Abstract—Ambient-PRISMA is an architectural approach for specifying aspect-oriented software architecture and generating code of distributed and mobile systems. Ambient-PRISMA lacks a precise semantics due to the fact that it is based only on a metamodel. In this paper, Ambient-PRISMA is mapped into a formal language called Channel Ambient Calculus, a process algebra for specifying mobile applications that provides channels and ambients as first-class citizens. We argue that the formalization in Channel Ambient Calculus is particularly well-suited for modelling Ambient-PRISMA.

Keywords—software architecture, mobile systems, process algebras

I. INTRODUCTION

Ambient-PRISMA [1] is a software architectural approach for specifying distributed and mobile systems. It enriches PRISMA [2], an aspect-oriented software architectural approach, with primitives inspired from Ambient Calculus (AC) [3]. In AC, locations are abstractly modelled through the concept of ambient, which is a bounded place where computation happens such as a PC, a folder or a network. Ambients have mobility capabilities for entering and exiting other ambients.

Ambient-PRISMA includes ambients as architectural elements that describe the locations (bounded places) where other ambients, components, and connectors (architectural elements) are distributed. Ambients provide architectural elements with entering and exiting capabilities. In addition, ambients control the communication among architectural elements located in different ambients.

The Ambient-PRISMA approach follows Model Driven Engineering (MDE) and provides a metamodel, an Aspect Oriented Architecture Description Language (AOADL), middleware and CASE Tool that allows designing models using graphical notations and generating their code [1]. The graphical notation for designing the architectural models and the building of the CASE Tool as plugins were possible thanks to MDE. Such CASE Tool includes a modelling interface and code generation facilities. However, metamodels can be ambiguous and lack a precise and concise semantics. In particular, dynamic behavioural aspects are germane to Ambient-PRISMA because mobility is a dynamic behaviour that changes the configuration of the software architecture.

In this paper, Ambient-PRISMA formalization is presented. The concepts of Ambient-PRISMA are precisely defined in process algebra in order to express in an accurate way the behaviour captured in its specifications. PRISMA concepts have been formalized in π-calculus such as aspects and weavings [4]. In this work, we focus on formalizing the concepts that are affected from including ambients, as well as the primitives related to architectural elements communication, and mobility.

The formalization is performed by mapping Ambient-PRISMA into Channel Ambient Calculus (ChAC) [5], a variant of Ambient Calculus. ChAC is chosen as our formal language because it has an intuitive representation of ambients, entering and exiting capabilities, and channels for communication purposes in a similar way to π-calculus [6].

The paper is organized as follows. In Section II, we give an overview of Ambient-PRISMA. Section III motivates the choice of ChAC and explains the constructs that have been used in the formalization. Section IV presents the formalization of Ambient-PRISMA by mapping the Ambient-PRISMA concepts into ChAC. Section V presents related work. Finally, Section 5 concludes and sketches further work.

II. AMBIENT-PRISMA OVERVIEW

In this section, we give a brief account of Ambient-PRISMA and refer the reader to [1] for details on its metamodel, middleware, CASE tool and AOADL.

Ambient-PRISMA is an approach that enriches PRISMA with the concept of ambient. Intuitively, an ambient represents a bounded place where architectural elements are located (see Figure 1). In our context, ambients also provide mobility services. Ambients extend the concept of an architectural connector because they can coordinate architectural elements that are inside their boundary from the ones that are outside.

In Ambient-PRISMA (as its predecessor PRISMA), architectural elements which are components, connectors and ambients are defined by ports, aspects, and weavings. Ports are the points where architectural elements send and receive service invocations. Architectural elements that
communicate with each other need to be connected through attachments. An attachment is a channel that connects a port of an architectural element to a port of another one (lines in Figure 1).

Figure 1. An ambient locating other architectural elements

In Figure 1, A is a component and B is a connector. Both are located in the same ambient called Root. Also, an ambient can have other subambients. For example, subambient is located in the Root ambient. Ambients can be nested to form a tree structure which has a root ambient. Each software architecture configuration needs an ambient called Root.

An ambient offers two kinds of services to architectural elements located in it: mobility services and distributed services. Architectural elements that need these services are connected to their parent ambient through attachments (the lines in Figure 1).

An ambient uses different aspects to specify the services it offers and requests. As Figure 1 shows, each ambient must have a mobility aspect called MobilityAspect, a coordination aspect called ACoordination, and a distribution Aspect. Noteworthy, the Mobility aspect and the Coordination Aspect of an ambient are uniform in all ambients and each ambient can have its own definition of a distribution aspect. Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects [2].

As ambients are responsible for the mobility concern, all ambients must have the mobility aspect to provide mobility services to moving architectural elements. The MobilityAspect differentiates an ambient from other PRISMA connectors. The following services are specified in the MobilityAspect:
- exit: architectural elements that need to exit an ambient must invoke the exit service from their parent ambient (AC exit capability).
- enter: architectural elements that need to enter an ambient must invoke the enter service from their parent ambient (AC enter capability).

The ACoordination aspect is necessary for coordinating its architectural elements with external elements. This coordination aspect receives calls from external architectural elements of the ambient (distributed calls) and redirects them to corresponding architectural elements of the ambient. It also receives calls from its architectural elements and redirects them to the exterior.

A distribution aspect is used by all architectural elements (ambients and others) in order to specify their distribution policies and allow them to be aware of their parent ambient. A designer can specify a number of distribution aspects depending on the distribution requirements of a system. A distribution aspect stores the name of the parent ambient of an architectural element. This aspect also includes the specification of whether an element is mobile or not. A mobile element requests mobility services from its parent ambient. When an element is mobile, its parent ambient can be changed.

To clarify the concepts of Ambient-PRISMA, we are going to use a simplified running example of an electronic auction system. The auction system allows customers to bid on the products or to use a mobile agent and bid on their behalf. A more detailed specification of this example in Ambient-PRISMA AOADL can be found in [1] and [7].

For example, Figure 2 shows a hierarchy of ambients where the Root ambient locates two ambients called ClientSite and AuctionSite. The ClientSite ambient locates the Customer1 component, the Bidder1 component and the AgentCustCnct connector. The AuctionSite ambient locates the AuctionCnctr1 connector and the Auction1 component.

Figure 2. A configuration of an auction mobile agent

An ambient controls what can permeate its boundary. For example in Figure 2, when the Bidder1 component sends a service request to the AuctionCnctr1 connector, the request has to be synchronized by the ClientSite ambient in order to allow it to be sent to the AuctionSite ambient, and the AuctionSite ambient forwards the service request to the AuctionCnctr1 connector. Also, if Bidder1 needs to exit from the ClientSite ambient to the Root ambient, then the ClientSite ambient has to coordinate the Bidder1 component with the Root ambient in order to allow the Root ambient to accept the Bidder1 component. In this way, the Bidder1 component crosses the boundary of the ClientSite ambient into the one of the Root ambient.

Elements of different ambients do not have direct attachments between each other. An attachment only connects elements of the same ambient. For example in Figure 2, an attachment between a port of the ClientSite ambient and a port of the AuctionSite ambient is possible because the ClientSite and the AuctionSite are siblings (they share the same parent ambient). An attachment is also
possible between Customer1 and the AgentCustCnct connector because both of them are in the ClientSite ambient.

![Diagram of a mobile agent's configuration](image)

Figure 3. Another Configuration of an auction mobile agent

When an architectural element moves from one ambient to another, the attachments associated to an ambient are reconfigured in order to provide the mobile architectural element with the services it needs. For example, when the Bidder1 component in Figure 2 moves from the ClientSite ambient to the AuctionSite one, the attachments associated to Bidder1 are reconfigured (as in Figure 3); where Bidder1 has attachments with the AuctionSite ambient instead of the ClientSite ambient, and attachments that connect Bidder1 to the elements located in ClientSite are removed.

III. CHANNEL AMBIENT CALCULUS

In this section, we motivate the use of ChAC as formalism for Ambient-PRISMA. Also, we present the notation that we have adopted in this paper.

A. Choosing Channel Ambient Calculus

π-calculus [6] is a process calculus for modeling concurrent interacting computations. It is based on the notion of naming. Names can be channels or data. The basic notion of action that can be performed is interaction between processes. Interaction is achieved by sending or receiving values (names) through channels (names). The variant of π-calculus called polyadic π-calculus [8] allows a channel to send or receive many names.

In π-calculus, channels can be transferred from a process to another. In this way, π-calculus models mobility and reconfiguration (evolution of structure). It is important to emphasize that the mobility which can be modelled in π-calculus is called virtual mobility (as denominated by Milner). In virtual mobility, movement is represented by the change of links among processes i.e., channels between processes move but processes do not. On the other hand, other forms of mobility exist such as physical mobility. In physical mobility, processes move from one physical space to another. This type of mobility is the one supported by Ambient Calculus [3]. In Ambient Calculus, names are ambients and the basic notion of action is movement from one ambient to another (instead of interaction as in π-calculus).

To model real distributed and mobile systems both interaction and mobility are essential. This is due to the fact that components need to interact among each other while they are in different physical spaces and in some cases they need to move from one physical space to another. In addition, virtual mobility and physical mobility are important because physical mobility can cause the change of links among processes. Therefore, a process algebra that provides both ambients and channels is of great interest.

ChAC [5] is a process algebra that specifies mobile applications. It is inspired by π-calculus [6], and the Ambient Calculus [3]. ChAC uses the notion of ambient, as introduced in Ambient Calculus, to model the entities of a mobile application. It also defines channels as first-class citizens. Channels are used to allow processes in ambients to communicate internally or externally and to allow ambients to move in and out of each other.

ChAC is well-suited to formalize Ambient-PRISMA for the following reasons:

- it is suitable to model the mobility and communication mechanisms of Ambient-PRISMA, since it provides channels for communication as well as ambient capabilities for mobility in a way similar to Ambient-PRISMA;
- the formalization of PRISMA [4] without distribution and mobility has been performed in π-calculus. Since ChAC is inspired by the π-calculus, the PRISMA formalization only has to be extended to incorporate the mobility and distribution characteristics of Ambient-PRISMA. In addition, π-calculus primitives can be easily encoded in ChAC.

B. Syntax and Semantics

This section borrows the main definitions of ChAC. The interested reader is referred to [5] for more details on ChAC.

shows the notations used for formalizing Ambient-PRISMA. A name can be a name of an ambient or a channel. For example, x, y, z are names. A tuple of names can be defined. A process can be an inactive process a, a parallel composition of processes P|Q, a non deterministic choice of processes P+Q i.e. either P or Q is executed, a process with a restricted (or local) name vn P, a process localized in an ambient a[P], or a process prefixed by an action a.P.

A prefix can be either a communication or a migration action. Communication actions allow ambient processes to send and receive values to and from processes of other ambients using channels. The action a. x(v) sends a value v on channel x to a sibling ambient a. The action x! (v) sends a value v on channel x from the parent ambient. The action x(u) receives a value u on channel x from outside the current ambient. The x'(u) receives a value u on channel x from the current ambient. The x(v) sends a value v on channel x locally. The a/x(v) sends a value v on channel x to a child ambient.
The following sections present the mapping for each Ambient-PRISMA construct.

A. Components and Connectors

A component/connector is formed by a set of aspects, their weaving relationships, and one or more ports. In Ambient-PRISMA, components and connectors can be mobile elements. The movement of a component/connector involves moving its aspects, weavings and ports.

### Table 1. The notations used in this paper

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y, z</td>
<td>Names</td>
</tr>
<tr>
<td><img src="x" alt="in" /></td>
<td>tuple of names</td>
</tr>
<tr>
<td><img src="x" alt="out" /></td>
<td>Parent Output</td>
</tr>
<tr>
<td><img src="a" alt="in" /></td>
<td>Internal Input</td>
</tr>
<tr>
<td><img src="a" alt="out" /></td>
<td>External Input</td>
</tr>
<tr>
<td><img src="x" alt="local" /></td>
<td>Local Output</td>
</tr>
<tr>
<td><img src="v" alt="in a x" /></td>
<td>Child Output</td>
</tr>
<tr>
<td><img src="x" alt="out x" /></td>
<td>Exit</td>
</tr>
<tr>
<td><img src="x" alt="in x" /></td>
<td>Accept</td>
</tr>
<tr>
<td><img src="x" alt="out x" /></td>
<td>Release</td>
</tr>
</tbody>
</table>

Migration actions allow an ambient to move in and out of each other over channels. The actions are:
- **in a x** which enters a sibling ambient over a channel x.
- **out x** which leaves a parent ambient over channel x.
- **![in](x)** which accepts a sibling ambient over x.
- **![out](x)** which releases a child ambient over channel x.

Every process identifier $A$ has a defining equation:

$$A(\hat{x}) = P_A$$

where $P_A$ is a process and $\hat{x} = x_1, \ldots, x_n$ (all distinct).

To indicate the parallel composition of a set of processes the following notation is used:

$$\prod_{i=1}^{n} P_i$$

The conditional operator is a derived operator that checks the semantic equality of two names, if they are equal it executes a process $P_1$ otherwise it executes a process $P_2$. Also, we will use $\text{if } x = y \text{ then } P_1$ as a shortcut for $\text{if } x = y \text{ then } P_1 \text{ else } \alpha$.

Table 2 shows the reduction rules for executing processes in ChAC. In the following, we give their intuitive description:

1. An ambient $b$ can send a value $v$ to a sibling $a$ over a channel $x$, if $a$ contains a sibling output $a \times v.P$, and there is a sibling ambient $a$ with an external input $x(u).Q$.
2. An ambient $a$ can send a value $v$ to its parent over a channel $x$, if ambient $a$ contains a parent output $x(u).P$, and there is an internal input $x(u).Q$ in parallel with $b$. Note that in the communication interactions, the sent value $v$ is replaced with the value $u$ in the communication of the receiver.
3. An ambient $b$ can send a value $v$ to its child ambient $a$ over a channel $x$, if ambient $a$ contains a child output $a / x(v).P$, and there is an input $x(u).Q$ in parallel in ambient $a$.
4. An ambient $b$ can enter a sibling $a$ over a channel $x$, if ambient $b$ contains an enter $in a.x.P$, and ambient $a$ has an accept $\overline{in} x.Q$.
5. An ambient $a$ can leave its parent ambient $b$ over a channel $x$, if the ambient $b$ contains a leave $out x.P$, and ambient $a$ has a release $\overline{out} x.Q$.
6. and (7) are not ChAC based reductions. An ambient $b$ can execute a process $P$ if the semantic equality of two names are equal, otherwise it executes process $Q$.
A component is mapped into a ChAC ambient containing the \( P_{IdAE} \) process that executes. The definition of a component is as follows:

\[
\begin{align*}
\text{IdComp}(\text{Asp}, \text{W}, \text{Port}) &= \text{IdComp}(P_{IdAE}(\text{Asp}, \text{W}, \text{Port})) \\
\text{P}_{IdAE}(\text{Asp}, \text{W}, \text{Port}) &= \prod_{i=1}^{a} P_{\text{Asp}_i} \prod_{i=1}^{w} P_{\text{W}_i} \prod_{i=1}^{p} P_{\text{Port}_i}
\end{align*}
\]

A component which has \( IdComp \) as its identifier is mapped into a ChAC ambient called \( IdComp \). The identifier of a component is its name. The \( IdComp \) ambient contains the \( P_{IdAE} \) process which models the parallel composition of its aspects, its set of weavings, and its set of ports, where \( IdAE \) is the identifier of an architectural element, \( Asp \) is a tuple of aspects (of length \( a \)), \( W \) is a tuple of weavings (of length \( w \)), and \( Port \) is a tuple of ports (of length \( p \)). We assume there is a process that represents the behaviour of the parameter of the definition.

For example, consider the \( Bidder \) component given in Figure 4 which specifies the \( Bidder \) component instance of Figure 2 and Figure 3. The component has a name (\( Bidder \)), a distribution aspect (\( BidderDist \)), a functional aspect (\( BidderFunct \)), a weaving process and a set of ports. Its formalization is an ambient called \( Bidder \) which contains the processes that map its aspects, weavings and ports as follows:

\[
\begin{align*}
\text{Bidder}[P_{\text{BidderDist}} | P_{\text{BidderFunct}} | P_{\text{W}} | P_{\text{DCapPort}} | P_{\text{DMovingPort}} | P_{\text{CustBidderPort}} | P_{\text{BidderAuctPort}}].
\end{align*}
\]

Figure 4. Specification of the \( Bidder \) component

\[
\begin{align*}
\text{Component_type Bidder} \\
\text{End Component_type Bidder;}
\end{align*}
\]

B. \( \text{Ports} \)

\( \text{Ports} \) are the interaction points of architectural elements. An architectural element receives invocations and sends requests of a set of services through its ports. These services are specified in an interface. Services have \( \text{input} \) parameters which are the arguments, and \( \text{output} \) parameters which are the return values. For example, the \( \text{DMovingPort} \) port of Figure 4 can receive and send services of an interface called \( \text{IMobility} \). The \( \text{IMobility} \) interface is specified in Figure 5 and it consists of a service called \( \text{move} \) that takes an ambient as input.

\[
\begin{align*}
\text{Interface IMobility} \\
\text{move(input NewAmbient: Ambient);}
\end{align*}
\]

Figure 5. Specification of the \( \text{IMobility} \) interface

A port is mapped into the parallel composition of two processes: the process that can receive services that are invoked from the exterior of the architectural element, and the process that sends requests of services from the aspects of the architectural element to the exterior. Composing that process is important because an architectural element can send and receive services at the same time.

As a result, a port is mapped into the following process:

\[
\begin{align*}
P_{\text{Port}}(i s, n, x, y, s) &= \prod_{x = s n} \text{PortSender}(i s, x, y, s) \mid \text{PortRec}(n, x, y, s)
\end{align*}
\]

where \( Port \) is the name of a port, \( i s \) is a tuple of service names of an aspect, \( n \) is a name of an invoked service, \( x \) is a tuple of channels for the input parameters, \( y \) is a tuple of channels for the output parameters, and \( s \) is a tuple of service names, \( \text{PortSender} \) sends requests of services while \( \text{PortRec} \) receives invocations of services in \( s \).

\( \text{PortSender} \) is defined as follows:

\[
\begin{align*}
P_{\text{PortSender}}(i s, x, y, s) &= \prod_{x = s n} \text{is}(x, y). \\
\text{if} \ i s = s_1 \ \text{then} \ 
\text{Request}(s_1, x, y), P_{\text{PortSender}}(i s, x, y, s) \\
\text{else} \ ... \\
\text{if} \ i s = s_n \ \text{then} \ 
\text{Request}(s_n, x, y), P_{\text{PortSender}}(i s, x, y, s) \\
\text{Request}(s, x, y) = s^j \langle x, y \rangle \cdot r_{y_j}^+ (y_0) ... r_{y_m}^+ (y_m)
\end{align*}
\]

A port requests a service only when an aspect of an architectural element requests it through an \( i s \) channel (the channel that represents a service invoked by an aspect). The \( \text{PortSender} \) process compares the \( s \) channel with the \( \langle s \rangle \) channels in order to match it with an \( s \) channel. Once the \( i s \) channel is matched with a \( s \) channel, the process forwards the request on a channel \( s \) that sends the input parameters
and the channels created for the output parameters to a process outside the ambient which represents the architectural element. The process that receives the input parameters and the created channels are the attachments (see section IV.E). Then, the process waits in order to receive the results of the output parameters \( y_1, \ldots, y_m \) through the \( r_1, \ldots, r_m \) channels. Finally, the process is prepared to receive a new service request.

For example, the process of the \( \text{DMovingPort} \) port in Figure 4 is formalized as follows:

\[
\text{move}(\text{NewAmbient}, \text{ack}).\text{move} \langle \text{NewAmbient}, \text{ack} \rangle . \text{ack} \langle \text{true} \rangle
\]

\[
\text{move} \langle \text{NewAmbient}, \text{ack} \rangle . \text{ack} \langle \text{true} \rangle
\]

where \( \text{ack} \) is \( r_{\text{ij}} \) and \( \text{true} \) is \( y_0 \) (or return value).

The \( \text{PPortRec} \) process is defined in a similar way to \( \text{PPortSender} \) process but instead of sending the message using the parent output \( \text{move} \langle \text{NewAmbient}, \text{ack} \rangle \), an external input is used because the process receives input from outside of the ambient that represents the architectural element \( \text{move} \langle \text{NewAmbient}, \text{ack} \rangle \). For example, the process of the \( \text{DMovingPortRec} \) port in Figure 4 is formalized as follows:

\[
\text{move} \langle \text{NewAmbient}, \text{ack} \rangle . \text{move} \langle \text{NewAmbient}, \text{ack} \rangle . \text{ack} \langle \text{true} \rangle
\]

The exit service is invoked by an architectural element to its ambient in order to allow it to leave. The enter service is invoked by an architectural element to its ambient in order to allow it to be accepted in a child ambient. The exit and enter services are different from other services. This is due to the fact that the satisfaction of the exit request means that the architectural element can exit its parent ambient through a channel and that the satisfaction of the enter request means that the architectural element can enter its sibling ambient. As a result, the exit and enter services receive an exit channel or an enter channel, respectively through their return channels. In this way, an architectural element can execute the \text{in} and \text{out} actions of ChAC.

The exit service is defined as follows:

\[
P_{\text{DCapPortSender}}(\text{exit}, \text{enter}, \ldots, \text{Name}, \text{NewAmbient}, \ldots, r_y, \text{exit}, \text{def} \text{enter}, \ldots) =
\]

\[
\text{if} \ iexit(\text{Name}, r_y).
\]

\[
\text{then} \ \text{EXIT}(\text{Name}, \text{NewAmbient}, \text{ry})
\]

\[
\text{else} \ ... \text{else} \ ...
\]

\[
\text{then} \ \text{Request}(s_\mathcal{N}, x, r_y)). P_{\text{DCapPortSender}}(i, s, x, y, s)
\]

\[
\text{EXIT}(\text{Name}, r_y) =
\]

\[
\text{exit} \langle \text{Name}, r_y \rangle . r_y \langle \text{true} \rangle \langle \text{true} \rangle
\]

\[
\text{ENTER}(\text{Name}, \text{NewAmbient}, r_y) =
\]

\[
\text{enter} \langle \text{Name}, r_y \rangle . r_y \langle \text{true} \rangle \langle \text{true} \rangle
\]

\[
in \text{NewAmbient} \cdot y_i
\]

where \( \text{iexit} \) and \( \text{ienter} \) are channels of the requested services shared between a distribution aspect and the \( \text{DCapPort} \); \( \text{Name} \) and \( \text{NewAmbient} \) are channels for the \text{input} parameters, \( r_y \) is a tuple of channels for the \text{output} parameters, and \( \text{exit} \) and \( \text{enter} \) are the channels for services for sending the requests.

Referring to the processes above, when an architectural element requests to exit an ambient (and the ambient accepts to release it), the architectural element receives through the

### Figure 7 The IC\text{Capability} interface

The IC\text{Capability} interface is defined as follows:

\[
\begin{array}{|l|}
\hline
\text{Interface IC\text{Capability}} \\
\hline
\text{exit } (\text{input Name: ArchitecturalElement}); \\
\text{enter } (\text{input Name: ArchitecturalElement}, \\
\text{input NewAmbient: Ambient}); \\
\hline
\end{array}
\]

### Bidder component

A mobile architectural element needs to have a port called \( \text{DCapPort} \). The \( \text{DCapPort} \) port allows an architectural element to invoke mobility services of the IC\text{Capability} interface, including the \text{exit} and the \text{enter} services as defined in Figure 7.

### DCapPort

A mobile architectural element needs to have a port called \( \text{DCapPort} \). The \( \text{DCapPort} \) port allows an architectural element to invoke mobility services of the IC\text{Capability} interface, including the \text{exit} and the \text{enter} services as defined in Figure 7.

### Figure 6. Simplified and partial formalization of the Bidder component into ChAC

![Figure 6. Simplified and partial formalization of the Bidder component into ChAC](image-url)

### Key:

- **Ambient**
- **Process**
- **channel**
channel $r_i$ the name of another channel that is used for leaving a parent ambient ($\text{out } y_i$). Also, when an architectural element requests to enter a sibling ambient and the sibling ambient accepts it, the architectural element receives through the $r_i$ channel a name which is used as a channel for allowing an element to enter a sibling ambient (in NewAmbient $\cdot y_i$).

D. Ambients

Ambient-PRISMA ambients are distinguished from other architectural elements (components and connectors) because they can release and accept new architectural elements in them. These architectural elements that can be released or accepted can be components, connectors, or other ambients.

Each Ambient-PRISMA ambient is defined by a ChAC ambient that contains a set of subambients, components, connectors, attachments (for the formalization of attachments see section IV.E), together with the process definition of its own aspects, weavings, and ports. In the following, we show the definition of an ambient in ChAC:

$$
\text{IdAmb}(\text{Amb}, \text{Comp}, \text{Con}, \text{ATCH}, \text{Asp}, \text{W}, \text{Port}) = \\
\prod_{i=1}^{a} \text{IdAmb}_i \mid \prod_{i=1}^{c} \text{IdCon}_i \mid \prod_{i=1}^{n} \text{IdComp}_i \mid \prod_{i=1}^{t} \text{PATCH}_i
$$

$$
P_{\text{IdAmb}}(\text{Asp}, \text{W}, \text{Port})
$$

$$
\text{Client}(\prod_{i=1}^{a} \text{IdAmb}_i \mid \prod_{i=1}^{c} \text{IdCon}_i \mid \prod_{i=1}^{n} \text{IdComp}_i \mid \prod_{i=1}^{t} \text{PATCH}_i)
$$

$$
P_{\text{IdAmb}}(\text{Asp}, \text{W}, \text{Port})
$$

where $\text{Amb} \cdot \text{Comp} \cdot \text{Con}$ is a tuple of subambients (of length $a$) located in an ambient and where $\text{IdAmb}$ cannot be part of $\text{Amb} \cdot \text{Comp} \cdot \text{Con}$. $\text{Asp}$ is a tuple of components (of length $c$), $\text{Con}$ is a tuple of connectors, $\text{ATCH}$ is the tuple of attachments; an attachment can be either connect a port of an architectural element located in an ambient to a port of another architectural element located in the same ambient, or connect a port of an architectural element located in an ambient to a port of the same ambient, $\text{Asp} \cdot \text{Patch}$ is the tuple of aspects of the ambient where the MobilityAspect is part of, $\text{Asp} \cdot \text{Port}$ is the tuple of ports of the ambient.

In addition, the $P_{\text{MobilityAspect}}$ process which must form part of the $\text{Asp}$ of an ambient has to perform the following process in order to create a channel for releasing and another for accepting ambient, and perform the accepting and releasing actions as follows:

$$
v \text{exitingCh.out} \text{exitingCh} + v \text{enteringCh.in} \text{enteringCh}
$$

Figure 8 shows the HostSite ambient specification in the AOADL. As it can be observed, an ambient is specified as an architectural element (similarly to a component and a connector), but it also has attachments that connect ports of architectural elements (components, connectors, and subambients) and attachments that connect architectural elements with its ports.

```plaintext
Ambient_Site type HostSite
Import Mobility Aspect MobilityAspect;
Import Coordination Aspect ACoordination;
Import Distribution Aspect ADist;
Weavings
ADist.getLocation(Location) instead
MobilityAspect.getParent(Parent);
End_Weavings
Ports
InCapabilitiesPort: ICapability;
EServicesPort: ICall;
InServicesPort: ICall;
End_Ports
End_Ambient_Site type HostSite;
```

E. Attachments

Attachments are the connections that connect a port of an architectural element to a port of another architectural element. Attachments can only connect two architectural elements that are located in the same ambient or connect an architectural element to its parent ambient. Architectural elements connected through attachments do not have the references (names) of the elements they are connected to. This information is contained in an attachment.

An Ambient-PRISMA attachment is mapped into a process in ChAC. In addition, an attachment is not a mobile entity of the software architecture which emphasizes that an attachment does not need to be an ambient. In this way, attachments are process of an Ambient-PRISMA ambient which is the parent of the connected elements. Therefore, the processes (ports) do not need to know the name of the receivers or senders.

The formalization of an attachment process is given according to the following classification:

- Attachments between Architectural Elements in the same Ambient
- Attachments between the parent architectural element and its children.

In the following, we present how we formalized an attachment that connects two sibling architectural elements. An attachment of this kind can connect a port of a component to a port of a sibling connector, or a port of an ambient to a port of another sibling ambient. In this case, an attachment is a process that exists with other sibling ambients as follows:

$$
a[\text{IdAE1} [P_{AE1}] | ... | P_{ATCH} | \text{IdAE2} [P_{AE2}] | ...]
$$
where \( Id_{AE1} \) and \( Id_{AE2} \) are the ambients which represent the architectural elements and \( a \) is the ambient where these architectural elements are executing.

The process that models an attachment when it receives a service from a client architectural element is shown below:

\[
P_{ATCH}(Id_{AE1}, Id_{AE2}, s) = \left\{ \begin{array}{ll}
\rightarrow & \rightarrow \\
(\rightarrow x, r, \rightarrow y) & (Id_{AE1}/\rightarrow y) (s_1 (x, \rightarrow y)). \\
\rightarrow y (y_0, \ldots, y_m) & ((Id_{AE1}/\rightarrow x) (y_0, \ldots, y_m)). \\
+ \ldots +
\rightarrow x, r, \rightarrow y) & (Id_{AE2}/\rightarrow x) (s_2 (x, \rightarrow y)). \\
\rightarrow y (y_0, \ldots, y_m) & ((Id_{AE2}/\rightarrow y) (s_2 (x, \rightarrow y)).
\end{array} \right.
\]

where \( Id_{AE1} \neq Id_{AE2} \) are names of architectural elements, and \( s \) is a tuple of services that the ports of the architectural elements publish.

The \( P_{ATCH} \) process listens to the \( s_n \) channel in order to receive the \( input \) parameters and the names of the channels that are needed to return the \( output \) parameters. Once, it receives this data, it creates a channel (\( \rightarrow y \)) to receive the \( output \) parameters (or the acknowledgements). Then, it sends the information to the server architectural element (\( Id_{AE2} \)), which is a ChAC ambient. It then receives the \( output \) parameters and sends them to the \( Id_{AE1} \) ambient.

However, since an attachment in Ambient-PRISMA is bidirectional (i.e. it represents the communication when \( Id_{AE1} \) is a client and also when \( Id_{AE2} \) is a client), the attachment process is modeled as follows:

\[
P_{ATCH}(Id_{AE1}, Id_{AE2}, s) = P_{ATCH}(Id_{AE2}, Id_{AE1}, s) = \left\{ \begin{array}{ll}
\rightarrow & \rightarrow \\
(\rightarrow x, r, \rightarrow y) & (Id_{AE2}/\rightarrow x) (s_2 (x, \rightarrow y)). \\
\rightarrow y (y_0, \ldots, y_m) & ((Id_{AE2}/\rightarrow y) (s_2 (x, \rightarrow y)).
\end{array} \right.
\]

\[
ATT1 = \text{new CustCnctrAtt(Customer1, AgentsPort, CustAgentPort, AgentCnct)};
\]

F. Architectural Model Configuration

An architectural model configuration defines a specific software architecture by instantiating types of ambients, components, and connectors and indicating where the instances are located. Then, attachments instances are created in order to connect architectural element instances together. The user only creates attachment instances that connect components’ to connectors’ instances.

In Ambient-PRISMA, an architectural model configuration \( Id_{AM} \) is represented by an ambient called \( Root \) which is defined as follows:

\[
Id_{AM}(Id_{new}) = \left\{ \begin{array}{ll}
\rightarrow & \rightarrow \\
Id_{new} (\rightarrow Amb, \rightarrow Comp, \rightarrow Con, \rightarrow ATCH, \rightarrow Asp, \rightarrow W, \rightarrow Port)
\end{array} \right.
\]

shows a fragment of the AOADL that specifies the MobileAgentsAuctionConf configuration. It can be also observed that each ambient (\( Root, ClientSite, \) and \( AuctionSite, \) components (\( Customer1, Bidder1, \) connectors, and attachments are instances. The parent ambient is assigned at the creation of each element. For example, the \( Customer1 \) component has \( ClientSite \) as its parent ambient.

Architectural Model Configuration

MobileAgentsAuctionConf =

\[
\text{New MobileAgentsAuction} \\
\{ \begin{array}{l}
\rightarrow Root = \text{new Root}();
\rightarrow ClientSite = \text{new HostSite(Root, IP1)};
\rightarrow AuctionSite = \text{new HostSite(Root, IP2)};
\rightarrow Customer1 = \text{new Customer("ClientSite")};
\rightarrow Bidder1 = \text{new Bidder("ClientSite")};
\rightarrow AgentCustCnct1 = \text{new AgentCustCnct("ClientSite")};
\rightarrow AuctionHouseCnct1 = \text{new AuctionHouseCnct("AuctionSite")};
\rightarrow AttachAuctCnct1 = \text{new AttachAuctCnct(AuctionCnct1, CnctAuctPortBidder, AuctionHouse1, BidderAuctPort)};
\rightarrow AttachCustAuct1 = \text{new AttachCustAuct(Customer1, CUSTAUCTPort, AuctionCnct1, CustPortAuct)};
\rightarrow ...
\end{array} \}
\]

Figure 10. Specification of the MobileAgentsAuctionConf configuration

The \( MobileAgentsAuctionConf \) configuration is formalized as follows:

For example, using the Ambient-PRISMA AOADL this can be instantiated into the attachment between the \( Customer1 \) component and the \( AgentCustCnct \) connector when they are located in the \( ClientSite \) ambient as follows (see Figure 9):

\[
\text{New ClientSite} \\
\{ \begin{array}{l}
\rightarrow Customer1 = \text{new Customer1("ClientSite")};
\rightarrow Bidder1 = \text{new Bidder1("ClientSite")};
\rightarrow AgentCustCnct1 = \text{new AgentCustCnct1("ClientSite")};
\rightarrow AuctionHouseCnct1 = \text{new AuctionHouseCnct1("AuctionSite")};
\rightarrow AttachAuctCnct1 = \text{new AttachAuctCnct1(AuctionCnct1, CnctAuctPortBidder, AuctionHouse1, BidderAuctPort)};
\rightarrow AttachCustAuct1 = \text{new AttachCustAuct1(Customer1, CUSTAUCTPort, AuctionCnct1, CustPortAuct)};
\rightarrow ...
\end{array} \}
\]

Figure 9. Customer1 component and AgentCustCnct connector connected in ClientSite ambient
Darwin [14]. The Darwin Architecture Description software architectures for distributed systems is known as concurrent and interacting systems. Process algebra provides abstract mechanisms for describing formalization of software architecture. This is because as a change in a value.

architecture. However in Community, mobility is modeled as a reconfiguration of the software components to model locations. In most approaches, Moto provides special kinds of components called physical composite components for representing locations. Con-

of a variable. Other approaches such as LAM-Model use a data type where a location is represented by a value uniquely. For example, Community models locations and mobility in a unique way. For example, Community models locations using a data type where a location is represented by a value of a variable. Other approaches such as LAM-Model use composite components for representing locations. Con-Moto provides special kinds of components called physical components to model locations. In most approaches, mobility is modeled as a reconﬁguration of the software architecture. However in Community, mobility is modeled as a change in a value.

Process algebras have been widely used in the formalization of software architecture. This is because process algebra provides abstract mechanisms for describing concurrent and interacting systems.

One of the first approaches for the specification of software architectures for distributed systems is known as Darwin [14]. The Darwin Architecture Description Language has used \(\pi\)-calculus to deﬁne its semantics. This allows Darwin to deﬁne both static and dynamic structures. Darwin is also able to specify and implement distributed systems. However, since it is based on \(\pi\)-calculus, mobility can only be simulated by the movement of channels (the type of mobility that \(\pi\)-calculus offers). It lacks primitives to express the movement of entities that cross boundaries. Darwin has also used the Finite State Processes (FSP) to support its behavioural view. Tools have been developed in order to analyze and animate the FSP speciﬁcations.

\(\pi\)-ADL is another architectural approach that is based on \(\pi\)-calculus for deﬁning its semantics for mobility [15]. It simulates mobility of software architectural elements by dynamically deleting a subcomponent from a composite component and adding it to another component.

All the above software architecture approaches do not provide an explicit notion of location and mobility as Ambient-PRISMA. As a result, another kind of process algebra had to be used. Initially, our ﬁrst approach was to formalize Ambient-PRISMA with AC. However, the formalization with AC was very low level and complex since AC only provides the ambient primitive and channels had to be simulated with ambients. Formalizing Ambient-PRISMA with ChAC is appropriate since ChAC includes primitives appropriate for current distributed networks.

V. RELATED WORK

Many software architectural approaches are based on formal and mathematical notations to deﬁne their semantics and provide development based support [7]. Different kinds of formalisms have been used focusing on speciﬁc aspects and characteristics of the software architecture. For example, the Wright Architecture Description Language (ADL) [10] has uses CSP to specify components and connectors behaviour. Wright provides a tool to perform model checking of the speciﬁcations. However, since CSP is only suitable for specifying static software architectures, further extensions have to be made for dynamic architectures.

Some architectural approaches have modelled mobility in a technology independent way, such as Community [10], which is based on category theory; Con-Moto [12] based on \(\pi\)-calculus; and LAM model [13] based on Petri nets. Each one of these approaches models locations and mobility in a unique way. For example, Community models locations using a data type where a location is represented by a value of a variable. Other approaches such as LAM-Model use composite components for representing locations. Con-Moto provides special kinds of components called physical components to model locations. In most approaches, mobility is modeled as a reconfiguration of the software architecture. However in Community, mobility is modeled as a change in a value.

Process algebras have been widely used in the formalization of software architecture. This is because process algebra provides abstract mechanisms for describing concurrent and interacting systems.

One of the first approaches for the speciﬁcation of software architectures for distributed systems is known as Darwin [14]. The Darwin Architecture Description Language has used \(\pi\)-calculus to deﬁne its semantics. This allows Darwin to deﬁne both static and dynamic structures. Darwin is also able to specify and implement distributed systems. However, since it is based on \(\pi\)-calculus, mobility can only be simulated by the movement of channels (the type of mobility that \(\pi\)-calculus offers). It lacks primitives to express the movement of entities that cross boundaries. Darwin has also used the Finite State Processes (FSP) to support its behavioural view. Tools have been developed in order to analyze and animate the FSP speciﬁcations.

\(\pi\)-ADL is another architectural approach that is based on \(\pi\)-calculus for deﬁning its semantics for mobility [15]. It simulates mobility of software architectural elements by dynamically deleting a subcomponent from a composite component and adding it to another component.

All the above software architecture approaches do not provide an explicit notion of location and mobility as Ambient-PRISMA. As a result, another kind of process algebra had to be used. Initially, our ﬁrst approach was to formalize Ambient-PRISMA with AC. However, the formalization with AC was very low level and complex since AC only provides the ambient primitive and channels had to be simulated with ambients. Formalizing Ambient-PRISMA with ChAC is appropriate since ChAC includes primitives appropriate for current distributed networks.

VI. CONCLUSIONS AND FURTHER WORK

In this paper, the concepts of Ambient-PRISMA have been formalized. The primitives that provide distribution and mobility of architectural elements have been mapped into ChAC. Ambient-PRISMA components, connectors and ambients can be mobile architectural elements that move from an ambient to another. In addition, these elements can communicate. A summary of formalization is the following:

- components and connectors are mapped into ChAC ambients
- ports are mapped into ChAC processes that can send and receive inputs/outputs to or from processes outside ambients.
- ambients of Ambient-PRISMA are mapped into ChAC ambients which can accept and release ambients by creating channels for mobility.
- attachments are mapped into ChAC processes that can send and receive inputs/outputs to or from processes inside ambients.

This provides Ambient-PRISMA speciﬁcations of the software architectures of distributed and mobile software to have a proper formal semantics. This formalization can prepare the ground for tool support for semantically and syntactically analyze and validate the speciﬁcations.

In the near future, we are going to use this formalization alongside the existing MDE based CASE Tool. The graphical models can be automatically generated into the ChAC speciﬁcations. This will be used to verify the correctness of applications modelled in Ambient-PRISMA and model check Ambient-PRISMA speciﬁcations.

We plan also to extend Ambient-PRISMA with security primitives and study how ChAC or some variants can be used to deﬁne an architectural approach based on ambients for security.

ACKNOWLEDGMENT

The work presented in this paper has been possible thanks to a grant awarded PD106/2608 to Nour Ali by the Polytechnic University of Valencia to perform a research stay at the University of Leicester. The work has also been partially supported by Science Foundation Ireland grant 10/CE/I1855.
REFERENCES


