Splitting Mobility and Communication in Boxed Ambients

Pablo Garralda 1,2 Adriana Compagnoni 3

Computer Science Department
Stevens Institute of Technology
Hoboken, NJ - U.S.

Abstract

Stemming from our previous work on BACI, a boxed ambients calculus with communication interfaces, we define a new calculus that further enhances communication mechanisms and mobility control by introducing multiple communication ports, access control lists, and port hiding.

The development of the calculus is mainly focused on three objectives: separation of concerns between mobility and communication, fine-grained controls, and locality. Communication primitives use ports to establish communication channels between ambients while ambient names are only used for mobility. Port names are used in communications with children ambients as well as in communications with parent ambients, providing extra information to the communicating parties. The introduction of multiple ports allows for extra control in communications and a direct implementation of dedicated channels such as those used for ftp, ssh, or other services.

In order to control mobility, the calculus includes co-capabilities à la Safe Ambients, but with the addition of access control lists. These lists contain the names of the ambients that are allowed to enter (or exit) the ambient with that co-capability.

The resulting calculus not only provides more flexibility and expressiveness than BACI, but also enables simpler implementations using more powerful constructs for communication and mobility. We establish the basic meta-theory of the calculus by proving a subject reduction result.

Key words: AMBIENTS, MOBILITY, COMPUTING MODELS

1 This research was supported by the National Science Foundation project No. CCR-0220286 ITR: Secure Electronic Transactions. The funding bodies are not responsible for any use that might be made of the results presented here.
2 Email: pablo@cs.stevens.edu
3 Email: abc@cs.stevens.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
1 Introduction

In Cardelli and Gordon’s Mobile Ambients (MA)\[6\], ambients represent nested computational environments containing data and live computation. Ambients are capable of moving under the influence of the process they enclose and can dissolve their perimeter with an open operation. Mobile Ambients provide a direct characterization of computational processes as well as computational devices.

Boxed Ambients (BA) \[3\] evolved from MA, by removing the ability of an ambient of dissolving its boundary. In BA, an ambient is a "box" that cannot be opened. This notion of closed ambient provides a complete encapsulation of the agents they contain. To enable the communication lost by disabling the open operation, ambients are equipped with communication channels to exchange information with adjacent ambients (parent and children ambients).

Both in MA and BA, ambient mobility is commanded by processes inside the ambient. The commands for mobility are called capabilities. The capabilities tell an ambient to open or move inside or outside another ambient. Unrestricted mobility, however, can lead to undesired interferences between two concurrent processes. Addressing this concern, control over capabilities was first introduced in Safe Ambients \[10\] and later used in New Boxed Ambients (NBA) \[4\] in the form of co-capabilities. A capability can be exercised only in the presence of a matching co-capability. Hence, in order to enter an ambient using the in capability, that ambient must contain a matching in co-capability authorizing that access; similarly for exiting using the out capability.

Bonelli et al \[1\] introduced the notion of local views in the calculus of Boxed Ambients with Communication Interfaces (BACI). In this calculus, each ambient has an associated communication port and a local view. The communication port is used for sending and receiving messages to and from other ambients, and the local view represents the communication types that are used by the processes enclosed inside the ambient. BACI is flexible enough to allow an ambient to communicate with different parents using different types. However, this flexibility came with the price of a rather complex syntax and some run-time type checking required to guarantee type safety.

In this paper, we present an enhanced and simplified version of BACI called BACI v2. In this version, we share the same goals present in the original version: to stress the separation between communication and mobility and to reduce the amount of global information in the calculus. However, this time, we achieve a better trade-off between locality, expressiveness and the calculus complexity.

In BACI v2, processes inside an ambient can use multiple ports –each one with its own communication type– to communicate with other neighboring processes within or outside the ambient. This allows, for instance, the straightforward specification of a host exposing several services like ssh or
ftp, using a different port for each service.

Notice that ports can be encoded using dedicated ambients, however, having multiple ports as primitives has the advantage of not requiring the co-capabilities necessary for the mobility of such ambients.

Another application of multiple ports is the implementation of data structures such as a stack. We can encapsulate the implementation of the stack with a single ambient, using different ports for each of the stack operations as depicted in Figure 1(a).

![Diagram](image)

(a) Stack ambient using multiple ports  
(b) Dynamic ambient access control

Fig. 1. Some examples in BACIv2.

BACIv2 also enhances the mobility control by using fine-grained co-capabilities, where each co-capabilities contains an access control list to restrict access to those ambients in the list, enabling dynamic access control. Access control lists can be used to implement access control using passwords similar to the mechanisms found in NBA [5].

Figure 1(b) depicts an ambient that, in order to enter a restricted ambient, needs to authenticate itself by sending the password to an authenticator process located outside the restricted ambient. Next, the authenticator process validates the password and instructs the restricted ambient to grant access to the validated ambient. Finally, the restricted ambient allows the access of the incoming ambient, and the operation is completed.

BACIv2 also introduces port name restriction. This restriction is used to create truly private communication channels, preventing undesired communication interferences. Moreover, new ports can be created dynamically using a special primitive called connect. Connect binds two different ports: one from the parent ambient and one from a child ambient, using a new (private) port name. This construct creates new communication channels without requiring any previous knowledge of the parent or child ambient names or the ports they use.
### Messages:

\[ M, N ::= \alpha \quad \text{name} \quad P, Q ::= 0 \quad \text{nil process} \]

\[ C \quad \text{capability} \quad P \mid Q \quad \text{composition} \]

### Capabilities:

\[ C, D ::= \text{in } \alpha \quad \text{enter} \quad \pi.P \quad \text{replication} \]

\[ \text{outTo } \alpha \quad \text{exit} \quad (\nu n)P \quad \text{restriction} \]

\[ C, D \quad \text{path} \quad (\nu_p c : \tilde{\varphi})P \quad \text{port restr.} \]

\[ x \quad \text{variable} \quad \pi.P \quad \text{prefixing} \]

\[ \alpha[P] \quad \text{ambient} \]

\[ [M = N]\{P\}\{Q\} \quad \text{equality} \]

### Prefixes:

\[ \pi ::= \text{Capabilities} \quad \text{Names:} \]

\[ \alpha, \beta ::= n \quad \text{constant} \]

\[ \alpha, \beta ::= x \quad \text{variable} \]

### Access control list:

\[ \chi ::= \text{any} \quad \text{unrestricted} \quad \Sigma ::= \emptyset \quad \text{empty} \]

\[ \hat{\alpha} \quad \text{list of names} \quad \Sigma, x : \varphi \quad \text{variable} \]

### Ports:

\[ c, u, v \quad \text{constant} \quad \Gamma ::= \emptyset \quad \text{empty} \]

\[ \Gamma, \tau \quad \text{interface} \]

### Basic types:

\[ \varphi ::= \text{amb} \quad \text{ambient} \quad \text{Communication types} \]

\[ \tau ::= c : \tilde{\varphi} \quad \text{typed port} \]

### Typing environments

\[ \chi ::= \text{any} \quad \text{unrestricted} \quad \Sigma ::= \emptyset \quad \text{empty} \]

\[ \hat{\alpha} \quad \text{list of names} \quad \Sigma, x : \varphi \quad \text{variable} \]

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax of BACIV2</td>
</tr>
</tbody>
</table>

### 2 BACIV2

#### 2.1 Syntax and Semantics

The complete syntax of the calculus is summarized in Table 1. It includes two main syntactic categories: *processes* and *messages*. Messages, ranged over by \( M \) and \( N \), include ambient *names* and *capabilities*. Ambient names, ranged over by \( \beta \) and \( \alpha \), can be either constant ambient names or name variables. Capabilities, ranged over by \( C \) and \( D \), can be either the capabilities for entering and exiting an ambient, variables or a "path" which is a sequence of capabilities describing a mobility path. In addition to the sets of variables and ambient names, we also have the set of *ports* used for communication. We presuppose these three sets to be pairwise disjoint.

Processes, ranged over by \( P, Q, R, \) and \( S \), are built from the constructors of *inactivity*, showing the end of a process; *parallel composition* of two processes; *replication*, used for recursion; ambient and port name *restriction*; *prefixing*, where \( \pi \) is an operation that is followed by a continuation process \( P \); a named ambient encapsulating a process; and, finally, *message equality testing* for branching to either \( P \) or \( Q \) depending whether \( M \) is equal to \( N \) or not.

The syntax for name and port restriction includes the name of the hidden
ambient or port paired with the appropriate type. This is done in order to unify the syntax for the structural congruence rules (omitted here). The communication type in port hiding is decorated with — in order to distinguish ambient name restriction from port name restriction.

Process prefixes can be divided into four different groups: capabilities, message send and receive, restricted co-capabilities and connects. Capabilities command an ambient to move inside or outside another ambient. Send and receive exchange messages between processes possibly at different locations. A location is merely a port located in the parent or a child ambient or locally in the same ambient. (Restricted) co-capabilities allow the entrance or exit of a particular set of ambassadors depending on whether the ambient name is included in its access control list. Finally, connect can be used to dynamically create a new port (of type \( \tilde{\varphi} \)) between child and parent ambients.

Capabilities in \textbf{BACIv2} are slightly different from the ones in Safe Ambients or NBA. Here, co-capabilities are not included as capabilities \(^4\). Moreover, the \texttt{out(To)} capability refers to the target ambient instead of referring to the ambient that is being exited. This implies that the moving ambient must know its destination. However, after executing \texttt{outTo}, a process can be certain of it current location. This is more difficult to assert with the standard \texttt{out} capability found in other ambient calculi.

Send and receive use locations to address a particular port in a parent or a child. In order to establish a communication, both send and receive must have matching port names.

This can be seen in the \texttt{INPUT} reduction rule:

\[(\texttt{INPUT} \downarrow \cdot)\]

\[(\tilde{x} : \tilde{\varphi})^k.P \mid n[\langle \tilde{M} \rangle^k.Q \mid R] \rightarrow P\{\tilde{x} := \tilde{M}\} \mid n[Q \mid R]\]

where \(\downarrow c\) matches \(\uparrow c\).

Only capabilities and ambient names can be sent as messages. However, the syntax can be easily extended to allow other kinds of messages, such as integers or booleans.

Co-capabilities \(\overline{\text{in}}\{}\) and \(\overline{\text{out}}\{}\) have a list of ambients that are allowed to enter (or exit) using that co-capability. Additionally, the label any can be used to denote unrestricted access. Here is the reduction rule for \texttt{ENTER}:

\[(\texttt{ENTER})\]

\[n[\text{in} m.P_1 \mid P_2] \mid m[\overline{\text{in}}\{}\chi\}.Q_1 \mid Q_2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2]\]

where \(\chi\) is \{\(\tilde{n}_1, n, \tilde{n}_2\)\} or any

Finally, the \texttt{connect} prefix allows a process to create a new private channel shared with another process located in the parent or a child ambient.

\(^4\) This means that co-capabilities cannot appear as messages and; therefore, they cannot be sent or received.
(CONNECT)
\[
\text{connect} \lfloor (u : \tilde{\phi}) \rfloor P \mid n[\text{connect} \lfloor (v : \tilde{\phi}) \rfloor Q \mid R] \rightarrow \\
\lfloor (\nu_p c : \tilde{\phi}) \rfloor (P\{u := c\} \mid n\{Q\{v := c\} \mid R\})
\]
where \(c\) is a fresh port name.

3 Typing Judgments

Typing environments are defined by the following grammar.

\[
\Sigma ::= \emptyset \quad \text{empty environment} \\
\mid \Sigma, x : \varphi \quad \text{extension}
\]

Typing environments are assumed to assign a unique type to each name in its domain.

There exist two different typing judgments, one for messages and one for processes:

- \(\Sigma \Vdash M : \varphi\), read “\(M\) is a well-formed message of type \(\varphi\) in \(\Sigma\)”, and
- \(\Sigma \triangleright P : \Gamma\), read “\(P\) is a well-formed process of type \(\Gamma\) in \(\Sigma\)”.  

In contrast to other systems, there is no communication type associated with an ambient name, instead an ambient name has the constant type \text{amb}.

In the judgment \(\Sigma \triangleright P : \Gamma\), the process interface \(\Gamma\) exhibits the communication types used by \(P\). \(\Gamma\) assigns communication types to each free port \(c\) used in \(P\). Ports hidden using port restriction \((\nu_p c : \tilde{\phi})\) do not appear in \(\Gamma\), since their type is declared in the restriction operation. This fact is reflected in the corresponding typing rule:

( Proceedings )

\[
\Sigma \triangleright P : \Gamma, c : \tilde{\phi} \quad \frac{}{\Sigma \triangleright (\nu_p c : \tilde{\phi})P : \Gamma}
\]

Similarly, the \text{connect} operation also abstracts port names hiding them from resulting \(\Gamma\).

The type system guarantees that communication inside ambients and across ambient boundaries never leads to type mismatches. This is formalized as the Subject Reduction theorem:

**Theorem 3.1 (Subj. reduction)** If \(\Sigma \triangleright P : \Gamma\) and \(P \rightarrow Q\), then \(\Sigma \triangleright Q : \Gamma\).

**Proof.** By induction on the derivation of \(P \rightarrow Q\). \(\square\)

4 Examples

4.1 Stack

In this example, we model a stack (data-structure) for storing names using multiple ports. We consider three primitive operations applied to stacks: \text{push},
pop, isEmpty. Push takes an element an insert it on the top of the stack; pop, on the other hand, removes the element on top and returns it as a result of the operation. Finally, isEmpty is used to query the stack whether it is empty or not.

For the implementation of these operations we use different ports. In fact, we use two ports per operation: one for receiving the request and the other to deliver the response (possibly excepting the push operation that do not require a response). In general, we name each port using the name of the operation and a subindex indication whether it is receiving a request or submitting a response.

The stack is represented by a ambient named accordingly.

\[ STACK = stack[\text{INTERNALS} \mid \text{!PUSH} \mid \text{!POP} \mid \text{!ISEMPTY}] \]

The processes inside the stack can divided into four parts: INTERNALS which keeps the internal state and manage some internal operations, and the processes that manage each operation, respectively, PUSH, POP and ISEMPHY.

The implementation uses (internally) a linked list of nodes. Each node is an ambient containing a "stacked" value. A local port called top is used as a variable to store the name of the node that holds the top value of the stack.

The INTERNALS part holds the current state of the stack. Initially, the stack is empty, so INTERNALS only contains the initial state: INITSTATE.

\[ \text{INITSTATE} = \langle \text{empty} \rangle^{\text{top}} \]

We use the name empty to denote that the stack is empty; therefore, empty is "stored" as the top.

In order to manipulate this internal state, we introduce two syntactic definitions: gettop and settop.

\[ \text{gettop}(x).P = (x)^{\text{top}}. (P \mid (x)^{\text{top}}) \]
\[ \text{settop}(n).P = P \mid (x)^{\text{top}}. (n)^{\text{top}} \]

Gettop retrieves the name of the node ambient that holds the top value of the stack and binds it to a variable. On the other hand, settop takes an ambient name (i.e. a node name) as an argument and sets that name to by the top of the stack.

Using these macros, we can define the ISEMPHY operation:

\[ \text{ISEMPTY} = \]
\[ (_)^{\text{isEmpty}_{req}}. \text{gettop}(t). [t == \text{empty}] \{ (true)^{\text{isEmpty}_{req}} \} \{ (false)^{\text{isEmpty}_{req}} \} \]

First, a request is received on isEmpty_{req} port. Then, after retrieving the top name, the process checks if it is equal to empty or not, returning the

\[ ^{5} \text{In general, we omit the nil process at the end a process expression for sake of readability.} \]
\[ ^{6} \text{We use a "_" instead of variable name to notice that the communication is done just to signal the operation, and the value received on this port is a meaningless dummy value.} \]
names true or false over the port isEmpty\textsubscript{res}.

The \textit{PUSH} operation receives a value \textit{v} over the port \textit{push\textsubscript{req}} and creates a new node with that value.

\textit{PUSH} = (v)\textsubscript{push\textsubscript{req}}.NEWNODE(\textit{v})

The newly created node stores the value at the top of the stack and, it also records the name of the old top of the stack as the next node in the linked list.

\textit{NEWNODE}(\textit{x}) =

\text{(\textit{v node})gettop(\textit{t}).(node\{ NODE\_INT(\textit{x}, \textit{t}) \} \mid settop(node))}

Internally, each node has a process that is ready to release the stored value after it is triggered by the entrance of a special messenger ambient called \textit{popper}. As its name suggests, the sole purpose of the \textit{popper} ambient is to signal the node that is being popped from the stack.

\textit{NODE\_INT}(\textit{v}, \textit{t}) = [\text{popper}].\text{(} (\textit{v}, \textit{t}) \text{)}^{\text{release}}

After this signal, the node retrieves the stored value and also the name of the following node in the stack. This feature of the nodes is used by the \textit{POP} operation. After receiving a request via the \textit{pop\textsubscript{req}} port, the \textit{POP} operation send a messenger ambient to the top node to start the removing process.

\textit{POP} = (\text{\_})\text{\textsubscript{pop\textsubscript{req}} gettop(\textit{t}).(popper[\text{in } \textit{t}].(v, t')^{\text{release}} \text{ settop(}t').(v)\text{\textsubscript{pop\textsubscript{res}}})}

Simultaneously, a parallel process waits the response from the top node though the \textit{release} port. After that response is received, the next node in the stack is placed at the top of the stack and; finally, the value that used to be on top is retrieved from the stack ambient using the \textit{pop\textsubscript{res}} port.

\subsection{4.2 Authentication using passwords}

This example shows a simplified implementation of a mechanism similar to the capabilities in NBA \cite{5} that use passwords as a access control to enter or exit into other ambients. The example corresponds to the situation depicted in Figure 1(b).

An \textit{agent} request access to a system \textit{sys} by sending its credentials to \textit{AUTH} the process in charge of the authentication procedure.

\textit{agent[ in\{ sys with passwd\}.P ] \mid AUTH \mid sys[ ACCESS | SYS ]}

\textit{AUTH} is continuously listening for incoming requests. On each request, it tests the given password either granting the requested access or informing that the access was denied.

\textit{AUTH} = !(m, m_{pwd})^{\text{auth\textsubscript{req}}.\text{[}m_{pwd} = \text{good}_{pwd}\text{]}\text{GRANT}\text{]}\text{DENY}

\textit{GRANT} = \langle\text{granted}\rangle^{\text{auth\textsubscript{res}.}(m)\text{grant}}

\textit{DENY} = \langle\text{denied}\rangle^{\text{auth\textsubscript{res}}}
The agent uses the macro in \{n with passwd\} to request access.

\[
P = REQUEST_{agent}(sys, passwd) | (\_)^{\text{go}}.in. n. P
\]

The REQUEST process continuously try and re-try to get access by sending the its credentials and waiting for a response each time. Therefore, the continuation process \( P \) is blocked until the access is granted.

\[
REQUEST_m(n, pwd) = (\nu_p \text{ do}_\text{auth} : \text{amb})(\langle \text{try}^{\text{do}_\text{auth}} | !((\_)^{\text{do}_\text{auth}}. \langle m, pwd \rangle)\text{authreq}. (\langle \text{ok} \rangle)^{\text{authres}}[\text{res} = \text{granted}]\langle \text{retry}^{\text{do}_\text{auth}} \rangle)
\]

where \( m \) is the name of the ambient requesting permission to enter the ambient \( n \).

Finally, after AUTH authorizes the access, it communicates with the system to allow the access of the authenticated process via a port named grant.

\[
ACCESS = !(a)_{\text{grant}}.in\{a\}
\]

5 Summary and Conclusions

Continuing our earlier work on BACI[1], the calculus presented here aims at further decoupling communication from mobility. This is achieved by the introduction of multiple ports that are exclusively used for communication. The addition of access control lists to co-capabilities enables a better mobility control against interferences than standard co-capabilities. These fine-grained co-capabilities are useful in encoding specific mechanisms that can be used transparently, without any side-effect on the ambient mobility control. Multiple ports reduce the need of complex encodings to implement several communication channels, rendering specifications closer to the reality being modeled. However, excessive use of ports could be reduced by extending the calculus with session types [9] for ports. A type system including session types would enforce stronger type safety guarantees. Moreover, the introduction of co-capabilities that also bind the name of the entering ambient – similar to NBA [5] co-capabilities – could give each ambient additional control over its child ambients.

Acknowledgement

We are grateful to Mariangiola Dezani-Ciancaglini for enlightening discussions and to Elsa Gunter, Eduardo Bonelli, Ricardo Medel and Alejandro Russo for many useful comments and constructive criticism. This work was supported in part by the NSF Grant No. CCR-0220286 ITR: Secure Electronic Transactions.
References


