Finite-Control Mobile Ambients

Witold Charatonik[1,2], Andrew D. Gordon[3], and Jean-Marc Talbot[4]

Max-Planck-Institut für Informatik, Germany.[1]
University of Wrocław, Poland.[2]
Microsoft Research, United Kingdom.[3]
Laboratoire d'Informatique Fondamentale de Lille, France.[4]

Abstract. We define a finite-control fragment of the ambient calculus, a formalism for describing distributed and mobile computations. A series of examples demonstrates the expressiveness of our fragment. In particular, we encode the choice-free, finite-control, synchronous π-calculus. We present an algorithm for model checking this fragment against the ambient logic (without composition adjunct). This is the first proposal of a model checking algorithm for ambients to deal with recursively-defined, possibly nonterminating, processes. Moreover, we show that the problem is PSPACE-complete, like other fragments considered in the literature. Finite-control versions of other process calculi are obtained via various syntactic restrictions. Instead, we rely on a novel type system that bounds the number of active ambients and outputs in a process; any typable process has only a finite number of derivatives.

1 Introduction

The ambient calculus [6] is a formalism for describing distributed and mobile computation in terms of *ambients*, named collections of running processes and nested subambients. A state of computation has a tree structure induced by ambient nesting. Mobility is represented by re-arrangement of this tree (an ambient may move inside or outside other ambients) or by deletion of a part of this tree (a process may dissolve the boundary of some ambient, revealing its content).

There are proposals for analysing systems expressed in the ambient calculus and its variants [2,14] via several techniques, such as equational reasoning [3], type systems [8], control flow analysis [16], and abstract interpretation [13]. Still, the ambient calculus is Turing-complete, and little attention has been paid to finding expressive finite-state fragments that admit automatic verification via state-space exploration. The goal of this work is to identity such a fragment, and to develop a model checking algorithm for verifying properties expressible in the ambient logic [5,7]. The long term intention is that automatic verification tools for a finite-state ambient calculus will be useful either by themselves or in conjunction with methods for obtaining finite-state abstractions of infinite-state systems. Similar abstractions [9] are being developed for the π-calculus [15], the formalism from which the ambient calculus derives.

A finite-state version of π exists [12]. It is described as a finite-control calculus because its control structure is finite. Starting in any state, the number of states reachable via internal reduction steps is finite. However, if we allow inputs of external data, the number of reachable states may be infinite. We define in this paper a

finite-control ambient calculus. It is a substantial extension of the replication-free fragment [5,10,11] (sometimes referred to as the "finite-state ambient calculus"). In particular, in the replication-free fragment every process can make only a finite number of computation steps—no recursion or iteration is possible.

We begin, in Section 2, by presenting a variant of the ambient calculus in which recursion is defined by means of an explicit recursive definition instead of replication. We specify standard spatial rearrangements via a structural congruence relation on processes, and specify the operational semantics as a reduction relation on processes. This variant is easily seen to simulate the original one. Then, in Section 3, we design a type system for ambient processes and show that typability of a process guarantees finitary behaviour. The basic idea of the type system is to count the number of active outputs and ambients in a process. Theorem 1 asserts that the number of processes, up to structural congruence, reachable from any typable process is finite. We define the finite-control fragment as those processes that are typable. In contrast, finite-control fragments of the π-calculus are defined via simple syntactic restrictions. In Section 4, we explore the expressivity of our calculus by presenting and developing some standard examples, including an encoding of a finite-control π-calculus.

Turning to the verification problem, Section 5 reviews the syntax and semantics of the ambient logic we use to specify process properties. We prove that the verification problem—model checking against the ambient logic without composition adjunct—is decidable for the finite-control fragment. To achieve this, we adapt the model checking algorithm from [11]. Theorem 2 states that the algorithm is correct with respect to the semantics of the logic. Moreover, our final result is Theorem 3, that the verification problem remains PSPACE-complete, which is the same complexity as verifying the replication-free fragment against the same logic.

A difficulty in designing a finite-control fragment of a process calculus is striking a balance between the expressivity of the fragment and the complexity of the verification problem. The general goal is to make the calculus as expressive as possible while keeping the verification problem decidable. The methods we use differ substantially from the methods used to define finite-control π-calculi. Therefore, for the sake of a simple exposition we omit several possible features from our finite-control ambient calculus while including enough to model interesting iterative computations. Section 6 discusses some of these additional features. Section 7 concludes the paper.

2 An Ambient Calculus with Recursion

We present in this section an ambient calculus with recursive definitions instead of replication. We give examples of the calculus in Section 4; see [6] for more elementary examples.

The following table defines the syntax of *capabilities* and *processes* of our calculus. We assume countably many *names* ranging over by $n, m, a, b, c, \ldots$ and countably many *identifiers* ranging over by $A, B, C, \ldots$. For the sake of a simple presentation we allow only names to be communicated whereas the original calculus allows also the transmission of sequences of capabilities.

Processes and Capabilities:

α ::=	capabilities		
$\mathsf{in}\ n$	can enter n	$\mathsf{out}\ n$	can exit n
$\mathsf{open}\ n$	can open n		
P, Q, R ::=	processes		
$\mathbf{0}$	inactivity	$P \mid Q$	composition
$n[P]$	ambient	$\alpha.P$	action prefix
$(n).P$	input	$\langle n \rangle$	output
$(\nu n)P$	name restriction	A	identifier
$(\mathsf{fix}\ A{=}P)$	recursion		

We consider input (n), name restriction (νn) to be binders for the name n and $\mathsf{fix}\ A$ to be a binder for the identifier A. A name n or an identifier A occurring in the scope of respectively (n),(νn) and $\mathsf{fix}\ A$ is *bound*. Otherwise it is *free*. We write $fn(P)$ for the set of free names in P. We say that a process is *closed* if it contains no free identifier. We identify processes up-to capture-avoiding α-renaming of both bound names and bound identifiers. For instance, $(\mathsf{fix}\ A{=}(\nu n)open\ n.A)$ and $(\mathsf{fix}\ B{=}(\nu m)open\ m.B)$ are identical processes. Slightly abusing the notation, we write $bn(P)$ for the bound names in an implicitly given syntactic representation of P. We write $P\{m{\leftarrow}n\}$ for the outcome of substituting n for each free occurrence of m in P. Similarly, $P\{A{\leftarrow}Q\}$ is the outcome of substituting Q for each free occurrence of A in P. We will assume without loss of generality that two distinct bound identifiers are different as well as being distinct from any free identifier.

The semantics of our calculus is given by two relations. The *reduction relation* $P \rightarrow Q$ describes the evolution of ambient processes over time. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. The *structural congruence* relation $P \equiv Q$ relates different syntactic representations of the same process; it is used to define the reduction relation.

Structural Congruence $P \equiv Q$:

$P \equiv P$	(Str Refl)	$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Str Res)
$P \equiv Q \Rightarrow Q \equiv P$	(Str Symm)	$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Str Par)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Str Trans)	$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Str Amb)
$P \mid \mathbf{0} \equiv P$	(Str Par Zero)	$P \equiv Q \Rightarrow \alpha.P \equiv \alpha.Q$	(Str Action)
$P \mid Q \equiv Q \mid P$	(Str Par Comm)	$P \equiv Q \Rightarrow (n).P \equiv (n).Q$	(Str Input)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Str Par Assoc)	$P \equiv Q \Rightarrow (\mathsf{fix}\ A{=}P) \equiv (\mathsf{fix}\ A{=}Q)$	(Str Fix)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Str Res Zero)	$(\mathsf{fix}\ A{=}A) \equiv \mathbf{0}$	(Str Fix Id)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Str Res Res)	$(\mathsf{fix}\ A{=}P) \equiv P\{A{\leftarrow}(\mathsf{fix}\ A{=}P)\}$	(Str Fix Rec)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$	(Str Res Par)		
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Str Res Amb)		

Reduction: $P \rightarrow Q$

$n[\mathsf{in}\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[\mathsf{out}\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$\mathsf{open}\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$\langle m \rangle \mid (n).P \rightarrow P\{n{\leftarrow}m\}$	(Red I/O)

$P \to Q \Rightarrow P \mid R \to Q \mid R$	(Red Par)
$P \to Q \Rightarrow n[P] \to n[Q]$	(Red Amb)
$P \to Q \Rightarrow (\nu n)P \to (\nu n)Q$	(Red Res)
$P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q'$	(Red $\equiv$)

As in other process calculi with recursion, it is convenient to regard certain unwanted recursive processes as ill-formed, and to disregard them. An example is $(\mathsf{fix}\, A{=}A \mid A)$. We define a well-formed process as follows.

Definition 1. *A process P is said to be* well-formed *if every recursive subprocess $(\mathsf{fix}\, A{=}Q)$ of P satisfies the following two requirements: (i) A is the only free identifier in Q, and (ii) A occurs at most once in Q.*

From now on, we only consider well-formed processes. These processes are stable with respect to structural congruence and reduction: if P is well-formed and either $P \equiv P'$ or $P \to P'$ then P' is well-formed.

As in the π-calculus [15], we can easily simulate replication with recursion. To simulate $!P$, which behaves like an unbounded number of replicas of P running in parallel, we introduce a new identifier A_P and replace $!P$ by $(\mathsf{fix}\, A_P{=}P \mid A_P)$. The resulting process is well-formed. This encoding of replication fulfils the axioms of structural congruence for replication $!P \equiv P \mid !P$ and $!\mathbf{0} \equiv \mathbf{0}$ given in [6]. It does not obey the two additional axioms $!P \equiv !!P$ and $!(P \mid Q) \equiv !P \mid !Q$ from [5], but these axioms are unnecessary for computing reduction steps.

Sangiorgi [17] also considers an ambient calculus with recursion. A difference is that our formulation allows recursively defined ambient structures. In a recursion $(\mathsf{fix}\, A{=}P)$, the identifier A can appear in P within an ambient construct; for example, the processes $(\mathsf{fix}\, A{=}m[A])$ and $(\mathsf{fix}\, A{=}\mathsf{open}\ n.m[A])$ are well-formed. The latter but not the former belongs to the finite-control fragment defined next.

3 A Finite-Control Ambient Calculus

The finite-control (synchronous) π-calculus [12] is obtained by disallowing parallel composition through recursion. So, a finite-control π-calculus process is a finite parallel composition of threads each of which is a recursive process without parallel composition. This ensures that there is only finitely many pairwise non-congruent configurations reachable from such a process. In the ambient calculus this restriction is both too strong and too weak. It is too strong because it limits the admissible computation too much. In particular, due to the asynchronous communication mechanism in the ambient calculus, it completely excludes communication in recursive programs. On the other hand, it is too weak, as exemplified below, because it does not ensure finitary behaviour of processes. The example shows that in the ambient calculus a bound on the number of parallel threads gives neither a bound on the size of reachable processes nor a bound on the number of possible interactions between threads. This is in contrast to the situation in the π-calculus, and seems to arise from the spatial characteristics of the ambient calculus.

Example 1. Consider the two ambient processes P_A and P_B defined respectively by $(\mathsf{fix}\, A{=}n[\mathsf{open}\ m.A])$ and $(\mathsf{fix}\, B{=}m[\mathsf{in}\ n.B])$. Neither process contains parallel composition. However, the process $P_A \mid P_B$ reduces in $2k$ steps to $\underbrace{n[\cdots n[}_{k}\, P_A \mid P_B]\cdots]$, denoted $(P_A \mid P_B)_k$. Since for $k \neq k'$, $(P_A \mid P_B)_k \not\equiv (P_A \mid P_B)_{k'}$, there are infinitely many non-congruent processes reachable from $P_A \mid P_B$.

Now, if we place another process $m[(\mathsf{fix}\, C{=}\mathsf{in}\ n.C) \mid (\mathsf{fix}\, D{=}\mathsf{out}\ n.D)]$ in parallel with $P_A \mid P_B$, this process can traverse the structure of $(P_A \mid P_B)_k$ in an arbitrary way. Thus, although there are only four recursively defined processes and none of them contains parallel composition, they may create an arbitrary number of locations and interact in any of these locations.

This example shows that directly adopting the syntactic restriction from the finite-control synchronous π-calculus is problematic, but it does not show undecidability of the verification problem. However, if we adopt a more liberal condition from the finite-control asynchronous π-calculus [1] (which ensures that there is only a bounded number of active threads and seems more appropriate here due to the asynchronous communication used in ambient calculus), we obtain undecidability even for the reachability problem. One can adapt the encoding of the Post correspondence problem from [11], in which only finitely many active threads are used.

Thus, one may consider a more severe syntactic restriction, to forbid both parallel composition and ambient construction within recursion. We will see later on that this restriction indeed ensures finite-control. Still, we can see at once that it is too drastic. In such a restricted process only sequences of action prefixes, each invoking a capability, could be defined recursively. Moreover, those sequences would operate on a process whose spatial structure has a bounded size. Inspecting the individual effect of such sequences, one sees that the open capability is somehow the more powerful as it changes the spatial structure of the process by deleting part of it whereas both capabilities in and out only re-arrange this structure. As a consequence, only finitely many occurrences of the powerful capability open can be executed by such a restricted process.

Instead of defining finite-control by means of syntactic restrictions over processes, we adopt a semantic point of view based on a type system. Intuitively, a type of a process P is a natural number that bounds the number of active outputs and ambients in any process reachable from P. We present the type system in Section 3.1. In Section 3.2 we show that typability ensures finitary computation.

3.1 The Type System FC

A *type environment* Γ is a finite set of pairs $\{(A_1, \tau_1), \ldots, (A_n, \tau_n)\}$ such that each A_i is an identifier, τ_i is a natural number and for any two pairs (A_i, τ_i) and (A_j, τ_j), $i \neq j$ implies $A_i \neq A_j$. We say that an environment Γ is defined for A if Γ contains a pair (A, τ). Whenever Γ is defined for no identifier, we simply write $\varnothing$.

Definition 2. *Given a type environment Γ, a type judgment $\Gamma \vdash Q : \tau$ holds for a process Q and a natural number τ if there exists a finite proof tree built with the inference rules from the table below such that its root is labelled by $\Gamma \vdash Q : \tau$ and none of its leaves contains a type judgment.*

Process Typing : $\Gamma \vdash P \,:\, \tau$

(Identifier) $\dfrac{A \text{ is identifier, } (A,\tau) \in \Gamma}{\Gamma \vdash A \,:\, \tau}$ (Zero) $\dfrac{}{\Gamma \vdash \mathbf{0} \,:\, 0}$ (Par) $\dfrac{\Gamma \vdash P \,:\, \tau \;,\; \Gamma \vdash Q \,:\, \theta}{\Gamma \vdash P \mid Q \,:\, \tau + \theta}$ (Res) $\dfrac{\Gamma \vdash P \,:\, \tau}{\Gamma \vdash (\nu n)P \,:\, \tau}$

(Output) $\dfrac{}{\Gamma \vdash \langle n \rangle \,:\, 1}$ (Input) $\dfrac{\Gamma \vdash P \,:\, \tau}{\Gamma \vdash (n).P \,:\, \max(\tau - 1, 1)}$ (In/Out) $\dfrac{\Gamma \vdash P \,:\, \tau \,,\, \mathsf{cap} \in \{\mathsf{in}\,, \mathsf{out}\,\}}{\Gamma \vdash \mathsf{cap}\, n.P \,:\, \max(\tau, 1)}$

(Amb) $\dfrac{\Gamma \vdash P \,:\, \tau}{\Gamma \vdash n[P] \,:\, \tau + 1}$ (Open) $\dfrac{\Gamma \vdash P \,:\, \tau}{\Gamma \vdash \mathsf{open}\; n.P \,:\, \max(\tau - 1, 1)}$ (Fix) $\dfrac{\Gamma \cup \{(A,\tau)\} \vdash P \,:\, \theta \,,\quad \theta \leq \tau}{\Gamma \vdash (\mathsf{fix}\, A{=}P) \,:\, \tau}$

The basic idea of the type system is to bound the number of active outputs and ambients in all processes reachable from a given one. In the rules (Input) and (Open) the process P is guarded and thus not active. These rules express that P may become active only after dissolving some active output or ambient. The function $\max(\cdot, 1)$ is used to avoid negative types. Without it some processes of unbounded (or even infinite) size like $(\mathsf{fix}\, A{=}\mathsf{open}\; n.\mathbf{0} \mid m[] \mid A)$ could be still typable. We take maximum with 1 and not 0 to obtain a property—used in some proofs—that $\mathbf{0}$ is the only process of type 0.

Example 2. The proof tree stating that the type judgment $\varnothing \vdash (\mathsf{fix}\, A{=}\mathsf{open}\; n.m[A]) \mid n[\mathbf{0}] \;:\; 2$ holds is given below. In a similar way we can build a proof tree for $\varnothing \vdash (\mathsf{fix}\, A{=}\mathsf{open}\; n.m[A]) \mid n[\mathbf{0}] \;:\; 3$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A,1) \in \{(A,1)\}}{\{(A,1)\} \vdash A \,:\, 1}}{\{(A,1)\} \vdash m[A] \,:\, 2}}{\{(A,1)\} \vdash \mathsf{open}\; n.m[A] \,:\, 1 \,,\quad 1 \leq 1} \qquad \dfrac{\dfrac{}{\varnothing \vdash \mathbf{0} \,:\, 0}}{\varnothing \vdash n[\mathbf{0}] \,:\, 1}}{\varnothing \vdash (\mathsf{fix}\, A{=}\mathsf{open}\; n.m[A]) \mid n[\mathbf{0}] \,:\, 2}$$

Recall Example 1. The process $(\mathsf{fix}\, A{=}n[\mathsf{open}\; m.A]) \mid (\mathsf{fix}\, B{=}m[\mathsf{in}\; n.B])$ is not typable because $(\mathsf{fix}\, B{=}m[\mathsf{in}\; n.B])$ is not typable.

Definition 3. *A type environment* Γ well-types *a process* P *if there exists some natural number* τ_P *such that the type judgment* $\Gamma \vdash P \;:\; \tau_P$ *holds. A process* P *is* typable *if there exists a type environment* Γ *that well-types* P.

If P is typable and does not contain free identifiers, then $\varnothing$ well-types P.

We say that a process P is *balanced* if the number of occurrences of ambients and outputs in P is equal to the number of occurrences of open capabilities and inputs in P. We say that a recursive process $(\mathsf{fix}\, A{=}P)$ has a balanced type if for every type environment $\Gamma \cup \{(A : \tau)\}$ such that $\Gamma \cup \{(A,\tau)\} \vdash P : \theta$ and $\theta \leq \tau$ we have $\theta = \tau$. In most natural examples, like in $(\mathsf{fix}\, A{=}n[\mathsf{open}\; n.A])$, if P is balanced then $(\mathsf{fix}\, A{=}P)$ is typable. But there are exceptions like $(\mathsf{fix}\, A{=}A \mid \mathsf{open} n.n[\mathbf{0}])$ which is balanced but not typable. If P is not balanced because it contains more outputs and ambients than inputs and opens (like in $(\mathsf{fix}\, A{=}n[A])$) then $(\mathsf{fix}\, A{=}P)$ is not typable. Finally, note that not all

typable processes have balanced types. For example $(\mathsf{fix}\, A{=}\mathsf{open}\, n.A)$ is typable but it does not have a balanced type (the environment $\{(A, 2)\}$ being a counter-example).

In most examples of typable recursive processes that are considered in this paper we will want the types to be balanced. This is because if such a process does not have a balanced type then in each execution it consumes some (strictly more than it creates) messages or ambients in the global context in which it is placed, and thus it can be executed only finitely many times.

For a given type environment Γ and a given process P there may be many natural numbers τ such that the type judgment $\Gamma \vdash P : \tau$ holds. For example, $\varnothing \vdash (\mathsf{fix}\, A{=}A) : \tau$ holds for any natural τ. However, since every set of natural numbers has a least element, we may define a least type.

Definition 4 (Least Type). *For any process P and any type environment Γ that well-types P, the* least *type of P with respect to Γ, denoted $\mathcal{L}^{\mathsf{FC}}(P, \Gamma)$, is the least natural number τ such that $\Gamma \vdash P : \tau$ holds.*

Proposition 1 (Type Stability). *Let P and P' be typable processes and let Γ be a type environment that well-types P and P'. Then $P \equiv P'$ implies $\mathcal{L}^{\mathsf{FC}}(P, \Gamma) = \mathcal{L}^{\mathsf{FC}}(P', \Gamma)$.*

Proof. The proof goes by induction over the proof tree for $P \equiv P'$. The only difficult case is for the axiom $(\mathsf{fix}\, A{=}Q) \equiv Q\{A{\leftarrow}(\mathsf{fix}\, A{=}Q)\}$ (Str Fix Rec) which requires an induction over the structure of Q. □

Proposition 2 (Type Checking - Type Inference). *Type checking (that is, deciding, given Γ, P and τ, whether the type judgment $\Gamma \vdash P : \tau$ holds) is decidable. For any process P and type environment Γ, we can decide whether Γ well-types P and compute $\mathcal{L}^{\mathsf{FC}}(P, \Gamma)$.*

Proof. Both type checking and type inference amount to solving easy systems of inequalities with addition, subtraction of a constant, and $\max$ as the only arithmetic operations. □

Proposition 3 (Subject Reduction). *Let P be a process and Γ an environment that well-types P. Then for all processes P' such that $P \to P'$, Γ well-types P' and $\mathcal{L}^{\mathsf{FC}}(P', \Gamma) \leq \mathcal{L}^{\mathsf{FC}}(P, \Gamma)$.*

Sketch of proof. The reductions (Red In) and (Red Out) do not change the type of a process at all. The reductions (Red Open) and (Red I/O) reducing processes of the form $\mathsf{open}\, n.Q$ or $(n).Q$, respectively, do not change the type if $\mathcal{L}^{\mathsf{FC}}(Q, \Gamma) > 1$. Otherwise, they strictly decrease the type by removing the ambient n or consuming a message. For the other reductions (Red Par), (Red Amb), (Red Res), and (Red $\equiv$), it follows from induction over P, using Proposition 1 in case of (Red $\equiv$). □

Due to Proposition 1, the least type of any process congruent to $\mathbf{0}$ is 0. Conversely, we have:

Proposition 4. *For all closed and typable processes P, if $\mathcal{L}^{\mathsf{FC}}(P, \varnothing) = 0$ then $P \equiv \mathbf{0}$.*

Proof. It is easy to see that if a closed process P contains either an ambient construct, a capability, an input or an output, then its least type is greater or equal to one. Therefore, it is enough to show that closed and well-formed processes built up with identifiers, $\mathbf{0}$, parallel composition, name restriction, and fix are congruent to $\mathbf{0}$. The proof goes by induction on the structure of P. □

Additionally, we can prove some other properties. All recursion-free processes are typable. The encoding of replication $!P$ given earlier is not typable for any P non-congruent to $\mathbf{0}$. Processes built without parallel composition and ambient construct are typable. This last property implies that processes are typable if they satisfy the syntactic restriction—to forbid both composition and ambients within recursion—considered in Section 3. As we see in the next section, it follows that processes obeying this syntactic restriction are finite-state.

3.2 Typability and Finite-Control

The goal of this section is to prove that for a typable and closed process, there exist finitely many $\equiv$-congruence classes $\mathcal{K}_1, \ldots, K_n$ such that for all processes P' with $P \rightarrow^* P'$, there exists i for which $P' \in \mathcal{K}_i$. Instead of proving this directly, we show that for any typable and closed process P and any process P' reachable from P, there exists a *representative* P'' of P' (that is, $P' \equiv P''$) such that:

- the size of P'' is bounded and depends only on P;
- the set of free names of P'' is a subset of the free names of P.

Here, by the size $|P|$ of a process P we mean the number of nodes in the tree representation of P. The two statements above imply that there exist only finitely many pairwise non-congruent processes reachable from P. Simply showing the size is bounded is insufficient as there are infinitely many different names. For example, processes from the set $\{n[0] \mid n \text{ being a name}\}$ have a bounded size, but being non-congruent with each other, they represent infinitely many $\equiv$-congruence classes.

The second requirement about free names is straightforward and actually does not rely on typability.

Proposition 5. *For all processes P, P', if $P \equiv P'$ or $P \rightarrow P'$ then $fn(P') \subseteq fn(P)$.*

The first requirement is much more involved for various reasons. We need to characterize representatives of structural congruence classes of reachable processes; this requires to consider a process split into several parts.

First, we define pre-normalized processes. Let a process P be *pre-normalized* if it takes the form $(\nu n_1) \ldots (\nu n_k) Q$ and, (i) every n_j occurs free in Q, (ii) $n_1, \ldots, n_k$ are pairwise distinct, and (iii) any other name restriction occurring in Q appears in the scope of some input or of some action prefix. Intuitively, pre-normalization is rewriting a process using the scope extrusion rules (Str Res Par) and (Str Res Amb) to a kind of prenex normal form.

The second part of processes consists in *outermost guarded* subprocesses. A process P is *guarded* if either $P \equiv \mathbf{0}$, $P \equiv \langle M \rangle$, $P \equiv \alpha.P'$ for some P' and some α, $P \equiv (x).P'$ for some P', or recursively $P \equiv (\nu n)Q$ for some guarded Q. This property is clearly stable with respect to structural congruence, that is, if P is guarded and $P \equiv P'$ then P' is guarded as well. Let a subprocess P' be *outermost guarded* in a process P if P' is guarded and for any subprocess P'' of P enclosing P', P'' is not guarded. For instance, $\mathsf{out}\ m.\mathbf{0}$ is outermost guarded in $n[\mathsf{out}\ m.\mathbf{0}]$ and as a consequence, $\mathbf{0}$ is guarded but not outermost guarded. In $n[\mathsf{out} m.\mathbf{0} \mid \mathbf{0}]$, $\mathsf{out} m.\mathbf{0} \mid \mathbf{0}$ is outermost guarded

(because $\mathsf{out}\ m.\mathbf{0} \mid \mathbf{0}$ is congruent to a process of the form $\alpha.P$, namely $\mathsf{out}\ m.\mathbf{0}$) and thus, $\mathsf{out}\ m.\mathbf{0}$ is not outermost guarded. This last example shows that outermost guardedness is a pure syntactic condition and is not stable with respect to structural congruence.

Finally, the remaining part of the process is captured by a context. A *context* $\mathcal{C}$ with l holes (or, for short, an l-context) is a process where exactly l subprocesses have been replaced by a hole $\star_i$ occurring exactly once in $\mathcal{C}$. We write $\mathcal{C}[P_1, \ldots, P_l]$ for the process obtained by filling each hole $\star_i$ in $\mathcal{C}$ with P_i.

A context is *active* if it consists only of holes, ambients, parallel compositions, and void processes $\mathbf{0}$ and furthermore if each process $\mathbf{0}$ occurs as a child node of an ambient in the tree representation of the context.

A process is *normalized* if this process is either $\mathbf{0}$ or a pre-normalized process of the form $(\nu n_1) \ldots (\nu n_k) Q$, where Q is of the form $\mathcal{C}[P_1, \ldots, P_l]$ such that:

- $\mathcal{C}$ is an active l-context,
- $P_1, \ldots, P_l$ are the outermost guarded subprocesses from Q that are not congruent to $\mathbf{0}$.

By the *one-step unfolding* of a process P we mean the process obtained from P by replacing every subprocess of the form $(\mathsf{fix}\, A{=}Q)$ by $Q\{A{\leftarrow}(\mathsf{fix}\, A{=}Q)\}$. If Q is obtained by one-step unfolding from P then $|Q| \leq |P|^2$.

Lemma 1. *Any typable and closed process P admits a congruent normalized process Q such that $|Q| \leq |P|^2$.*

Sketch of proof. First, by structural induction we prove that in any recursive process $(\mathsf{fix}\, A{=}Q)$ either A is guarded in Q or $(\mathsf{fix}\, A{=}Q)$ is congruent to $\mathbf{0}$. We obtain a normalized version of a pre-normalized process by replacing all recursive definitions congruent to $\mathbf{0}$ by $\mathbf{0}$, applying a one-step unfolding to the result, and then removing all $\mathbf{0}$'s from the context that are not child nodes of an ambient. □

We say that a process Q is a *subprocess up to renaming* of a process P if Q can be obtained from some subprocess of P by renaming its free names.

Proposition 6. *Let P be a closed, typable, normalized and non-congruent to $\mathbf{0}$ process and P' be its one-step unfolding. Then for all processes Q reachable from P (that is, such that $P \rightarrow^* Q$), there exists a normalized process $(\nu n_1) \ldots (\nu n_k) \mathcal{C}[P_1, \ldots, P_l]$ structurally congruent to Q and such that*

- *k is bounded by the size of $\mathcal{C}$,*
- *the size of $\mathcal{C}$ is bounded by $3 \cdot \mathcal{L}^{\mathsf{FC}}(P, \varnothing)$,*
- *each Q_j is a subprocess up to renaming of some outermost guarded part from P'.*

Sketch of proof. Since $(\nu n_1) \ldots (\nu n_k) \mathcal{C}[P_1, \ldots, P_l]$ is pre-normalized, every restricted name from the set $\{n_1, \ldots, n_k\}$ must occur freely in $\mathcal{C}$. Thus k is bounded by the size of $\mathcal{C}$.

Since the process is normalized, the subprocesses $P_1, \ldots, P_l$ are not congruent to $\mathbf{0}$ and thus have strictly positive types. $\mathcal{C}$ is an active context, so its tree representation consists of four kinds of nodes:

– leaves representing a hole, whose number is smaller than $\mathcal{L}^{\mathsf{FC}}(P_1, \varnothing) + \ldots + \mathcal{L}^{\mathsf{FC}}(P_l, \varnothing)$,
– leaves representing $\mathbf{0}$, whose number is smaller than the number of unary nodes,
– binary nodes representing parallel compositions whose number is smaller than the number of leaves,
– unary nodes representing ambients; the number of such nodes summed with $\mathcal{L}^{\mathsf{FC}}(P_1, \varnothing) + \ldots + \mathcal{L}^{\mathsf{FC}}(P_l, \varnothing)$ gives $\mathcal{L}^{\mathsf{FC}}(Q, \varnothing)$.

This together with the subject reduction theorem (Proposition 3) gives that the size of $\mathcal{C}$ is bounded by $3 \cdot \mathcal{L}^{\mathsf{FC}}(P, \varnothing)$.

Finally, the processes Q_j are either directly subprocesses up to renaming of the initial process P or of unfoldings of the recursive definitions, which are already unfolded in P'. This is because the only possibility (apart from using the structural congruence) to modify a process below a guard is to substitute some of its free names with other names coming from communication. □

The following theorem is a direct corollary from Propositions 5 and 6.

Theorem 1 (Finite-State). *For any closed and typable process P, there exist only finitely many pairwise non-congruent processes reachable from P.*

4 Examples

The model checking algorithm from [5] is limited to replication-free processes. We want to have at least some restricted version of recursion that would help us in modelling mobile computations while keeping model checking decidable. This section gives examples of programs that are typable and that therefore, by Theorem 1, are finite-state.

4.1 Simple Examples with Infinite Behaviour

Probably the simplest possible example with infinite behaviour is $n[P_A] \mid m[]$ where P_A is the process $(\mathsf{fix}\, A{=}\mathsf{in}\ m.\mathsf{out}\ m.A)$. It is typable with the type of P_A equal to 1 and the type of the whole process being 3. We have $n[P_A] \mid m[] \to m[n[\mathsf{out}\, m.P_A]] \to n[P_A] \mid m[]$, which creates an infinite loop.

Another simple example is $P_A \mid P_B$ where P_A is $(\mathsf{fix}\, A{=}a[\mathsf{open}\ b.A])$ and P_B is $(\mathsf{fix}\, B{=}\mathsf{open}\ a.b[B])$. Here the least type of P_A is 2 and the least type of P_B is 1. One can see it as a simple synchronization mechanism—we will use such a mechanism later in the encoding of the (synchronous) finite-control π-calculus. We have $P_A \mid P_B \equiv a[\mathsf{open}\ b.P_A] \mid \mathsf{open}\ a.b[P_B] \to \mathsf{open}\ b.P_A \mid b[P_B] \to P_A \mid P_B$.

A similar behaviour can be obtained from a simpler process $(\mathsf{fix}\, A{=}\mathsf{open}\ a.A) \mid (\mathsf{fix}\, B{=}a[B])$, but we cannot use it since it is not typable.

Our last example in this section shows that we can obtain not only infinite computation paths, but also infinitely many syntactically different processes along these paths. Consider the process $P_A \mid P_B$ where P_A is $(\mathsf{fix}\, A{=}(\nu a)\mathsf{open}\ n.\mathsf{open}\ m.(\langle a \rangle \mid a[A]))$ and P_B is $(\mathsf{fix}\, B{=}n[m[(x).\mathsf{open}\ x.B]])$. The process is typable with the least types of P_A and P_B being respectively 1 and 3. Here, in every iteration, the process P_A creates a new fresh name and sends it to P_B.

4.2 Objective Moves

The only iterative definition in the encoding of objective moves in [6] is $\texttt{allow}\ n \triangleq !\mathsf{open}\ n$. This can be directly translated to $(\mathsf{fix}\, A{=}\mathsf{open}\ n.A)$, but such a translation leads to a definition of $\texttt{mv}\ \texttt{in}\ n.P$ where the type of $\texttt{mv}\ \texttt{in}\ n.P$ is one greater than the type of P, and so does not allow the use of objective moves inside recursion. Therefore we propose an alternative definition.

$$\begin{aligned}
\texttt{allow}\ n &\triangleq (\mathsf{fix}\, A{=}\mathsf{open}\ n.n[A]) \\
n^{\downarrow}[P] &\triangleq n[P \mid \texttt{allow}\ in] \\
n^{\uparrow}[P] &\triangleq n[P] \mid \texttt{allow}\ out \\
n^{\downarrow\uparrow}[P] &\triangleq n[P \mid \texttt{allow}\ in] \mid \texttt{allow}\ out \\
\texttt{mv}\ \texttt{in}\ n.P &\triangleq (\nu k)k[\mathsf{in}\ n.in[\mathsf{out}\ k.\mathsf{open}\ k.\mathsf{open}\ in.P]] \\
\texttt{mv}\ \texttt{out}\ n.P &\triangleq (\nu k)k[\mathsf{out}\ n.out[\mathsf{out}\ k.\mathsf{open}\ k.\mathsf{open}\ out.P]]
\end{aligned}$$

It is easy to see that all these processes are typable and are balanced; the least type of $\texttt{allow}\ n$ is 1, the least type of $\texttt{mv}\ \texttt{in}\ n.P$ and $\texttt{mv}\ \texttt{out}\ n.P$ is the maximum of the type of P and 2. One can check that $n^{\downarrow}[Q] \mid \texttt{mv}\ \texttt{in}\ n.P \rightarrow^* n^{\downarrow}[P \mid Q]$ and $n^{\uparrow}[\texttt{mv}\ \texttt{out}\ n.P \mid Q] \rightarrow^* n^{\uparrow}[Q] \mid P$.

4.3 Firewalls

Consider the firewall from [6]. This is a replication-free process

$$\mathit{firewall} = (\nu w)k[\mathsf{in}\ k.\mathsf{in}\ w] \mid w[\mathsf{open}\ k.P],$$

but it allows only one agent to enter the firewall. Let us first extend this example to allow for more agents. To avoid some confusion we replace one of the two occurrences of the name k with k': $\mathit{firewall} = (\nu w)!k[\mathsf{in}\ k'.\mathsf{in}\ w] \mid w[!\mathsf{open}\ k' \mid P]$. Then we have $k'[\mathsf{open}\ k.Q] \mid \mathit{firewall} \rightarrow^* (\nu w)!k[\mathsf{in}\ k'.\mathsf{in}\ w] \mid w[!\mathsf{open}\ k' \mid P \mid Q]$ and the firewall is still ready to allow more agents that are aware of the password (k, k').

We still have a little problem with modelling this firewall as a typable program. The process $k[\mathsf{in}\ k'.in\ w]$ is at the beginning outside the ambient w, but at the end (after the agent enters the firewall) it is inside w. In typable programs we need to always start a recursion in the same place that we end it. Therefore we first modify the firewall: $\mathit{firewall} = (\nu w)w[!k[\mathsf{out}\ w.\mathsf{in}\ k'.\mathsf{in}\ w] \mid !\mathsf{open}\ k' \mid P]$.

Now it is easy to see that this process behaves in the same way as the following program *firewall* where

$$\begin{aligned}
\mathit{firewall} &= (\nu w)w[hook \mid initiator \mid P] \\
hook &= (\mathsf{fix}\, A{=}k[\mathsf{out}\ w.in\ k.\mathsf{in}\ \ w.\mathsf{open}\ \ b.A]) \\
initiator &= (\mathsf{fix}\, B{=}\mathsf{open}\ k.b[B])
\end{aligned}$$

We use the additional ambient b to balance the bodies of the procedures $hook$ and $initiator$.

4.4 Routable Packets

Following [6] we define *packet pkt* as an empty packet named *pkt* that can be routed repeatedly to various destinations. Contrary to [6], we do not model routing as communicating the path to be followed (we restricted the calculus not to contain communication of compound messages), but by sending it another ambient containing the path.

$$\begin{aligned} \mathit{packet\ pkt} &\triangleq \mathit{pkt}[(\mathsf{fix}\, R{=}\mathsf{open}\ \mathit{route}.\mathit{route}[R])] \\ \mathit{route\ pkt\ with\ } P \mathit{\ to\ } M &\triangleq \mathit{route}[\mathsf{in}\ \mathit{pkt}.\mathsf{open}\ \mathit{route}.M \mid P] \\ \mathit{forward\ pkt\ to\ } M &\triangleq \mathit{route\ pkt\ with\ } \mathbf{0} \mathit{\ to\ } M \end{aligned}$$

Then there is an execution $\mathit{packet\ pkt} \mid \mathit{route\ pkt\ with\ } P \mathit{\ to\ } M \rightarrow^* \mathit{pkt}[M \mid P \mid P_R]$ where P_R is the process $(\mathsf{fix}\, R{=}\mathsf{open}\ \mathit{route}.\mathit{route}[R])$. Similarly, $\mathit{pkt}[P \mid P_R] \mid \mathit{forward\ pkt\ to\ } M \rightarrow^* \mathit{pkt}[M \mid P \mid P_R]$.

4.5 A Finite-Control π-Calculus

Here we encode a version of the finite-control π-calculus [12] without name passing in recursive procedures (that is, with parameterless recursive definitions) and without nondeterministic choice. An encoding of the full finite-control π-calculus seems possible using the extensions of our calculus discussed in Section 6.

Processes of the finite-control π-calculus

P ::=	process		
$(\nu n)P$	name restriction		
$T_1 \mid \ldots \mid T_k$	parallel threads		
T ::=	thread		
$\mathbf{0}$	inactivity	$(\nu n)T$	name restriction
$\alpha.T$	action	A	identifier
$(\mathsf{fix}\, A{=}T)$	recursion		
α::=	action		
$n(x)$	input on channel n		
$n\langle x\rangle$	output on channel n		

The encoding of the (asynchronous) π-calculus given in [6] cannot be used here for at least two reasons. First, the finite-control π-calculus uses synchronous communication while the communication in the ambient calculus is asynchronous. In order to simulate synchronous communication we have to run a synchronisation protocol. Second, dynamic generation of new channels strictly increases the size of the encoding and thus cannot be typable. Instead of this, we create new channels for every single communication and we destroy this channel immediately after the communication is finished.

To synchronize the communication, for every thread T_i we introduce an ambient $\mathit{sync}_i[]$ that avoids mixing the order of actions taken by this thread: every thread can send or receive at most one message at a time. Additionally we introduce one ambient $\mathit{lock}[]$ that allows processing only one communication at a time. These ambients are

present at the beginning, but they disappear (that is, they get opened) when the respective action starts and they reappear when the action is finished.

In the encoding given below we use an ambient named ch as a place where communication happens. The idea of this encoding is as follows. If two processes $n\langle M\rangle.P$ and $n(x).Q$ in threads T_i and T_j are willing to communicate, they start by opening the respective ambients $sync_i$ and $sync_j$ (if some of these ambients are not present, it means that the thread is busy with some other action, and the process has to wait). Then the output process leaves an ambient $n[]$ (this is the information that there is a message sent over the channel n) and moves inside the ambient ch. There it sends the message $\langle M\rangle$ within another ambient n. The input process opens the ambient $n[]$ (if there is no such ambient, it means that there is no message sent over channel n and the process has to wait), then it opens $lock[]$ (again, if there is no $lock[]$ ambient, it means that there is another communication just taking place and the process has to wait until it is finished) and goes inside ch and inside n where it reads the message M. The rest of the encoding is just to clean up afterwards: both processes go out of ch and together with the auxiliary processes $Sync_i$ and $Sync_j$ they synchronize the two threads and release the lock on communication, and remove all auxiliary ambients used in the meantime (more precisely, after the communication the ambient b gets opened, $done_i$ goes out of n and n goes inside $done_i$ where it gets opened; then $done_i$ moves outside ch, c gets opened, and the two *Sync* processes open $done_i$ and $done_j$; at this moment the ambients $sync_i[]$, $sync_j[]$ and $lock[]$ appear again at the top level). The ambients a, b, c are used to balance the process and to move inside ch.

Formally, the encoding is defined by the function $[\![\cdot]\!]$ from processes of the finite-control π-calculus to the finite-control ambient calculus. Except from communication, the encoding is quite straightforward: we have $[\![(\nu n)P]\!] \triangleq (\nu n)[\![P]\!]$,

$$[\![T_1 \mid \ldots \mid T_k]\!] \triangleq [\![T_1]\!]_1 \mid Sync_1 \mid \ldots \mid [\![T_k]\!]_k \mid Sync_k \mid ch^{\downarrow\uparrow}[] \mid lock[]$$

$[\![\mathbf{0}]\!]_i \triangleq \mathbf{0}$, $[\![(\nu n)T]\!]_i \triangleq (\nu n)[\![T]\!]_i$, $[\![A]\!]_i \triangleq A$, $[\![(\mathsf{fix}\, A{=}T)]\!]_i \triangleq (\mathsf{fix}\, A{=}[\![T]\!]_i)$,

$$[\![n\langle M\rangle.P]\!]_i \triangleq \mathsf{open}\ sync_i.(n[] \mid \mathtt{mv}\ \mathtt{in}\ ch.n[\langle M\rangle \mid \mathsf{open}\ a.PostOut_i(P)]$$
$$[\![n(x).Q]\!]_j \triangleq \mathsf{open}\ sync_j.\mathsf{open}\ n.\mathsf{open}\ lock.\mathtt{mv}\ \mathtt{in}\ ch.a[\mathsf{in}\ n.(x).PostIn_j(Q)]$$

where

$$Sync_i \triangleq sync_i[] \mid (\mathsf{fix}\, S_i{=}\mathsf{open}\ done_i.(sync_i[] \mid S_i))$$
$$PostOut_i(P) \triangleq \mathsf{open}\ b.(\mathsf{in}\ done_i.lock[] \mid done_i[\mathsf{out}\ n.\mathsf{open}\ n.\mathsf{out}\ ch.\mathsf{open}\ c.[\![P]\!]_i])$$
$$PostIn_j(Q) \triangleq b[c[done_j[[\![Q]\!]_j]]]$$

5 Ambient Logic and Model Checking

To reason about distributed and mobile computations programmed in the ambient calculus, Cardelli and Gordon [5] introduce a modal logic that apart from standard temporal modalities for describing the evolution of processes includes novel spatial modalities for describing the tree structure of ambient processes. In a recent paper, Cardelli and Gordon extend the logic with the constructs for describing private names [7].

The *model checking* problem is to decide whether a given object (in our case, an ambient process) satisfies (that is, is a model of) a given formula. Cardelli and Gordon [5]

give a model checking algorithm for the fragment of the calculus in which the processes contain no replications and no dynamic name generation against a fragment of the logic in which formulas contain no composition adjunct. It was then proved in [10] that model checking this fragment of the calculus against this fragment of the logic is PSPACE-complete. Recently, in [11] it has been shown that on the one hand, extending the calculus with name restriction and the logic with corresponding logical operators is harmless for the complexity of model checking—it remains PSPACE—and on the other hand that either considering replication in the calculus or composition adjunct in the logic makes the model checking problem undecidable.

5.1 Ambient Logic

We recall in this section definitions concerning this logic (omitting the composition adjunct).

In addition to the reduction relation and the structural congruence, we introduce an additional relation called *location* and denoted $\downarrow$ to reason about the shape of ambients (that is, space). The location relation is defined as $P \downarrow Q$ if there exists Q', n such that $P \equiv n[Q] \mid Q'$. We write $\downarrow^*$ for the reflexive and transitive closure of $\downarrow$.

We describe the syntax of the ambient logic and its satisfaction relation in the following tables.

Logical Formulas:

η	a name n or a variable x		
$\mathcal{A}, \mathcal{B} ::=$	formula	$\eta[\mathcal{A}]$	location
$\mathbf{T}$	true	$\mathcal{A}@\eta$	location adjunct
$\neg\mathcal{A}$	negation	$\eta\circledR\mathcal{A}$	revelation
$\mathcal{A} \vee \mathcal{B}$	disjunction	$\mathcal{A} \oslash \eta$	revelation adjunct
$\mathbf{0}$	void	$\Diamond\mathcal{A}$	sometime modality
$\mathcal{A} \mid \mathcal{B}$	composition match	$\diamondsuit\mathcal{A}$	somewhere modality
		$\exists x.\mathcal{A}$	existential quantification

We assume that names and variables belong to two disjoint vocabularies. We write $\mathcal{A}\{x \leftarrow m\}$ for the outcome of substituting each free occurrence of the variable x in the formula $\mathcal{A}$ with the name m. We say a formula $\mathcal{A}$ is closed if and only if it has no free variables (though it may contain free names).

The *satisfaction relation* $P \models \mathcal{A}$ provides the semantics of our logic. It is stable with respect to structural congruence, that is, if $P \models \mathcal{A}$ and $P \equiv P'$ then $P' \models \mathcal{A}$.

Satisfaction $P \models \mathcal{A}$ (for $\mathcal{A}$ closed):

$$
\begin{array}{ll}
P \models \mathbf{T} & \\
P \models \neg\mathcal{A} & \triangleq \neg(P \models \mathcal{A}) \\
P \models \mathcal{A} \vee \mathcal{B} & \triangleq P \models \mathcal{A} \vee P \models \mathcal{B} \\
P \models \mathbf{0} & \triangleq P \equiv \mathbf{0} \\
P \models \mathcal{A} \mid \mathcal{B} & \triangleq \exists P', P''.\ P \equiv P' \mid P'' \wedge P' \models \mathcal{A} \wedge P'' \models \mathcal{B} \\
P \models n[\mathcal{A}] & \triangleq \exists P'.P \equiv n[P'] \wedge P' \models \mathcal{A} \\
P \models \mathcal{A}@n & \triangleq n[P] \models \mathcal{A} \\
P \models n\circledR\mathcal{A} & \triangleq \exists P'.P \equiv (\nu n)P' \wedge P' \models \mathcal{A} \\
P \models \mathcal{A} \oslash n & \triangleq (\nu n)P \models \mathcal{A} \\
P \models \Diamond\mathcal{A} & \triangleq \exists P'.P \rightarrow^* P' \wedge P' \models \mathcal{A} \\
P \models \diamondsuit\mathcal{A} & \triangleq \exists P'.P \downarrow^* P' \wedge P' \models \mathcal{A} \\
P \models \exists x.\mathcal{A} & \triangleq \exists m.P \models \mathcal{A}\{x \leftarrow m\}
\end{array}
$$

5.2 Model Checking Finite-Control Mobile Ambients

In this section we show how closed and typable processes can be model checked against formulas of the ambient logic. We assume that in any process bound identifiers are pairwise distinct and that bound names are pairwise distinct and different from free names (not only free names from the process itself, but also free names occurring in formulas).

We consider here normalized processes as introduced in Section 3.2. To single out name restrictions, we write a normalized process $(\nu n_1) \ldots (\nu n_k)\mathcal{C}[P_1, \ldots, P_l]$ as a pair $\langle \{n_1, \ldots, n_k\}, \mathcal{C}[P_1, \ldots, P_l] \rangle$, separating name restriction prefix from the rest of the process and considering these name restrictions as a set of names. [1]

In a normalized process, only the active-context part is addressed by spatial modalities from the logic, that is, the modalities $\mathcal{A} \mid \mathcal{B}$, $n[\mathcal{A}]$, $\Diamond\mathcal{A}$ and $\mathbf{0}$. This allows us to control the size and the number of normalized processes considered for model checking these spatial modalities.

The following propositions (Propositions 7–10) express that in polynomial space we can test whether a process is congruent to $\mathbf{0}$, we can decompose it in all possible ways to a parallel composition of two other processes, we can remove the given leading ambient (if it exists), and we can compute all sublocations of the process. This is possible because it requires examining only the active context of the given process. The proof of these propositions is based on the following lemma.

Lemma 2 (Inversion). *Let P, Q, Q' be normalized processes.*

1. *$(\nu n)P \equiv \mathbf{0}$ if and only if $P \equiv \mathbf{0}$.*
2. *If n and m are different names, then $(\nu n)P \equiv m[Q]$ if and only if there exists a normalized process R such that $P \equiv m[R]$ and $Q \equiv (\nu n)R$.*
3. *$(\nu n)P \equiv Q \mid Q'$ if and only if there exist normalized processes R, R' such that $P \equiv R \mid R'$ and either $Q \equiv (\nu n)R$ and $Q' \equiv R'$ and $n \notin fn(Q')$ or $Q \equiv R$ and $Q' \equiv (\nu n)R'$ and $n \notin fn(Q)$.*

Proof. The proof is the same as for the replication-free fragment of the ambient calculus (Proposition 5.1 in [11]), observing that we have to examine only the active context of a normalized process. □

Proposition 7. *For any normalized process $\langle N, \tilde{P} \rangle$, $\langle N, \tilde{P} \rangle \equiv \mathbf{0}$ if and only if $\tilde{P} \equiv \mathbf{0}$. Furthermore, we can test whether $\tilde{P}$ is congruent to $\mathbf{0}$ in polynomial-time.*

Proposition 8. *For any normalized process $\langle N, \tilde{P} \rangle$, we can compute in polynomial space a finite set of pairs of normalized processes that we denote $Decomp(\langle N, \tilde{P} \rangle)$ and defined as $\{(\langle N_1, \tilde{Q}_1 \rangle, \langle N'_1, \tilde{R}_1 \rangle), \ldots, (\langle N_p, \tilde{Q}_p \rangle, \langle N'_p, \tilde{R}_p \rangle)\}$ satisfying:*

- *for all Q,R satisfying $\langle N, \tilde{P} \rangle \equiv Q \mid R$, there exists i such that $\langle N_i, \tilde{Q}_i \rangle \equiv Q$ and $\langle N'_i, \tilde{R}_i \rangle \equiv R$.*
- *for all i in $1 \ldots p$, $N_i \cup N'_i = N$ and $N_i \cap N'_i = \varnothing$, $fn(\tilde{Q}_i) \cap N'_i = \varnothing$ and $fn(\tilde{R}_i) \cap N_i = \varnothing$ and $|\tilde{Q}_i|, |\tilde{R}_i| \leq |\tilde{P}|$.*

[1] Whenever two different normalized processes P and P' have the same pair-representation, then $P \equiv P'$ by the axiom (Str Res Res) from the structural congruence.

Proposition 9. *For any normalized process $\langle N, \tilde{P}\rangle$ and any name n, we can test in polynomial time if there exists Q such that $\langle N, \tilde{P}\rangle \equiv n[Q]$. Moreover, if such a Q exists, then $n \notin N$ and we can compute in polynomial time the normalized version $\langle N, \tilde{Q}\rangle$ of Q such that $|\tilde{Q}| \leq |\tilde{P}|$.*

Combining the two previous propositions, we obtain:

Proposition 10. *For any normalized process $\langle N, \tilde{P}\rangle$, we can compute a finite set of normalized processes $Sublocations(\langle N, \tilde{P}\rangle) = \{\langle N_1, \tilde{Q_1}\rangle, \ldots, \langle N_p, \tilde{Q_p}\rangle\}$ such that (i) for all Q such that $\langle N, \tilde{P}\rangle \downarrow^* Q$, there exists i satisfying that $Q \equiv \langle N_i, \tilde{Q_i}\rangle$ and (ii) for all i in $1 \ldots p$, $N_i \subseteq N$ and $|\tilde{Q_i}| \leq |\tilde{P}|$.*

Moreover, using results from Section 3.2, we obtain:

Proposition 11. *For any typable and normalized process $\langle N, \tilde{P}\rangle$, we can compute a finite set of normalized processes $Reachable(\langle N, \tilde{P}\rangle) = \{\langle N_1, \tilde{Q_1}\rangle, \ldots, \langle N_p, \tilde{Q_p}\rangle\}$ such that (i) for all Q such that $\langle N, \tilde{P}\rangle \rightarrow^* Q$, there exists i satisfying that $Q \equiv \langle N_i, \tilde{Q_i}\rangle$ and (ii) for all i in $1 \ldots p$, $N_i \subseteq N$ and $|\tilde{Q_i}| \leq |\tilde{P}|^2$.*

The algorithm presented here is very close to the one given in [11].

Model Checking Algorithm: $Check(\langle N, \tilde{P}\rangle, \mathcal{A})$ **where** $N \cap fn(\mathcal{A}) = \varnothing$**, by convention**

$$
\begin{array}{l}
Check(\langle N, \tilde{P}\rangle, \mathbf{T}) \triangleq \mathbf{T} \\
Check(\langle N, \tilde{P}\rangle, \neg\mathcal{A}) \triangleq \neg Check(\langle N, \tilde{P}\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, \mathcal{A} \vee \mathcal{B}) \triangleq Check(\langle N, \tilde{P}\rangle, \mathcal{A}) \vee Check(\langle N, \tilde{P}\rangle, \mathcal{B}) \\
Check(\langle N, \tilde{P}\rangle, \mathbf{0}) \triangleq \begin{cases} \mathbf{T} \text{ if } \tilde{P} \equiv \mathbf{0} \\ \mathbf{F} \text{ otherwise} \end{cases} \\
Check(\langle N, \tilde{P}\rangle, \mathcal{A} \mid \mathcal{B}) \triangleq \bigvee_{(P_1,P_2)\in Decomp(\langle N,\tilde{P}\rangle)} Check(P_1, \mathcal{A}) \wedge Check(P_2, \mathcal{B}) \\
\qquad\qquad P_1, P_2 \text{ being respectively } \langle N_1, \tilde{P_1}\rangle \text{ and } \langle N_2, \tilde{P_2}\rangle \\
Check(\langle N, \tilde{P}\rangle, n[\mathcal{A}]) \triangleq \tilde{P} \equiv n[\tilde{Q}] \wedge Check(\langle N, \tilde{Q}\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, \mathcal{A}@n) \triangleq Check(\langle N, n[\tilde{P}]\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, n\circledR\mathcal{A}) \triangleq \bigvee_{m\in N} Check(\langle N - \{m\}, \tilde{P}\{m\leftarrow n\}\rangle, \mathcal{A}) \\
\qquad\qquad \vee (n \notin fn(\tilde{P}) \wedge Check(\langle N, \tilde{P}\rangle, \mathcal{A})) \\
Check(\langle N, \tilde{P}\rangle, \mathcal{A} \oslash n) \triangleq Check(\langle N \cup \{n\}, \tilde{P}\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, \Diamond\mathcal{A}) \triangleq \bigvee_{\langle N',\tilde{P'}\rangle\in Reachable(\langle N,\tilde{P}\rangle)} Check(\langle N', \tilde{P'}\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, \diamond\mathcal{A}) \triangleq \bigvee_{\langle N,\tilde{P'}\rangle\in Sublocations(N,P)} Check(\langle N, \tilde{P'}\rangle, \mathcal{A}) \\
Check(\langle N, \tilde{P}\rangle, \exists x.\mathcal{A}) \triangleq \text{let } n_0 \notin N \cup fn(\tilde{P}) \cup bn(\tilde{P}) \cup fn(\mathcal{A}) \text{ be a fresh name in} \\
\qquad\qquad \bigvee_{n\in fn(N,\tilde{P})\cup fn(\mathcal{A})\cup\{n_0\}} Check(\langle N, \tilde{P}\rangle, \mathcal{A}\{x\leftarrow n\})
\end{array}
$$

Theorem 2 (Correctness). *For all normalized and typable processes $\langle N, \tilde{P}\rangle$ and all closed formulas $\mathcal{A}$, we have $\langle N, \tilde{P}\rangle \models \mathcal{A}$ if and only if $Check(\langle N, \tilde{P}\rangle, \mathcal{A}) = \mathbf{T}$.*

Sketch of proof. The proof follows the lines of the proof of Theorem 5.1 in [11] and goes by induction on the formula $\mathcal{A}$. In the cases of $\mathbf{T}$, $\neg\mathcal{A}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A}@n$, $\mathcal{A} \oslash n$ the result follows directly from the definition of the satisfaction relation. In the case of $\mathbf{0}$ and $n[\mathcal{A}]$ it follows from Propositions 7 and 9, and in the case of $\mathcal{A} \mid \mathcal{B}$ from Proposition 8. The cases of $\diamond\mathcal{A}$ and $\Diamond\mathcal{A}$ follow from Propositions 10 and 11. The case of $\exists x.\mathcal{A}$ follows

the lines of the proof of Proposition 4.11 in [4]. Finally, the case of $n®\mathcal{A}$ reflects the two possibilities that either n is one of the bounded names occurring in the process or it does not occur there (in the latter case observe that for all processes Q, $n \notin fn(Q)$ implies $(\nu n)Q \equiv Q$). □

Theorem 3 (PSPACE-complete). *The model checking problem for finite-control processes against the ambient logic is decidable. Moreover, it is PSPACE-complete.*

Sketch of proof. Decidability follows from Theorem 2. One obtains the PSPACE upper bound by implementing disjunction in polynomial space, as is done in [10]. The PSPACE lower bound is proved in [10]. □

6 Extensions of the System

In this section we discuss some extensions that are possible to the calculus without affecting decidability or complexity of the model checking problem. We did not introduce these extensions before because we want to keep our formal presentation of a finite-control ambient calculus as simple as possible.

Parameters in recursive definitions. In the system we defined the identifiers used in recursive definitions do not carry any name parameters. It is however quite straightforward to allow definitions of the form $(\mathsf{fix}\, A(\boldsymbol{x})=P[A\langle \boldsymbol{y}\rangle])$ — one has to clearly distinguish between definitions (λ-abstractions) and calls (λ-application) of such functions and then respectively handle the renaming of parameters.

Nondeterministic choice. In the ambient calculus one may encode an internal nondeterministic choice $P + Q$ (see [6] for an encoding of an external choice) as the process $(\nu n)(n[\mathbf{0}] \mid \mathsf{open}\ n.P \mid \mathsf{open}\ n.Q)$.

Then reducing $P + Q$ leads to either $P \mid (\nu n)\mathsf{open}\ n.Q$ or to $Q \mid (\nu n)\mathsf{open}\ n.P$. As $(\nu n)\mathsf{open}\ n.R$ is bisimilar to $\mathbf{0}$, this is a good approximation of nondeterministic choice. However, even this simple encoding is no longer possible in recursive processes of the finite-control fragment, since it goes beyond well-formed processes. But even if we ignore the well-formedness restriction, such an encoding does not work because $(\nu n)\mathsf{open}\ n.R$ is not congruent (it is only bisimilar) to $\mathbf{0}$, which means that its type must be strictly positive, so it is not possible to balance the type of $P + Q$ if P and Q are balanced (and thus $P + Q$ is not typable). In the encoding of the finite-control π-calculus in Section 4.5 all recursive processes have balanced types and thus to extend the encoding to accommodate nondeterministic choice we need a balanced encoding of choice.

A possible solution is to add nondeterministic choice as a primitive construct in the calculus. To do so, we need to relax the definition of well-formed processes from one occurrence of identifier in a recursive definition to one occurrence per option of a nondeterministic choice. The reduction rules for processes can be then extended in a straightforward way, and an appropriate typing rule is:

$$\text{(Choice)}\quad \frac{\Gamma \vdash P \,:\, \tau\ ,\ \Gamma \vdash Q \,:\, \theta}{\Gamma \vdash P + Q \,:\, \max(\tau, \theta)}$$

Replication-free fragment of the ambient calculus. Our initial motivation was to find a fragment of the ambient calculus that extends the replication-free fragment (for which the decidability and complexity of the model checking problem was known [5,10]) to allow some infinite computation, while retaining a decidable model checking problem. The calculus of this paper does not extend the replication-free fragment because it does not allow for sending capabilities inside messages. It is however quite obvious that a typable finite-control process can be put in a replication-free context without any change to the model checking algorithm. The only subtle point is that if one wants to achieve a PSPACE algorithm one should apply the data structure from [10] only to the replication-free context; otherwise storing an explicit substitution for every communication might lead (in the case of recursive communications) to infinitely growing substitutions.

Sending capabilities in communication. In the current version of the calculus we allow for sending only names. The extension to sending single capabilities is however not difficult. The problem with sending single capabilities is probably best shown in the following process : $\langle \mathsf{in}\, n\rangle \mid (\mathsf{fix}\, A{=}(x).(\langle \mathsf{in}\, x\rangle \mid A))$. After the first iteration the process sends the message $\mathsf{in}\ (\mathsf{in}\ n)$, then $\mathsf{in}\ (\mathsf{in}\ (\mathsf{in}\ n))$, and so on, growing infinitely. Probably the simplest solution is to observe that $\mathsf{in}\ x$ cannot be executed if x is not a name, so it is enough to introduce a special deadlock capability and replace these complex capabilities with the deadlock capability.

An intriguing alternative possibility to solve the problem would be to combine the calculus with the type system of [8], where $\mathsf{in}\ (\mathsf{in}\ n)$ cannot be well-typed.

Sending sequences of capabilities. In the original definition of the ambient calculus [6] it is possible to send not only single capabilities but also sequences of capabilities. We do not see a very easy solution to this problem. Consider as an example the following process: $\langle \mathsf{in}\ n\rangle \mid (\mathsf{fix}\, A{=}(x).(\langle x.x\rangle \mid A))$. Here, after k iterations we obtain a sequence of capabilities $\mathsf{in}\ n \ldots \mathsf{in}\ n$ of length 2^k. Thus the process grows infinitely. A possible solution is to distinguish between simple communications (sending a name or a single capability) and complex communications (sending a sequence of length ≥ 2), and to give a simple output type 1 (as in the current version of the system) while a complex output is typed 2 (that is, to introduce an additional typing rule $\Gamma \vdash \langle M\rangle : 2$ for complex M). Then the above process is not typable, but for example the process $(\mathsf{fix}\, A{=}(x).(y).(\langle x.y\rangle \mid A))$ is typable. The decidability of model checking relies then on the observation that such a process cannot be executed infinitely many times (roughly, it can be executed as many times as many outputs are present in the context around).

Again, to achieve PSPACE complexity one has to be careful about substitutions—one should apply the substitution in the case of simple communication but one should store the substitution in the data structure (as is done in [10]) in the case of complex communication.

7 Conclusion

Previous work on model checking the spatial and temporal logic of the ambient calculus is limited to processes lacking any form of recursion or iteration. This work shows the

possibility of model checking a richer, more expressive class of mobile behaviours. We hope it will lead to the discovery of further applications of the ambient logic.

References

1. R.M. Amadio and Ch. Meyssonnier. On the decidability of fragments of the asynchronous pi-calculus. In *Electronic Notes in Theoretical Computer Science, Proceedings EXPRESS 2001*, 2001.
2. M. Bugliesi and G. Castagna. Secure safe ambients. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 222–235, 2001.
3. L. Cardelli and A. D. Gordon. Equational properties of mobile ambients. In *Proceedings FoSSaCS'99*, volume 1578 of *LNCS*, pages 212–226. Springer, 1999.
4. L. Cardelli and A. D. Gordon. Modal logics for mobile ambients: Semantic reasoning. Unpublished annex to [5], 1999.
5. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings POPL'00*, pages 365–377. ACM, January 2000.
6. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
7. L. Cardelli and A. D. Gordon. Logical properties of name restriction. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA'01)*, volume 2044 of *LNCS*, pages 46–60. Springer, 2001.
8. L. Cardelli and A.D. Gordon. Types for mobile ambients. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 79–92, 1999.
9. S. Chaki, S.K. Rajamani, and J.Rehof. Types as models: Model checking message-passing programs. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002. To appear.
10. W. Charatonik, S. Dal Zilio, A. D. Gordon, S. Mukhopadhyay, and J.-M. Talbot. The complexity of model checking mobile ambients. In *Proceedings FoSSaCS'01*, volume 2030 of *LNCS*, pages 152–167. Springer, 2001. An extended version appearas as Technical Report MSR–TR–2001–03, Microsoft Research, 2001.
11. W. Charatonik and J.-M. Talbot. The decidability of model checking mobile ambients. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *LNCS*, pages 339–354. Springer, 2001.
12. M. Dam. Model checking mobile processes. *Information and Computation*, 121(1):35–51, 1996.
13. R.R. Hansen, J.G. Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. In *Static Analysis (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1999.
14. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 352–364, 2000.
15. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
16. F. Nielson, H. Riis Nielson, R.R. Hansen, and J.G. Jensen. Validating firewalls in mobile ambients. In *Concurrency Theory (Concur'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 463–477. Springer, 1999.
17. D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proceedings POPL'01*, pages 4–13. ACM, January 2001.