enclosing agent, the request on *prime* is matched with a local atom; a fresh name k' is created which acts as a dummy continuation in the requesting agent (i.e. we keep some way to be called later from inside that agent). Locally, process Q is triggered, which will perform the remote procedure call (and another atom *local* is released so that another computation can take place concurrently):

$$Q = (\nu k'')\ def\ k'\langle\rangle \ || \ k''\langle x\rangle \triangleright k\langle x\rangle \ || \ \mathbf{0}\ in\ R$$

We create a fresh name k'' on which we intend to receive the result of that RPC call. When it is available, we match it against our remote dummy continuation k' waiting in b, which is then replaced by the "real" continuation $k\langle x\rangle$ that contains the final result.

The remote call is implemented with a new agent named p, which goes out of a into some server, where the computation will take place.

$$R = (\nu p)\ p()[\ go(\uparrow.server, S)\ ]$$

There, we call $prime\langle n, k'''\rangle$ with a fresh continuation k''' to get the result. When we receive it, we go out of the sever back inside a, and trigger the continuation k'' with the actual result.

$$S = (\nu k''')\ def\ k'''\langle x\rangle \triangleright go(\uparrow.a,\ k''\langle x\rangle)\ in\ prime\langle n,k'''\rangle$$

The reader can check that the full system:

$$A \mid server(prime\langle n,k\rangle \triangleright k\langle ...\rangle)[\mathbf{0}]$$

where ... denotes the result of the computation on n, will effectively trigger process P in agent b (we do not detail this process as it now contains useless pattern rules for continuations in a and b; those rules can however be safely garbage-collected).

Packet routing Let us consider two sites a and b. Site a contains data from an application that need to be sent to b. For some reason, they are not allowed to travel directly, but should be enclosed into an IP packet first. The following system provides an encoding for such a situation: data first goes into a packet, tells it its destination, packet moves along the network and releases the data agent when arrived (this protocol is quite close to the taxi protocol for ambient presented in [TZH02]).

$$\begin{aligned}
P &= (\nu packet)\ packet(J)\ [\ packet_ready\langle packet\rangle \mid wait_dest\langle\rangle\ ] \\
J &= move_to\langle y\rangle \ || \ wait_dest\langle\rangle \ \triangleright \\
&\qquad\qquad \mathbf{0} \ || \ go(\uparrow.y,\ def\ is_arrived\langle k\rangle \ \triangleright\ go(\uparrow, k\langle\rangle)\ in\ \mathbf{0}) \\
Q &= data()[\ data_ready\langle\rangle \mid move_to\langle b\rangle \mid is_arrived\langle k\rangle\ ] \\
S &= a(packet_ready\langle x\rangle \ || \ data_ready\langle\rangle \ \triangleright\ \mathbf{0} \ || \ go(x,\mathbf{0}))[\ P \mid Q\ ] \ \mid \ b(\ldots)[\ldots]
\end{aligned}$$

First of all, data and packet agents synchronize by sending atoms $data\_ready\langle\rangle$ and $packet\_ready\langle packet\rangle$ (the pattern rule in agent a is triggered). The packet agent sends its name (which is private), and it is given to the data agent which use it to go inside the packet with a go.

Then, the two agents synchronize again (but this time inside the packet) with atoms $move\_to\langle b\rangle$ and $wait\_dest\langle\rangle$ (pattern rule J). The destination name b is communicated to the packet, and it uses it by going first out of a, then into b.

Afterwards, the new definition $def\ \ldots\ in\ \ldots$ gets activated. This means now that the atom $is\_arrived\langle k\rangle$ in the data agent is consumed and replaced with $go(\uparrow, k\langle\rangle)$, consequently the data agent will first go out of the packet agent, and then the continuation k will be called for further computation.

We invite the reader to check that:

$$S \quad \rightarrow^* \quad a(\ldots)[\mathbf{0}] \ \mid\ b(\ldots)[\ (\nu packet)packet(J)[\mathbf{0}] \ \mid\ data()[k\langle\rangle] \ \mid\ \ldots\ ]$$

5 Expressiveness of CAC

Encoding $\boldsymbol{\pi}$-calculus In order to show that CAC is fully expressive, we will first give an encoding of the π-calculus [MPW92]. To be more precise, and for a matter of convenience, we will encode only a subfragment of the full calculus, namely its monadic version with asynchronous output, replicated input and no matching. It is widely known that this does not affect its expressive power [HT91, Bou92].

The full grammar of this fragment is given in Fig. 4, as well as a simple encoding into CAC. Outputs in π and atoms in CAC are in fact the same object, so their encoding is direct. Replicated inputs $!x(v).P$ actually behave as servers that trigger a copy of P each time they are called. This can be easily emulated with a unary pattern rule $J = x\langle v\rangle \triangleright P$, so that replicated inputs are simply encoded via the corresponding process $def\ J\ in\ \mathbf{0}$. Non-replicated inputs require some care, as we do not want the corresponding pattern rule to be triggered more than once. The workaround is quite simple: we create a unique atom with a fresh name k, that can be consumed only once together with the atom on x. In other words, the encoding of $x(v).P$ is:

$$(\nu k)\ (def\ x\langle v\rangle \ ||\ k\langle\rangle \triangleright [\![P]\!]_\pi \ ||\ \mathbf{0}\ in\ k\langle\rangle)$$

The encoding is trivial for all other constructs. Finally, a π-calculus process P is simulated by the following CAC process: $world()[\ [\![P]\!]_\pi\ ]$, as we need at least one enclosing agent to store the definitions created by inputs. Note that we actually need only one agent for the encoding, and that the hierarchical structure of CAC is not useful.

Encoding $\boldsymbol{\lambda}$-calculus Milner [Mil92] gave encodings for the λ-calculus into the π-calculus. For reasons that will be made clear, we choose a slight reformulation of the general encodings given by Sangiorgi [San98] for both the call-by-value and

Grammar for the monadic asynchronous π-calculus with replicated input:

$$P ::= \mathbf{0} \mid P|P' \mid (\nu x)P \mid \bar{x}\langle v\rangle \mid x(v).P \mid !x(v).P$$

Encoding into CAC:

$[\![\mathbf{0}]\!]_\pi = \mathbf{0}$
$[\![P \mid P']\!]_\pi = [\![P]\!]_\pi \mid [\![P']\!]_\pi$
$[\![(\nu x)\ P]\!]_\pi = (\nu x)\ [\![P]\!]_\pi$
$[\![\bar{x}\langle v\rangle]\!]_\pi = x\langle v\rangle$
$[\![!x(v).P]\!]_\pi = def\ x\langle v\rangle \triangleright [\![P]\!]_\pi\ in\ \mathbf{0}$
$[\![x(v).P]\!]_\pi = (\nu k)\ (def\ x\langle v\rangle \mid\mid k\langle\rangle \triangleright [\![P]\!]_\pi \mid\mid \mathbf{0}\ in\ k\langle\rangle)$ with $k \neq x$ and $k \notin fn(P)$

Fig. 4. The π-calculus and its encoding in CAC

call-by-name λ-calculi. Those encodings are detailed in Fig. 5 and take a return channel p in parameter (we refer to [San98] for details). They provide a uniform encoding for functions and variables, while only the encoding of applications depends on the chosen reduction strategy (for call-by-value there exists a direct and simpler encoding).

Note that the encoding of a λ-calculus term is a process from the subfragment of π that we have introduced in the previous Section. As a consequence, we obtain an encoding of a λ-calculus term M into CAC by composing the two encodings: $[\![[\![M]\!]_p]\!]_\pi$ for any fresh name p (choosing either call-by-name or call-by-value).

Context-sensitive computation The last result is not that impressive and surprising. What is more interesting is that it allows us to turn the notion of computation dependent on local rules. What we would like to achieve is a uniform encoding of λ-calculus into CAC, such that the reduction strategy is chosen by the current environment. For this purpose, we turn the main reduction rule of λ-calculus (namely β-reduction) into a pattern rule, in such a way that different locations may provide different forms of β-reductions. More precisely, an application will be encoded as:

$$[\![MN]\!]_p = (\nu m, n)\ (def\ m\langle q\rangle \triangleright [\![M]\!]_q\ ,\ n\langle q\rangle \triangleright [\![N]\!]_q\ in\ appl\langle m, n, p\rangle)$$

In other words, the local agent requesting such an application MN provides two servers on fresh channels m and n, waiting to receive a return name q to compute M and N on q, and waits for an enclosing agent to trigger the actual β-reduction by calling $appl\langle m, n, p\rangle$ (the name *appl* must not be used for any other purpose than application, and m and n must not appear in M, N or p). Other λ-terms are encoded as before, that is via π-calculus (we give the direct encoding into CAC here):

$$[\![\lambda x M]\!]_p = (\nu v)\ (def\ v\langle x, r\rangle \triangleright [\![M]\!]_r\ in\ p\langle v\rangle)$$
$$[\![x]\!]_p = x\langle p\rangle$$

Grammar for the λ-calculus:

$$M ::= \ x \ \mid \ \lambda x M \ \mid \ M \ N$$

Encoding of functions and variables:

$[\![\lambda x M]\!]_p = (\nu v)\ (\bar{p}\langle v\rangle \mid !v(x,r).\ [\![M]\!]_r)$
$[\![x]\!]_p = \bar{x}\langle p\rangle$

Encoding of application for call-by-value λ-calculus:

$[\![MN]\!]_p^{cbv} = (\nu q)\ ([\![M]\!]_q \mid q(v).\ (\nu r)\ ([\![N]\!]_r \mid r(w).\ (\nu x)\ (\bar{v}\langle x,p\rangle \mid !x(r').\ \bar{r'}\langle w\rangle)))$

Encoding of application for call-by-name λ-calculus:

$[\![MN]\!]_p^{cbn} = (\nu q)\ ([\![M]\!]_q \mid q(v).\ (\nu x)\ (\bar{v}\langle x,p\rangle \mid !x(r).\ [\![N]\!]_r))$

Fig. 5. Encoding of λ-calculus into π-calculus

It is now the duty of the enclosing agent to take care of β-reduction through an appropriate pattern rule for *appl*. We can express such two possibilities very simply by reusing encodings for π-calculus:

$$J_{cbv} = appl\langle m,n,p\rangle \ \triangleright \ [\![[\![m\ n]\!]_p^{cbv}]\!]_\pi$$
$$J_{cbn} = appl\langle m,n,p\rangle \ \triangleright \ [\![[\![m\ n]\!]_p^{cbn}]\!]_\pi$$

As a consequence, the process $world(J_{cbv})[\ a()[\ [\![M]\!]_p\ ]\ ]$ will reduce M following a call-by-value strategy, while its counterpart $world(J_{cbn})[\ a()[\ [\![M]\!]_p\ ]\ ]$ would follow call-by-name.

We can even go further and enrich the λ-calculus with the movement primitives of CAC, allowing agent a to move around in the network. As a consequence, it may happen that the reduction strategy would change in the middle of the computation, depending on the β-pattern provided by the environment where a finds itself. We argue that this provides us with a basic model of agents able to move between physical locations that may use very different notions of computation.

6 Remarks

In this Section, we make a few remarks concerning implementation issues of CAC, and the detection of absence.

Implementation What about the efficiency of the semantics ? Since atoms have to be sent to all enclosing agents, it might be quite inefficient if they lie on different physical locations. Moreover, due to latency of communication, we may have interference problems between possible concurrent reductions and/or we may have to synchronize all the network, in particular when we try to enforce the priority between concurrent reductions.

For those reasons, we need some assumptions about the real layout of agents. The intended layout is to have at toplevel all the wide-area physical locations, while their subagents are all local to the same machine or at least the same local network:

$$site_a(\ldots)[\ldots] \mid site_b(\ldots)[\ldots] \mid \ldots$$

To be consistent, we can also enclose this process in a *world* agent with no definition and no atom:

$$world()[\ site_a(\ldots)[\ldots] \mid site_b(\ldots)[\ldots] \mid \ldots]$$

With such a layout, we are sure that every atom can be sent only locally, and there is no communication on the wide-area network, except for agent movement. This still means we need a synchronizing scheduler on every local physical location, taking care of all local agents.

We can also define a more general model, where it is not necessary that physical locations appear at toplevel. All we need to do is to syntactically distinguish those locations with a specific notation such as: $a(D)[|P|]$.

The semantics needs little changes: atoms are only sent locally, i.e. they cannot cross double-bracketed barriers. This is quite easy, by saying that $a(D)[|C|]$ is *not* a valid context, while $a(D)[|E|]$ is a valid evaluation context. Moreover, we define $msg(a(D)[|P|]) = \emptyset$. For what concerns movement, it is not yet clear if allowing those physical locations to move is relevant or not.

Detecting Absence With the current communication mechanism of CAC, it is easy to detect the presence of a subagent, but it is much more difficult to detect an absence. One possible way to achieve such a feature might be to add terms $\neg x$ in pattern rules, with the meaning that there should not be any atom on x available for the rule to match.

For example, in the following process:

$$a(x\langle\rangle \mid\mid \neg y \triangleright P\ ,\ x\langle\rangle \mid\mid \neg z \triangleright Q)\ [\ x\langle\rangle \mid y\langle\rangle\ ]$$

the former pattern cannot be activated, while the latter can be.

7 Conclusion and Future Work

This paper is a foundational work, whose aim is to trigger interest among the community for a process calculus modelling context-awareness. We have proposed a first step in the form of CAC, and shown how it can be used as a generic framework to model such different things as context-awareness for the access and availability of resources or libraries, for RPC or packet routing, for co-located multi-agent synchronization, and even context-awareness of computation itself.

The expressiveness of CAC still remains to be fully explored. Also, the specific and context-dependent multi-agent synchronization mechanism used in CAC raises new specific and challenging issues about its behavioural theory, as an equational theory for CAC would probably have to consider not only processes but also contexts. Finally, we also need ways for agents to trust and control the information provided by their inner agents.

Acknowledgements I would like to thank Mogens Nielsen for providing insightful suggestions and the opportunity to work on this subject.

References

[BCC01] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *LNCS*, pages 38–63. Springer-Verlag, 2001.

[Bou92] Gerard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, Rapport de Recherche INRIA Sophia Antipolis, 1992.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer Verlag, 1998.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, LNCS, pages 406–421. Springer-Verlag, 1996.

[Hen04] Matthew Hennessy. Context-awareness: Models and analysis, 2004. Course given at 2nd UK-UbiNet Workshop: Security, trust, privacy and theory for ubiquitous computing, Cambridge UK, May 2004.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *LNCS*. Springer-Verlag, 1991.

[MD01] Thomas P. Moran and Paul Dourish. Context-aware computing. *Special Issue of Human-Computer Interaction*, 16, 2001.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[RJP04] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A formal treatment of context-awareness (invited paper). In *Proceedings of FASE'04*, volume 2984 of *LNCS*. Springer-Verlag, 2004.

[San98] Davide Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, Rapport de Recherche INRIA Sophia Antipolis, 1998.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[TZH02] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using ambients to control resources. In *Proceedings of CONCUR 2002*, volume 2421 of *LNCS*, pages 288–303, 2002.