A Study On The Potential
Active Names of $\pi$-Agents

Ana C. V. de Melo

Department of Computer Science
University of São Paulo (USP)
Rua do Matão, 1010 - Cidade Universitária
05508 090 - São Paulo - SP - Brazil
e-mail: acvm@ime.usp.br

Abstract

Verification techniques for CCS [5] cannot be directly used to verify $\pi$-calculus [6,7,10] agents. Montanari and Pistore [8] have pointed out that, under certain restrictions, if only active names are considered, labelled transition systems (LTS) for $\pi$-agents can be built in order to have tools and algorithms for CCS also used to check some equivalences in $\pi$-calculus.

There, they suggested a method for calculation of agents active names based on semantic models – the whole LTS is built first and the active names are calculated later on. This paper presents a syntactic characterisation of active names for $\pi$-agents. In this way, active names can be calculated for agents expressions instead of its corresponding LTS. The results from this study can be applied to reduce the size of $\pi$-agents in verification techniques based on both rewriting and behavioural models.

Keywords: Mobile agents, $\pi$-calculus, formal verification, active names.

1 Introduction

The problem of formal verification for $\pi$-calculus agents has been widely studied in the last decade. Regarding equivalences checking, we can roughly split verification techniques into those that address the problem at the syntactic level using rewriting systems, for example [2]; and those based on labelled transition systems (LTS) for the behavioural equivalences. In both cases, the problem of checking equivalences for $\pi$-agents is not trivial, and each technique has limitations.

For the behavioural equivalences checking, state explosion is still a problem since input actions may generate an infinite number of transitions when
agents have infinite sets of names. Montanari and Pistore \cite{8,9} have proved
that finite automata can be efficiently built for finitary $\pi$-calculus without
matching if representatives are chosen as inputs. As a result, minimal real-
ization automata of finitary $\pi$-agents can be found (irredundant unfolding),
and techniques and tools for checking CCS equivalences can be applied to
such automata. However, to build irredundant unfolding automata, active
names and representative new names must be discovered (they are used to
build finite branching automata).

The notion of active names is related to the idea of used names for con-
structing finite state transitions for CCS agents with value passing \cite{4}. Mota-
nari and Pistore redefined this idea for $\pi$-calculus: a name $x$ is active in $P$ if
$(\nu x) P$ is not bisimilar to $P$. This means that active names are a subset of
free names that play a role on agents’ behaviour and early bisimilar agents
have identical sets of active names. They also proposed an algorithm to calcu-
late active names given an agent labelled transition system: (i) a compacted
automaton is first constructed with the whole set of free names; (ii) active
names are calculated over this compacted automaton; and (iii) the inactive
names are removed from the compacted automaton (irredundant unfolding).
Once the compacted automata are built, bisimulations of such agents can be
checked with tools for CCS.

Working on active names instead of free names has a great significance for
verification techniques using the semantic approach: this may reduce the state
space of automata. This is also of interest to rewriting verification techniques
as agents expressions can be reduced in size with the elimination of actions
never engaged (namely, inactive actions). The costs of using active names to
check equivalences, as with \cite{8,9}, is on the calculation of compacted automata
which can be applied to the semantic approach, but not directly applied to
rewriting systems. On the other hand, if potential active names are calcu-
lated on agents expressions, inactive names can be removed from expressions,
reducing them in size and, consequently, reducing the state space of agents
transitions. As a result, active names could be applied to both syntactic
(rewriting) and semantic verification techniques.

This paper presents a study on the syntactic characterisation of potential
active names in order to calculate them on agents expressions. This is not de-
cidable in general and internal communications are necessary to be performed
to calculate active names of certain expressions. This study is devoted to iden-
tify classes of expressions that can be syntactically calculated, and those that
require internal actions. It first addresses the fragment of $\pi$-calculus in which
active names are trivially calculated from syntax. The fragment in which
internal communications are necessary to find the potential active names is
presented afterwards.

Firstly, \( \pi \)-calculus preliminaries are presented: the language fragment and its semantics. The following section is devoted to the study of active names for this language fragment. Section 4 presents some examples to show calculation of \( \pi \)-agents active names based on the previous study. The last section analyses the limitations of that study and give directions on how it could be applied to verification techniques.

2 \( \pi \)-calculus Preliminaries

The \( \pi \)-calculus fragment (monadic \( \pi \)-calculus) used in the present work\(^1\) is concerned with the one used in [8] plus guarded replicated agents \(^2\):

\[
Q := 0 \mid \alpha.P \mid (\nu x) \, P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid ! \alpha.P
\]

The language elements have the usual meaning. \( 0 \) represents the stop agent and cannot perform actions. \( (\nu x) \, P \) makes all elements in \( x \) restricted to agent \( P \). \( P_1 + P_2 \) represents the choice of either agents \( P_1 \) or \( P_2 \), while \( P_1 \mid P_2 \) is the composition of both agents. Finally, \( ! \alpha.P \) replicates \( \alpha.P \) as much as required; there is an infinite number of \( \alpha.P \)'s in composition.

Agent actions \( \alpha \) are defined as follows:

\[
\alpha := \tau \mid a(b) \mid \overline{a}(b) \mid \overline{\alpha}(b)
\]

\( \tau \) is the silent (or internal) action: an action with no observable behaviour. \( a(b) \) denotes an action receiving \( b \) along port \( a \), \( \overline{a}(b) \) denotes an action sending \( b \) along port \( a \), and \( \overline{\alpha}(b) \) sends the internal name \( b \) along port \( a \) to its context (scope extrusion). The restriction \( (\nu b) \) and the input action \( a(b) \) both bind name \( b \). \( b \) is a bound name (\( bn \)) in both cases, and a free name (\( fn \)) otherwise. Moreover, a name is bound to an agent if it appears bound in any of its actions and free otherwise.

2.1 Structural Congruence

One way of defining \( \pi \)-calculus semantics is first capturing the notion of structural congruence; agents that intuitively have the same behaviour and can be identified from their structure. The structural congruence \( \equiv \) is defined as the smallest congruence satisfying laws in Table 1 [10].

\(^1\) Matching and mismatching are not considered in this language fragment.

\(^2\) Replication has been included here to be concerned with some of the rewriting techniques for \( \pi \)-calculus.
(i) If $P$ and $Q$ are variants of alpha-conversion then $P \equiv Q$.

(ii) The Abelian monoid laws for Parallel and Sum:
(a) commutativity: $P \parallel Q \equiv Q \parallel P$, $P + Q \equiv Q + P$;
(b) associativity: $(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$, $(P + Q) + R \equiv P + (Q + R)$;
and
(c) 0 as unit: $P \parallel 0 \equiv P$ and $P + 0 \equiv P$.

(iii) The scope extension laws:
(a) $(\nu x) 0 \equiv 0$
(b) $(\nu x) P \equiv P$, if $x \notin \text{fn}(P)$
(c) $(\nu x) (P \parallel Q) \equiv (P \parallel (\nu x) Q)$, if $x \notin \text{fn}(P)$
(d) $(\nu x) (P + Q) \equiv (P + (\nu x) Q)$, if $x \notin \text{fn}(P)$
(e) $(\nu x) (\nu y) P \equiv (\nu y) (\nu x) P$

(iv) The replication laws:
(a) $!\alpha.P \parallel !\alpha.P \equiv !\alpha.P$
(b) $!\alpha.P \parallel !\alpha.P \equiv !\alpha.P$

Table 1
Structural Congruence Rules

2.2 Transition Rules

Structural congruence is not enough to define behavioural equivalences in process algebras. Besides structural congruence, an operational semantics of each combinator is necessary. Here, the semantics of $\pi$-calculus is given by a labelled transition system based on late semantics (Table 2) [10].

Rule Struc is introduced to take structural congruence accounted into the semantics. This also simplifies the transition system rules. Since commutativity laws are defined for summation and parallel composition in the structural congruence, there is no need to define the dual rules of Sum and Par.

The Close transition rule is not necessary to be defined here since it is a conclusion from structural congruence and rule Open; both have been defined here to explicitly have the notions of scope intrusion and extrusion.

2.3 Bisimulation

Behavioural equivalences of $\pi$-agents are required for practical use. There is nowadays a set of equivalences defined for $\pi$-calculus[10]. For the study purposes presented here, only early bisimulation has been considered.

Definition 2.1 An early bisimulation ($\approx_E$) with late semantics is a symmetric binary relation $R$ on agents satisfying the following: $P R Q$ and $P \xrightarrow{\alpha} P'$ where bn($\alpha$) is fresh implies that
1. if \( \alpha = a(x) \) then \( \forall u : \exists Q' : Q \overset{a(x)}{\rightarrow} Q' \land P'\{u/x\} R Q'\{u/x\} \), and;

2. if \( \alpha \) is not an input, then \( \exists Q' : Q \overset{\alpha}{\rightarrow} Q' \land P' R Q' \).

\( P \) and \( Q \) are (strongly) early bisimilar, written \( P \sim_E Q \), if they are related by an early bisimulation.

### 3 A Characterisation of Active Names

Active names of \( \pi \)-calculus agents (\( \pi \)-agents for short) were studied by Montanari and Pistore in [8] based on the idea of used names for value passing CCS. Pistore and Sangiorgi [11,12,13] proposed a partition refinement algorithm to check early and open bisimulations based on active names. That is a step forward to the Montanari’s work [8] who argued that bisimilar agents have the same set of active names. In fact, actions with inactive names, namely inactive actions, could be dropped from agents without changing their behaviours. Due to agents reconfiguration allowed in \( \pi \)-calculus, finding active names is as hard as checking bisimulations and a semantic approach was followed in both works. In the present work, a study on recognising active names from \( \pi \)-agents expressions is carried out, giving a syntactic characterisation instead of following the semantic approach as in the previous works.

A name is said semantically active in agent \( P \) if it is a free name and can be performed by \( P \). Clearly, actions having their port names restricted cannot be performed in isolation (except when an internal communication occur), and
all names involved in are not relevant to the agent behaviour. An important result from this rests on fact that bisimilar agents have the same set of active names: the active names of an agent is the smallest subset of free names which affects the agent behaviour. This is formally stated in [8] as follows:

**Definition 3.1** A name $a$ is active for an agent $P$ iff $P \not\sim (\nu a) \ P$; $\text{an}(P) = \{ a | P \not\sim (\nu a) \ P \}$ is the set of active names for the agent $P$.

**Proposition 3.2** If $P \sim P'$ then $\text{an}(P) = \text{an}(P')$.

Basically, a name is inactive in an agent if it is unable to change such agent from the external context point-of-view. Certain names may play a role only on internal actions of agents; they are thus unable to interfere on the external context. Besides that, certain actions are never performed due to names restrictions. So, agents names exclusively involved in either internal actions or actions never engaged in that agent are the inactive names.

In order to have active and inactive names of agents identified, active names of single actions must be defined firstly:

**Definition 3.3** Active names of single actions are as follows [8]:

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\text{an}(\alpha)$</th>
<th>$\overline{\text{an}}(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>$\overline{ab}$</td>
<td>{a, b}</td>
<td>{}</td>
</tr>
<tr>
<td>$\overline{a(b)}$</td>
<td>{a}</td>
<td>{b}</td>
</tr>
<tr>
<td>$a(b)$</td>
<td>{a}</td>
<td>{b}</td>
</tr>
</tbody>
</table>

Active names of action $\alpha$ ($\text{an}(\alpha)$) is the set of its free names. This corresponds to the port names of input and output actions and output information. On the other hand, input and output bound information are represented by inactive names ($\overline{\text{an}}(\alpha)$). The silent action has neither active or inactive names. Despite being essential to identify agents inactive names, the definition of active and inactive names of actions are not enough to calculate the whole set of active names of agents due to agents reconfiguration.

In order to study a characterisation of agents active names, we must consider a standard form: the standard form of agents by Milner [6] is used here with the limitation that all replicated agents have a prefixing action \footnote{This is also a limitation on the rewriting systems for $\pi$-calculus (to guarantee uniqueness of normal forms) and has been adopted here to make the analysis shorter.} $\pi$. The standard form of $\pi$-agents includes parallel and replicated agents, and all names restrictions are on the top of expressions.
Definition 3.4 [6] A process expression

\[(\nu \vec{x}) M_1 | \ldots | M_m \; | \; \alpha_1.Q_1 \ldots \; | \; \alpha_n.Q_n\]

is said to be in a standard form if each \(M_i\) is a non-empty sum, and each \(Q_j\) is itself in a standard form.

For that standard form, alpha-conversion is considered in a way that all bound names are new (free and bound names do not clash in agents). As such, operations over agents names set can be conducted without checking the subsequent names in expressions.

The forthcoming sections summarise an analysis on active names of agents based on the expression structures of \(\pi\)-agents (syntax). This is a syntactic characterisation of active names, instead of the transition (semantic) one defined by Montanari and Pistore. The study has been carried out concerned with a structural induction approach: single \(\pi\)-calculus operators are treated followed by their combinations. The \(\pi\)-calculus fragment in which active names can be calculated directly from expressions is first analysed, followed by the fragment in which internal communications must be engaged to have active names calculated.

3.1 Active Names without Reaction

Certain active names can be calculated from agents expressions without engaging internal actions. This is particularly the case of prefixing actions. Prefixing actions, even if they are input actions, can have their active names calculated looking at agents expressions without performing transitions.

The set of agents active names is mainly the set of their free names, except those appearing exclusively in actions never performed by the agent. Looking at \(\pi\)-calculus expressions, agents active names calculation is analysed over each operator separately.

3.1.1 Stop

\[Q \xleftarrow{\text{def}} 0, \text{ then} \]

\[\text{an}(Q) = \text{fn}(Q) = \{ \}

The free names of the 0 agent is given by the empty set; it has neither bound or free names.

3.1.2 Prefixing

\[Q \xleftarrow{\text{def}} \alpha.P, \text{ then} \]

\[\text{an}(Q) = \text{fn}(\alpha) \cup (\text{an}(P) - \text{bn}(\alpha))\]
For agents with a prefixing action, the active names of the prefixing action is first calculated and the subsequent agent has its active names calculated. Note that bound names are removed from the set of active names.

### 3.1.3 Summation

If \( Q \overset{\text{def}}{=} P_1 + P_2 \), then

\[
\text{an}(Q) = \text{an}(P_1) \cup \text{an}(P_2)
\]

The summation agent has its active names calculated by the union of each summand separately. There is no need to eliminate the bound names of an agent from the other (eliminate \( \text{bn}(P_1) \) from \( P_2 \) and vice-versa) since no bound names clash to free names in standard form agents.

### 3.1.4 Restriction

(i) Prefixing:

\[
Q \overset{\text{def}}{=} (\nu \vec{x}) \alpha.P:
\]

\[
\text{an}(Q) = \begin{cases} 
\{
\end{cases} 
\]

\[
\text{fn}(\alpha) \cup (\text{an}((\nu \vec{x}) - \{b\}P) - \text{bn}(\alpha)), \alpha = \overline{a}(b) \land a \notin \vec{x}
\]

\[
\text{fn}(\alpha) \cup (\text{an}((\nu \vec{x}) P) - \text{bn}(\alpha)), \text{otherwise}
\]

In the case above, output bound actions have been treated as special case because it must be removed from the list of restricted names from that point on. Output bound names are not active because they are fresh names created by the system. However, their extrusion extends the agent scope drastically and may change the new active actions to be calculated. If such bound output becomes a channel name in the remaining agent expression, the agent has a chance of not deadlocking because of the scope extrusion. Consider the following single example:\(^\text{4}\):

\[
P \overset{\text{def}}{=} (\nu b) a(x).\overline{b}.\overline{a}.d(y)
\]

\[
\text{an}(P) = \{a\} \cup (\text{an}((\nu b) \overline{b}.\overline{a}.d(y)) - \{x\})
\]

\[
= \{a\} \cup (\{c\} \cup (\text{an}(\overline{b}.d(y)) - \{b\}) - \{x\}
\]

\[
= \{a\} \cup (\{c\} \cup (\{a, b\} \cup (\text{an}(d(y))) - \{b\}) - \{x\}
\]

\[
= \{a\} \cup (\{c\} \cup (\{a, b\} \cup (\{d\} - \{y\}) - \{b\}) - \{x\}
\]

\[
= \{a, b, c, d\} - \{y, b, x\} = \{a, c, d\}
\]

\(^\text{4}\) The stop \((0)\) agent is omitted from the end of agents for short.
If no scope extrusion was considered, actions $\bar{ba}$ and $d(y)$ could not be performed because $b$ was restricted. So, with the $b$ scope extrusion (from action $\bar{b}b$ on), the following actions become active and their free names are also active.

(ii) Summation:
\[
Q \overset{\text{def}}{=} (\nu \vec{x}) (P_1 + P_2), \text{ then }
\]
\[
\text{an}(Q) = \text{an}((\nu \vec{x}) P_1) \cup \text{an}((\nu \vec{x}) P_2)
\]

For summation with restriction, summands are treated separately since no communication can be performed between both agents. As such, all active names from both restricted agents are also active in the whole agent (a single union is considered).

3.1.5 Composition

(i) Prefixing:
\[
Q \overset{\text{def}}{=} \alpha.P_1 | \beta.P_2, \text{ then }
\]
\[
\text{an}(Q) = \text{an}(\alpha.P_1) \cup \text{an}(\beta.P_2)
\]

For the asynchronous composition, agents are evaluated in isolation since no restriction is observed and agents are in a standard form: no bound names of $P_1$ may coincide with free names of $P_2$ (and vice-versa). As a result, all active names from both agents are also active in the whole agent and a single union can be considered.

Even when scope extrusion is applied, due to communications involving bound output actions, active names of each agents remain active in the compound agent. Considering that composition is asynchronous in $\pi$-calculus, the set of active names of both agents in composition cannot be reduced due to their internal communications. On the other hand, no new active name would come up with internal communications since there is no restricted names.

(ii) Summation:
\[
Q \overset{\text{def}}{=} \sum_{i=1}^{n} \alpha_i.P_i | \sum_{j=n+1}^{n+m} \alpha_j.P_j, \text{ then }
\]
\[
\text{an}(Q) = \bigcup \{\text{an}(\alpha_i.P_i) | 1 \leq i \leq n + m\}
\]

This is a generalisation of the summation and composition and has been introduced here to cover the standard form.

3.1.6 Replication

(i) Prefixing:
Whenever an agent is reconfigured, only its bound names may change. Then, no new active name can emerge from agents due to reconfiguration (active names is a subset of free names). Also, if no restriction is applied, there is no way to reach deadlocked agents. As a result, no free names are removed from the set of active names.

Scope restriction must be applied whenever a communication is taken. However, this agent is normalized, and no restriction is applied a priori. So, even if bound actions are found and scope extrusion is applied, all free names are active since no restriction is found on the top. This is, in fact, a particular case of composition.

(ii) Composition:

\[
Q \overset{\text{def}}{=} ! \alpha.P \mid ! \beta.P, \text{ then } \]

\[
an(Q) = an(\alpha.P) \cup an(\beta.P)
\]

Since no restriction is applied \textit{a priori} to the agent, all free names are active and active names can be calculated without reactions (justifications are similar to the above).

The cases discussed above show the fragment of \(\pi\)-calculus language in which a “single spelling” of expressions is enough to identify (calculate) agents active names. This result is a step forward to the conclusions by Pistore and Sangiorgi [11] who argued that for the fragment of \(\pi\)-calculus without parallel composition and matching active names could be syntactically calculated since they coincide to the set of free names. Here, we have shown that this can be extended to parallel composition without restricted names (this also includes replicated agents). The forthcoming section shows the calculation of active names including restriction, parallel composition and replication together.

3.2 Active Names under Reaction

For the general case, this is not possible to calculate all agents active names without engaging into internal communications. For agents comprising sub-agents in parallel with no restricted names, active names can be calculated without engaging into internal actions, as presented in the previous section. For agents that can exclusively engage into internal actions, however, such internal steps must be performed in order to check what actions come next and, then, calculate their active names. The present section summarises the calculation of potential agents active names whenever reactions are needed.
3.2.1 Restriction

(i) Composition: \( Q \overset{\text{def}}{=} (\nu \bar{x}) \alpha.\ P_1 \mid \beta.\ P_2, \) then
\[
\text{an}(Q) = \bigcup \left\{ \begin{array}{l}
(\text{an}((\nu \bar{x}) \beta.\ P_2)) \cup (\text{an}((\nu \bar{x}) \alpha.\ P_1)), \\
(\text{an}((\nu \bar{x}) (P_1\{c/b\} \mid P_2)) - \text{bn}(\alpha)), \quad \alpha = a(b), \\
(\text{an}(((\nu \bar{x}) \cup \{c\}) (P_1\{c/b\} \mid P_2)) - \text{bn}(\alpha) - \text{bn}(\beta)), \quad \alpha = a(b), \\
\end{array} \right.
\]
\[\beta = \pi c, \quad a \in \bar{x}
\]

Reactions are needed to calculate active names if only internal communications can be engaged. This leads to an explosion of new expressions because agents are reconfigured with such internal actions. Even so, this is limited by the number of internal actions.

For this calculation, three main special cases must be treated. In the first case, active names calculation is made of active names of each agent in isolation. The last two take care of all potential active names that can emerge as internal actions are performed. They capture the idea of having agents deadlocking in isolation, but progressing as internal communications are engaged.

The second definition treats calculation of active names as agents are reconfigured due to an internal communication. The last one captures the idea of scope extrusion as internal actions are engaged. Note that agents that cannot progress a priori may become progressing as scope extrusion is performed. So, names in actions coming after such extrusion may become active due that internal communication.

(ii) Composition-Summation:
\[
Q \overset{\text{def}}{=} (\nu \bar{x}) (\sum_{i=1}^{n} \alpha_i.\ P_i \mid \sum_{j=n+1}^{m} \alpha_j.\ P_j), \] then
\[
\text{an}(Q) = \bigcup \{ \text{an}((\nu \bar{x}) (\alpha_i.\ P_i \mid \alpha_j.\ P_j)) | 1 \leq i \leq n, n + 1 \leq j \leq m \}
\]
For this case, the combination of agents in composition must be performed. This is necessary to cover all possible internal communications between pairs of agents.

(iii) Replication:
\[
Q \overset{\text{def}}{=} (\nu \bar{x})! \alpha.P, \] then
\[
\text{an}(Q) = \text{an}((\nu \bar{x}) \prod_{i=1}^{n} \alpha.P), \quad n = \exp\text{size}(\alpha.P) + 1
\]
Active names cannot emerge from reconfiguration; they are a subset of free names. Replicated agents can then have their active names calculated by composition of a finite number of agent copies, instead of infinite copies as suggested by the replication operator.

The number of copies considered is based on the size of expressions involving the prefixing and composition operators. In principle, we may consider a single agent to calculate its active names. However, for restricted compound agents (which is the case of replicated agents), certain compositions might enlarge the set of active names due to reconfiguration and scope extrusion. In order to get all possible internal communications, we must then consider at most the number of subsequent actions an agent can perform. This corresponds to the size of expressions \( \text{expsize}(E) \) considering only prefixing and composition operators.

(iv) Replication-Composition:

\[
Q \overset{\text{def}}{=} (\nu \vec{x}) \prod_{i=1}^{n} ! \alpha_i.P_i, \text{ then } \\
\text{an}(Q) = \bigcup \left\{ \bigcup \{ \text{an}(\langle \nu \vec{x} \rangle (! \alpha_i.P_i)) | 1 \leq i \leq n \} \right\} \\
\bigcup \left\{ \bigcup \{ \text{an}(\langle \nu \vec{x} \rangle (\alpha_i.P_i | \alpha_j.P_j)) | 1 \leq i, j \leq n \land i \neq j \} \right\}
\]

This is similar to the case above. Here, each individual replicated agent is considered together with composition of all pairs of replicated agents.

As can be noted, the fragment of \( \pi \)-calculus for which internal actions must be engaged to have active names calculated involves the restriction operator combined with one kind of composition. This results in combination of restriction with composition (and composition-summation to cover the normal form), and restriction with replication which is a kind of composition (infinitely many compositions).

Considering the definitions above, we may note that many cases lead to an explosion of new agents, even before reactions are actually performed. Algorithms for calculation of active names must consider that certain individual agents may appear in various places and make its unification in calculation. In this way, the number of new expressions might be reduced and their active names uniquely calculated.

4 Calculating Active Names: Examples

This section presents a set of examples to show how to calculate active names from \( \pi \)-agents expressions. First, some examples of agents that can have their active names calculated without engaging into internal actions are presented. The calculation of active names for agents expressions with internal reactions
is presented in the forthcoming section.

Here, agents active names are calculated by a straightforward application of active names definitions from Section 3 (over agents expressions). A recursive algorithm for this calculation is derivable from definitions since they are based on structural induction of $\pi$-expressions. However, this is not the best algorithm (as discussed above) particularly for those calculations in which internal reactions are necessary. For that case, a better algorithm can be developed to recognise agents congruence up to alpha-conversion. A study of such algorithm is out of the scope of this paper.

4.1 Without Reactions

This section first shows few examples for the fragment of $\pi$-calculus involving only prefixing and summation. For agents with such characterisation, active names are calculated throughout expressions spelling; no internal action is performed.

Example 4.1 $Q1 \overset{\text{def}}{=} a(x).x(y).\overline{y}b$

\[
an(Q1) = fn(a(x)) \cup (an(x(y)).\overline{y}b) - bn(a(x))
= \{a\} \cup (fn(x(y)) \cup (an(\overline{y}b) - bn(x(y))) - \{x\})
= \{a\} \cup (\{x\} \cup (fn(\overline{y}b) \cup (an(0) - bn(\overline{y}b)) - \{y\}) - \{x\})
= \{a\} \cup (\{x\} \cup (\{y, b\} \cup (\{\} - \{\}) - \{y\}) - \{x\})
= \{a, b\}
\]

In the example above, the agent expression is a sequence of prefixing actions. To calculate its active names, we only need to read each action from left to right and add the free names and remove all bound names from the whole set of the agent active names. This is a straightforward application of agent active names definitions for prefixing actions.

Example 4.2 $Q2 \overset{\text{def}}{=} a(x).x(y).\overline{y}b + bc.c(z)$

\[
an(Q2) = an(a(x).x(y).\overline{y}b) \cup an(bc.c(z))
= an(Q1) = \{a, b\}
= \{b, c\} \cup (\{c\} \cup (\{\} - \{z\}) - \{\})
= \{b, c\}
= \{a, b\} \cup \{b, c\} = \{a, b, c\}
\]
In the case of summation, as above, the calculation is split in two separate cases: one case per summand. The active names of the whole agent is given by a union of both summand agents active names. Note that this single union is only possible because the agent is in a normal form: no free and bound names clash.

For agents with restricted names, scope extrusion has to be considered. If an action has its port name restricted, it cannot be engaged a priori. There are, however, two situations in which such restricted action could be engaged: its port name is previously extruded to a wider scope, or this action is able to communicate to another inside the restricted scope. The first case is illustrated in the following example.

**Example 4.3** 

\[ Q_3 \overset{\text{def}}{=} (\nu b) (a(x).x(w) + \overline{\nu b}(b).\overline{bc}.c(z)) \]

\[
\begin{align*}
\text{an}(Q3) & = \text{an}((\nu b) a(x).x(w)) \cup \text{an}((\nu b) \overline{\nu b}(b).\overline{bc}.c(z)) \\
\text{an}((\nu b) a(x).x(y)) & = \{a\} \\
\text{an}((\nu b) \overline{\nu b}(b).\overline{bc}.c(z)) & = \text{fn}(\overline{\nu b}(b)) \cup (\text{an}(\overline{bc}.c(z)) - \text{bn}(\overline{\nu b}(b))) \\
& = \{y\} \cup \text{fn}(\overline{bc}) \cup ((\text{an}(c(z)) - \text{bn}(bc) - \{b\})) \\
& = \{y\} \cup \{b, c\} \cup \{c\} \cup (\{\} - \{z\}) - \{\} - \{b\} \\
& = \{y, c\} \\
\text{an}(Q3) & = \{a\} \cup \{y, c\} = \{a, y, c\}
\end{align*}
\]

If scope extrusion was not considered in the agent above, only the first action from the second summand would engage and, as a result, name \(c\) would be missed from the set the agent active names. Considering scope extrusion, however, name \(b\) is extruded making \(c\) to be accounted as active name in that agent. Note that \(b\) is no longer restricted in expression as soon as it is extruded to the environment.

Active names of compound agents without restrictions can also be calculated on expressions without making internal actions. It does not matter if the compound agents can make communications since reconfiguration does not affect the set of active names of unrestricted agents. The following example illustrates a compound agent:

**Example 4.4** 

\[ Q_4 \overset{\text{def}}{=} a(x).x(y) | \overline{ab}.\overline{bc}.c(z) \]
\[
an(Q4) = \text{an}(a(x).x(y)) \cup \text{an}(\overline{ab}.\overline{bc}.c(z))
\]
\[
an(a(x).x(y)) = \{a\}
\]
\[
an(\overline{ab}.\overline{bc}.c(z)) = \text{fn}(\overline{ab}) \cup (\text{an}(\overline{bc}.c(z)) - \text{bn}(\overline{ab}))
\]
\[
= \{a, b\} \cup (\text{fn}(\overline{bc}) \cup (\text{an}(c(z)) - \text{bn}(\overline{bc}) - \{\})
\]
\[
= \{a, b\} \cup (\{b, c\} \cup (\{c\} \cup (\{\} - \{z\}) - \{\}) - \{\})
\]
\[
= \{a, b, c\}
\]
\[
an(Q4) = \{a\} \cup \{a, b, c\} = \{a, b, c\}
\]

Observe that the compound agents can engage into internal communications. Even so, actions that can communicate internally can also communicate with the external environment because there is no restriction on the top of expression. As a result, no scope extrusion is made and all actions in the expression can be engaged (independently from input names). For agents with no restricted names, input names does not affect their sets of active names.

### 4.2 With Reactions

As we have seen with this study, internal communications must be performed to calculate active names of certain expressions. In particular, whenever restriction and composition appear in expressions, reconfiguration must be performed due to internal communications. This is necessary because certain reconfigurations may enhance the agents capability of communications. This section presents how to calculate active names as internal communications are necessary.

**Example 4.5** Suppose agents \(Q5\) and \(Q6\) defined as follows:
\[ Q_5 \overset{\text{def}}{=} \tau d.a(x).\pi y \]
\[ Q_6 \overset{\text{def}}{=} \tau (b).b(z).\pi w \]
\[(\nu a) \ Q_5 \overset{\tau d}{\rightarrow} (\nu a) \ (a(x).\pi y) \sim 0 \]
\[(\nu a) \ Q_6 \sim 0 \]
\[(\nu a) \ (Q_5 \mid Q_6) \overset{\tau d}{\rightarrow} (\nu a) \ (a(x).\pi y \mid Q) \overset{\tau}{\rightarrow} (\nu a, b) \ (b y \mid b(z).\pi w) \]
\[ \tau (\nu a, b) \ (0 \mid \pi w) \overset{\pi w}{\rightarrow} (\nu a, b) \ (0 \mid 0) \]
\[ \text{an}( (\nu a) \ Q_5 ) = \{ c, d \} \]
\[ \text{an}( (\nu a) \ Q_6 ) = \{ \} \]
\[ \text{an}( (\nu a) \ (Q_5 \mid Q_6) ) = \{ c, d, y, w \} \]

The example above shows a case in which scope extrusion is performed and then actions can be engaged due to such names extrusion, otherwise the agent would stop. Agent \((\nu a) \ Q_5\), for example, is unable to perform action \(a(x)\) because name \(a\) is restricted on the top of the expression. For the same reason \((\nu a) \ Q_6\) has a behaviour bisimilar to agent stop (cannot perform any action). Although both agents are not able to progress in isolation due to those restricted actions, they can engage into internal communications if they are composed and restriction is put on the top of the whole agent scope. As a result, \((\nu a) \ (Q_5 \mid Q_6)\) can perform all \(Q_5\) and \(Q_6\) actions and the active names set must comprise all those actions. Note that scope extrusion of name \(b\) is first performed due to the internal action, and this makes a second internal action possible.

## 5 Final Considerations

This paper presented a study on the syntactic characterisation of active names of \(\pi\)-agents expressions. Since bisimilar agents have the same set of active names, the calculation of that set of names is of particular interest to \(\pi\)-calculus verification techniques. The main results from this study can be applied to both semantic and syntactic approaches of verification.

Checking bisimulation can be performed by rewriting rules concerned with structural congruence. There, a normal form for \(\pi\)-expressions is used and agents are bisimilar if they can be rewritten to the same (identical) normal form. Besides that, Hirschkoff et al [2,3] have developed a technique to extend the idea of checking structural congruence to up-to-bisimulation developed by Sangiorgi [12,13].
With the syntactic characterisation of active names, we can also find the potential inactive names of agents. From that, we may discover part of expressions never used because it is preceded by names semantically inactive (namely inactive actions). Such situation leads to the stop agent. In other words, we may use the characterisation of active names to reduce $\pi$-expressions and then enhance $\pi$-calculus rewriting verification techniques. This work on the development of new rewriting rules that eliminate part of expressions progressing to the stop agent is underway (new rules have been created but their proofs are still under development).

For the semantic approach, the precalculation of active names may reduce state explosion. If active names are calculated in expressions, there is no need to engage into transitions of free names that are not active, reducing the space of input transitions. In [8], for example, the transition system is first created considering the whole set of free names and further reduced to the set of active names. For certain expressions, only active names might be considered to build the transition system instead of the whole set of free names. For expressions comprising inactive names, the state space can be reduced a priori (there is no need to build the automaton with spare transitions and states and eliminate them later).

In this paper, calculation of agents active names is a straightforward application of active names definitions from the study. However, an efficient algorithm must be developed to such calculation. We have noted that when internal actions are needed to the active names calculation, reconfigured expressions could be alpha-conversion of others (mainly for replicated agents). For such a case, a unification on alpha-convertible expressions must be applied to enhance the performance of active names calculation. An algorithm for this has not been suggested so far, despite of being essential to measure the complexity of applying active names calculation to the existing verification techniques (suggested above).

References


