On Hierarchical Communication Topologies in the π-calculus

Emanuele D'Osualdo[1(✉)] and C.-H. Luke Ong[2]

[1] TU Kaiserslautern, Kaiserslautern, Germany
dosualdo@cs.uni-kl.de
[2] University of Oxford, Oxford, UK
lo@cs.ox.ac.uk

Abstract. This paper is concerned with the shape invariants satisfied by the communication topology of π-terms, and the automatic inference of these invariants. A π-term P is *hierarchical* if there is a finite forest $\mathcal{T}$ such that the communication topology of every term reachable from P satisfies a $\mathcal{T}$-shaped invariant. We design a static analysis to prove a term hierarchical by means of a novel type system that enjoys decidable inference. The soundness proof of the type system employs a non-standard view of π-calculus reactions. The coverability problem for hierarchical terms is decidable. This is proved by showing that every hierarchical term is depth-bounded, an undecidable property known in the literature. We thus obtain an expressive static fragment of the π-calculus with decidable safety verification problems.

1 Introduction

Concurrency is pervasive in computing. A standard approach is to organise concurrent software systems as a dynamic collection of processes that communicate by message passing. Because processes may be destroyed or created, the number of processes in the system changes in the course of the computation, and may be unbounded. Moreover the messages that are exchanged may contain process addresses. Consequently the *communication topology* of the system—the hypergraph [19,20] connecting processes that can communicate directly—evolves over time. In particular, the connectivity of a process (i.e. its neighbourhood in this hypergraph) can change dynamically. The design and analysis of these systems is difficult: the dynamic reconfigurability alone renders verification problems undecidable. This paper is concerned with *hierarchical systems*, a new subclass of concurrent message-passing systems that enjoys decidability of safety verification problems, thanks to a shape constraint on the communication topology.

The π-calculus of Milner, Parrow and Walker [20] is a process calculus designed to model systems with a dynamic communication topology. In the π-calculus, processes can be spawned dynamically, and they communicate by exchanging messages along synchronous channels. Furthermore channel names can themselves be created dynamically, and passed as messages, a salient feature known as *mobility*, as this enables processes to modify their neighbourhood at runtime.

P. Thiemann (Ed.): ESOP 2016, LNCS 9632, pp. 149–175, 2016.
DOI: 10.1007/978-3-662-49498-1_7

It is well known that the π-calculus is a Turing-complete model of computation. Verification problems on π-terms are therefore undecidable in general. There are however useful fragments of the calculus that support automatic verification. The most expressive such fragment known to date is the *depth-bounded* π-calculus of Meyer [13]. Depth boundedness is a constraint on the shape of communication topologies. A π-term is *depth-bounded* if there is a number k such that every simple path[1] in the communication topology of every reachable π-term has length bounded by k. Meyer [15] proved that termination and coverability (a class of safety properties) are decidable for depth-bounded terms.

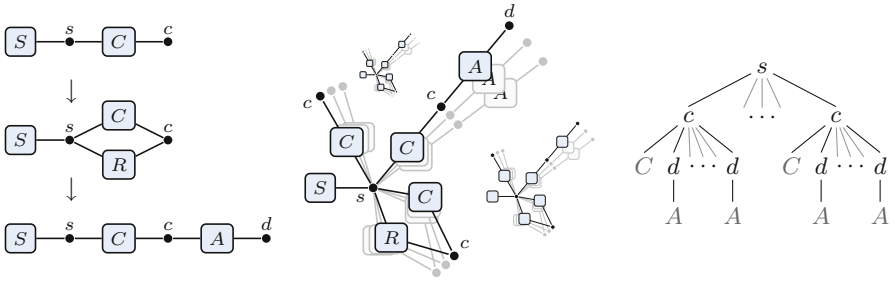
Unfortunately depth boundedness itself is an undecidable property [15], which is a serious impediment to the practical application of the depth-bounded fragment to verification. This paper offers a two-step approach to this problem. First we identify a (still undecidable) subclass of depth-bounded systems, called *hierarchical*, by a shape constraint on communication topologies (as opposed to numeric, as in the case of depth-boundedness). Secondly, by exploiting this richer structure, we define a type system, which in turn gives a *static* characterisation of an expressive and practically relevant fragment of the depth-bounded π-calculus.

Example 1 (Client-server pattern). To illustrate our approach, consider a simple system implementing a client-server pattern. A server S is a process listening on a channel s which acts as its address. A client C knows the address of a server and has a private channel c that represents its identity. When the client wants to communicate with the server, it asynchronously sends c along the channel s. Upon receipt of the message, the server acquires knowledge of (the address of) the requesting client; and spawns a process A to answer the client's request R asynchronously; the answer consists of a new piece of data, represented by a new name d, sent along the channel c. Then the server forgets the identity of the client and reverts to listening for new requests. Since only the requesting client knows c at this point, the server's answer can only be received by the correct client. Figure 1a shows the communication topology of a server and a client, in the three phases of the protocol.

The overall system is composed of an unbounded number of servers and clients, constructed according to the above protocol. The topology of a reachable configuration is depicted in Fig. 1b. While in general the topology of a mobile system can become arbitrarily complex, for such common patterns as client-server, the programmer often has a clear idea of the desired shape of the communication topology: there will be a number of servers, each with its cluster of clients; each client may in turn be waiting to receive a number of private replies. This suggests a hierarchical relationship between the names representing servers, clients and data, although the communication topology itself does not form a tree.

$\mathcal{T}$-compatibility and Hierarchical Terms. Recall that in the π-calculus there is an important relation between terms, $\equiv$, called *structural congruence*,

[1] A simple path is a path with no repeating edges.



(a) the protocol (b) a reachable configuration (c) forest representation

Fig. 1. Evolution of the communication topology of a server interacting with a client. R represents a client's pending request and A a server's pending answer.

which equates terms that differ only in irrelevant presentation details, but not in behaviour. For instance, the structural congruence laws for restriction tell us that the order of restrictions is irrelevant—$\nu\, x.\, \nu\, y.P \equiv \nu\, y.\, \nu\, x.P$—and that the scope of a restriction can be extended to processes that do not refer to the restricted name—i.e., $(\nu\, x.P) \parallel Q \equiv \nu\, x.(P \parallel Q)$ when x does not occur free in Q—without altering the meaning of the term. The former law is called *exchange*, the latter is called *scope extrusion*.

Our first contribution is a formalisation in the π-calculus of the intuitive notion of hierarchy illustrated in Example 1. We shall often speak of the *forest representation* of a π-term P, forest(P), which is a version of the abstract syntax tree of P that captures the nesting relationship between the active restrictions of the term. (A restriction of a π-term is *active* if it is not in the scope of a prefix.) Thus the internal nodes of a forest representation are labelled with (active) restriction names, and its leaf nodes are labelled with the sequential subterms. Given a π-term P, we are interested in not just forest(P), but also forest(P') where P' ranges over the structural congruents of P, because these are all behaviourally equivalent representations. See Fig. 4 for an example of the respective forest representations of the structural congruents of a term. In our setting a hierarchy $\mathcal{T}$ is a *finite* forest of what we call *base types*. Given a finite forest $\mathcal{T}$, we say that a term P is $\mathcal{T}$*-compatible* if there is a term P', which is structurally congruent to P, such that the parent relation of forest(P') is consistent with the partial order of $\mathcal{T}$.

In Example 1 we would introduce base types srv, cl and data associated with the restrictions $\nu\, s$, $\nu\, c$ and $\nu\, d$ respectively, and we would like the system to be compatible to the hierarchy $\mathcal{T} = \mathsf{srv} \prec \mathsf{cl} \prec \mathsf{data}$, where $\prec$ is the is-parent-of relation. That is, we must be able to represent a configuration with a forest that, for instance, does not place a server name below a client name nor a client name below another client name. Such a representation is shown in Fig. 1c.

In the Example, we want every reachable configuration of the system to be compatible with the hierarchy. We say that a π-term P is *hierarchical* if there is a hierarchy $\mathcal{T}$ such that every term reachable from P is $\mathcal{T}$-compatible. Thus the hierarchy $\mathcal{T}$ is a shape invariant of the communication topology under reduction.

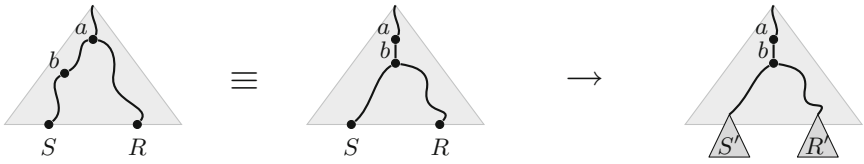


Fig. 2. Standard view of π-calculus reactions

It is instructive to express depth boundedness as a constraint on forest representation: a term P is depth-bounded if there is a constant k such that every term reachable from P has a structurally congruent P' whereby forest(P') has height bounded by k. It is straightforward to see that hierarchical terms are depth-bounded; the converse is however not true.

A Type System for Hierarchical Terms. While membership of the hierarchical fragment is undecidable, by exploiting the forest structure, we have devised a novel type system that guarantees the invariance of $\mathcal{T}$-compatibility under reduction. Furthermore type inference is decidable, so that the type system can be used to infer a hierarchy $\mathcal{T}$ with respect to which the input term is hierarchical. To the best of our knowledge, our type system is the first that can infer a shape invariant of the communication topology of a system.

The typing rules that ensure invariance of $\mathcal{T}$-compatibility under reduction arise from a new perspective of the π-calculus reaction, one that allows compatibility to a given hierarchy to be tracked more readily. Suppose we are presented with a $\mathcal{T}$-compatible term $P = C[S, R]$ where $C[\text{-}, \text{-}]$ is the *reaction context*, and the two processes $S = \overline{a}\langle b\rangle.S'$ and $R = a(x).R'$ are ready to communicate over a channel a. After sending the message b, S continues as the process S', while upon receipt of b, R binds x to b and continues as $R'' = R'[b/x]$. Schematically, the traditional understanding of this transaction is: first extrude the scope of b to include R, then let them react, as shown in Fig. 2.

Instead, we seek to implement the reaction *without scope extrusion*: after the message is transmitted, the sender continues in-place as S', while R'' is split in two parts $R''_{\text{mig}} \parallel R''_{\neg\text{mig}}$, one that uses the message (the *migratable* part) and one that does not. As shown in Fig. 3, the migratable part of R'', R''_{mig}, is "installed" under b so that it can make use of the acquired name, while the non-migratable one, $R''_{\neg\text{mig}}$, can simply continue in-place.

Crucially, the *reaction context*, $C[\text{-}, \text{-}]$, is left unchanged. This means that if the starting term is $\mathcal{T}$-compatible, the reaction context of the *reactum* is $\mathcal{T}$-compatible as well. We can then focus on imposing constraints on the use of names of R' so that the migration does not result in R''_{mig} escaping the scope of previously bound names.

By using these ideas, our type system is able to statically accept π-calculus encodings of such system as that discussed in Example 1. The type system can be used, not just to *check* that a given $\mathcal{T}$ is respected by the behaviour of a term, but also to *infer* a suitable $\mathcal{T}$ when it exists. Once typability of a term

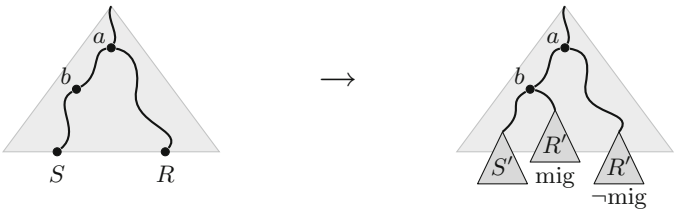


Fig. 3. $\mathcal{T}$-compatibility preserving reaction

is established, safety properties such as unreachability of error states, mutual exclusion or bounds on mailboxes, can be verified algorithmically. For instance, in Example 1, a coverability check can prove that each client can have at most one reply pending in its mailbox. To prove such a property, one needs to construct an argument that reasons about dynamically created names with a high degree of precision. This is something that counter abstraction and uniform abstractions based methods have great difficulty attaining.

Our type system is (necessarily) incomplete in that there are depth-bounded, or even hierarchical, systems that cannot be typed. The class of π-terms that can be typed is non-trivial, and includes terms which generate an unbounded number of names and exhibit mobility.

Outline. In Sect. 2 we review the π-calculus, depth-bounded terms, and related technical preliminaries. In Sect. 3 we introduce $\mathcal{T}$-compatibility and the hierarchical terms. We present our type system in Sect. 4. Section 5 discusses soundness of the type system. In Sect. 6 we give a type inference algorithm; and in Sect. 7 we present results on expressivity and discuss applications. We conclude with related and future work in Sects. 8 and 9. All missing definitions and proofs can be found in [5].

2 The π-calculus and the Depth-Bounded Fragment

2.1 Syntax and Semantics

We use a π-calculus with guarded replication to express recursion [17]. Fix a universe $\mathcal{N}$ of *names* representing channels and messages. The syntax is defined by the grammar:

$$\begin{aligned} \mathcal{P} \ni P, Q &::= \mathbf{0} \mid \nu x.P \mid P_1 \parallel P_2 \mid M \mid !M && \text{process} \\ M &::= M + M \mid \pi.P && \text{choice} \\ \pi &::= a(x) \mid \overline{a}\langle b \rangle \mid \boldsymbol{\tau} && \text{prefix} \end{aligned}$$

Definition 1. *Structural congruence, $\equiv$, is the least relation that respects α-conversion of bound names, and is associative and commutative with respect*

to + (choice) and ∥ (parallel composition) with $\mathbf{0}$ *as the neutral element, and satisfies laws for restriction:* $\nu\, a.\mathbf{0} \equiv \mathbf{0}$ *and* $\nu\, a.\, \nu\, b.P \equiv \nu\, b.\, \nu\, a.P$*, and*

$$!P \equiv P \parallel !P \qquad \text{Replication}$$

$$P \parallel \nu\, a.Q \equiv \nu\, a.(P \parallel Q) \quad (\text{if a} \notin \text{fn(P)}) \qquad \text{Scope Extrusion}$$

In $P = \pi.Q$, we call Q the *continuation* of P and will often omit Q altogether when $Q = \mathbf{0}$. In a term $\nu\, x.P$ we will occasionally refer to P as the *scope* of x. The name x is bound in both $\nu\, x.P$, and in $a(x).P$. We will write $\text{fn}(P)$, $\text{bn}(P)$ and $\text{bn}_\nu(P)$ for the set of free, bound and restriction-bound names in P, respectively. A sub-term is *active* if it is not under a prefix. A name is active when it is bound by an active restriction. We write $\text{act}_\nu(P)$ for the set of active names of P. Terms of the form M and $!M$ are called *sequential*. We write $\mathcal{S}$ for the set of sequential terms, $\text{act}_{\mathcal{S}}(P)$ for the set of active sequential processes of P, and P^i for the parallel composition of i copies of P.

Intuitively, a sequential process acts like a thread running finite-control sequential code. A term $\boldsymbol{\tau}.(P \parallel Q)$ is the equivalent of spawning a process Q and continuing as P —although in this context the rôles of P and Q are interchangeable. Interaction is by *synchronous* communication over channels. An *input prefix* $a(x)$ is a blocking receive on the channel a binding the variable x to the message. An *output prefix* $\overline{a}\langle b\rangle$ is a blocking send of the message b along the channel a; here b is itself the name of a channel that can be used subsequently for further communication: an essential feature for mobility. A non-blocking send can be simulated by spawning a new process doing a blocking send. Restrictions are used to make a channel name private. A replication $!(\pi.P)$ can be understood as having a server that can spawn a new copy of P whenever a process tries to communicate with it. In other words it behaves like an infinite parallel composition $(\pi.P \parallel \pi.P \parallel \cdots)$.

For conciseness, we assume channels are unary (the extension to the polyadic case is straightforward). In contrast to process calculi without mobility, replication and systems of tail recursive equations are equivalent methods of defining recursive processes in the π-calculus [18, Sect. 3.1].

We rely on the following mild assumption, that the choice of names is unambiguous, especially when selecting a representative for a congruence class:

Name Uniqueness Assumption. *Each name in* P *is bound at most once and* $\text{fn}(P) \cap \text{bn}(P) = \emptyset$.

Normal Form. The notion of hierarchy, which is central to this paper, and the associated type system depend heavily on structural congruence. These are criteria that, given a structure on names, require the existence of a specific representative of the structural congruence class exhibiting certain properties. However, we cannot assume the input term is presented as that representative; even worse, when the structure on names is not fixed (for example, when inferring types) we cannot fix a representative and be sure that it will witness the desired properties. Thus, in both the semantics and the type system, we manipulate a neutral type of representative called *normal form*, which is a variant of the *standard form* [20].

In this way we are not distracted by the particular syntactic representation we are presented with.

We say that a term P is in *normal form* ($P \in \mathcal{P}_{\mathsf{nf}}$) if it is in standard form and each of its inactive subterms is also in normal form. Formally, normal forms are defined by the grammar

$$\begin{aligned} \mathcal{P}_{\mathsf{nf}} \ni N &::= \nu\, x_1.\cdots \nu\, x_n.(A_1 \parallel \cdots \parallel A_m) \\ A &::= \pi_1.N_1 + \cdots + \pi_n.N_n \;\mid\; !\,(\pi_1.N_1 + \cdots + \pi_n.N_n) \end{aligned}$$

where the sequences $x_1 \ldots x_n$ and $A_1 \ldots A_m$ may be empty; when they are both empty the normal form is the term $\mathbf{0}$. We further assume w.l.o.g. that normal forms satisfy Name Uniqueness. Given a finite set of indexes $I = \{i_1, \ldots, i_n\}$ we write $\prod_{i\in I} A_i$ for $(A_{i_1} \parallel \cdots \parallel A_{i_n})$, which is $\mathbf{0}$ when I is empty; and $\sum_{i\in I} \pi_i.N_i$ for $(\pi_{i_1}.N_{i_1} + \cdots + \pi_{i_n}.N_{i_n})$. This notation is justified by commutativity and associativity of the parallel and choice operators. Thanks to the structural laws of restriction, we also write $\nu\, X.P$ where $X = \{x_1, \ldots, x_n\}$, or $\nu\, x_1\, x_2 \cdots x_n.P$, for $\nu\, x_1.\cdots \nu\, x_n.P$; or just P when X is empty. When X and Y are disjoint sets of names, we use juxtaposition for union.

Every process $P \in \mathcal{P}$ is structurally congruent to a process in normal form. The function $\mathrm{nf}\colon \mathcal{P} \to \mathcal{P}_{\mathsf{nf}}$, defined in [5], extracts, from a term, a structurally congruent normal form.

Given a process P with normal form $\nu\, X.\prod_{i\in I} A_i$, the *communication topology*[2] of P, written $\mathcal{G}[\![P]\!]$, is defined as the labelled hypergraph with X as hyperedges and I as nodes, each labelled with the corresponding A_i. An hyperedge $x \in X$ is connected with i just if $x \in \mathrm{fn}(A_i)$.

Semantics. We are interested in the reduction semantics of a π-term, which can be described using the following rule.

Definition 2 (Semantics of π-calculus). *The operational semantics of a term $P_0 \in \mathcal{P}$ is defined by the (pointed) transition system $(\mathcal{P}, \to, P_0)$ on π-terms, where P_0 is the initial term, and the transition relation, $\to \subseteq \mathcal{P}^2$, is defined by $P \to Q$ if either (i) to (iv) hold, or (v) and (vi) hold, where*

(i) $P \equiv \nu\, W.(S \parallel R \parallel C) \in \mathcal{P}_{\mathsf{nf}}$,
(ii) $S = (\overline{a}\langle b\rangle.\nu\, Y_s.S') + M_s$,
(iii) $R = (a(x).\nu\, Y_r.R') + M_r$,
(iv) $Q \equiv \nu\, W Y_s Y_r.(S' \parallel R'[b/x] \parallel C)$,
(v) $P \equiv \nu\, W.(\boldsymbol{\tau}.\nu\, Y.P' \parallel C) \in \mathcal{P}_{\mathsf{nf}}$,
(vi) $Q \equiv \nu\, W Y.(P' \parallel C)$.

We define the set of reachable terms from P as $\mathrm{Reach}(P) := \{Q \mid P \to^* Q\}$, *writing $\to^*$ to mean the reflexive, transitive closure of $\to$. We refer to the restrictions, $\nu\, Y_s$, $\nu\, Y_r$ and $\nu\, Y$, as the restrictions* activated *by the transition $P \to Q$.*

Notice that the use of structural congruence in the definition of $\to$ takes unfolding replication into account.

[2] This definition arises from the "flow graphs" of [20]; see e.g. [15, p. 175] for a formal definition.

Example 2 (Client-server). We can model a variation of the client-server pattern sketched in the introduction, with the term $\nu\, s\, c.P$ where $P = !S \parallel !C \parallel !M$, $S = s(x).\nu\, d.\overline{x}\langle d\rangle$, $C = c(m).(\overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle)$ and $M = \boldsymbol{\tau}.\nu\, m.\overline{c}\langle m\rangle$. The term $!S$ represents a server listening to a port s for a client's requests. A request is a channel x that the client sends to the server for exchanging the response. After receiving x the server creates a new name d and sends it over x. The term $!M$ creates unboundedly many clients, each with its own private mailbox m. A client on a mailbox m repeatedly sends requests to the server and concurrently waits for the answer on the mailbox before recursing.

In the following examples, we use CCS-style nullary channels, which can be understood as a shorthand: $c.P := c(x).P$ and $\overline{c}.P := \nu\, x.\overline{c}\langle x\rangle.P$ where $x \notin \mathrm{fn}(P)$.

Example 3 (Resettable counter). A counter with reset is a process reacting to messages on three channels *inc*, *dec* and *rst*. An *inc* message increases the value of the counter, a *dec* message decreases it or causes a deadlock if the counter is zero, and a *rst* message resets the counter to zero. This behaviour is exhibited by the process $C_i = !\left(p_i(t).\left(inc_i.(\overline{t} \parallel \overline{p_i}\langle t\rangle) + dec_i.(t.\overline{p_i}\langle t\rangle) + rst_i.(\nu\, t_i'.\overline{p_i}\langle t_i'\rangle)\right)\right)$. Here, the number of processes $\overline{t}$ in parallel with $\overline{p_i}\langle t\rangle$ represents the current value of the counter i. A system $\left(\nu\, p_1\, t_1.(C_1 \parallel \overline{p_1}\langle t_1\rangle) \parallel \nu\, p_2\, t_2.(C_2 \parallel \overline{p_2}\langle t_2\rangle)\right)$ can for instance simulate a two-counter machine when put in parallel with a finite control process sending signals along the channels inc_i, dec_i and rst_i.

Example 4 (Unbounded ring). Let $R = \nu\, m.\nu\, s_0.(M \parallel \overline{m}\langle s_0\rangle \parallel \overline{s_0})$, $S = !(s.\overline{n})$ and $M = !\left(m(n).s_0.\nu\, s.(S \parallel \overline{m}\langle s\rangle \parallel \overline{s})\right)$. The term R implements an unboundedly growing ring. It initialises the ring with a single "master" node pointing at itself (s_0) as the next in the ring. The term M, implementing the master node's behaviour, waits on s_0 and reacts to a signal by creating a new slave with address s connected with the previous next slave n. A slave S simply propagates the signals on its channel to the next in the ring.

2.2 Forest Representation of Terms

In the technical developement of our ideas, we will manipulate the structure of terms in non-trivial ways. When reasoning about these manipulations, a term is best viewed as a forest representing (the relevant part of) its abstract syntax tree. Since we only aim to capture the active portion of the term, the active sequential subterms are the leaves of its forest view. Parallel composition corresponds to (unordered) branching, and names introduced by restriction are represented by internal (non-leaf) nodes.

A *forest* is a simple, acyclic, directed graph, $f = (N_f, \prec_f)$, where the edge relation $n_1 \prec_f n_2$ means "n_1 is the parent of n_2". We write $\leq_f$ and $<_f$ for the reflexive transitive and the transitive closure of $\prec_f$ respectively. A *path* is a sequence of nodes, $n_1 \ldots n_k$, such that for each $i < k$, $n_i \prec_f n_{i+1}$. Henceforth we drop the subscript f from $\prec_f, \leq_f$ and $<_f$ (as there is no risk of confusion),

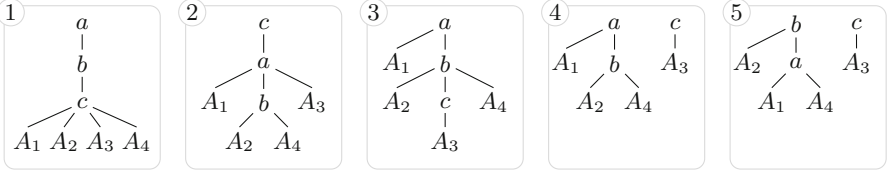


Fig. 4. Examples of forests in $\mathcal{F}[\![P]\!]$ where $P = \nu\, a\, b\, c.(A_1 \parallel A_2 \parallel A_3 \parallel A_4)$, $A_1 = a(x)$, $A_2 = b(x)$, $A_3 = c(x)$ and $A_4 = \overline{a}\langle b\rangle$.

and assume that all forests are finite. Thus every node has a unique path to a root (and that root is unique).

An *L-labelled forest* is a pair $\varphi = (f_\varphi, \ell_\varphi)$ where f_φ is a forest and $\ell_\varphi \colon N_\varphi \to L$ is a labelling function on nodes. Given a path $n_1 \dots n_k$ of f_φ, its *trace* is the induced sequence $\ell_\varphi(n_1) \dots \ell_\varphi(n_k)$. By abuse of language, a *trace* is an element of L^* which is the trace of some path in the forest.

We define L-labelled forests inductively from the empty forest $(\emptyset, \emptyset)$. We write $\varphi_1 \uplus \varphi_2$ for the disjoint union of forests φ_1 and φ_2, and $l[\varphi]$ for the forest with a single root, which is labelled with $l \in L$, and whose children are the respective roots of the forest φ. Since the choice of the set of nodes is irrelevant, we will always interpret equality between forests up to isomorphism (i.e. a bijection on nodes respecting parent and labeling).

Definition 3 (Forest representation). *We represent the structural congruence class of a term $P \in \mathcal{P}$ with the set of labelled forests $\mathcal{F}[\![P]\!] := \{\mathrm{forest}(Q) \mid Q \equiv P\}$ with labels in $\mathrm{act}_\nu(P) \uplus \mathrm{act}_{\mathcal{S}}(P)$ where $\mathrm{forest}(Q)$ is defined as*

$$\mathrm{forest}(Q) := \begin{cases} (\emptyset, \emptyset) & \text{if } Q = \mathbf{0} \\ Q[(\emptyset, \emptyset)] & \text{if } Q \text{ is sequential} \\ x[\mathrm{forest}(Q')] & \text{if } Q = \nu\, x.Q' \\ \mathrm{forest}(Q_1) \uplus \mathrm{forest}(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \end{cases}$$

Note that leaves (and only leaves) are labelled with sequential processes.

The restriction height, $\mathrm{height}_\nu(\mathrm{forest}(P))$, *is the length of the longest path formed of nodes labelled with names in* $\mathrm{forest}(P)$.

In Fig. 4 we show some of the possible forest representations of an example term.

2.3 Depth-Bounded Terms

Definition 4 (Depth-bounded term [13]). *The* nesting of restrictions *of a term is given by the function*

$$\begin{aligned} \mathrm{nest}_\nu(M) := \mathrm{nest}_\nu(!M) := \mathrm{nest}_\nu(\mathbf{0}) &:= 0 \\ \mathrm{nest}_\nu(\nu\, x.P) &:= 1 + \mathrm{nest}_\nu(P) \\ \mathrm{nest}_\nu(P \parallel Q) &:= \max(\mathrm{nest}_\nu(P), \mathrm{nest}_\nu(Q)). \end{aligned}$$

The depth *of a term is defined as the minimal nesting of restrictions in its congruence class,* $\mathrm{depth}(P) := \min\{\mathrm{nest}_\nu(Q) \mid P \equiv Q\}$. *A term* $P \in \mathcal{P}$ *is* depth-bounded *if there exists* $k \in \mathbb{N}$ *such that for each* $Q \in \mathrm{Reach}(P)$, $\mathrm{depth}(Q) \leq k$. *We write* $\mathcal{P}_{\mathsf{db}}$ *for the set of terms with bounded depth.*

It is straightforward to see that the nesting of restrictions of a term coincides with the height of its forest representation, i.e., for every $P \in \mathcal{P}$, $\mathrm{nest}_\nu(P) = \mathrm{height}_\nu(\mathrm{forest}(P))$.

Example 5 (Depth-bounded term). The term in Example 2 is depth-bounded: all the reachable terms are congruent to terms of the form

$$Q_{ijk} = \nu\, s\, c.\big(P \parallel N^i \parallel Req^j \parallel Ans^k\big)$$

for some $i, j, k \in \mathbb{N}$ where $N = \nu\, m.\overline{c}\langle m\rangle$, $Req = \nu\, m.(\overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle)$ and $Ans = \nu\, m.(\nu\, d.\overline{m}\langle d\rangle \parallel m(y).\overline{c}\langle m\rangle)$. For any i, j, k, $\mathrm{nest}_\nu(Q_{ijk}) \leq 4$.

Example 6 (Depth-unbounded term). Consider the term in Example 4 and the following run:

$$\begin{aligned} R &\to^* \nu\, m\, s_0.(M \parallel \nu\, s_1.(!(s_1.\overline{s_0}) \parallel \overline{m}\langle s_1\rangle \parallel \overline{s_1})) \\ &\to^* \nu\, m\, s_0.(M \parallel \nu\, s_1.(!(s_1.\overline{s_0}) \parallel \nu\, s_2.(!(s_2.\overline{s_1}) \parallel \overline{m}\langle s_2\rangle \parallel \overline{s_2}))) \to^* \ldots \end{aligned}$$

The scopes of s_0, s_1, s_2 and the rest of the instantiations of $\nu\, s$ are inextricably nested, thus R has unbounded depth: for each $n \geq 1$, a term with depth n is reachable.

Depth boundedness is a semantic notion. Because the definition is a universal quantification over reachable terms, analysis of depth boundedness is difficult. Indeed the membership problem is undecidable [15]. In the communication topology interpretation, depth has a tight relationship with the maximum length of the simple paths. A path $v_1 e_1 v_2 \ldots v_n e_n v_{n+1}$ in $\mathcal{G}[\![P]\!]$ is *simple* if it does not repeat hyper-edges, i.e., $e_i \neq e_j$ for all $i \neq j$. A term is depth-bounded if and only if there exists a bound on the length of the simple paths of the communication topology of each reachable term [13]. This allows terms to grow unboundedly in *breadth*, i.e., the degree of hyper-edges in the communication topology.

A term P is *embeddable* in a term Q, written $P \preceq Q$, if $P \equiv \nu\, X.\prod_{i\in I} A_i \in \mathcal{P}_{\mathsf{nf}}$ and $Q \equiv \nu\, XY.(\prod_{i\in I} A_i \parallel R) \in \mathcal{P}_{\mathsf{nf}}$ for some term R. In [13] the term embedding ordering, $\preceq$, is shown to be both a simulation relation on π-terms, and an effective well-quasi ordering on depth-bounded terms. This makes the transition system $(\mathrm{Reach}(P)/_\equiv, \to/_\equiv, P)$ a *well-structured transition system* (WSTS) [1,8] under the term embedding ordering. Consequently a number of verification problems are decidable for terms in $\mathcal{P}_{\mathsf{db}}$.

Theorem 1 (Decidability of termination [13]). *The termination problem for depth-bounded terms, which asks, given a term* $P_0 \in \mathcal{P}_{\mathsf{db}}$, *if there is an infinite sequence* $P_0 \to P_1 \to \ldots$, *is decidable.*

Theorem 2 (Decidability of coverability [13,25]). *The coverability problem for depth-bounded terms, which asks, given a term* $P \in \mathcal{P}_{\mathsf{db}}$ *and a* query $Q \in \mathcal{P}$, *if there exists* $P' \in \mathrm{Reach}(P)$ *such that* $Q \preceq P'$, *is decidable.*

3 $\mathcal{T}$-compatibility and Hierarchical Terms

A hierarchy is specified by a finite forest $(\mathcal{T}, \prec)$. In order to formally relate *active* restrictions in a term to nodes of the hierarchy $\mathcal{T}$, we annotate restrictions with types. For the moment we view types abstractly as elements of a set $\mathbb{T}$, equipped with a map base: $\mathbb{T} \to \mathcal{T}$. An annotated restriction $\nu(x:\tau)$ where $\tau \in \mathbb{T}$ will be associated with the node $\text{base}(\tau)$ in the hierarchy $\mathcal{T}$. Elements of $\mathbb{T}$ are called *types*, and those of $\mathcal{T}$ are called *base types*. In the simplest case and, especially for Sect. 3, we may assume $\mathbb{T} = \mathcal{T}$ and $\text{base}(t) = t$. In Sect. 4 we will consider a set $\mathbb{T}$ of types generated from $\mathcal{T}$, and a non-trivial base map.

Definition 5 (Annotated term). *A* $\mathbb{T}$-annotated π-term *(or simply* annotated π-term*)* $P \in \mathcal{P}^{\mathbb{T}}$ *has the same syntax as ordinary* π*-terms except that restrictions take the form* $\nu(x:\tau)$ *where* $\tau \in \mathbb{T}$*. In the abbreviated form* νX*,* X *is a set of annotated names* $(x:\tau)$*.*

Structural congruence, $\equiv$, of annotated terms, is defined by Definition 1, with the proviso that the type annotations are invariant under α-conversion and replication. For example, $!\,(\pi.\,\nu\,(x:\tau).P) \equiv \pi.\,\nu\,(x:\tau).P \parallel !\,(\pi.\,\nu\,(x:\tau).P)$ and $\nu\,(x:\tau).P \equiv \nu\,(y:\tau).P[y/x]$; observe that the annotated restrictions that occur in a replication unfolding are necessarily inactive.

The forest representation of an annotated π-term is obtained from Definition 3 by replacing the case of $Q = \nu\,(x:\tau).Q'$ by

$$\text{forest}(\nu\,(x:\tau).Q') := (x,t)[\text{forest}(Q')]$$

where $\text{base}(\tau) = t$. Thus the forests in $\mathcal{F}[\![P]\!]$ have labels in $(\text{act}_\nu(P) \times \mathcal{T}) \uplus \text{act}_{\mathcal{S}}(P)$. We write $\mathcal{F}_{\mathcal{T}}$ for the set of forests with labels in $(\mathcal{N} \times \mathcal{T}) \uplus \mathcal{S}$. We write $\mathcal{P}^{\mathbb{T}}_{\mathsf{nf}}$ for the set of $\mathbb{T}$-annotated π-terms in normal form.

The definition of the transition relation of annotated terms, $P \to Q$, is obtained from Definition 2, where W, Y_s, Y_r and Y are now sets of annotated names, by replacing clauses (iv) and (vi) by

(iv') $Q \equiv \nu W Y_s' Y_r'.(S' \parallel R'[b/x] \parallel C)$ *(vi')* $Q \equiv \nu W Y'.(P' \parallel C)$

respectively, such that $Y_s \restriction \mathcal{N} = Y_s' \restriction \mathcal{N}$, $Y_r \restriction \mathcal{N} = Y_r' \restriction \mathcal{N}$, and $Y \restriction \mathcal{N} = Y \restriction \mathcal{N}$, where $X \restriction \mathcal{N} := \{x \in \mathcal{N} \mid \exists \tau.(x:\tau) \in X\}$. I.e. the type annotation of the names that are activated by the transition (i.e. those from Y_s, Y_r and Y) are not required to be preserved in Q. (By contrast, the annotation of every active restriction in P is preserved by the transition.) While in this context inactive annotations can be ignored by the transitions, they will be used by the type system in Sect. 4, to establish invariance of $\mathcal{T}$-compatible.

Now we are ready to explain what it means for an annotated term P to be $\mathcal{T}$-compatible: there is a forest in $\mathcal{F}[\![P]\!]$ such that every trace of it projects to a chain in the partial order $\mathcal{T}$.

Definition 6 ($\mathcal{T}$-compatibility). *Let* $P \in \mathcal{P}^{\mathbb{T}}$ *be an annotated* π*-term. A forest* $\varphi \in \mathcal{F}[\![P]\!]$ *is* $\mathcal{T}$-compatible *if for every trace* $((x_1,t_1)\ldots(x_k,t_k)\ A)$ *in* φ *it holds that* $t_1 < t_2 < \ldots < t_k$*. The* π*-term* P *is* $\mathcal{T}$-compatible *if* $\mathcal{F}[\![P]\!]$ *contains a* $\mathcal{T}$*-compatible forest. A term is* $\mathcal{T}$-shaped *if each of its subterms is* $\mathcal{T}$*-compatible.*

As a property of annotated terms, $\mathcal{T}$-compatibility is by definition invariant under structural congruence.

A term $P' \in \mathcal{P}^{\mathbb{T}}$ is a *type annotation* (or simply *annotation*) of $P \in \mathcal{P}$ if its *type-erasure*, written $\ulcorner P' \urcorner$, coincides with P. (We omit the obvious definition of type-erasure.) A *consistent annotation* of a transition of terms, $P \to Q$, is a choice function that, given an annotation P' of P, returns an annotation Q' of Q such that $P' \to Q'$. Note that it follows from the definition that the annotation of every active restriction in P' is preserved in Q'. The effect of the choice function is therefore to pick a possibly new annotation for each restriction in Q' that is activated by the transition. Thus, given a semantics $(\mathcal{P}, \to, P)$ of a term P, and an annotation P' of P, and a consistent annotation for every transition of the semantics, there is a well-defined pointed transition system $(\mathcal{P}^{\mathbb{T}}, \to', P')$ such that every transition sequence of the former lifts to a transition sequence of the latter. We call $(\mathcal{P}^{\mathbb{T}}, \to', P')$ a *consistent annotation* of the semantics $(\mathcal{P}, \to, P)$.

Definition 7 (Hierarchical term). *A term $P \in \mathcal{P}$ is* hierarchical *if there exist a finite forest $\mathcal{T} = \mathbb{T}$ and a consistent annotation $(\mathcal{P}^{\mathbb{T}}, \to', P')$ of the semantics $(\mathcal{P}, \to, P)$ of P, such that all terms reachable from P' are $\mathcal{T}$-compatible.*

Example 7. The term in Examples 2 and 5 is hierarchical: take the hierarchy $\mathcal{T} = \mathsf{s} \prec \mathsf{c} \prec \mathsf{m} \prec \mathsf{d}$ and annotate each name in Q_{ijk} as follows: $s : \mathsf{s}$, $c : \mathsf{c}$, $m : \mathsf{m}$ and $d : \mathsf{d}$. The annotation is consistent, and forest(Q_{ijk}) is $\mathcal{T}$-compatible for all i,j and k.

Example 4 gives an example of a term that is not hierarchical. The forest representation of the reachable terms shown in Example 6 does not have a bounded height, which means that if $\mathcal{T}$ has n base types, there is a reachable term with a representation of height bigger than n, which implies that there will be a path repeating a base type.

Let us now study this fragment. First it is easy to see that invariance of $\mathcal{T}$-compatibility under reduction $\to$, for some finite $\mathcal{T}$, puts a bound $|\mathcal{T}|$ on the height of the $\mathcal{T}$-compatible reachable forests, and consequently a bound on depth.

Theorem 3. *Every hierarchical term is depth-bounded. The converse is false.*

Thanks to Theorem 2, an immediate corollary of Theorem 3 is that coverability and termination are decidable for hierarchical terms.

Unfortunately, like the depth-bounded fragment, membership of the hierarchical fragment is undecidable. The proof is by adapting the argument for the undecidability of depth boundedness [15].

Lemma 1. *Every terminating π-term is hierarchical.*

Proof. Since the transition system of a term, quotiented by structural congruence, is finitely branching, by König's lemma the computation tree of a terminating term is finite, so it contains finitely many reachable processes and therefore finitely many names. Take the set of all (disambiguated) active names of the reachable terms and fix an arbitrary total order $\mathcal{T}$ on them. The consistent annotation with $(x : x)$ for each name will prove the term hierarchical.

Theorem 4. *Determining whether an arbitrary π-term is hierarchical, is undecidable.*

Proof. The π-calculus is Turing-complete, so termination is undecidable. Suppose we had an algorithm to decide if a term is hierarchical. Then we could decide termination of an arbitrary π-term by first checking if the term is hierarchical; if the answer is yes, we can decide termination for it by Theorem 1, otherwise we know that it is not terminating by Lemma 1.

Theorem 4—and the corresponding version for depth-bounded terms—is a serious impediment to any practical application of hierarchical terms to verification: when presented with a term to verify, one has to prove that it belongs to one of the two fragments, manually, before one can apply the relevant algorithms.

While the two fragments have a lot in common, hierarchical systems have a richer structure, which we will exploit to define a type system that can prove a term hierarchical, in a feasible, sound but incomplete way. Thanks to the notion of hierarchy, we are thus able to statically capture an expressive fragment of the π-calculus that enjoys decidable coverability.

4 A Type System for Hierarchical Topologies

The purpose of this section is to devise a static check to determine if a term is hierarchical. To do so, we define a type system, parametrised over a forest $\mathcal{T}$, which satisfies subject reduction. Furthermore we prove that if a term is typable then $\mathcal{T}$-shapedness is preserved by reduction of the term. Typability together with $\mathcal{T}$-shapedness of the initial term would then prove the term hierarchical.

As we have seen in the introduction, the typing rules make use of a new perspective on π-calculus reactions. Take the term $\nu\, a.(\nu\, b.\overline{a}\langle b\rangle.S \parallel \nu\, c.a(x).R)$ where $\nu\, a.(\nu\, b.[-] \parallel \nu\, c.[-])$ is the *reaction context*. Standardly the synchronisation of the two sequential processes over a is preceded by an extrusion of the scope of b to include $\nu\, c.a(x).R$, followed by the actual reaction:

$$\begin{aligned}\nu\, a.\big(\nu\, b.(\overline{a}\langle b\rangle.S) \parallel \nu\, c.a(x).R\big) &\equiv \nu\, a.\,\nu\, b.\big(\overline{a}\langle b\rangle.S \parallel \nu\, c.a(x).R\big)\\ &\to \nu\, a.\,\nu\, b.\big(S \parallel \nu\, c.(R[b/x])\big)\end{aligned}$$

This dynamic reshuffling of scopes is problematic for establishing invariance of $\mathcal{T}$-compatibility under reduction: notice how $\nu\, c$ is brought into the scope of $\nu\, b$, possibly disrupting $\mathcal{T}$-compatibility. (For example, the preceding reduction would break $\mathcal{T}$-compatibility of the forest representations if the tree $\mathcal{T}$ is either $a \prec c \prec b$ or $b \succ a \prec c$.) We therefore adopt a different view. After the message is transmitted, the sender continues in-place as S, while R is split into two parts $R_{\text{mig}} \parallel R_{\neg\text{mig}}$, one that uses the message (the *migratable* one) and one that does not. The migratable portion R_{mig} is "installed" under $\nu\, b$ so that it can make use of the acquired name, while the non-migratable one can simply continue in-place:

$$\nu\, a.\big(\nu\, b.(\overline{a}\langle b\rangle.S) \parallel \nu\, c.a(x).R\big) \to \nu\, a.\big(\nu\, b.(S \parallel R_{\text{mig}}[b/x]) \parallel \nu\, c.R_{\neg\text{mig}}\big)$$

Crucially, the *reaction context*, $\nu\, a.(\nu\, b.[-] \parallel \nu\, c.[-])$, is unchanged. This means that if the starting term is $\mathcal{T}$-compatible, the context of the *reactum* is $\mathcal{T}$-compatible as well. Naturally, this only makes sense if R_{mig} does not use c. Thus our typing rules impose constraints on the use of names of R so that the migration does not result in R_{mig} escaping the scope of bound names such as c.

The formal definition of "migratable" is subtle. Consider the term

$$\nu\, f.a(x).\nu\, c\, d\, e.\big(\overline{x}\langle c\rangle \parallel \overline{c}\langle d\rangle \parallel \overline{a}\langle e\rangle.\overline{e}\langle f\rangle\big)$$

Upon synchronisation with $\nu\, b.\overline{a}\langle b\rangle$, surely $\overline{x}\langle c\rangle$ will need to be put under the scope of $\nu\, b$ after substituting b for x, hence the first component of the continuation, $\overline{x}\langle c\rangle$, is migratable. However this implies that the scope of $\nu\, c$ will need to be placed under $\nu\, b$, which in turn implies that $\overline{c}\langle d\rangle$ needs to be considered migratable as well. On the other hand, $\nu\, e.\overline{a}\langle e\rangle.\overline{e}\langle f\rangle$ must be placed in the scope of f, which may not be known by the sender, so it is not considered migratable. The following definition makes these observations precise.

Definition 8 (Linked to, tied to, migratable). *Given a normal form* $P = \nu\, X.\prod_{i\in I} A_i$ *we say that* A_i *is* linked to A_j in P, *written* $i \leftrightarrow_P j$, *if* $\mathrm{fn}(A_i) \cap \mathrm{fn}(A_j) \cap X \neq \emptyset$. *We define the* tied-to *relation as the transitive closure of* $\leftrightarrow_P$. *I.e.* A_i *is* tied to A_j, *written* $i \frown_P j$, *if* $\exists k_1, \ldots, k_n \in I.\, i \leftrightarrow_P k_1 \leftrightarrow_P k_2 \ldots \leftrightarrow_P k_n \leftrightarrow_P j$, *for some* $n \geq 0$. *Furthermore, we say that a name* y *is* tied to A_i in P, *written* $y \triangleleft_P i$, *if* $\exists j \in I.\, y \in \mathrm{fn}(A_j) \wedge j \frown_P i$. *Given an input-prefixed normal form* $a(y).P$ *where* $P = \nu\, X.\prod_{i\in I} A_i$, *we say that* A_i is migratable in $a(y).P$, *written* $\mathrm{Mig}_{a(y).P}(i)$, *if* $y \triangleleft_P i$.

These definitions have an intuitive meaning with respect to the communication topology of a normal form P: two sequential subterms are linked if they are connected by an hyperedge in the communication topology of P, and are tied to each other if there exists a path between them.

The following lemma indicates how the tied-to relation fundamentally constrains the possible shape of the forest of a term.

Lemma 2. *Let* $P = \nu\, X.\prod_{i\in I} A_i \in \mathcal{P}_{\mathsf{nf}}$, *if* $i \frown_P j$ *then if a forest* $\varphi \in \mathcal{F}[\![P]\!]$ *has leaves labelled with* A_i *and* A_j *respectively, they belong to the same tree in* φ *(i.e., have a common ancestor in* φ*).*

Example 8. Take the normal form $P = \nu\, a\; b\; c.(A_1 \parallel A_2 \parallel A_3 \parallel A_4)$ where $A_1 = a(x)$, $A_2 = b(x)$, $A_3 = c(x)$ and $A_4 = \overline{a}\langle b\rangle$. We have $1 \leftrightarrow_P 4$, $2 \leftrightarrow_P 4$, therefore $1 \frown_P 2 \frown_P 4$ and $a \triangleleft_P 2$. In Fig. 4 we show some of the forests in $\mathcal{F}[\![P]\!]$. Forest 1 represents forest(P). The fact that A_1, A_2 and A_4 are tied is reflected by the fact that none of the forests place them in disjoint trees. Now suppose we select only the forests in $\mathcal{F}[\![P]\!]$ that respect the hierarchy $a \prec b$: in all the forests in this set, the nodes labelled with A_1, A_2 and A_4 have a as common ancestor (as in forests 1, 2, 3 and 4). In particular, in these forests A_2 is necessarily a descendent of a even if a is not one of its free names.

In Sect. 3 we introduced annotations in a rather abstract way by means of a generic domain of types $\mathbb{T}$. In Definition 7 we ask for the existence of an annotation for the semantics of a term. Specifically, one can decide an arbitrary annotation for each active name. A type system however will examine the term statically, which means that it needs to know what could be a possible annotation for a variable, i.e., the name bound in an input action. This information is directly related to the notion of data-flow, that is the set of names that are bound to a variable during runtime. Since a static method cannot capture this information precisely, we make use of *sorts* [18], also known as *simple types*, to approximate it. The annotation of a restriction will carry not only which base type should be associated with its instances, but also instructions on how to annotate the messages received or sent through those instances. Concretely, we define

$$\mathbb{T} \ni \tau ::= t \mid t[\tau]$$

where $t \in \mathcal{T}$ is a base type.

A name with type t cannot be used as a channel but can be used as a message; a name with type $t[\tau]$ can be used to transmit a name of type τ. We will write $\text{base}(\tau)$ for t when $\tau = t[\tau']$ or $\tau = t$. By abuse of notation we write, for a set of types X, $\text{base}(X)$ for the set of base types of the types in X.

As is standard, we keep track of the types of free names by means of a typing environment. An environment Γ is a partial map from names to types, which we will write as a set of *type assignments*, $x : \tau$. Given a set of names X and an environment Γ, we write $\Gamma(X)$ for the set $\{\Gamma(x) \mid x \in X \cap \text{dom}(\Gamma)\}$. Given two environments Γ and Γ' with $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$, we write $\Gamma\Gamma'$ for their union. For a type environment Γ we define

$$\min_{\mathcal{T}}(\Gamma) := \{(x : \tau) \in \Gamma | \forall (y : \tau') \in \Gamma.\, \text{base}(\tau') \not< \text{base}(\tau)\}.$$

A judgement $\Gamma \vdash_{\mathcal{T}} P$ means that $P \in \mathcal{P}^{\mathbb{T}}_{\text{nf}}$ can be typed under assumptions Γ, over the hierarchy $\mathcal{T}$; we say that P is *typable* if $\Gamma \vdash_{\mathcal{T}} P$ is provable for some Γ and $\mathcal{T}$. An arbitrary term $P \in \mathcal{P}^{\mathbb{T}}$ is said to be *typable* if its normal form is. The typing rules are presented in Fig. 5.

The type system presents several non-standard features. First, it is defined on normal forms as opposed to general π-terms. This choice is motivated by the fact that different syntactic presentations of the same term may be misleading when trying to analyse the relation between the structure of the term and $\mathcal{T}$. The rules need to guarantee that a reduction will not break $\mathcal{T}$-compatibility, which is a property of the congruence class of the term. As justified by Lemma 2, the scope of names in a congruence class may vary, but the tied-to relation puts constraints on the structure that must be obeyed by all members of the class. Therefore the type system is designed around this basic concept, rather than the specific scoping of any representative of the structural congruence class. Second, no type information is associated with the typed term, only restricted names hold type annotations. Third, while the rules are compositional, the constraints

$$\frac{\begin{array}{c}\forall i \in I.\, \Gamma, X \vdash_{\mathcal{T}} A_i \\ \forall i \in I.\, \forall x : \tau_x \in X.\, x \lhd_P i \implies \mathrm{base}(\Gamma(\mathrm{fn}(A_i))) < \mathrm{base}(\tau_x)\end{array}}{\Gamma \vdash_{\mathcal{T}} \nu X.\textstyle\prod_{i\in I} A_i}\ \textsc{Par}$$

$$\frac{\forall i \in I.\, \Gamma \vdash_{\mathcal{T}} \pi_i.P_i}{\Gamma \vdash_{\mathcal{T}} \sum_{i\in I} \pi_i.P_i}\ \textsc{Choice} \qquad \frac{\Gamma \vdash_{\mathcal{T}} A}{\Gamma \vdash_{\mathcal{T}} !A}\ \textsc{Repl} \qquad \frac{\Gamma \vdash_{\mathcal{T}} P}{\Gamma \vdash_{\mathcal{T}} \boldsymbol{\tau}.P}\ \textsc{Tau}$$

$$\frac{a : t_a[\tau_b] \in \Gamma \qquad b : \tau_b \in \Gamma \qquad \Gamma \vdash_{\mathcal{T}} Q}{\Gamma \vdash_{\mathcal{T}} \overline{a}\langle b\rangle.Q}\ \textsc{Out}$$

$$\frac{\begin{array}{c}a : t_a[\tau_x] \in \Gamma \qquad \Gamma, x : \tau_x \vdash_{\mathcal{T}} \nu X.\textstyle\prod_{i\in I} A_i \\ \mathrm{base}(\tau_x) < t_a \vee \big(\forall i \in I.\ \mathrm{Mig}_{a(x).P}(i) \implies \mathrm{base}(\Gamma(\mathrm{fn}(A_i) \setminus \{a\})) < t_a\big)\end{array}}{\Gamma \vdash_{\mathcal{T}} a(x).\nu X.\textstyle\prod_{i\in I} A_i}\ \textsc{In}$$

Fig. 5. A type system for hierarchical terms. The term P stands for $\nu X.\prod_{i\in I} A_i$.

on base types have a global flavour due to the fact that they involve the structure of $\mathcal{T}$ which is a global parameter of typing proofs.

Let us illustrate intuitively how the constraints enforced by the rules guarantee preservation of $\mathcal{T}$-compatibility. Consider the term

$$P = \nu\, e\, a.\Big(\nu\, b.\big(\overline{a}\langle b\rangle.A_0\big) \parallel \nu\, d.\big(a(x).Q\big)\Big)$$

with $Q = \nu\, c.(A_1 \parallel A_2 \parallel A_3)$, $A_0 = b(y)$, $A_1 = \overline{x}\langle c\rangle$, $A_2 = c(z).\overline{a}\langle e\rangle$ and $A_3 = \overline{a}\langle d\rangle$. Let $\mathcal{T}$ be the forest with $t_e \prec t_a \prec t_b \prec t_c$ and $t_a \prec t_d$, where t_x is the base type of the (omitted) annotation of the restriction $\nu\, x$, for $x \in \{a, b, c, d, e\}$. The reader can check that forest(P) is $\mathcal{T}$-compatible.

In the traditional understanding of mobility, we would interpret the communication of b over a as an application of scope extrusion to include $\nu\, d.\big(a(x).Q\big)$ in the scope of b and then syncronisation over a with the application of the substitution $[b/x]$ to Q; note that the substitution is only valid because the scope of b has been extended to include the receiver.

Our key observation is that we can instead interpret this communication as a migration of the subcomponents of Q that do get their scopes changed by the reduction, from the scope of the receiver to the scope of the sender. For this operation to be sound, the subcomponents of Q migrating to the sender's scope cannot use the names that are in the scope of the receiver but not of the sender.

In our specific example, after the synchronisation between the prefixes $\overline{a}\langle b\rangle$ and $a(x)$, b is substituted to x in A_1 resulting in the term $A_1' = \overline{b}\langle c\rangle$ and A_0, A_1', A_2 and A_3 become active. The scope of A_0 can remain unchanged as it cannot know more names than before as a result of the communication. By contrast, A_1 now knows b as a result of the substitution $[b/x]$: A_1 needs to migrate under the scope of b. Since A_1 uses c as well, the scope of c needs to be moved under b; however A_2 uses c so it needs to migrate under b with the scope

of c. A_3 instead does not use neither b nor c so it can avoid migration and its scope remains unaltered.

This information can be formalised using the tied-to relation: on one hand, A_1 and A_2 need to be moved together because $1 \frown_Q 2$ and they need to be moved because $x \lhd_Q 1, 2$. On the other hand, A_3 is not tied to neither A_1 nor A_2 in Q and does not know x, thus it is not migratable. After reduction, our view of the reactum is the term

$$\nu\, a.\Big(\nu\, b.\big(A_0 \parallel \nu\, c.(A_1' \parallel A_2)\big) \parallel \nu\, d.A_3\Big)$$

the forest of which is $\mathcal{T}$-compatible. Rule Par, applied to A_1 and A_2, ensures that c has a base type that can be nested under the one of b. Rule In does not impose constraints on the base types of A_3 because A_3 is not migratable. It does however check that the base type of e is an ancestor of the one of a, thus ensuring that both receiver and sender are already in the scope of e. The base type of a does not need to be further constrained since the fact that the synchronisation happened on it implies that both the receiver and the sender were already under its scope; this implies, by $\mathcal{T}$-compatibility of P, that c can be nested under a.

We now describe the purpose of the rules of the type system in more detail. Most of the rules just drive the derivation through the structure of the term. The crucial constraints are checked by Par, In and Out.

The Out *Rule.* The main purpose of rule Out is enforcing types to be consistent with the dataflow of the process: the type of the argument of a channel a must agree with the types of all the names that may be sent over a. This is a very coarse sound over-approximation of the dataflow; if necessary it could be refined using well-known techniques from the literature but a simple approach is sufficient here to type interesting processes.

The Par *Rule.* Rule Par is best understood imagining the normal form to be typed, P, as the continuation of a prefix $\pi.P$. In this context a reduction exposes each of the active sequential subterms of P which need to have a place in a $\mathcal{T}$-compatible forest for the reactum. The constraint in Par can be read as follows. A "new" leaf A_i may refer to names already present in the forests of the reaction context; these names are the ones mentioned in both $\mathrm{fn}(A_i)$ and Γ. Then we must be able to insert A_i so that we can find these names in its path. However, A_i must belong to a tree containing all the names in X that are tied to it in P. So by requiring every name tied to A_i to have a base type greater than any name in the context that A_i may refer to, we make sure that we can insert the continuation in the forest of the context without violating $\mathcal{T}$-compatibility. Note that $\Gamma(\mathrm{fn}(A_i))$ contains only types that annotate names both in Γ and $\mathrm{fn}(A_i)$, that is, names which are not restricted by X and are referenced by A_i (and therefore come from the context).

The In *Rule.* Rule In serves two purposes: on the one hand it requires the type of the messages that can be sent through a to be consistent with the use of the variable x which will be bound to the messages; on the other hand, it constrains

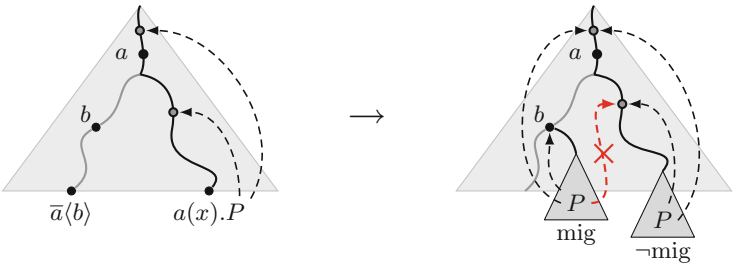


Fig. 6. Explanation of constraints imposed by rule In. The dashed lines represent references to names restricted in the reduction context.

the base types of a and x so that synchronisation can be performed without breaking $\mathcal{T}$-compatibility.

The second purpose is achieved by distinguishing two cases, represented by the two disjuncts of the condition on base types of the rule. In the first case, the base type of the message is an ancestor of the base type of a in $\mathcal{T}$. This implies that in any $\mathcal{T}$-compatible forest representing $a(x).P$, the name b sent as message over a is already in the scope of P. Under this circumstance, there is no real mobility, P does not know new names by the effect of the substitution $[b/x]$, and the $\mathcal{T}$-compatibility constraints to be satisfied are in essence unaltered.

The second case is more complicated as it involves genuine mobility. This case also requires a slightly non-standard feature: not only do the premises predicate on the direct subcomponents of an input prefixed term, but also on the direct subcomponents of the continuation. This is needed to be able to separate the continuation in two parts: the one requiring migration and the one that does not. The situation during execution is depicted in Fig. 6. The non migratable sequential terms behave exactly as the case of the first disjunct: their scope is unaltered. The migratable ones instead are intended to be inserted as descendents of the node representing the message b in the forest of the reaction context.

For this to be valid without rearrangement of the forest of the context, we need all the names in the context that are referenced in the migratable terms, to be also in the scope at b; we make sure this is the case by requiring the free names of any migratable A_i that are from the context (i.e. in Γ) to have base types smaller than the base type of a. The set $\mathrm{base}(\Gamma(\mathrm{fn}(A_i) \setminus \{a\}))$ indeed represents the base types of the names in the reaction context referenced in a migratable continuation A_i. In fact a is a name that needs to be in the scope of both the sender and the receiver at the same time, so it needs to be a common ancestor of sender and receiver in any $\mathcal{T}$-compatible forest. Any name in the reaction context and in the continuation of the receiver, with a base type smaller than the one of a, will be an ancestor of a—and hence of the sender, the receiver and the node representing the message—in any $\mathcal{T}$-compatible forest. Clearly, remembering a is not harmful as it must be already in the scope of receiver and sender, so we exclude it from the constraint.

Example 9. Take the normal form in Example 2. Let us fix $\mathcal{T}$ to be the forest $\mathsf{s} \prec \mathsf{c} \prec \mathsf{m} \prec \mathsf{d}$ and annotate the normal form with the following types: $s : \tau_s = \mathsf{s}[\tau_m]$, $c : \tau_c = \mathsf{c}[\tau_m]$, $m : \tau_m = \mathsf{m}[\mathsf{d}]$ and $d : \mathsf{d}$. We want to prove $\emptyset \vdash_{\mathcal{T}} \nu\, s\, c.P$. We can apply rule Par: in this case there are no conditions on types because, being the environment empty, we have $\text{base}(\emptyset(\text{fn}(A))) = \emptyset$ for every active sequential term A of P. Let $\Gamma = \{(s : \tau_s), (c : \tau_c)\}$. The rule requires $\Gamma \vdash_{\mathcal{T}} !S$, $\Gamma \vdash_{\mathcal{T}} !C$ and $\Gamma \vdash_{\mathcal{T}} !M$, which can be proved by proving typability of S, C and M under Γ by rule Repl.

To prove $\Gamma \vdash_{\mathcal{T}} S$ we apply rule In; we have $s : \mathsf{s}[\tau_m] \in \Gamma$ and we need to prove that $\Gamma, x : \tau_m \vdash_{\mathcal{T}} \nu\, d.\overline{x}\langle d\rangle$. No constraints on base types are generated at this step since the migratable sequential term $\nu\, d.\overline{x}\langle d\rangle$ does not contain free variables typed by Γ making $\Gamma(\text{fn}(\nu\, d.\overline{x}\langle d\rangle) \setminus \{a\}) = \Gamma(\{x\})$ empty. Next, $\Gamma, x : \tau_m \vdash_{\mathcal{T}} \nu\, d.\overline{x}\langle d\rangle$ can be proved by applying rule Par which amounts to checking $\Gamma, x : \tau_m, d : \mathsf{d} \vdash_{\mathcal{T}} \overline{x}\langle d\rangle.\mathbf{0}$ (by a simple application of Out and the axiom $\Gamma, x : \tau_m, d : \mathsf{d} \vdash_{\mathcal{T}} \mathbf{0}$) and verifying the condition—true in $\mathcal{T}$—$\text{base}(\tau_m) < \text{base}(\tau_d)$: in fact d is tied to $\overline{x}\langle d\rangle$ and, for $\Gamma' = \Gamma \cup \{x : \tau_m\}$, $\text{base}(\Gamma'(\text{fn}(\overline{x}\langle d\rangle))) = \text{base}(\Gamma'(\{x, d\})) = \text{base}(\{\tau_m\})$. The proof for $\Gamma \vdash_{\mathcal{T}} M$ is similar and requires $\mathsf{c} < \mathsf{m}$ which is true in $\mathcal{T}$.

Finally, we can prove $\Gamma \vdash_{\mathcal{T}} C$ using rule In; both the two continuations $A_1 = \overline{s}\langle m\rangle$ and $A_2 = m(y).\overline{c}\langle m\rangle$ are migratable in C and since $\text{base}(\tau_m) < \text{base}(\tau_c)$ is false we need the other disjunct of the condition to be true. This amounts to checking that $\text{base}(\Gamma(\text{fn}(A_1) \setminus \{c\})) = \text{base}(\Gamma(\{s, m\})) = \text{base}(\{\tau_s\}) = \mathsf{s} < \mathsf{c}$ (note $m \notin \text{dom}(\Gamma)$) and $\text{base}(\Gamma(\text{fn}(A_a) \setminus \{c\})) = \text{base}(\Gamma(\emptyset)) < \mathsf{c}$ (that holds trivially).

To complete the typing we need to show $\Gamma, m : \tau_m \vdash_{\mathcal{T}} A_1$ and $\Gamma, m : \tau_m \vdash_{\mathcal{T}} A_2$. The former can be proved by a simple application of Out which does not impose further constraints on $\mathcal{T}$. The latter is proved by applying In which requires $\text{base}(\tau_c) < \mathsf{m}$, which holds in $\mathcal{T}$.

Note how, at every step, there is only one rule that applies to each subproof.

Example 10. The term Example 4 is not typable under any $\mathcal{T}$. To see why, one can build the proof tree without assumptions on $\mathcal{T}$ by assuming that each restriction $\nu\, x$ has base type t_x. When typing $\overline{m}\langle s\rangle$ we deduce that $t_s = t_n$, which is in contradiction with the constraint that $t_n < t_s$ required by rule Par when typing $\nu\, s.(S \parallel \overline{m}\langle s\rangle \parallel \overline{s})$.

5 Soundness of the Type System

We now establish the soundness of the type system. Theorem 5 will show how typability is preserved by reduction. Theorem 6 establishes the main property of the type system: if a term is typable then $\mathcal{T}$-shapedness is invariant under reduction. This allows us to conclude that if a term is $\mathcal{T}$-shaped and typable, then every term reachable from it will be $\mathcal{T}$-shaped.

The subtitution lemma states that substituting names without altering the types preserves typability.

Lemma 3 (Substitution). *Let $P \in \mathcal{P}^{\mathbb{T}}_{\mathsf{nf}}$ and Γ be a typing environment such that $\Gamma(a) = \Gamma(b)$. Then it holds that if $\Gamma \vdash_{\mathcal{T}} P$ then $\Gamma \vdash_{\mathcal{T}} P[b/a]$.*

Before we state the main theorem, we define the notion of P-safe type environment, which is a simple restriction on the types that can be assigned to names that are free at the top-level of a term.

Definition 9 (P-safe environment). *A type environment Γ is said to be P-*safe *if for each $x \in \mathrm{fn}(P)$ and $(y : \tau) \in \mathrm{bn}_{\nu}(P)$, $\mathrm{base}(\Gamma(x)) < \mathrm{base}(\tau)$.*

Theorem 5 (Subject Reduction). *Let P and Q be two terms in $\mathcal{P}^{\mathbb{T}}_{\mathsf{nf}}$ and Γ be a P-safe type environment. If $\Gamma \vdash_{\mathcal{T}} P$ and $P \to Q$, then $\Gamma \vdash_{\mathcal{T}} Q$.*

The proof is by careful analysis of how the typing proof for P can be adapted to derive a proof for Q. The only difficulty comes from the fact that some of the subterms of P will appear in Q with a substitution applied. However, typability of P ensures that we are only substituting names for names with the same type, thus allowing us to apply Lemma 3.

To establish that $\mathcal{T}$-shapedness is invariant under reduction for typable terms, we will need to show that starting from a typable $\mathcal{T}$-shaped term P, any step will reduce it to a (typable) $\mathcal{T}$-shaped term. The hypothesis of $\mathcal{T}$-compatibility of P can be used to extract a $\mathcal{T}$-compatible forest φ from $\mathcal{F}[\![P]\!]$. While many forests in $\mathcal{F}[\![P]\!]$ can be witnesses of the $\mathcal{T}$-compatibility of P, we want to characterise the shape of a witness that *must* exist if P is $\mathcal{T}$-compatible. The proof of invariance relies on selecting a φ that does not impose unnecessary hierarchical dependencies among names. Such forest is identified by $\Phi_{\mathcal{T}}(\mathrm{nf}(P))$: it is the shallowest among all the $\mathcal{T}$-compatible forests in $\mathcal{F}[\![P]\!]$.

Definition 10 ($\Phi_{\mathcal{T}}$). *The function $\Phi_{\mathcal{T}} \colon \mathcal{P}^{\mathbb{T}}_{\mathsf{nf}} \to \mathcal{F}_{\mathcal{T}}$ is defined inductively as*

$$\Phi_{\mathcal{T}}\Big(\prod_{i\in I} A_i\Big) := \biguplus_{i\in I} \{A_i[]\}$$

$$\Phi_{\mathcal{T}}(P) := \Big(\biguplus \Big\{(x, \mathrm{base}(\tau))[\Phi_{\mathcal{T}}(\nu\, Y_x.\textstyle\prod_{j\in I_x} A_j)] \mid (x:\tau) \in \min_{\mathcal{T}}(X)\Big\}\Big) \uplus \Phi_{\mathcal{T}}(\nu\, Z.\textstyle\prod_{r\in R} A_r)$$

where $X \neq \emptyset$, $P = \nu\, X.\prod_{i\in I} A_i$ $I_x = \{i \in I \mid x \lhd_P i\}$ and

$$Y_x = \{(y:\tau) \in X \mid \exists i \in I_x.\, y \in \mathrm{fn}(A_i)\} \setminus \min_{\mathcal{T}}(X)$$
$$Z = X \setminus \big(\textstyle\bigcup_{(x:\tau)\in \min_{\mathcal{T}}(X)} Y_x \cup \{x:\tau\}\big)$$
$$R = I \setminus \big(\textstyle\bigcup_{(x:\tau)\in \min_{\mathcal{T}}(X)} I_x\big)$$

Forest 4 of Fig. 4 is $\Phi_{\mathcal{T}}(P)$ when every restriction $\nu\, x$ has base type x (for $x \in \{a, b, c\}$) and $\mathcal{T}$ is the forest with nodes a, b and c and a single edge $a \prec b$.

Lemma 4. *Let $P \in \mathcal{P}_{\mathsf{nf}}^{\mathbb{T}}$. Then:*

(a) $\Phi_{\mathcal{T}}(P)$ is a $\mathcal{T}$-compatible forest;
(b) $\Phi_{\mathcal{T}}(P) \in \mathcal{F}[\![P]\!]$ if and only if P is $\mathcal{T}$-compatible;
(c) if $P \equiv Q \in \mathcal{P}^{\mathbb{T}}$ then $\Phi_{\mathcal{T}}(P) \in \mathcal{F}[\![Q]\!]$ if and only if Q is $\mathcal{T}$-compatible.

Theorem 6 (Invariance of $\mathcal{T}$-shapedness). *Let P and Q be terms in $\mathcal{P}_{\mathsf{nf}}^{\mathbb{T}}$ such that $P \to Q$ and Γ be a P-safe environment such that $\Gamma \vdash_{\mathcal{T}} P$. Then, if P is $\mathcal{T}$-shaped then Q is $\mathcal{T}$-shaped.*

The key of the proof is (a) the use of $\Phi_{\mathcal{T}}(P)$ to extract a specific $\mathcal{T}$-compatible forest, (b) the definition of a way to insert the subtrees of the continuations of the reacting processes in the forest of reaction context, in a way that preserves $\mathcal{T}$-compatibility. Thanks to the constraints of the typing rules, we will always be able to find a valid place in the reaction context where to attach the trees representing the reactum.

6 Type Inference

In this section we will show that it is possible to take any non-annotated normal form P and derive a forest $\mathcal{T}$ and an annotated version of P that can be typed under $\mathcal{T}$.

Inference for simple types has already been proved decidable in [9,24]. In our case, since our types are not recursive, the algorithm concerned purely with the constraints imposed by the type system of the form $\tau_x = t[\tau_y]$ is even simpler. The main difficulty is inferring the structure of $\mathcal{T}$.

Let us first be more specific on assigning simple types. The number of ways a term P can be annotated with types are infinite, simply from the fact that types allow an arbitrary nesting as in t, $t[t]$, $t[t[t]]$ and so on. We observe that, however, there is no use annotating a restriction with a type with nesting deeper than the size of the program: the type system cannot inspect more deeply nested types. Thanks to this observation we can restrict ourselves to annotations with bounded nesting in the type's structure. This also gives a bound on the number of base types that need to appear in the annotated term. Therefore, there are only finitely many possible annotations and possible forests under which P can be proved typably hierarchical. A naïve inference algorithm can then enumerate all of them and type check each.

Theorem 7 (Decidability of inference). *Given a normal form $P \in \mathcal{P}_{\mathsf{nf}}$, it is decidable if there exists a finite forest $\mathcal{T}$, a $\mathcal{T}$-annotated version $P' \in \mathcal{P}^{\mathbb{T}}$ of P and a P'-safe environment Γ such that P' is $\mathcal{T}$-shaped and $\Gamma \vdash_{\mathcal{T}} P'$.*

While enumerating all the relevant forests, annotations and environments is impractical, more clever strategies for inference exist.

We start by annotating the term with type variables: each name x gets typed with a type variable t_x. Then we start the type derivation, collecting all the constraints on types along the way. If we can find a $\mathcal{T}$ and type expressions to

associate to each type variable, so that these constraints are satisfied, the process can be typed under $\mathcal{T}$.

By inspecting rules Par and In we observe that all the "tied-to" and "migratable" predicates do not depend on $\mathcal{T}$ so for any given P, the type constraints can be expressed simply by conjunctions and disjunctions of two kinds of basic predicates:

1. *data-flow constraints* of the form $\mathfrak{t}_x = t_x[\mathfrak{t}_y]$ where t_x is a base type variable;
2. *base type constraints* of the form $\text{base}(\mathfrak{t}_x) < \text{base}(\mathfrak{t}_y)$ which correspond to constraints over the corresponding base type variables, e.g. $t_x < t_y$.

Note that the P-safety condition on Γ translates to constraints of the second kind. The first kind of constraint can be solved using unification in linear time. If no solution exists, the process cannot be typed. This is the case of processes that cannot be *simply typed*. If unification is successful we get a set of equations over base type variables. Any assignment of those variables to nodes in a suitable forest that satisfies the constraints of the second kind would be a witness of typability. An example of the type inference in action can be found in [5].

First we note that if there exists a $\mathcal{T}$ which makes P typable and $\mathcal{T}$-compatible, then there exists a $\mathcal{T}'$ which does the same but is a linear chain of base types (i.e. a single tree with no branching). To see how, simply take $\mathcal{T}'$ to be any topological sort of $\mathcal{T}$.

Now, suppose we are presented with a set $\mathcal{C}$ of constraints of the form $t < t'$ (no disjuctions). One approach for solving them could be based on reductions to SAT or CLP(FD). We instead outline a direct algorithm. If the constraints are acyclic, i.e. it is not possible to derive $t < t$ by transitivity, then there exists a finite forest satisfying the constraints, having as nodes the base type variables. To construct such forest, we can first represent the constraints as a graph with the base type variables as vertices and an edge between t and t' just when $t < t' \in \mathcal{C}$. Then we can check the graph for acyclicity. If the test fails, the constraints are unsatisfiable. Otherwise, any topological sort of the graph will represent a forest satisfying $\mathcal{C}$.

We can modify this simple procedure to support constraints including disjuctions by using backtracking on the disjuncts. Every time we arrive at an acyclic assignment, we can check for $\mathcal{T}$-shapedness (which takes linear time) and in case the check fails we can backtrack again.

To speed up the backtracking algorithm, one can merge the acyclicity test with the $\mathcal{T}$-compatibility check. Acyclicity can be checked by constructing a topological sort of the constraints graph. Every time we produce the next node in the sorting, we take a step in the construction of $\Phi(P)$ using the fact that the currently produced node is the minimal base type among the remaining ones. We can then backtrack as soon as a choice contradicts $\mathcal{T}$-compatibility.

The complexity of the type checking problem is easily seen to be linear in the size of the program. This proves, in conjnuction with the finiteness of the candidate guesses for $\mathcal{T}$ and annotations, that the type inference problem is in NP. We conjecture that inference is also NP-hard.

We implemented the above algorithm in a tool called 'James Bound' (`jb`), available at http://github.com/bordaigorl/jamesbound.

7 Expressivity and Verification

7.1 Expressivity

Typably hierarchical terms form a rather expressive fragment. Apart from including common patterns as the client-server one, they generalise powerful models of computation with decidable properties.

Relations with variants of CCS are the easiest to establish: CCS can be seen as a syntactic subset of π-calculus when including 0-arity channels, which are very easily dealt with by straightforward specialisations of the typing rules for actions. One very expressive, yet not Turing-powerful, variant is CCS$^!$ [10] which can be seen as our π-calculus without mobility. Indeed, every CCS$^!$ process is typably hierarchical [4, Sect. 11.4].

Reset nets can be simulated by using resettable counters as defined in Example 3. The full encoding can be found in [5]. The encoding preserves coverability but not reachability.

CCS$^!$ was recently proven to have decidable reachability [10] so it is reasonable to ask whether reachability is decidable for typably hierarchical terms.

We show this is not the case by introducing a weak encoding of Minsky machines (in [5]). The encoding is weak in the sense that not all of the runs represent real runs of the encoded Minsky machine; however with reachability one can distinguish between the reachable terms that are encodings of reachable configurations and those which are not. We therefore reduce reachability of Minsky machines to reachability of typably hierarchical terms.

Theorem 8. *The reachability problem is undecidable for (typably) hierarchical terms.*

Theorem 8 can be used to clearly separate the (typably) hierarchical fragment from other models of concurrent computation as Petri Nets, which have decidable reachability and are thus less expressive.

7.2 Applications

Although reachability is not decidable, coverability is often quite enough to prove non-trivial safety properties. To illustrate this point, let us consider Example 2 again. In our example, each client waits for a reply reaching its mailbox before issuing another request; moreover the server replies to each request with a single message. Together, these observations suggest that the mailboxes of each client will contain at most one message at all times. To automatically verify this property we could use a coverability algorithm for depth-bounded systems: since the example is typable, it is depth-bounded and such algorithm is guaranteed to terminate with a correct answer. To formulate the property

as a coverability problem, we can ask for coverability of the following query: $\nu\, s\ m.(!S \parallel m(y).\overline{c}\langle m\rangle \parallel \nu\, d.\overline{m}\langle d\rangle \parallel \nu\, d'.\overline{m}\langle d'\rangle)$. This is equivalent to asking whether a term is reachable that embeds a server connected with a client with a mailbox containing two messages. The query is not coverable and therefore we proved our property.[3]

Other examples of coverability properties are variants of secrecy properties. For instance, the coverability query $\nu\, s\ m\ m'.(!S \parallel m(y).\overline{c}\langle m\rangle \parallel m'(y).\overline{c}\langle m'\rangle \parallel \nu\, d.(\overline{m}\langle d\rangle \parallel \overline{m'}\langle d\rangle))$ encodes the property "can two different clients receive the same message?", which cannot happen in our example.

It is worth noting that this level of accuracy for proving such properties automatically is uncommon. Many approaches based on counter abstraction [7, 23] or CFA-style abstractions [6] would collapse the identities of clients by not distinguishing between different mailbox addresses. Instead a single counter is typically used to record the number of processes in the same control state and of messages. In our case, abstracting the mailbox addresses away has the effect of making the bounds on the clients' mailboxes unprovable in the abstract model.

A natural question at this point is: how can we go about verifying terms which cannot be typed, as the ring example? Coverability algorithms can be applied to untypable terms and they yield sound results when they terminate. But termination is not guaranteed, as the term in question may be depth-unbounded.

However, even a failed typing attempt may reveal interesting information about the structure of a term. For instance, in Example 10 one may easily see that the cyclic dependencies in the constraints are caused by the names representing the "next" process identities. In the general case heuristics can be employed to automatically identify a minimal set of problematic restrictions. Once such restrictions are found, a counter abstraction could be applied *to those restrictions only* yielding a term that simulates the original one but introducing some spurious behaviour. Type inference can be run again on the abstracted term; on failure, the process can be repeated, until a hierarchical abstraction is obtained. This abstract model can then be model checked instead of the original term, yielding sound but possibly imprecise results.

8 Related Work

Depth boundedness in the π-calculus was first proposed in [13] where it is proved that depth-bounded systems are well-structured transition systems. In [25] it is further proved that (forward) coverability is decidable even when the depth bound k is not known *a priori*. In [26] an approximate algorithm for computing the *cover set*—an over-approximation of the set of reachable terms—of a system of depth bounded by k is presented. All these analyses rely on the assumption of depth boundedness and may even require a known bound on the depth to terminate.

[3] To fully prove a bound on the mailbox capacity one may need to also ask another coverability question for the case where the two messages bear the same data-value d.

Several other interesting fragments of the π-calculus have been proposed in the literature, such as name bounded [11], mixed bounded [16], and structurally stationary [14]. Typically defined by a non-trivial condition on the set of reachable terms – a *semantic* property, membership becomes undecidable. Links with Petri nets via encodings of proper subsets of depth-bounded systems have been explored in [16]. Our type system can prove depth boundedness for processes that are breadth and name unbounded, and which cannot be simulated by Petri nets. In [2], Amadio and Meyssonnier consider fragments of the asynchronous π-calculus and show that coverability is decidable for the fragment with no mobility and bounded number of active sequential processes, via an encoding to Petri nets. Typably hierarchical systems can be seen as an extension of the result for a synchronous π-calculus with unbounded sequential processes and a restricted form of mobility.

Recently Hüchting et al. [12] proved several relative classification results between fragments of π-calculus. Using Karp-Miller trees, they presented an algorithm to decide if an arbitrary π-term is bounded in depth by a given k. The construction is based on an (accelerated) exploration of the state space of the π-term, with non primitive recursive complexity, which makes it impractical. By contrast, our type system uses a very different technique leading to a quicker algorithm, at the expense of precision. Our forest-structured types can also act as specifications, offering more intensional information to the user than just a bound k.

Our types are based on Milner's sorts for the π-calculus [9,18], later refined into I/O types [21] and their variants [22]. Based on these types is a system for termination of π-terms [3] that uses a notion of levels, enabling the definition of a lexicographical ordering. Our type system can also be used to determine termination of π-terms in an approximate but conservative way, by using it in conjunction with Theorem 1. Because the respective orderings between types of the two approaches are different in conception, we expect the terminating fragments isolated by the respective systems to be incomparable.

9 Future Directions

The type system we presented in Sect. 4 is conservative: the use of simple types, for example, renders the analysis context-insensitive. Although we have kept the system simple so as to focus on the novel aspects, a number of improvements are possible. First, the extension to the polyadic case is straightforward. Second, the type system can be made more precise by using subtyping and polymorphism to refine the analysis of control and data flow. Third, the typing rule for replication introduces a very heavy approximation: when typing a subterm, we have no information about which other parts of the term (crucially, which restrictions) may be replicated. By incorporating some information about which names can be instantiated unboundedly in the types, the precision of the analysis can be greatly improved. The formalisation and validation of these extensions is a topic of ongoing research.

Another direction worth exploring is the application of this machinery to heap manipulating programs and security protocols verification.

Acknowledgement. We would like to thank Damien Zufferey for helpful discussions on the nature of depth boundedness, and Roland Meyer for insightful feedback on a previous version of this paper.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: Symposium on Logic in Computer Science, pp. 313–321. IEEE Computer Society (1996)
2. Amadio, R.M., Meyssonnier, C.: On decidability of the control reachability problem in the asynchronous π-calculus. Nordic J. Comput. **9**(2), 70–101 (2002)
3. Deng, Y., Sangiorgi, D.: Ensuring termination by typability. Inf. Comput. **204**(7), 1045–1082 (2006)
4. D'Osualdo, E.: Verification of Message Passing Concurrent Systems. Ph.D. thesis, University of Oxford (2015). http://ora.ox.ac.uk/objects/uuid:f669b95b-f760-4de9-a62a-374d41172879
5. D'Osualdo, E., Ong, C.-H.L.: On hierarchical communication topologies in the pi-calculus. CoRR (2016). http://arxiv.org/abs/1601.01725
6. D'Osualdo, E., Kochems, J., Ong, C.-H.L.: Automatic verification of Erlang-style concurrency. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 454–476. Springer, Heidelberg (2013)
7. Emerson, E.A., Trefler, R.J.: From asymmetry to full symmetry: new techniques for symmetry reduction in model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 142–157. Springer, Heidelberg (1999)
8. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1–2), 63–92 (2001)
9. Gay, S.J.: A sort inference algorithm for the polyadic π-calculus. In: Deusen, M.S.V., Lang, B. (eds.) Principles of Programming Languages (POPL), pp. 429–438. ACM Press (1993)
10. He, C.: The decidability of the reachability problem for CCS$^!$. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 373–388. Springer, Heidelberg (2011)
11. Hüchting, R., Majumdar, R., Meyer, R.: A theory of name boundedness. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013 – Concurrency Theory. LNCS, vol. 8052, pp. 182–196. Springer, Heidelberg (2013)
12. Hüchting, R., Majumdar, R., Meyer, R.: Bounds on mobility. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 357–371. Springer, Heidelberg (2014)
13. Meyer, R.: On boundedness in depth in the π-calculus. In: IFIP International Conference on Theoretical Computer Science, IFIP TCS, pp. 477–489 (2008)
14. Meyer, R.: A theory of structural stationarity in the π-calculus. Acta Informatica **46**(2), 87–137 (2009)
15. Meyer, R.: Structural stationarity in the π-calculus. Ph.D. thesis, University of Oldenburg (2009)

16. Meyer, R., Gorrieri, R.: On the relationship between π-calculus and finite place/transition Petri nets. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
17. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(02), 119–141 (1992)
18. Milner, R.: The polyadic pi-calculus: a tutorial. Technical Report CS-LFCS-91-180, University of Edinburgh (1993)
19. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
20. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I, II. Inf. Comput. **100**(1), 1–77 (1992)
21. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. In: Symposium on Logic in Computer Science, pp. 376–385 (1993)
22. Pierce, B.C., Sangiorgi, D.: Behavioral equivalence in the polymorphic pi-calculus. J. ACM **47**(3), 531–584 (2000)
23. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
24. Vasconcelos, V.T., Honda, K.: Principal typing schemes in a polyadic π-calculus. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 524–538. Springer, Heidelberg (1993)
25. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)
26. Zufferey, D., Wies, T., Henzinger, T.A.: Ideal abstractions for well-structured transition systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 445–460. Springer, Heidelberg (2012)