Saarland University
Computer Science

Seminar
Concurrency Theory

Mobile Ambients

Calculi

Author:
Max-Ferdinand Gerhard Suffel

Supervisor:
Luis María Ferrer Fioriti

Wintersemester - March 6, 2012

0 Introduction

Today it's all about *mobility*. In case of information technologies the most simple mobile entity we could imagine is moving data. Chunks of information, we share all around the world. Since the introduction of the World Wide Web in the '80s this became an elementary thing to our today's life. To access these information everywhere we developed laptops, tablets, smartphones, i.e., mobile computers. Hence, this is the physical part of mobility and therefore called *mobile computing*. Let's consider some situation in which you work on a text document with your laptop while travelling by a plane. After some time you expect to send a draft of that document to some colleague at the office. Because of the benefits of mobility you share the editor including the text file. Hence, you exchange mobile code and data. This would be *mobile computation*, i.e., the logical part of mobility. Technically this is possible today, e.g. Java applets. But the underlying system architectures are still not well understood and therefore in research. One aspect to consider is the difficulty of *administrative domains*. That means, nowadays the internet is not a flat domain. Moreover it is partitioned by firewalls into a huge bundle of (administrative) domains. Hence, data and of course executable code is not always allowed to access or leave an arbitrary location, i.e., the firewall blocks this or some code is only running on a local computer without network access. In the following section we will study a mobility calculi named **Mobile Ambients** developed by Luca Cardelli and Andrew D. Gordon [3]. We will use this calculi to model mobile systems in such a way that mobile computations are self-contained nested environments consisting of data and live computation [3, p.178]. Since we expect communication especially when mobile code interacts it should be technically required that only information of some common type will be shared. Hence, we need *type safety* [1] which is described in section 2. Although we are going to describe mobile systems in a formal way, we are not able to assert easily any runtime behaviour of the total system. For example, assertions which include a certain behaviour over *time*. To achieve this we study a *modal logic* [2] for mobile ambients to apply some basic model checking. This is covered in section 3.

0.1 Ambients

Mobile computation happens at a bounded and discrete place, the so called *ambient*. Hence, that allows us to distinguish between the in- and outside of an ambient. This is important because we want to move ambients. As an example think about a Java applet (ambient) which is bounded by a file and stored in a file system on a hard drive of a certain computer (parent ambient) will be transferred to another computer (top-level ambient) to be executed. This leads us to the structure of an ambient. Each ambient therefore consists of other *subambients* which coincides to the aspect of administrative domains. To enter and leave an ambient some other ambient needs the *capabilities* to do so, e.g., needs a password to access a firewall ambient. This corresponds to the safety aspect of domains. To achieve this an ambient has an unique local *name* we can extract its capabilities from. Furthermore it can be used to create and name new ambients or passed around by interaction. Since we talk about computation each ambient also consists of *agents*, i.e., processes, threads, logics which control the behaviour of the surrounding ambient. Hence, an ambient has the following structure:

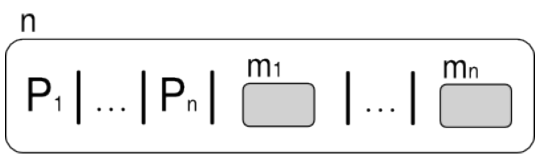

Figure 1: General structure of an ambient n where $P_1, \dots, P_n$ denote processes and $m_1, \dots, m_n$ are sub-ambients.

1 Ambient calculi

In this section we will see how to describe mobile systems with the ambient calculi. We begin with some syntactic notations and continue with their semantics. Since we expect to describe movement of mobile computation between administrative domains some kind of synchronisation like locks or authentication mechanism between ambients are our first goal. We shall see that the calculi is Turing-complete which is done by a direct encoding of Turing-Machines. That gives us the intuition that the calculi is quite powerful. In the end we look at some basic communication techniques, i.e., cells, records, channels.

1.1 Syntax

Let's start with a first look at the concrete syntax of the ambient calculi. There are two elementary syntactic categories: *processes* and *capabilities*. Both form in combination the mobility and communication primitives of mobile systems as seen in the calculi.

1.1.1 Processes

As long as we talk about mobile systems, we have to capture concurrent processes. A process therefore describes a behaviour of some part of the total system independently of all other counterparts. The most simple process is the process 0 which does nothing. Based on this process we will form new processes as a combination of unary and binary operators as in the following.

Definition 1.1 (Processes)**.**
P, Q ::=

0	*(Inactivity)*
$P\|Q$	*(Composition)*
$!P$	*(Replication)*
$(vn)P$	*(Restriction)*
$M[P]$	*(Ambient)*
$M.P$	*(Capability action)*
$(x).P$	*(Input action)*
$\langle M\rangle$	*(Async output action)*

1.1.2 Capabilities

Since we organized ambients in a hierarchical manner which can move and therefore change the structure of the hole system we need basic capabilities like *in*, *out* or *open* to describe the natural behaviour of crossing named domains, i.e., ambients. The definitions are given below.

Definition 1.2 (Capabilities)**.**
M ::=

$in\, M$	*(Can enter into M)*
$out\, M$	*(Can exit out of M)*
$open\, M$	*(Can open M)*
x	*(Variable)*
n	*(Name)*
ε	*(Null)*
$M.M'$	*(Path)*

☞ When defining a process $M[P]$ we restrict ourselves to instantiate M by a name or a variable. Similarly in a capability action $M.P$, M should be restricted to Null, Path, Entry, Exit and Open primitives. Otherwise syntactic anomalies may occur.

For simplicity there are some syntactic conventions and abbreviations given:

$(vn)P\|Q$	$\triangleq$	$((vn)P)\|Q$
$!P\|Q$	$\triangleq$	$(!P)\|Q$
$M.P\|Q$	$\triangleq$	$(M.P)\|Q$
$(vn_1 \dots n_m)P$	$\triangleq$	$(vn_1)\dots(n_m)P$
$(x).P\|Q$	$\triangleq$	$((x).P)\|Q$
$n[]$	$\triangleq$	$n[0]$
M	$\triangleq$	$M.0$

1.2 Semantic

Next we define the operational semantics of the primitives. This is done implicitly by reduction rules defining a reduction relation $\rightarrow$. Since we do not plan to distinguish processes up to their syntactic structures we enhance the $\rightarrow$ by a structural congruence $\equiv$, i.e., forming syntactic equivalence classes. This gives us more freedom when describing mobile systems.

1.2.1 Free names and variables

Before we continue with the semantic we first look at names and variables as used in the definition of the capabilities in (1.2). Names in the way we use them are identifiers for ambients to extract the capabilities. But they are not unique. It is possible that some process iteratively produces ambients but all share the same name. Imagine a web service which is distributed to several web servers offering the same service. Therefore a name can have many *occurrences* in a process. An occurrence of a name n is *bounded* if it appears inside an expression of the form $(vn)P$. An occurrence is *free* if it is not bounded. As an example consider the process $(vn)n[P]|n[Q]$. On the left hand of the parallel composition n is bounded, on the right hand it is free. Hence, a name n is free in a process if there is at least one free occurrence of n. Otherwise n is bounded. The set of free names of a process is inductively defined through the function fn see definition (1.3).

Definition 1.3 (Free names)**.**

$fn(0)$	$\triangleq$	$\emptyset$
$fn(P\|Q)$	$\triangleq$	$fn(P) \cup fn(Q)$
$fn(!P)$	$\triangleq$	$fn(P)$
$fn((vn)P)$	$\triangleq$	$fn(P) - \{n\}$
$fn(M[P])$	$\triangleq$	$fn(M) \cup fn(P)$
$fn(M.P)$	$\triangleq$	$fn(M) \cup fn(P)$
$fn((x).P)$	$\triangleq$	$fn(P)$
$fn(\langle M \rangle)$	$\triangleq$	$fn(M)$
$fn(in\, M)$	$\triangleq$	$fn(M)$
$fn(out\, M)$	$\triangleq$	$fn(M)$
$fn(open\, M)$	$\triangleq$	$fn(M)$
$fn(x)$	$\triangleq$	$\emptyset$
$fn(n)$	$\triangleq$	$\{n\}$
$fn(\varepsilon)$	$\triangleq$	$\emptyset$
$fn(M.M')$	$\triangleq$	$fn(M) \cup fn(M')$

We define $P\{n \leftarrow m\}$ to denote the substitution of all free occurrences of the name n in process P with m.
When a process sends out a value and some other process asynchronously receives it they communicate by value passing. To store the received value we need variables. So far we do not distinguish variable types which means it is allowed to receive arbitrary values, i.e., names or capabilities. Variable can also be free or bounded which is defined in the same way as we did with names despite some changes. First, we replace fn by fv. Next, we change the definitions of these rules:

$fv((vn)P)$	$\triangleq$	$fv(P)$
$fv((x).P)$	$\triangleq$	$fv(P) - \{x\}$
$fv(x)$	$\triangleq$	$\{x\}$
$fv(n)$	$\triangleq$	$\emptyset$

And last we define $P\{x \leftarrow M\}$ to denote the substitution of all free occurrences of the variable x in process P with the capability M.

1.2.2 Structural congruence

As mentioned before we do not plan to distinguish processes up to their syntactic structures. This is defined by structural congruence as the following.

Definition 1.4 (Structural congruence $\equiv$)**.**

$P \equiv P$	*(Struct Relf)*
$P \equiv Q \Rightarrow Q \equiv P$	*(Struct Symm)*
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	*(Struct Trans)*
$P \equiv Q \Rightarrow (vn)P \equiv (vn)Q$	*(Struct Res)*
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	*(Struct Par)*
$P \equiv Q \Rightarrow !P \equiv !Q$	*(Struct Repl)*
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	*(Struct Amb)*
$P \equiv Q \Rightarrow M.P \equiv M.Q$	*(Struct Action)*
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	*(Struct Input)*
$P \mid Q \equiv Q \mid P$	*(Struct Par Comm)*
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	*(Struct Par Assoc)*
$!P \equiv P \mid !P$	*(Struct Repl Par)*
$(vn)(vm)P \equiv (vm)(vn)P$	*(Struct Res Res)*
$(vn)(P \mid Q) \equiv P \mid (vn)Q$ *if* $n \notin fn(P)$	*(Struct Res Par)*
$(vn)(m[P]) \equiv m[(vn)P]$ *if* $n \neq m$	*(Struct Res Amb)*
$P \mid 0 \equiv P$	*(Struct Zero Par)*
$(vn)0 \equiv 0$	*(Struct Zero Res)*
$!0 \equiv 0$	*(Struct Zero Repl)*
$\varepsilon.P \equiv P$	*(Struct ε)*
$(M.M').P \equiv M.M'.P$	*(Struct .)*

Processes then are identified up to renaming of bounded names and variables:

- $(vn)P = (vm)P\{n \leftarrow m\}$ if $m \notin fn(P)$
- $(x).P = (y).P\{x \leftarrow y\}$ if $y \notin fv(P)$

1.2.3 Reduction

Now we got all necessary definitions to specify the semantics of the full ambient calculi. We will do that by a step by step definition of a reduction relation $\rightarrow$.

- **Inactivity** denoted as 0 defines no reduction rule. Hence, it is a process that does nothing.
- **Composition** defines the parallel execution of two processes P, R:

  $$\frac{P \to Q}{P|R \to Q|R} \quad \text{(Red Par)}$$

  In this case the reduction is defined by reducing P to Q while R starves. Reducing R first can be achieved by congruence using (Struct Par Comm).
- **Replication** denoted as $!P$ is identified with $!P \equiv P|!P$ see (1.4)(Struct Repl Par). Hence, replication does not define any reduction rule but stands for iteration or recursion. It can be used to generate arbitrary many instances of process P which will then be executed in parallel denoted by the composition operator.
- **Restriction** can be used to make a name n in some process P unique. This is done by denoting $(vn)P$. Hence, all occurrences of n in $(vn)P$ are bounded ($n \notin fn(P)$). The rule is given by:

  $$\frac{P \to Q}{(vn)P \to (vn)Q} \quad \text{(Red Res)}$$
- **Ambients** are denoted as $M[P]$. Here, P is a process which controls the ambient. Generally P is a composition of two or more processes or ambients since we consider a hierarchical structure. The execution of P still happens even when the ambient moves, that's a design decision of the calculi and forms the following rule:

  $$\frac{P \to Q}{M[P] \to M[Q]} \quad \text{(Red Amb)}$$
- **Capability action** denoted as $M.P$ is a process which performs some action M before continuing with P. We will focus first on the three capabilities $in\, M$, $out\, M$ and $open\, M$.
  - **Entry** is used to define a process $in\, m.P$ which instructs the surrounding ambient n to enter a sibling ambient m. This is a blocking action. As long as no ambient m occurs in composition to n the process blocks.

    $$n[in\, m.P|Q]|m[R] \to m[n[P|Q]|R] \quad \text{(Red In)}$$
  - **Exit** denoted by a process $out\, m.P$ instructs its surrounding ambient n to leave the parent ambient m. Similar to (Entry) this blocks if n is a top-level ambient, i.e., not embedded into an ambient m.

    $$m[n[out\, m.P|Q]|R] \to n[P|Q]|m[R] \quad \text{(Red Out)}$$
  - **Open** is a capability to dissolve the boundary of an ambient m which appears in composition to an ambient n. It blocks as usual if m is not present. Since, the capability is given out by m itself, the operation is well defined despite what Q does.

    $$open\, m.P|m[Q] \to P|Q \quad \text{(Red Open)}$$
- **Communication** is provided by a local asynchronous value passing mechanism. Using the input and output actions this can be described by $(x).P|\langle M\rangle$. Hence, no long-range communication is possible but that's no a pitfall since we will define channels later on, see section 1.4. The reduction rule is given by:

  $$(x).P|\langle M\rangle \to P\{x \leftarrow M\} \quad \text{(Red Comm)}$$

  Hence, the capability M is bound to the variable x and all free occurrences of x in P are replaced by M. To sent multiple values one can build a *path* of capabilities denoted by $M.M'$. As a path delimiter one can use the empty path ε.

- **Equivalence** allows us to use the structural congruence $\equiv$ (see definition (1.4)) when applying reductions. This implies a rule:

$$\frac{P' \equiv P, P \to Q}{P' \to Q} \quad (\text{Red} \equiv)$$

Finally to argue over chains of reductions we define the reflexive and transitive closure of the reduction relation as $\to^*$, i.e., $P \equiv P_1 \to P_2 \dots P_{n-1} \to P_n \equiv Q$ which is similar to $P \to^* Q$.

1.3 Expressiveness

So far we only saw *how* we can build ambient calculi expression in a formal way. Now let's focus on the expressiveness power and answer the question: *What can we describe?*

1.3.1 Locks

Some requirement for concurrent system environments are *locks*. The ambient calculi offers no synchronous hand-shake but we can define some *acquire* and *release* capabilities using the given mobility primitives.

$$\mathit{acquire}\, n.P \triangleq \mathit{open}\, n.P$$
$$\mathit{release}\, n.P \triangleq n[]|P$$

The acquire capability is the same as open. Release denotes a composition of an new ambient n with P. To define a hand-shake we need two locks n and m. Hence, processes synchronize in a cross-over manner:

$$\mathit{acquire}\, n.\mathit{release}\, m.P|\mathit{release}\, n.\mathit{acquire}\, m.Q$$

Until both locks are acquired and released, P and Q are blocked.

1.3.2 Authentication

In the beginning our motivation was to define a calculi supporting the feature of administrative domains. A possible scenario might be that an agent leaves its home and comes back. Since we are restrictive not every agent is allowed to enter. To authorize a certain agent we can use the restriction primitive to agree a password n. Formally the scenario looks like:

$$\mathit{Home}[(vn)(\mathit{open}\, n|\mathit{Agent}[\mathit{out}\, \mathit{Home}.\mathit{in}\, \mathit{Home}.n[\mathit{out}\, \mathit{Agent}.\mathit{open}\, \mathit{Agent}.P]])]$$

There is a top-level ambient Home which contains a sub-ambient Agent, i.e., the agent which will leave and enter again. First, the agent leaves the home ambient by executing $\mathit{out}\, \mathit{Home}$. After that he decides to come back with $\mathit{in}\, \mathit{Home}$. Since we want to continue later on with P we have to get rid off the agent's ambient applying $\mathit{out}\, \mathit{Agent}$. After that there is a password authentication. We simulate this by $\mathit{open}\, n$. At this moment illegal agents would be detected, i.e., the processes block. Last but not least there is some garbage work to do by removing the agent and finally we got the initial process P inside of its home ambient.

$$\begin{array}{ll}
\mathit{Home}[(vn)(\mathit{open}\, n|\mathit{Agent}[\mathit{out}\, \mathit{Home}.\mathit{in}\, \mathit{Home}.n[\mathit{out}\, \mathit{Agent}.\mathit{open}\, \mathit{Agent}.P]])] & \\
\equiv (vn)\mathit{Home}[\mathit{open}\, n|\mathit{Agent}[\mathit{out}\, \mathit{Home}.\mathit{in}\, \mathit{Home}.n[\mathit{out}\, \mathit{Agent}.\mathit{open}\, \mathit{Agent}.P]]] & (\text{Struct Res Amb}) \\
\to (vn)(\mathit{Home}[\mathit{open}\, n]|\mathit{Agent}[\mathit{in}\, \mathit{Home}.n[\mathit{out}\, \mathit{Agent}.\mathit{open}\, \mathit{Agent}.P]]) & (\text{Red Out}) \\
\to (vn)\mathit{Home}[\mathit{open}\, n|\mathit{Agent}[n[\mathit{out}\, \mathit{Agent}.\mathit{open}\, \mathit{Agent}.P]]] & (\text{Red In}) \\
\to (vn)\mathit{Home}[\mathit{open}\, n|n[\mathit{open}\, \mathit{Agent}.P]|\mathit{Agent}[]] & (\text{Red Out}) \\
\to (vn)\mathit{Home}[0|\mathit{open}\, \mathit{Agent}.P|\mathit{Agent}[]] & (\text{Red Open}) \\
\to (vn)\mathit{Home}[0|P|0] & (\text{Red Open}) \\
\equiv (vn)\mathit{Home}[P] & (\text{Struct Zero Par})
\end{array}$$

1.3.3 Objective movement

The ambient calculi as defined supports only subjective moves so far. That means, a process *inside* an ambient has the capability to make the ambient move. Now we enhance this framework by objective moves. These are as usually offered by capabilities.

$$
\begin{array}{lcll}
\mathit{allow}\, n & \triangleq & !\mathit{open}\, n & \\
n^{\downarrow}[P] & \triangleq & n[P|\mathit{allow}\, \mathit{enter}] & (n \text{ allows move in}) \\
n^{\uparrow}[P] & \triangleq & n[P]|\mathit{allow}\, \mathit{exit} & (n \text{ allows move out}) \\
n^{\downarrow\uparrow}[P] & \triangleq & n[P|\mathit{allow}\, \mathit{enter}]|\mathit{allow}\, \mathit{exit} & (n \text{ allows both move in and out}) \\
\mathit{mv}\, \mathit{in}\, n.P & \triangleq & (vk)k[\mathit{in}\, n.\mathit{enter}[\mathit{out}\, k.\mathit{open}\, k.P]] & \\
\mathit{mv}\, \mathit{out}\, n.P & \triangleq & (vk)k[\mathit{out}\, n.\mathit{exit}[\mathit{out}\, k.\mathit{open}\, k.P]] &
\end{array}
$$

☞ *enter* and *exit* are arbitrary names.
An objective move of a process P into an ambient n allowing *move in* then works like this:

$$
\begin{array}{ll}
\mathit{mv}\, \mathit{in}\, n.P|n^{\downarrow}[Q] & \\
\equiv (vk)(k[\mathit{in}\, n.\mathit{enter}[\mathit{out}\, k.\mathit{open}\, k.P]]|n[Q|\mathit{allow}\, \mathit{enter}]) & (\text{Def. } \mathit{mv}\, \mathit{in}, \text{ Def. } n^{\downarrow}) \\
\rightarrow (vk)n[k[\mathit{enter}[\mathit{out}\, k.\mathit{open}\, k.P]]|Q|\mathit{allow}\, \mathit{enter}] & (\text{Red In}) \\
\rightarrow n[(vk)(k[\mathit{enter}[\mathit{out}\, k.\mathit{open}\, k.P]]|Q|\mathit{allow}\, \mathit{enter})] & (\text{Res} \equiv) \\
\rightarrow n[(vk)(k[]|\mathit{enter}[\mathit{open}\, k.P]|Q|\mathit{allow}\, \mathit{enter})] & (\text{Red Out}) \\
\rightarrow n[(vk)(k[]|\mathit{enter}[\mathit{open}\, k.P]|Q)|\mathit{allow}\, \mathit{enter}] & (\text{Red} \equiv) \\
\rightarrow n[(vk)(k[]|\mathit{enter}[\mathit{open}\, k.P]|Q|\mathit{open}\, \mathit{enter})|\mathit{allow}\, \mathit{enter}] & (\text{Red} \equiv, \text{ Def. } \mathit{allow}) \\
\rightarrow n[(vk)(k[]|\mathit{open}\, k.P|Q|0)|\mathit{allow}\, \mathit{enter}] & (\text{Red Open}) \\
\rightarrow n[(vk)(0|P|Q|0)|\mathit{allow}\, \mathit{enter}] & (\text{Red Open}) \\
\rightarrow n[0|P|Q|0|\mathit{allow}\, \mathit{enter}] & (\text{Red} \equiv) \\
\rightarrow n[P|Q|\mathit{allow}\, \mathit{enter}] & (\text{Red} \equiv) \\
\equiv n^{\downarrow}[P|Q] & (\text{Def. } n^{\downarrow})
\end{array}
$$

1.3.4 Choice

Sometimes one may need a choice operator to either continue with some process P or Q. This can be defined as:

$$n \Rightarrow P + m \Rightarrow Q \triangleq (v\, p\, q\, r)(p[\mathit{in}\, n.\mathit{out}\, n.q[\mathit{out}\, p.\mathit{open}\, r.P]]|p[\mathit{in}\, m.\mathit{out}\, m.q[\mathit{out}\, p.\mathit{open}\, r.Q]]|\mathit{open}\, q|r[])$$

Until no ambient n or m is present the process blocks. To check for their presence the ambient p tries to move in and out of n for example. If that works a $\mathit{out}\, p$ and $\mathit{open}\, q$ follows. To block the execution of P or Q during the reduction of the choice there exist an additional ambient r which will later be removed. As an example consider the following:

$$(n \Rightarrow P + m \Rightarrow Q)|n[R] \rightarrow^{*} P|n[R]$$

With that it is possible to define *booleans*. Let $flag\, n \triangleq n[]$ denote some arbitrary flag, e.g, $flag\, \mathrm{tt}$ for true and $flag\, \mathrm{ff}$ for false. Then we use the choice operator to define a conditional:

$$if\, \mathrm{tt}\, P,\, if\, \mathrm{ff}\, Q \triangleq \mathrm{tt} \Rightarrow \mathit{open}\, \mathrm{tt}.P + \mathrm{ff} \Rightarrow \mathit{open}\, \mathrm{ff}.Q$$

1.3.5 Turing-Machines

Next we show that the ambient calculi is Turing-complete. The tape consists of squares which are ambients. The outermost square is called *end*. Inside *end* there is an ambient which represents its value. Since we want a tape there is an embedded sub-square called *sq*. It has the same structure as *end*. Hence, the tape is designed in hierarchical structure. We assume that all initial values of the squares are the $flag$ ff. The tape then looks like that:

$$end^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]|sq^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]|sq^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]|sq^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]|\ldots]]]]$$

Since a Turing-Machine needs a head to read and modify the cells we define an ambient *head* which contains the program of the machine. As an example consider this concrete head:

$$\begin{aligned} head &\triangleq head[init|transitions] \quad \text{with} \\ init &\triangleq S_1[] \\ transitions &\triangleq !open\,S_1.mv\,out\,head.if\;\mathrm{tt}(\mathrm{ff}[]|mv\,in\,head.in\,sq.S_2),\;if\;\mathrm{ff}(\mathrm{tt}[]|mv\,in\,head.out\,sq.S_3[])|\ldots \end{aligned}$$

Hence, the initial state of the Turing-Machine is represented by the presence of an ambient S_1. The transitions are compositions of processes controlling the head. To achieve that each transition can be used everytime when possible we use the replication operator. The transition given in the example is only applicable in state S_1, i.e., this is checked by $open\,S_1$. Next the process moves out of the head and reads the value. If it is true it will be replaced by a false flag. After that the process moves back into the head and steps further to the square *sq* which means a move to the right. Hence, the head is in state S_2. If the value was false, it works similarly. Moving the head to left is then stepping out of the surrounding square *sq*.

Turing-Machines are as usually defined with an unbounded tape. To design this we need two tape stretchers for the left and right end. They are called $stretchRht$ and $stretchLft$ and defined like that:

$$\begin{aligned} stretchRht &\triangleq (vr)r[!open\,it.mv\,out\,r.(sq^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]]|mv\,in\,r.in\,sq.it[])|it[]] \\ stretchLft &\triangleq !open\,it.mv\,in\,end.(mv\,out\,end.end^{\downharpoonleft\upharpoonright}[sq^{\downharpoonleft\upharpoonright}[]|\mathrm{ff}[]]| \\ &\qquad in\,end.in\,sq.mv\,out\,end.open\,end.mv\,out\,sq.mv\,out\,end.it[])|it[] \end{aligned}$$

Finally the Turing-Machine is encoded as:

$$machine \triangleq stretchLft|end^{\downharpoonleft\upharpoonright}[\mathrm{ff}[]|head|stretchRht]$$

1.4 Communication

In section (1.2.3) we saw how the ambient calculi supports asynchronous communication by value passing between two processes. Next we will define some kind of data structures to store values. From simple cells we continue to records which store multiple values and finally channels. All of these implement shared memory which can be used to share values among a set of processes.

1.4.1 Cells

A cell is a data structure storing only one value. The content of the cell can either be replaced by a new one (*set* operation) or simply read (*get* operation) which happens synchronously. We denote a cell as:

$$cell\,c\,v \triangleq c^{\downharpoonleft\upharpoonright}[\langle v\rangle]$$

Hence, c is the location and v the initial value. The *set* operation $set\,c\langle v\rangle.P$ first implies a objective move into the cell c. This is followed by an input action which reads the current value of the cell and leaves an output action representing the new value v. Finally to continue with P we need an objective output move.

$$set\,c\langle v\rangle.P \triangleq mv\,in\,c.(x).(\langle v\rangle|mv\,out\,c.P)$$

Similarly works the *get* operation. First we move into the cell and read the current value which is then bounded to x. Since we do not want to overwrite the cell's value, we leave an output action with x inside the cell before moving out. Hence, we refreshed the value of the cell.

$$get\, c(x).P \triangleq mv\, in\, c.(x).(\langle x\rangle | mv\, out\, c.P)$$

1.4.2 Records

One may store multiple values in a data structure which supports get and set operations like the cells. To achieve that we define a bundle of cells and name this a record.

$$record\, r(l_1 = v_1 \dots l_n = v_n \triangleq r^{\downharpoonleft\upharpoonleft}[cell\, l_1 v_1 | \dots | cell\, l_n c_n]$$

Hence, to read a value out of a concrete cell l we first move into the record r and then apply the usual get operation for cells. Same for set.

$$getr\, r\, l(x).P \triangleq mv\, in\, r.get\, l(x).mv\, out\, r.P$$

$$setr\, r\, l\langle v\rangle.P \triangleq mv\, in\, r.set\, l\langle v\rangle.mv\, out\, r.P$$

1.4.3 Channels

Mobile systems often use communication links ($\triangleq$ channel buffers) to communicate asynchronously over long distances. It is possible to simulate such a channel by an named ambient n. Hence, channel communication is defined as local communication inside a common location. First a channel buffer is given as:

$$buf\, n \triangleq n[!open\, io]$$

Channel input and of course output requests represented through an ambient *io* then are directly processed inside the buffer n.

$$\begin{aligned} n(x).P &\triangleq (vp)(io[in\, n.(x).p[out\, n.P]] | open\, p) \\ n\langle M\rangle &\triangleq io[in\, n.\langle M\rangle] \end{aligned}$$

For example some output request $n\langle M\rangle$ first enters the buffer and then waits until some input request $n(x).P$ follows. To open the requests *open io* capability actions are used. Since this happens a lot of times it is designed as replication. After value passing the output request is removed. To hint P continuing its execution it is embedded in an ambient p which has to exit the buffer and then be opened.

2 Type safety for ambients

The ambient calculus describes mobile computation as ambients consisting of other sub-ambients and processes. Ambients are allowed to enter or exit others which implies mobility. Processes therefore control the movement. Additionally when processes share the same ambient, message exchanges are possible which is the communication part of the calculus. As opposed to movement, which is in generally only restricted by the given capabilities, value passing is arbitrary. Imagine some process which sends an integer value but its counterpart awaits a boolean. This is a dangerous behaviour and implies the system to crash. To avoid such situations we extend the calculus by type information, for example:

$$a[(x : Int).P | open\, b] | b[in\, a.\langle 3\rangle] \quad \rightarrow^* \quad a[P\{x \leftarrow 3\}]$$

Hence, this reduction is only possible whether a and b exchange values of same type which is *type safety*. Since the syntax of the calculus only allows the exchange of names and capabilities (or paths of them) we will focus on that, see section (1.1). Furthermore the syntax allows certain misleading processes (e.g. $in\, n[0]$) which should be detected as errors.

2.1 Polyadic ambient calculi

Before we continue with the type system we enhance the ambient calculi by polyadic input/output actions, i.e., instead of single values (or paths) we exchange tuples of values. We do this because it strictly enforces the need for type safety. The syntax therefore is similar to the basic ambient calculus despite the fact that we added the exchange of tuples and annotate types especially when applying the restriction operator since it creates new names for ambients which allow internal communication by a certain type.

Definition 2.1 (Processes).
$P, Q ::=$

0	*(Inactivity)*
$P \vert Q$	*(Composition)*
$!P$	*(Replication)*
$(vn : W)P$	*(Restriction)*
$M[P]$	*(Ambient)*
$M.P$	*(Capability action)*
$(n_1 : W_1, \ldots, n_k : W_k).P$	*(Input action)*
$\langle M_1, \ldots, M_k \rangle$	*(Async output action)*

Definition 2.2 (Capabilities).
$M \quad ::=$

$in\, M$	*(Can enter into M)*
$out\, M$	*(Can exit out of M)*
$open\, M$	*(Can open M)*
n	*(Name)*
ε	*(Null)*
$M.M'$	*(Path)*

Note the missing variable definition in (2.2). So we will not distinguish lexically between names an variables. Hence, there exist only v-bound names (restriction operator) and free names which act as variables. Due the changes in the syntax we have to adapt the reduction rules (Red Comm) and (Red Res), compare that with section (1.2.3). All other rules behave the same.

$$\frac{P \to Q}{(vn : W)P \to (vn : W)Q} \quad \text{(Red Res)}$$

$$(n_1 : W_1, \ldots, n_k : W_k).P | \langle M_1, \ldots, M_k \rangle \to P\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\} \quad \text{(Red Comm)}$$

2.2 Type system

Since our motivation was to avoid that two processes exchange wrong messages we will now define the type system for the polyadic ambient calculus. Therefore we keep track of the *topic of conversation* [1, p.3]. That means, we annotate all processes with the type of messages they *may* exchange. Remark that an ambient does not exchange messages by itself. Instead all ambient names then are typed with the kind of messages the corresponding ambients *allow* to exchange in their internal boundaries. Other capabilities of course will also get typed. Enter's and Exit's type for example is arbitrary. Why is that the case? Well, they permit to perform subjective movements of ambients. Since ambients do not communicate with other processes when placed in parallel there is no need for any kind of restriction. For Open the things work differently here. It will be annotated with kind of messages exchanges it possibly unleashed. Remark that it removes the boundary of an ambient.

2.2.1 Types

There exist two classes of types defined as *message types* W and *exchange types* T. The former one is used to type ambient names and capabilities. The later one is needed for processes.

Definition 2.3 (Types)**.**

$W ::=$	*message types*	
	$Amb[T]$	*(Ambient name allows T exchange)*
	$Cap[T]$	*(Capability unleashs T exchange)*
$T ::=$	*exchange types*	
	Shh	*(No exchange)*
	$W_1 \times \ldots \times W_k$	*(Tuple exchange)*

For example:

- $n : Amb[Shh]$ some ambient name which allows no exchanges
- $open\, n : Cap[Shh]$ a capability which is harmless

2.2.2 Environment

Since miscellaneous types have to keep tracked we need an environment to store the type information of capabilities and processes. The empty environment will be denoted as ϕ. When binding a new name n to a type W in a given environment E we get a new environment E' defined as $E' := E\{n \leftarrow W\}$ if $n \notin dom(E)$. To look up a type in an environment we define three *judgements*:

- $E \vdash \diamond$
- $E \vdash M : W$
- $E \vdash P : T$

The first says that E is a *good* environment, i.e., everything is well-typed. The second looks up the capability (☞ names are capabilities) M in environment E and retrieves the message type W. The last one retrieves the exchange type T of a process P in E if possible.

2.2.3 Typing rules

Now we define the typing rules of the polyadic ambients calculus. Therefore we distinguish between the environment, the capability and process rules.

- **Environments** need to be consistent after mapping a name to a certain type. This behaves the same for the empty environment ϕ. Hence, this implies the rules:

$$\frac{}{\phi \vdash \diamond} \quad (\text{Evn}\,\phi)$$

$$\frac{E \vdash \diamond \quad n \notin dom(E)}{E\{n \leftarrow W\} \vdash \diamond} \quad \text{(Evn n)}$$

- **Capabilities** are typed with message types. Let's start with the name capability n. To get the type of n we have to look up the environment E. As a requirement n has to be mapped to a type before. The rules is given as follows:

  $$\frac{E' := E\{n \leftarrow W\} \vdash \diamond}{E' \vdash n : W} \quad \text{(Exp n)}$$

  The next rules are for paths. As defined in the basic ambient calculus (see section (1.1.2) it is possible to combine multiple capabilities to a path to exchange them as one message. Since an empty path ε does not unleash any message type it will be typed as $Cap[T]$ with some arbitrary exchange type T. For the path connector $M.M'$ we have to ensure that both parts M and M' are type the same. Hence, the rules look like that:

  $$\frac{E \vdash \diamond}{E \vdash \varepsilon : Cap[T]} \quad \text{(Exp}\,\varepsilon)$$

  $$\frac{E \vdash M : Cap[T] \quad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]} \quad \text{(Exp .)}$$

  The remaining capabilities are Enter,Exit and Open. Since types do not restrict subjective movement, *in* M and *out* M can be typed arbitrary, i.e., we do not keep track of their types. Hence, their rules are:

  $$\frac{E \vdash M : Amb[S]}{E \vdash in\,M : Cap[T]} \quad \text{(Exp In)}$$

  $$\frac{E \vdash M : Amb[S]}{E \vdash out\,M : Cap[T]} \quad \text{(Exp Out)}$$

  The *open* M capability is somehow special as we know because it removes the boundary of some ambient. Hence, the possibility to exchange some message of type T inside that ambient may be unleashed in the future. To keep track of that both the capability and the ambient are required to share the same type T. This is defined as the following rule:

  $$\frac{E \vdash M : Amb[T]}{E \vdash open\,M : Cap[T]} \quad \text{(Exp Open)}$$

- **Processes** are typed as following. Hence, the most simple process is an inactive process 0. Naturally this process does not exchange any message at all so it could be typed with *Shh*. But we do not want to restrict ourselves so the type is arbitrary.

  $$\frac{E \vdash \diamond}{E \vdash 0 : T} \quad \text{(Proc Zero)}$$

  For the parallel composition of two processes P and Q meaning $P|Q$ we have to ensure that both are typed the same. This implies the rule:

  $$\frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P|Q : T} \quad \text{(Proc Par)}$$

  The rule for replication is defined naturally as:

  $$\frac{E \vdash P : T}{E \vdash !P : T} \quad \text{(Proc Repl)}$$

  Next let us study a restricted process P, meaning $(\nu n : W)P$. The restriction operator was enhanced with a type annotation W, see definition (2.1). Since n is a name for some ambient that occurs in P, W is in generally a message type $Amb[T]$. Hence, we first map n to W and later on resolve the exchange type of P. This gives us the rule:

$$\frac{E\{n \leftarrow Amb[T]\} \vdash P : S}{E \vdash (\nu n : Amb[T])P : S} \quad \text{(Proc Res)}$$

An ambient $M[P]$ allows communication in its boundary. Itself does not exchange messages. So as a first observation its exchange type is arbitrary. But there can be a pitfall when arbitrary processes move into that ambient. They may exchange wrong messages. Since we typed the ambient name with $Amb[T]$ we can now restrict P to type T. So every process which wants to enter can guess what kind of messages will be exchanged inside. The formal typing rule is:

$$\frac{E \vdash M : Amb[T] \quad E \vdash P : T}{E \vdash M[P] : S} \quad \text{(Proc Amb)}$$

If we form a process $M.P$ that applies a capability action M we require that M is of capability type $Cap[T]$. This is because we need the exchange type T an open capability may unleash. Hence, this is a restriction to process P which has to be of type T.

$$\frac{E \vdash M : Cap[T] \quad E \vdash P : T}{E \vdash M.P : T} \quad \text{(Proc Action)}$$

Last but not least the typing rules for the input and output actions are missing. If we assume a process P which has the exchansge type $W_1 \times \ldots \times W_k$ then the process $(n_1 : W_1, \ldots, n_k : W_k).P$ may also exchanges messages of type $W_1 \times \ldots \times W_k$.

$$\frac{E\{n_1 \leftarrow W_1, \ldots, n_k \leftarrow W_k\} \vdash P : W_1 \times \ldots \times W_k}{E \vdash (n_1 : W_1, \ldots, n_k : W_k).P : W_1 \times \ldots \times W_k} \quad \text{(Proc Input)}$$

For an output process $\langle M_1, \ldots, M_k \rangle$ we state the rule:

$$\frac{E \vdash M_1 : W_1 \quad \ldots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ldots, M_k \rangle : W_1 \times \ldots \times W_k} \quad \text{(Proc Output)}$$

Obviously each message M_i has its own type W_i. Hence, the output process is typed with the crossproduct of all types W_i since we exchange tuples.

Finally we combine the reduction relation $\rightarrow$ with the type system since each well-typed process must be well-typed after a reduction and get the *subject reduction* property [1, p.4]:

$$\frac{E \vdash P : U \quad P \rightarrow Q}{E \vdash Q : U}$$

To demonstrate that the type system is strong enough to detect certain run-time errors consider the following syntactical possible process: $(\nu n : Amb[T])n.P$. When deriving its type we get a *type clash error*:

$$\dfrac{\dfrac{\phi\{n \leftarrow Amb[T]\} \vdash n : Cap[T] \quad \phi\{n \leftarrow Amb[T]\} \vdash P : T}{\phi\{n \leftarrow Amb[T]\} \vdash n.P : ?} \ \text{(Proc Action)} \ \lightning}{\phi \vdash (\nu n : Amb[T])n.P : ?} \ \text{(Proc Res)}$$

When applying the (Proc Action) rule, n is assumed to be typed as $Cap[T]$ but was typed as $Amb[T]$. Hence, a type clash error is detected. As a further example which contains movement and communication of processes, assume the environment:

$$E := \{w = Amb[Amb[Amb[Shh]]], n = Amb[Amb[Shh]], u = Amb[Shh]\}$$

and the process:

$$u[(v : Amb[Shh])v[]|open\, w]|w[in\, u|\langle n\rangle]$$

First we derive the types of $open\, w.0$ and $(n : Amb[Shh]).v[]$:

$$\dfrac{\dfrac{E\vdash w{:}Amb[Amb[Amb[Shh]]]}{E\vdash open\, w{:}Cap[Amb[Amb[Shh]]}\ \text{(Exp Open)} \quad \dfrac{E\vdash\diamond}{E\vdash 0{:}Amb[Amb[Shh]]}\ \text{(Proc Zero)}}{E \vdash open\, w.0 : Amb[Amb[Shh]]}\ \text{(Proc Action)}$$

$$\dfrac{\dfrac{E\{v{\leftarrow}Amb[Shh]\}\vdash v{:}Amb[Shh] \quad \dfrac{E\{v{\leftarrow}Amb[Shh]\}\vdash\diamond}{E\{v{\leftarrow}Amb[Shh]\}\vdash 0{:}Shh}\ \text{(Proc Zero)}}{E\{v{\leftarrow}Amb[Shh]\}\vdash v[]{:}Amb[Amb[Shh]]}\ \text{(Proc Amb)}}{E \vdash (v : Amb[Shh]).v[] : Amb[Amb[Shh]]}\ \text{(Proc Input)}$$

Hence, the type of their parallel composition is given as:

$$\dfrac{E \vdash (v : Amb[Shh]).v[] : Amb[Amb[Shh]] \quad E \vdash open\, w.0 : Amb[Amb[Shh]]}{E \vdash (v : Amb[Shh])v[]|open\, w : Amb[Amb[Shh]]}\ \text{(Proc Par)}$$

Finally, we get again a type clash error in the (Proc Amb) rule:

$$\dfrac{\dfrac{E\vdash u{:}Amb[Shh] \quad E\vdash(v{:}Amb[Shh])v[]|open\, w.0{:}Amb[Amb[Shh]]}{E\vdash u[(v{:}Amb[Shh])v[]|open\, w.0]{:}\,?}\ \text{(Proc Amb)}\ \lightning \quad E \vdash w[in\, u|\langle n\rangle] : ?}{E \vdash u[(v : Amb[Shh])v[]|open\, w.0]|w[in\, u|\langle n\rangle] : ?}\ \text{(Proc Par)}$$

3 Modal Logics for Mobile Ambients

In the previous sections we introduced the ambient calculi and studied how to model mobile system. Additionally we defined some kind of type-safety. For the following let's assume we have formalized a mobile system. Since we are not sure if everything works fine we want to check some specification properties to argue about the system. The most obvious thing we could do is doing a formal proof over the reduction relation. This works quite well as long as the system and the properties are simple, e.g. structural properties. But this technique seems to be too complex for properties like "eventually the exist some ambient n" or "always ambient n enters ambient m" that talk about the structure of the system over time. A more practical approach is the usage of a modal logic which provides formulas consisting of temporal and spatial operators. Hence, as an computational application we consider model checking which we shall see is decidable for some sub-logic.

3.1 Space and Time

In the definition of processes in (1.1), ambients are hierarchically organized. Let's assume a fixed state of our mobile system. Hence, each process consisting only of process 0, ambients, the replication and composition operator defines a *spatial configuration* over ambients or named locations. For example the process $a[b[0]]|c[0]$ defines a spatial configuration in which a and c are top-level ambients because of the composition operator. b instead is a nested ambient of a. Therefore talking about spatial configurations means talking about *space.* Since executing a process may instruct an ambient to move, executions change the spatial configurations. Hence, there exist an evolution of the initial spatial configuration over *time.* Hence, our logic should also consider spatial configurations and their evolutions. This leads to a modal logic over space and time.

3.2 Formulas

In the following we will only consider ambient calculi expressions without the restriction operator. This implies, there exist only public names for ambients. We do this because it simplifies the logic. The logical formulas are given by the following listing.

Definition 3.1 (Logical Formulas).
$\mathcal{A}, \mathcal{B}, \mathcal{C}$ ::=

$\mathbf{T}$	*(True)*
$\neg\mathcal{A}$	*(Negation)*
$\mathcal{A} \vee \mathcal{B}$	*(Disjunction)*
0	*(Void)*
$\eta[\mathcal{A}]$	*(Location)*
$\mathcal{A}\vert\mathcal{B}$	*(Composition)*
$\forall x.\mathcal{A}$	*(Universal quantification over names)*
$\Diamond\mathcal{A}$	*(Sometime modality)*
$\blacklozenge\mathcal{A}$	*(Somewhere modality)*
$\mathcal{A}@\eta$	*(Location adjunct)*
$\mathcal{A} \triangleright \mathcal{B}$	*(Composition adjunct)*

Note: η is a name or a variable.

The first three formulas define propositional logic. The next three ones are needed to specify spatial configurations. For example, 0 says there is nothing. $n[0]$ is a formula meaning there is an empty location n. Composition is used to describe contiguous locations, e.g. $n[0]|m[0]$. Additionally we have universal quantification over names to specify properties that should hold nevertheless which ambient we consider. Sometime and somewhere are our modality operators. The last two are used for security properties.

The set of free variables of a formula $\mathcal{A}$ is given as usually through a function $fv(\mathcal{A})$. The definition of fv is then straight forward. We have only to be careful with quantifiers since they bind variables, i.e., $fv(\forall x.A) = fv(A) - \{x\}$. Hence, we say a formula $\mathcal{A}$ is *closed* iff $fv(\mathcal{A}) = \emptyset$.

3.2.1 Satisfaction

Next we define a satisfaction relation $\models$. Informally we use this relation to express that some process P satisfies a formula $\mathcal{A}$ and denote it as $P \models \mathcal{A}$. Hence, $\models$ gives us a truth value when evaluating $\mathcal{A}$ in P. Before we study how exactly $\models$ is defined, let's note that the meaning of our modalities will be defined over reductions of the operational semantics given by the ambient calculus. Hence, for the somewhere modality $\blacklozenge$ we need a *sub-location* relation $P \downarrow P'$. It indicates that P' is embedded in P one step away in space, i.e., $P \downarrow P'$ iff $\exists n, P''.P \equiv n[P']|P''$. The reflexive and transitive closure of $\downarrow$ is then $\downarrow^*$. In the following definition of the satisfaction relation, Π is the set of processes, Φ is the set of formulas, ϑ is the set of variables and Λ is the set of names.

Definition 3.2 (Satisfaction relation $\models$).

$\forall P : \Pi.$	$P \models \mathbf{T}$		
$\forall P : \Pi, \mathcal{A} : \Phi.$	$P \models \neg\mathcal{A}$	$\triangleq$	$\neg P \models \mathcal{A}$
$\forall P : \Pi, \mathcal{A}, \mathcal{B} : \Phi.$	$P \models \mathcal{A} \vee \mathcal{B}$	$\triangleq$	$P \models \mathcal{A} \vee P \models \mathcal{B}$
$\forall P : \Pi.$	$P \models 0$	$\triangleq$	$P \equiv 0$
$\forall P : \Pi, n : \Lambda, \mathcal{A} : \Phi.$	$P \models n[\mathcal{A}]$	$\triangleq$	$\exists P' : \Pi.\, P \equiv n[P'] \wedge P' \models \mathcal{A}$
$\forall P : \Pi, \mathcal{A}, \mathcal{B} : \Phi.$	$P \models \mathcal{A}\vert\mathcal{B}$	$\triangleq$	$\exists P', P'' : \Pi.\, P \equiv P'\vert P'' \wedge P' \models \mathcal{A} \wedge P'' \models \mathcal{B}$
$\forall P : \Pi, x : \vartheta, \mathcal{A} : \Phi.$	$P \models \forall x.\mathcal{A}$	$\triangleq$	$\forall m : \Lambda.\, P \models \mathcal{A}\{x \leftarrow m\}$
$\forall P : \Pi, \mathcal{A} : \Phi.$	$P \models \Diamond\mathcal{A}$	$\triangleq$	$\exists P : \Pi.\, P \rightarrow^* P' \wedge P' \models \mathcal{A}$
$\forall P : \Pi, \mathcal{A} : \Phi.$	$P \models \blacklozenge\mathcal{A}$	$\triangleq$	$\exists P : \Pi.\, P \downarrow^* P' \wedge P' \models \mathcal{A}$
$\forall P : \Pi, \mathcal{A} : \Phi.$	$P \models \mathcal{A}@n$	$\triangleq$	$n[P] \models \mathcal{A}$
$\forall P : \Pi, \mathcal{A}, \mathcal{B} : \Phi.$	$P \models \mathcal{A} \triangleright \mathcal{B}$	$\triangleq$	$\forall P' : \Pi.\, P' \models \mathcal{A} \Rightarrow P\vert P' \models \mathcal{B}$

Let's take a closer look to some of these definitions in (3.2). Well, P satisfies $n[\mathcal{A}]$ if the top-level ambient of P is n and its inner process satisfies $\mathcal{A}$. $\mathcal{A}|\mathcal{B}$ is satisfied by a process P if it is a parallel composition of two processes P', P'' and P' satisfies $\mathcal{A}$ and P'' satisfies $\mathcal{B}$. A process P satisfies $\forall x.\mathcal{A}$ if nevertheless which binding m for variable x in A we consider, P satisfies $\mathcal{A}$. $\Diamond\mathcal{A}$ is satisfied by P if after some time its reduction P' satisfies $\mathcal{A}$. That means, at some point in the future (inclusive *now*) the spatial structure of the process changed to a structure which satisfies $\mathcal{A}$. Similarly is $\blacklozenge\mathcal{A}$ satisfied by a process P. Here we use $\downarrow^*$ to indicate that there exist a sub-location in the current spatial configuration were $\mathcal{A}$ is satisfied. P satisfies $\mathcal{A}@\,n$ if we place it in some ambient n and that still satisfies A. And finally, $\mathcal{A} \triangleright \mathcal{B}$ is satisfied by a process P if nevertheless which opponent P' that fulfils $\mathcal{A}$ we choose, P in parallel with P' always satisfies $\mathcal{B}$. Hence, P is safe under security attacks.

Additionally there exist some derived operators which are often useful when formalizing specification properties:

$\mathbf{F}$	$\triangleq$	$\neg\mathbf{T}$	(False)
$\mathcal{A} \wedge \mathcal{B}$	$\triangleq$	$\neg(\neg\mathbf{A} \vee \neg\mathcal{B})$	(Conjunction)
$\mathcal{A} \Rightarrow \mathcal{B}$	$\triangleq$	$\neg\mathbf{A} \vee \mathcal{B}$	(Implication)
$\mathcal{A} \Leftrightarrow \mathcal{B}$	$\triangleq$	$(\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\mathcal{B} \Rightarrow \mathcal{A})$	(Equivalence)
$\mathcal{A}\|\|\mathcal{B}$	$\triangleq$	$\neg(\neg\mathcal{A}\|\neg\mathcal{B})$	(Decomposition)
$\mathcal{A}^{\forall}$	$\triangleq$	$\mathcal{A}\|\|\mathbf{F}$	(Every component satisfies $\mathcal{A}$)
$\mathcal{A}^{\exists}$	$\triangleq$	$\mathcal{A}\|\mathbf{T}$	(Some component satisfies $\mathcal{A}$)
$\exists x.\mathcal{A}$	$\triangleq$	$\neg\forall x.\mathcal{A}$	(Existential quantification over names)
$\Box\mathcal{A}$	$\triangleq$	$\neg\Diamond\neg\mathcal{A}$	(Everytime modality)
$\blacksquare\mathcal{A}$	$\triangleq$	$\neg\blacklozenge\neg\mathcal{A}$	(Everywhere modality)
$\mathcal{A} \propto \mathcal{B}$	$\triangleq$	$\neg(\mathcal{B} \triangleright \neg\mathcal{A})$	(Fusion)
$\mathcal{A}\|_{\Rightarrow}\mathcal{B}$	$\triangleq$	$\neg(\mathcal{A}\|\neg\mathcal{B})$	(Fusion adjunct)

Some of these might need an explanation for what we use them. The decomposition $\mathcal{A}||\mathcal{B}$ for example is the dual of composition. It is satisfied if for every decomposition of P into $P'|P''$ either one of these processes satisfies $\mathcal{A}$ or the other $\mathcal{B}$. $\mathcal{A}^{\forall}$ is used to express that every decomposition of P must satisfy $\mathcal{A}$. The existential version is then satisfied if there exist at least one decomposition which satisfies $\mathcal{A}$. The fusion operators are useful when talking about contexts of processes. Since $\mathcal{A} \propto \mathcal{B}$ is satisfied by P if there exist a context of it that satisfies $\mathcal{B}$ and furthermore that helps to satisfy $\mathcal{A}$. Instead, P satisfies $\mathcal{A}|_{\Rightarrow}\mathcal{B}$ if one process of its decomposition fulfils $\mathcal{A}$ and implicitly the other one satisfies $\mathcal{B}$.

An useful property [2, p.4] of the satisfaction relation is its invariance under structural congruence:

$$\frac{P \models \mathcal{A} \wedge P \equiv P'}{P' \models \mathcal{A}}$$

Since structural congruence is decidable we can construct on top of this a model checking algorithm which we will do in the next section.

3.3 Decidability

Model checking in the ambient calculi will be the computational task to decide if some process P satisfies a formula $\mathcal{A}$. Unfortunately we achieve a Turing-complete problem when P is not replication-free or $\mathcal{A}$ contains the $\triangleright$-operator [2, p.11]. Consider $P \models \mathbf{T} \triangleright \mathcal{A}$ which requires to check if for all processes P', $P'|P$ satisfies $\mathcal{B}$. Hence, there is infinite choice for P'. Same problems encounters with the replication operator, since termination of a system is in generally not decidable. Therefore we will restrict P to replication-freedom and $\mathcal{A}$ to $\triangleright$-free in the following.

3.3.1 Normal form

The model-checking algorithm presented later on will use the so-called *normal forms*. A normal form is a product (=composition) of prime processes:

Definition 3.3 (Normal form).

$\Pi_{i\in 1\ldots k} P_i \triangleq P_1 \vert \ldots \vert P_k \vert 0$	*(Product)*
$\pi ::= M[P] \parallel n.P \parallel \mathit{in}\, M.P \parallel \mathit{out}\, M.P \parallel \mathit{open}\, M \parallel (n).P \parallel \langle M \rangle$	*(Prime process)*
$\Pi_{i\in 1\ldots k}\, \pi_i$	*(Normal form)*

To convert a replication-free process P into normal form we use the recursive procedure $Norm$ defined as:

$$
\begin{array}{lcl}
Norm(0) & \triangleq & [] \\
Norm(P|P') & \triangleq & [\pi_1 \ldots \pi_k, \pi_1' \ldots \pi_{k'}'] \text{ if } Norm(P) = [\pi_1 \ldots \pi_k] \wedge Norm(P') = [\pi_1' \ldots \pi_{k'}'] \\
Norm(M[P]) & \triangleq & [M[P]] \\
Norm(M.P) & \triangleq & [M.P] \text{ if } M \in \{n, \mathit{in}\, N, \mathit{out}\, N, \mathit{open}\, N\} \\
Norm(\varepsilon.P) & \triangleq & Norm(P) \\
Norm((M.N).P) & \triangleq & Norm(M.(N.P)) \\
Norm((n).P) & \triangleq & [(n).P] \\
Norm(\langle M \rangle) & \triangleq & [\langle M \rangle]
\end{array}
$$

Hence, one can show that if $Norm(P) = [\pi_1 \ldots \pi_k]$ then $P \equiv \Pi_{i\in 1\ldots k}\, \pi_i$ [2, p.10].
☞ $Norm(P)$ is unique up to structural congruence of P.

3.3.2 Model checking

Before we define the model checking algorithm we take a look at the formulas we want to satisfy. Since we consider $\triangleright$-free formulas, they may contain modalities like sometime and somewhere. For them we need two procedures *Reachable* and *SubLocations*. *Reachable*(P) computes the set of processes Q we get when applying $\rightarrow^*$ on process P. *SubLocations*(P) works similarly but uses the $\downarrow^*$ relation. Hence, their formal properties are [2, p.10]:

- If $Reachable(P) = [P_1, \ldots, P_k]$ then for all $i \in [1 \ldots k].\, P \rightarrow^* P_i$ and for all Q, if $P \rightarrow^* Q$ then $Q \equiv P_i$ for some $i \in [1 \ldots k]$.
- If $SubLocation(P) = [P_1, \ldots, P_k]$ then for all $i \in [1 \ldots k].\, P \downarrow^* P_i$ and for all Q, if $P \downarrow^* Q$ then $Q \equiv P_i$ for some $i \in [1 \ldots k]$.

Now we have all requirements for the model checking algorithm:

Definition 3.4 (Model checking of replication-free processes P and $\triangleright$-free formulas $\mathcal{A}$).

$$
\begin{array}{lcl}
Check(P, \mathbf{T}) & \triangleq & \mathbf{T} \\
Check(P, \neg\mathcal{A}) & \triangleq & \neg Check(P, \mathcal{A}) \\
Check(P, \mathcal{A} \vee \mathcal{B}) & \triangleq & Check(P, \mathcal{A}) \vee Check(P, \mathcal{B}) \\
Check(P, 0) & \triangleq & \mathit{if}\ Norm(P) = []\ \mathit{then}\ \mathbf{T}\ \mathit{else}\ \mathbf{F} \\
Check(P, n[\mathcal{A}]) & \triangleq & \mathit{if}\ Norm(P) = [n[Q]]\ \mathit{for\ some}\ Q,\ \mathit{then}\ Check(Q, \mathcal{A})\ \mathit{else}\ \mathbf{F} \\
Check(P, \mathcal{A}|\mathcal{B}) & \triangleq & \mathit{let}\ Norm(P) = [\pi_1, \ldots, \pi_k] \\
 & & \mathit{in}\ \exists I, J. I \cup J = 1 \ldots k \wedge I \cap J = \emptyset \wedge \\
 & & Check(\Pi_{i\in I}\, \pi_i, \mathcal{A}) \wedge Check(\Pi_{i\in J}\, \pi_i, \mathcal{B}) \\
Check(P, \forall x.\mathcal{A}) & \triangleq & \mathit{let}\ \{m_1, \ldots, m_k\} = fn(P) \cup fn(\mathcal{A})\ \mathit{and}\ m_0 \notin \{m_1, \ldots, m_k\} \\
 & & \mathit{in}\ \forall i \in [0 \ldots k].\, Check(P, \mathcal{A}\{x \leftarrow m_i\})
\end{array}
$$

$$
\begin{array}{lcl}
Check(P, \Diamond\mathcal{A}) & \triangleq & \mathit{let}\ [P_1,\dots,P_k] = Reachable(P)\ \mathit{in}\ \exists i \in [1\dots k].\, Check(P_i,\mathcal{A}) \\
Check(P, \blacklozenge\mathcal{A}) & \triangleq & \mathit{let}\ [P_1,\dots,P_k] = SubLocations(P)\ \mathit{in}\ \exists i \in [1\dots k].\, Check(P_i,\mathcal{A}) \\
Check(P, \mathcal{A}@\, n) & \triangleq & Check(n[P],\mathcal{A})
\end{array}
$$

Hence, given a replication-free process P and a $\triangleright$-free formula $\mathcal{A}$ one can proof that $P \models \mathcal{A}$ iff $Check(P,\mathcal{A}) = \mathbf{T}$ [2, p.10]. Termination is guaranteed because the recursive calls work on sub-formulas. The time-complexity of $Check$ is at least exponential in the size of P since for $Check(P,\mathcal{A}|\mathcal{B})$, 2^k subsets of $[1\dots k]$ have to be computed [2, p.10].

As an example consider the process: $P = a[p[out\,a.in\,b\langle m\rangle]]|b[open\,p.(x).x[]]$.
Let $an\,n \triangleq n[\mathbf{T}]^{\exists}$ and $p\,parents\,q \triangleq p[q[\mathbf{T}]^{\exists}]^{\exists}$. Then the following properties are evaluated:

$Check(P, an\,a) = \mathbf{T}$	"There is a top-level ambient a"
$Check(P, an\,p) = \mathbf{F}$	"There is a top-level ambient p"
$Check(P, \blacklozenge an\,p) = \mathbf{T}$	"There is an ambient p"
$Check(P, \Diamond\blacklozenge an\,a) = \mathbf{T}$	"Sometime there is an ambient m"
$Check(P, b\,parents\,p) = \mathbf{F}$	"p is parent of b"
$Check(P, \Diamond b\,parents\,p) = \mathbf{T}$	"Sometime p is parent of b"

4 Conclusion and future work

Initially our motivation was to study mobility in sense of moving code and data. Therefore we introduced the Mobile Ambients calculi which describes mobile computation as so-called ambients. These are places where computation and communication happens. Furthermore they are structured in a hierarchical manner to describe administrative domains. Therefore ambients are allowed to move which is controlled by embedded processes. On top of that we used the calculi to describe locks, authentication, communication mechanisms like channels. Later on we have seen that it is Turing-complete by a direct encoding of Turing-Machines. Since the syntax of the calculi allows several "illegal" processes and message exchanges between two processes it should be restricted to values of same type, we studied type-safety. Even if the types which have been presented are somehow unusual they are general enough to describe arbitrary scenarios. Depending on the size of a mobile system, proofing certain properties can be a complex task by hand. To overcome that we studied a modal logic for mobile ambients which is based on structural congruence. When restricting ourselves to replication-free processes and $\triangleright$-free formulas it is possible to apply model-checking to some decidable sub-logic. How to include replication in processes and composition adjuncts in formulas is an open question. All in all we got a quite powerful tool with the calculi to explore the interaction and properties of mobile systems. Especially security issues are of interest in nowadays internet. Current research introduced some dialects of the ambient calculi so-called *safe ambients* [5], *robust ambients* [8], *controlled ambients* [7] and *probabilistic ambients* [4]. Here the safe ambients calculi focuses on the interference of processes, meaning, the situation when processes damage other processes activities. The authors of the robust ambients come up with an improvement for the safe ambients calculi because it offers some security breaches to the hole calculi. Since concurrent systems nowadays are highly parallel and reconfigurable they rely on a lot of subsystems and communication protocols. But this is an open door for Denial of Service attacks since there may exist bugs whose number multiply in sense of parallelism. Therefore the controlled ambients approach tackles the control of resources in such systems. The probabilistic ambient calculi introduces a probabilistic choice operator and adapts the modal logic for this. Further research could be the development of new programming languages and techniques with use ambients as their semantics. One approach to that is *AmbIcobjs* [6]. It provides an ambient simulator that enables ambient programming in a graphical way. As an example, the authors developed a taxi demo with different cabs and clients which want to travel between different sites in a city.

References

[1] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Proc. 26th POPL*, pages 79–92. ACM Press, 1999.

[2] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *Proc. 27th POPL*, pages 365–377. ACM Press, 2000.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Theoretical Computer Science, 240(1)*, pages 177–213. Elsevier, 2000.

[4] M. Kwiatkowska, G. Norman, D. Parker, and M.G. Vigliotti. Probabilistic mobile ambients. *Theoretical Computer Science*, 410(12–13):1272–1303, 2009.

[5] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Proc. 27th ACM SIGPLAN-SIGACT*, pages 352–364. ACM, 2000.

[6] Mimosa Project. Ambient programming in icobjs. www-sop.inria.fr/mimosa/ambicobjs/.

[7] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using ambients to control resources. In *Proc. 13th CONCUR*, pages 288–303. Springer-Verlag, 2002.

[8] Yiling Yang Xudong Guan and Jinyuan You. Making ambients more robust, 2000.