Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 154 (2006) 31–46

Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Graphical Verification of a Spatial Logic for the π-calculus

Fabio Gadducci[1] and Alberto Lluch Lafuente[2]

Dipartimento di Informatica, Università di Pisa
largo Bruno Pontecorvo 3c, I-56127 Pisa, Italia

Abstract

The paper introduces a novel approach to the verification of spatial properties for finite π-calculus specifications. The mechanism is based on a recently proposed graphical encoding for mobile calculi: Each process is mapped into a (ranked) graph, such that the denotation is fully abstract with respect to the usual structural congruence (i.e., two processes are equivalent exactly when the corresponding encodings yield the same graph). Spatial properties for reasoning about the behavior and the structure of π-calculus processes are then expressed in a logic introduced by Caires, and they are verified on the graphical encoding of a process, rather than on its textual representation. More precisely, the graphical presentation allows for providing a simple and easy to implement verification algorithm based on the graphical encoding (returning true if and only if a given process verifies a given spatial formula).

Keywords: Process calculi, spatial logic, verification.

1 Introduction

A recent series of papers advocated *spatial logics* as a suitable formalism for expressing behavioral and spatial properties of system specifications, often given as processes of a calculus. Besides the temporal modalities of the Hennessy-Milner tradition, these logics include operators for reasoning about the structural properties of a system. For example, the connective `void` represents the

⋆ Research partly supported by the EU within the projects HPRN-CT-2002-00275 SegraVis (*Syntactic and Semantic Integration of Visual Modelling Techniques*) and IST-2004-16004 SEnSOria (*Software Engineering for Service-Oriented Overlay Computers*).

1 Email: `gadducci@di.unipi.it`

2 Email: `lafuente@di.unipi.it`

(processes structurally congruent to the) empty system, and the formula $\phi_1|\phi_2$ is satisfied by those processes that can be decomposed into two parallel components, satisfying ϕ_1 and ϕ_2, respectively. Moreover, these logics come equipped with mechanisms for reasoning about the names occurring in a system.

There are several approaches to the verification of spatial properties, on logics either for process calculi (see e.g. [2,3,4] and the references therein) or for other data structures such as heaps [14], trees [6], and graphs [5]. In this paper we propose a novel approach to the verification of spatial formulae [2] for finite π-calculus specifications, based on a graphical encoding for nominal calculi [8]. Even if a few articles have been already proposed on the verification of graphically described systems (see e.g [1,13,16]), to the best of our knowledge this is the first attempt to the model-checking of spatial properties for processes of nominal calculi, based on a graphical presentation.

Our paper is to be considered a combination of the graphical encoding of the π-calculus in [8] and of the verification techniques for spatial properties in [2], and it provides mechanisms for checking spatial formulae on the graphical representation of processes. Even if the present work focuses on the finite fragment of the π-calculus (hence on the recursion-free formulae of the spatial logic), we believe that it may offer novel insights on the model-checking of spatial formulae, possibly linking it to the standard logics for graphs; moreover, it offers further evidence of the adequacy of graph-based formalisms for system design and verification.

The structure of the paper is as follows. Section 2 presents the finite fragment of the π-calculus and the spatial logic for processes proposed in [2]. Section 3 recalls the main definitions concerning ranked graphs [7]. Section 4 presents an encoding of π-calculus processes into ranked graphs, streamlining the proposal already discussed in [8]. Section 5 introduces our algorithm for the verification of (closed) spatial formulae, briefly discussing its computational costs. The final section outlines future research avenues.

2 The π-calculus and a Spatial Logic

2.1 Synchronous (finite) π-calculus

This section briefly recalls the main definitions concerning the finite, deterministic fragment of the synchronous π-calculus.

Definition 2.1 (processes) *Let $\mathcal{N}$ be a set of* names, *ranged over by $a, b, c, \ldots$; and let $\Delta = \{a(b), \overline{a}b \mid a, b \in \mathcal{N}\}$ be the set of* prefix operators, *ranged over by δ. A* process *P is a term generated by the syntax*

$$P ::= \quad \mathtt{0} \quad | \quad (\nu a)P \quad | \quad P|P \quad | \quad \delta.P$$

We let $P, Q, R, \ldots$ range over the set $\mathcal{P}$ of processes.

The standard definition for the set of free names of a process P, denoted by $\mathtt{fn}(P)$, is assumed. Similarly for α-convertibility, with respect to the *restriction* operators $(\nu a)P$ and the *input* operators $b(a).P$: In both cases, the name a is bound in P, and it can be freely α-converted.

Using the definitions above, the behavior of a process P is described as a relation over *abstract processes*, i.e., a relation obtained by closing a set of basic rules under structural congruence.

Definition 2.2 (structural congruence) *The* structural congruence *for processes is the relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$, closed under process construction and α-conversion, inductively generated by the following set of axioms*

$$P \mid Q = Q \mid P \qquad P \mid (Q \mid R) = (P \mid Q) \mid R \qquad P \mid \mathtt{0} = P \qquad (\nu a)\mathtt{0} = \mathtt{0}$$
$$(\nu a)(\nu b)P = (\nu b)(\nu a)P \qquad (\nu a)(P \mid Q) = P \mid (\nu a)Q \text{ for } a \notin \mathtt{fn}(P)$$

Definition 2.3 (reductions) *The* reduction relation *for processes is the equivalence relation $R_\pi \subseteq \mathcal{P} \times \mathcal{P}$, closed under the structural congruence $\equiv$, inductively generated by the following set of axioms and inference rules*

$$\frac{}{a(b).P \mid \overline{a}c.Q \to P\{^c/_b\} \mid Q} \qquad \frac{P \to Q}{(\nu a)P \to (\nu a)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R}$$

where $P \to Q$ means that $(P, Q) \in R_\pi$.

The first rule denotes the communication between two processes: Process $\overline{a}c.Q$ is ready to communicate the (possibly global) name c along the channel a; it then synchronizes with process $a(b).P$, and the local name b is substituted by c on the residual process P (avoiding, as usual, the capture of name c). The latter rules state the closure of the reduction relation with respect to the operators of restriction and parallel composition.

Finally, we present the commitment relation, a variant of the standard labeled transition system semantics, introduced in [2] for verification purposes.

Definition 2.4 (commitments) *Let $\Lambda = \{\tau\} \uplus \{a[b], \overline{a}b \mid a, b \in \mathcal{N}\}$ be the set of* commitment labels*, ranged over by λ. The* commitment relation *for processes is the relation $R_c \subseteq \mathcal{P} \times \Lambda \times \mathcal{P}$, closed under the structural congruence*

$\equiv$, inductively generated by the following set of axioms and inference rules

$$\frac{P \to Q}{P \xrightarrow{\tau} Q} \quad \frac{a, c \notin N}{(\nu N)(\overline{a}c.P|Q) \xrightarrow{\overline{a}c} (\nu N)(P|Q)} \quad \frac{a, c \notin N}{(\nu N)(a(b).P|Q) \xrightarrow{a[c]} (\nu N)(P\{^c/_b\}|Q)}$$

where $P \xrightarrow{\lambda} Q$ means that $\langle P, \lambda, Q\rangle \in R_c$ and (νN) stands for $(\nu a_1)\dots(\nu a_k)$ for any finite $N = \{a_1, \dots, a_k\} \subset \mathcal{N}$.

Example 2.5 *Let us consider the process **race** $= (\nu a)\overline{b}a.\overline{a}a \mid b(d).\overline{d}c$. The sub-process on the left is ready to send a bound name a via a channel b. After a scope extension of the restriction operator, a possible commitment of **race** thus consists of a synchronization on b: **race** $\xrightarrow{\tau} (\nu a)(\overline{a}a \mid \overline{a}c)$. The residual process is deadlocked, since the restriction forbids a to be observed. Removing the restriction results in a process that may perform commitments $\overline{a}a \mid \overline{a}c \xrightarrow{\overline{a}a} \overline{a}c$ (a sent over a) and $\overline{a}a \mid \overline{a}c \xrightarrow{\overline{a}c} \overline{a}a$ (c sent over a).*

2.2 Spatial logic

This section recalls the finite fragment of the spatial logic presented in [2].

Definition 2.6 (logic syntax) *Let V be a set of* name variables*, ranged over by $x, y, \dots$, and let $\Xi = \Lambda \cup \{x[y], \overline{x}y \mid x, y \in V\}$ be the set of* observables*, ranged over by ξ. A* spatial formula *is a term generated by the syntax*

$$\phi ::= T \mid \neg\phi \mid \phi \vee \phi \mid \texttt{void} \mid \phi|\phi \mid \eta \circledR \phi \mid \exists x.\phi \mid Ⅎx.\phi \mid \eta = \eta \mid \langle\xi\rangle\phi$$

where $\eta \in V \uplus \mathcal{N}$. We let $\phi, \phi_1, \dots$ range over the set $\mathcal{SF}$ of spatial formulae.

Boolean connectives have the usual meaning; void characterizes processes that are structurally congruent to the empty process; $\phi_1|\phi_2$ holds for processes that are structurally congruent to the composition of two sub-processes, satisfying ϕ_1 and ϕ_2, respectively; $\eta \circledR \phi$ is true for those processes such that ϕ holds after the revelation of name η; $\exists x.\phi$ and $Ⅎx.\phi$ characterize processes such that ϕ holds for a name in $\mathcal{N}$ and a *fresh* name in $\mathcal{N}$ (see below), respectively; $\eta_1 = \eta_2$ requires η_1 and η_2 to be equal; and $\langle\lambda\rangle\phi$ is satisfied by a process P if P can be committed into Q with label λ and Q satisfies ϕ.

A formula is *closed* if all its variables occur inside the scope of either an existential or a fresh quantifier. The set of free names of a formula ϕ, denoted as $\texttt{ffn}(\phi)$, coincides with the set of names occurring in it, since the only binding operators are the variable quantifiers. A name is *fresh* in a formula (process) if it is different from any free name of the formula (process, respectively).

Definition 2.7 (logic semantics) *The* denotation $[\![\phi]\!]$*, mapping a closed* *formula ϕ into a set of abstract processes, is defined by*

$$\begin{array}{rlrl}
[\![T]\!] &= \mathcal{P} & [\![a®\phi]\!] &= \{P \mid \exists P'.P \equiv (\nu a)P' \textit{ and } P' \in [\![\phi]\!]\} \\
[\![\neg\phi]\!] &= \mathcal{P} \setminus [\![\phi]\!] & [\![\exists x.\phi]\!] &= \bigcup_{a \in \mathcal{N}} [\![\phi\{^a/_x\}]\!] \\
[\![\phi_1 \vee \phi_2]\!] &= [\![\phi_1]\!] \cup [\![\phi_2]\!] & [\![\text{И}x.\phi]\!] &= \bigcup_{a \notin \texttt{ffn}(\phi)} ([\![\phi\{^a/_x\}]\!] \setminus \{P \mid a \in \texttt{fn}(P)\}) \\
[\![\texttt{void}]\!] &= \{P \mid P \equiv \texttt{0}\} & [\![a = b]\!] &= \begin{cases} \mathcal{P} & \textit{if } a = b \\ \emptyset & \textit{otherwise} \end{cases}
\end{array}$$

$$[\![\phi_1|\phi_2]\!] = \{P \mid \exists P_1, P_2.P \equiv P_1|P_2 \textit{ and } P_1 \in [\![\phi_1]\!] \textit{ and } P_2 \in [\![\phi_2]\!]\}$$
$$[\![\langle\lambda\rangle\phi]\!] = \{P \mid \exists Q.P \xrightarrow{\lambda} Q \textit{ and } Q \in [\![\phi]\!]\}$$

In addition to the usual abbreviations, we shall use the *hidden name* quantifier ($\texttt{H}x.\phi \equiv \text{И}x.x®\phi$) for existentially quantifying over restricted names.

Example 2.8 (a spatial property) *In our running example two component processes are ready to send distinct names over the same restricted channel after a synchronization. We may express that property by the formula*

$$\textbf{\textit{crash}} \;=\; \texttt{H}x.\exists y.\exists z.y \neq z \wedge \langle\tau\rangle(\langle\overline{x}y\rangle T \mid \langle\overline{x}z\rangle T)$$

Explicitly, the formula first quantifies over all the possible restricted names x. Then, it quantifies over all pairs of different names y, z such that after a synchronization the residual process can be decomposed into two components, sending names y and z, respectively, on the same channel x.

2.3 Some technical results

We state some technical lemmas. The first recalls Gabbay-Pitts Property [2].

Proposition 2.9 (Gabbay-Pitts) *Let P be a process, and let ϕ be a formula such that x is the only free variable. Then*

(i) $P \in [\![\text{И}x.\phi]\!]$ *iff* $P \in \bigcap_{a \notin (\texttt{fn}(P) \cup \texttt{ffn}(\phi))} [\![\phi\{^a/_x\}]\!]$.

(ii) $P \in [\![\exists x.\phi]\!]$ *iff* $P \in [\![\text{И}x.\phi]\!]$ *or* $P \in \bigcup_{a \in (\texttt{fn}(P) \cup \texttt{ffn}(\phi))} [\![\phi\{^a/_x\}]\!]$.

These properties make existential and fresh quantification decidable. Consider item *1*: By definition, the semantics of the fresh name quantifier is given in terms of the union over the substitution with those names appearing neither in P nor in ϕ; hence, fresh quantification $\text{И}x.\phi$ can be decided by substituting *any* fresh name for variable x in ϕ, and then checking the resulting formula.

The second lemma describes a normal form for processes. This result is used on Proposition 2.11: It concerns the revelation operator, stating that only a finite set of instances for the channel to be revealed needs to be checked.

Lemma 2.10 (normal forms) *Let P be a process. Then, P is structurally congruent to a process $(\nu a_1)\dots(\nu a_n)(P_1 \mid \dots \mid P_m)$, such that all a_i's are different names, all P_j's are prefixed processes, and $\{a_1,\dots,a_n\} \subseteq \bigcup_j \mathtt{fn}(P_j)$.*

We then denote a normal form as $(\nu N)\mathcal{Q}$, for $\mathcal{Q}$ a set of prefixed processes, since the order of restriction and parallel composition operators is immaterial.

Proposition 2.11 (revelation set) *Let P be a process and $a®\phi$ a closed formula. Then, $P \in [\![a®\phi]\!]$ iff $a \notin \mathtt{fn}(P)$ and either (i) $P \in [\![\phi]\!]$; or (ii) $(\nu a)(\nu N)\mathcal{Q}$ is a normal form of P and $(\nu N)\mathcal{Q} \in [\![\phi]\!]$.*

In order to verify if $a®\phi$ holds in a process P, the check that a is not free in P is firstly performed; then it suffices either to check again P, or to fix a normal form for P and check all those processes obtained by revealing any restricted name as a. This result will simplify the verification procedure, since the normal form has an obvious counterpart the graphical representation.

3 Graphs and their Ranked Version

We recall a few definitions concerning (labeled hyper-)graphs, and their *ranked* extension, referring to [7] for a detailed introduction and a comparison with the standard presentation [11]. In the following we assume a chosen signature (Σ, S), for Σ a set of operators (edge labels), and S a set of sorts (node labels), such that the *arity* of an operator in Σ is a pair (s,ω), for $\omega \in S^*$ and $s \in S$.

Definition 3.1 (graphs) *A* graph *d (over (Σ,S)) is a tuple $\langle N,E,l,s,t\rangle$, where N, E are the sets of* nodes *and* edges*; l is the pair of* labeling *functions $l_e : E \to \Sigma$, $l_n : N \to S$; $s : E \to N$ and $t : E \to N^*$ are the* source *and* target *functions; and such that for all $e \in E$, the arity of $l_e(e)$ is $(l_n(s(e)), l_n^*(t(e)))$.*

We let l_n^* stand for the extension of the function l_n to strings of nodes, and we use l as a shorthand for l_n and l_e. We also denote the components of a graph d by N_d, E_d, l_d, s_d and t_d, dropping the subscript whenever clear.

Definition 3.2 (graph morphisms) *Let d, d' be graphs. A* (graph) morphism *$f : d \to d'$ is a pair of functions $f_n : N_d \to N_{d'}$, $f_e : E_d \to E_{d'}$ that preserve the labeling, source and target functions.*

In order to define the process encoding, we need operations on graphs. The first step is to equip them with "handles" for interacting with an environment.

Definition 3.3 (ranked graphs) *Let d_r, d_v be graphs with no edges. A* (d_r,d_v)-ranked graph *(a* graph of rank (d_r,d_v)*) is a triple $G = \langle r,d,v\rangle$, for d a graph and $r : d_r \to d$, $v : d_v \to d$ the* root *and* variable *morphisms.*

Let G, G' be ranked graphs of the same rank. A ranked graph morphism *$f : G \to G'$ is a graph morphism between the underlying graphs that preserves the root and variable morphisms.*

We let $d_r \overset{r}{\Rightarrow} d \overset{v}{\Leftarrow} d_v$ denote the (d_r, d_v)-ranked graph d. With an abuse of notation, we sometimes refer to the image of the root and variable morphisms as roots and variables, respectively. More importantly, in the following we will often refer implicitly to a ranked graph as the representative of its isomorphism class, still using the same symbols to denote it and its components.

Definition 3.4 (sequential and parallel composition) *Let $G = d_r \overset{r}{\Rightarrow} d \overset{v}{\Leftarrow} d_i$ and $H = d_i \overset{r'}{\Rightarrow} d' \overset{v'}{\Leftarrow} d_v$ be ranked graphs. Then, their* sequential composition *is the ranked graph $G \circ H = d_r \overset{r''}{\Rightarrow} d'' \overset{v''}{\Leftarrow} d_v$, for d'' the disjoint union $d \uplus d'$, modulo the equivalence on nodes induced by $v(x) = r'(x)$ for all $x \in N_{d_i}$, and $r'' : d_r \to d''$, $v'' : d_v \to d''$ the uniquely induced arrows.*

Let $G = d_r \overset{r}{\Rightarrow} d \overset{v}{\Leftarrow} d_v$ and $H = d'_r \overset{r'}{\Rightarrow} d' \overset{v'}{\Leftarrow} d'_v$ be ranked graphs. Then, their parallel composition *is the ranked graph $G \otimes H = (d_r \cup d'_r) \overset{r''}{\Rightarrow} d'' \overset{v''}{\Leftarrow} (d_v \cup d'_v)$, for d'' the disjoint union $d \uplus d'$, modulo the equivalence on nodes induced by $r(x) = r'(x)$ for all $x \in N_{d_r} \cap N_{d'_r}$ and $v(y) = v'(y)$ for all $y \in N_{d_v} \cap N_{d'_v}$, and $r'' : d_r \cup d'_r \to d'', v'' : d_v \cup d'_v \to d''$ the uniquely induced arrows.*

The sequential composition $G \circ H$ is obtained by taking the disjoint union of the graphs underlying G and H, and gluing the variables of G with the corresponding roots of H. Similarly, the parallel composition $G \otimes H$ is obtained by taking the disjoint union of the graphs underlying G and H, and gluing the roots (variables) of G with the corresponding roots (variables) of H.

The two operations are concretely defined, but they are intended to act on isomorphic classes of ranked graphs (hence, with the same rank). In fact, the result is independent of the choice of the representative, up to isomorphism.

Example 3.5 (some graphs) *Fig. 1 depicts two ranked graphs (part of the encoding of our running example): Their sequential composition appears in Fig. 2 (left). Fig. 3 represents the parallel composition of the graphs in Fig. 2.*

The nodes in the domain of the root (variable) morphism are depicted as a vertical sequence on the left (right, resp.); the variable and root morphisms are represented by dotted arrows, directed from right-to-left and left-to-right, respectively. Edges are represented by a boxed label, from where arrows pointing to the target nodes leave, and to where the arrow from the source node arrives; the sequence of target nodes is usually the clockwise order of the start points of the tentacles, even if sometimes it is indicated by a numbering on the tentacles: For the edge of the leftmost graph of Fig. 1 the sequence is $(v(p), v(b), v(a))$.

The leftmost graph of Fig. 1 has rank $(\{p\}, \{p, a, b\})$, four nodes and one

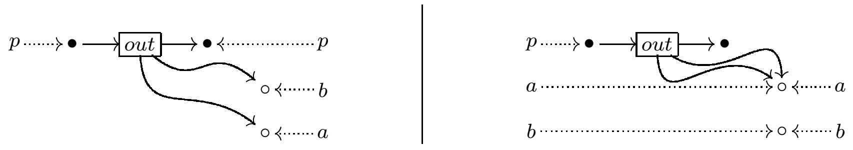

Fig. 1. Ranked graphs $out_{b,a}$ (left) and $[\![\bar{a}a]\!] \otimes id_{\{a,b\}}$ (right).

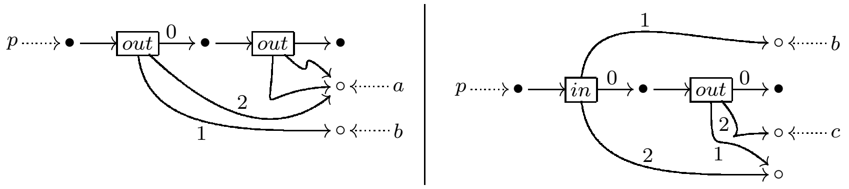

Fig. 2. Ranked graphs $out_{b,a} \circ ([\![\bar{a}a]\!] \otimes id_{\{a,b\}})$ (left) and $[\![b(d).\bar{d}c]\!]$ (right).

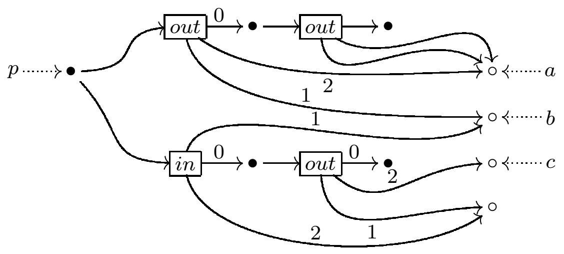

Fig. 3. The ranked graph $[\![\bar{b}a.\bar{a}a]\!] \otimes [\![b(d).\bar{d}c]\!]$.

edge labeled by out; the rightmost graph has rank $(\{p, a, b\}, \{a, b\})$*, four nodes and one edge labeled by out. For graphical convenience, nodes with different labels appearing in the underlying graph are also denoted differently.*

A *graph expression* is a term for the syntax containing ranked graphs as constants, and parallel and sequential composition as operators. An expression is *well-formed* if all the occurrences of these operators are defined for the rank of the sub-expressions, according to Definition 3.4: Its rank is inductively computed and its *value* is the graph obtained by evaluating its operators.

4 From Processes to Graphs

We now present the encoding of π-calculus processes into ranked graphs, based on the encoding presented in [8]. It is built out of a signature (Σ_π, S_π), and it preserves structural congruence. The set of sorts S_π is $\{s_p, s_n\}$: Intuitively, a graph reachable from a node of sort s_p corresponds to a process, while each node of sort s_n represents a name. The set Σ_π contains three operators: $\{in, out\}$ of sort $(s_p, s_p s_n s_n)$, and $\{\nu\}$ of sort (s_p, s_n). Clearly, the operators

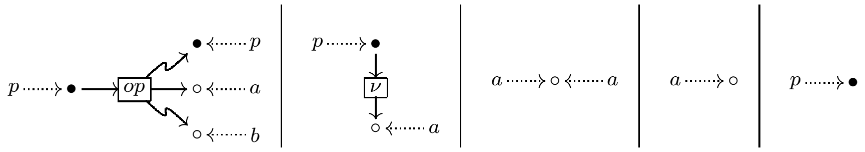

Fig. 4. Ranked graphs $op_{a,b}$ (for $op \in \{in, out\}$), ν_a, id_a, 0_a and 0_p.

in and *out* simulate the input and output prefixes, respectively; and operator ν stands for restriction. Furthermore, note that there is instead no explicit operator accounting for parallel composition.

The second step is the characterization of a class of graphs, such that all processes can be encoded into an expression containing only those graphs as constants, and parallel and sequential composition as binary operators. Let $p \notin \mathcal{N}$: Our choice is depicted in Fig. 4, for all $a, b \in \mathcal{N}$.

Finally, let id_Γ be a shorthand of $\bigotimes_{x\in\Gamma} id_x$, for a set Γ of names (since the ordering is immaterial). The encoding of processes into ranked graphs, mapping each finite process into a graph expression, is presented below.

Definition 4.1 (encoding for processes) *Let P be a process. The* encoding $\llbracket P \rrbracket$, *mapping a process P into a ranked graph, is defined by structural induction according to the following rules*

$$\begin{aligned}
\llbracket (\nu a)P \rrbracket &= \begin{cases} \llbracket P \rrbracket & \text{if } a \notin \mathtt{fn}(P) \\ (\llbracket P \rrbracket \otimes \nu_a) \circ (0_a \otimes id_{\mathtt{fn}(P)\setminus\{a\}}) & \text{otherwise} \end{cases} \\
\llbracket P \mid Q \rrbracket &= \llbracket P \rrbracket \otimes \llbracket Q \rrbracket \\
\llbracket 0 \rrbracket &= 0_p \\
\llbracket \bar{a}b.P \rrbracket &= out_{a,b} \circ (\llbracket P \rrbracket \otimes id_{\{a,b\}}) \\
\llbracket a(b).P \rrbracket &= in_{a,b} \circ (\llbracket P \rrbracket \otimes id_{\{a,b\}}) \circ (0_b \otimes id_{\mathtt{fn}(P)\setminus\{b\}})
\end{aligned}$$

Note the conditional rule for $(\nu a).P$: It is required for removing the occurrence of useless restriction operators, i.e., those binding a name not occurring in the process. The mapping is well-defined, since the resulting graph expression is well-formed, and the encoding $\llbracket P \rrbracket$ is a graph of rank $(\{p\}, \mathtt{fn}(P))$.

Example 4.2 (mapping a process) *In order to give some intuition about the intended meaning of the previous rules, we show the construction of the encoding for the process $\bar{b}a.\bar{a}a$ (a subprocess of our running example) whose*

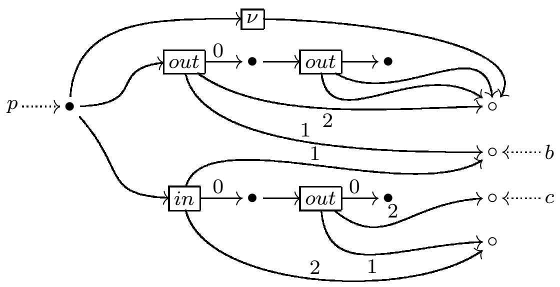

Fig. 5. The ranked graph $\llbracket(\nu a)\bar{b}a.\bar{a}a \mid b(d).\bar{d}c\rrbracket$.

graphical representation is depicted in Figure 2 (left)

$$\llbracket\bar{b}a.\bar{a}a\rrbracket \;=\; out_{b,a}\circ(\llbracket\bar{a}a\rrbracket\otimes id_{\{a,b\}}) \;=\; out_{b,a}\circ((out_{a,a}\circ(\mathsf{0}_p\otimes id_{\{a\}}))\otimes id_{\{a,b\}})$$

The denotation of $(\llbracket\bar{a}a\rrbracket\otimes id_{\{a,b\}})$ *coincides with* $(out_{a,a}\otimes id_{\{a,b\}})\circ(\mathsf{0}_p\otimes id_{\{a,b\}})$*, and the latter is clearly matched by its graphical representation. On the other hand, the graphical representation of* $\llbracket$***race***$\rrbracket$ *is depicted in Fig. 5.*

The mapping $\llbracket\cdot\rrbracket$ is not surjective, since there are graphs of rank $(\{p\},\Gamma)$ that are not (isomorphic to) the image of any process. Nevertheless, let us restrict our attention to processes verifying a mild syntactical condition, namely, forbidding the occurrences of input prefixes such as $a(a)$. Then, our encoding is sound and complete, as stated by the proposition below (adapted from [8]).

Proposition 4.3 *Let P, Q be processes. Then, $P\equiv Q$ iff $\llbracket P\rrbracket=\llbracket Q\rrbracket$.*

5 A Verification Algorithm

This section introduces an algorithm for verifying spatial formulae over the graphical representation of processes. It takes as input a closed formula ϕ to be verified and a ranked graph $G = r\Rightarrow d\Leftarrow v$ such that $G=\llbracket P\rrbracket$ for some process P, and returns a boolean, namely, *true* if $P\in[\![\phi]\!]$, *false* otherwise. It is defined by case induction on the formula to be verified, exploiting the structure of the graphical encoding. For any process P, the first call is *eval*$(\llbracket P\rrbracket,\phi)$.

Checking Booleans, Void and Name equality. The procedures to evaluate boolean formulae and name equality are self-explaining, and for checking `void` it suffices to determine whether d has no edge.

```
case $T$ return true;
case $\neg\phi$ return $\neg$eval$(G, \phi)$;
case $\phi_1 \vee \phi_2$ return eval$(G, \phi_1) \vee$ eval$(G, \phi_2)$;
case void  if $E_d = \emptyset$ then return true else return false;
case $a = b$ return $a = b$;
```

Checking Composition $(\phi_1 \mid \phi_2)$. The algorithm builds all pairs that correspond to a decomposition of the graph under consideration. These graphs are obtained by splitting the set of edges outgoing from the root that are not labeled with ν. This latter set is denoted by E in the pseudo-code below.

```
case $\phi_1|\phi_2$
    $E \leftarrow \{e \in E_d \mid s_d(e) = r(p)$ and $l_d(e) \neq \nu\}$;
    $R \leftarrow \{(e, n) \in E_d \times N_d \mid s_d(e) = r(p) \wedge l_d(e) = \nu \wedge t_d(e) = n\}$;
    foreach $E_1 \subseteq E$ do
        $G_1 \leftarrow$ sub-ranked graph of $G$ generated by $E_1$;
        $G_2 \leftarrow$ sub-ranked graph of $G$ generated by $E \setminus E_1$;
        foreach $(e, n) \in R$ do
            if $n \in d_1$ and $n \in d_2$ then continue outermost loop;
            if $n \in d_1$ then $d_1 \leftarrow d_1 \cup \{e\}$;
            if $n \in d_2$ then $d_2 \leftarrow d_2 \cup \{e\}$;
        if eval$(G_1, \phi_1) \wedge$ eval$(G_2, \phi_2)$ then return true;
    return false;
```

Intuitively, each edge in E corresponds to a prefixed sub-process of the process represented by G. However, not every graph decomposition corresponds to a correct process decomposition, and the reason for this is basically pinpointed by the structural axiom $(\nu a)(P \mid Q) = P \mid (\nu a)Q$ for $a \notin \mathtt{fn}(P)$. In different terms, after choosing a graph decomposition G_1 and G_2, it is necessary to consider all the names in the scope of a restriction operator placed on top of the process, and to check that each name occurs only in one of the two graphs. Hence, the procedure computes the set R of restricted nodes (together with the corresponding edges), and it checks for each restricted node n in R whether n belongs to both d_1 and d_2. If this is the case, then the chosen graph decomposition is not valid, since the name corresponding to $l_n(n)$ would occur free in both sub-processes. On the other hand, if n occurs in only one of the d_i's, the restriction edge is added to the corresponding ranked graph. After checking every restricted node in R, the algorithm recursively evaluates whether G_1 satisfies ϕ_1 and G_2 satisfies ϕ_2.

Sub-ranked graphs correspond to the usual sub-graphs reachable from a node (namely $r(p)$) and a set of adjacent edges, and they are built in linear complexity by a depth-first exploration.

Checking Name Quantification ($\exists x.\phi$). We exploit Proposition 2.9 and let x range on the nodes in $d_v \cup \mathtt{ffn}(\phi)$, since d_v represent the free names in the process encoded by G. If the result is negative in all such cases, we check if $\phi\{^a/_x\}$ holds for a fresh name a, relying on the case for fresh quantification.

```
case $\exists x.\phi$
    foreach $a \in d_v \cup \mathtt{ffn}(\phi)$ do  if eval$(G, \phi\{^a/_x\})$ then return true;
    return eval$(G, \text{И}x.\phi)$;
```

Checking Fresh Quantification ($\text{И}x.\phi$). Once more we exploit Proposition 2.9. We choose a name a neither in d_v nor in $\mathtt{ffn}(\phi)$, i.e., a name that is fresh for both the process and the formula. Then, we evaluate $\phi\{^a/_x\}$ on G.

```
case $\text{И}x.\phi$
    $a \leftarrow$ new name not in $d_v \cup \mathtt{ffn}(\phi)$;
    return  eval$(G, \phi\{^a/_x\})$;
```

Checking Commitment ($\langle\lambda\rangle\phi$). The algorithm distinguishes three different cases for λ. If λ is τ then the algorithm looks for an *out*-labeled edge and an *in*-labeled edge which operate on the same name node. Once such a pair is found a synchronization is simulated by building the residual graph, i.e., by coalescing the continuations of the two operators with the root of the process and the node being sent with the node being received. The procedure then removes the two involved edges, and it performs a garbage collection, deleting the useless occurrences of the restriction operator and all the isolated nodes (i.e., those nodes that appeared uniquely in the target sequence of the removed operators); finally, the algorithm checks whether ϕ holds in the resulting graph. Input and output commitments are computed similarly: Note that, according to the commitment semantics, the input action may receive a name b already occurring free in the process, hence the further control.

```
case $\langle\lambda\rangle\phi$
    if $\lambda = \tau$ then
        foreach $e1, e2 \in E_d$ with $l_d(e1) = out$ and $l_d(e2) = in$ do
            if $s_d(e1) = s_d(e2) = r(p)$ and $t_d(e1)[1] = t_d(e2)[1]$ then
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e1)[0]=t_d(e2)[0], t_d(e1)[2]=t_d(e2)[2]\}} \setminus \{e1, e2\}$;
                if eval($gc(G_1), \phi$) then return true;
    if $\lambda = \overline{a}b$ then
        foreach $e \in E_d$ with $l_d(e) = out$ do
            if $s_d(e) = r(p)$ and $t_d(e)[1] = v(a)$ and $t_d(e)[2] = v(b)$ then
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e)[0]\}} \setminus \{e\}$;
                if eval($gc(G_1), \phi$) then return true;
    if $\lambda = a[b]$ then
        foreach $e \in E_d$ with $l_d(e) = in$ do
            if $s_d(e) = r(p)$ and $t_d(e)[1] = v(a)$ then
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e)[0]\}} \setminus \{e\}$;
                if $b \in d_v$ then $d_1 \leftarrow d_{1\{v(b)=t_d(e)[2]\}}$;
                else $d_{v_1} \leftarrow d_v \cup \{b\}$; $v_1 \leftarrow v \cup \{b \mapsto t_d(e)[2]\}$;
                if eval($gc(G_1), \phi$) then return true;
    return false;
```

The garbage collection phase $gc(G_1)$ takes linear time, since it checks the connectivity for at most three nodes. It ensures that the resulting graph represents the encoding of the residual process after the commitment: To this end, garbage collection may also remove nodes from the variable graph.

Checking Revelation (a®ϕ). According to Proposition 2.11, the algorithm first checks whether a is free in the process represented by G, that is, if it belongs to d_v. If this fails, the algorithm then tries to check whether P satisfies ϕ. Finally, it reveals any restricted node as a: This is done by removing ν-labeled edges outgoing from the root of d and adding a to the variables.

```
case $a$®$\phi$
    if $a \in d_v$ then return false;
    if eval($G, \phi$) then return true;
    foreach $e \in E_d$ with $l_d(e) = \nu$ and $s_d(e) = r(p)$ do
        $G_1 \leftarrow G$;  $d_1 \leftarrow d \setminus \{e\}$;  $d_{v_1} \leftarrow d_v \cup \{a\}$;  $v_1 \leftarrow v \cup \{a \mapsto t_d(e)[0]\}$;
        if eval($G_1, \phi$) then return true;
    return false;
```

Example 5.1 *Does* **race** $= (\nu a)\overline{b}a.\overline{a}a \mid b(d).\overline{d}c$ *satisfy the property* **crash** $=$ $\mathtt{H}x.\exists y.\exists z.y \neq z \wedge \langle\tau\rangle(\langle\overline{x}y\rangle T \mid \langle\overline{x}z\rangle T)$? *The algorithm will first try and fix the variable x as a fresh name (say a) and try to reveal it as one of the*

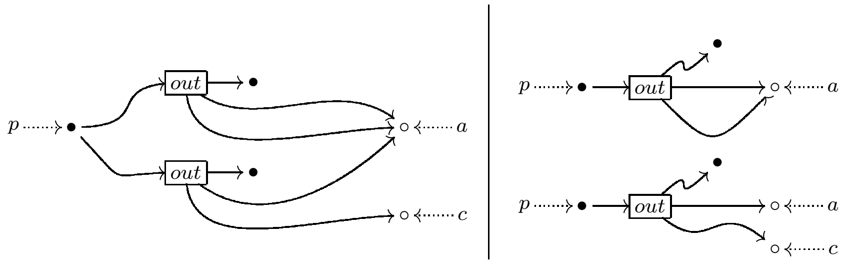

Fig. 6. The ranked graph $\llbracket \overline{a}a \mid \overline{a}c \rrbracket$ (left) and two sub-ranked graphs (right).

restricted names in $\llbracket$***race***$\rrbracket$*. Thus, x is revealed as the name a and the ranked graph depicted in Fig. 3 is constructed. Next, the algorithm will try and find a synchronization. The input and output edges, communicating on node b, are found and the residual graph is constructed: This latter is depicted in Fig. 6 (left). Then, the algorithm looks at every possible decomposition, which in this case (apart from the trivial ones where one component is void) are two, namely the two possibilities to form an ordered pair with the two out-labeled edges. The corresponding sub-ranked graphs are represented in Fig. 6 (right). In the decomposition formed with first the top graph and then the bottom graph the algorithm will successfully find the commitments sending the names a and c on channel a, thus returning* true.

We now state the correctness of the proposed evaluation procedure.

Theorem 5.2 (correct algorithms) *Let P be a process and ϕ a closed formula. Then, $P \in \llbracket \phi \rrbracket$ iff* $\mathrm{eval}(\llbracket P \rrbracket, \phi) = \mathrm{true}$.

Concerning the complexity of the algorithm, most of the operations rely on enumerating sets of edges or nodes and thus require polynomial time. The only exception is the verification of composition, where an exponential number of decompositions has to be considered.

6 Conclusions and Future Work

The paper introduced a graph-based technique for the verification of spatial properties of finite π-calculus specifications. We considered only the deterministic fragment of the calculus, in order to offer as simple a presentation as possible: The choice operator could be included with little effort.

Besides being intuition appealing, the graphical presentation offers canonical representatives for abstract processes, since two processes are structurally congruent iff they are mapped to the same ranked graph (up to isomorphism). The encoding has also a unique advantage with respect to most of the approaches to the graphical implementation of calculi with name mobility (such as Milner's *bigraphs* [10]): It allows for the reuse of standard graph transformation theory and tools for simulating the reduction semantics of the calculus [8].

The paper offers an effective mechanism for the verification of spatial properties, thus presenting a constructive alternative to the techniques proposed in [2]. In fact, even if no formal comparison is drawn, our algorithm on graphs exploits a "normal form" representation for processes that seems to be underlying also the model-checker proposed in [15]. Concerning efficiency, our worst case is the verification of parallel composition, since graph decomposition is exponential for general formulas. Again, no comparison can be traced to the results in [15], since the efficiency for their algorithms is not fully reported.

We are not aware of any other tool for model-checking formulas of spatial logics with respect to processes of the π-calculus. However, besides any consideration on the efficiency and usability of our algorithm, we believe that a main contribution of our paper is the further illustration of the usefulness of graphical techniques for the design and validation of concurrent systems: The claim is supported by a sound and complete encoding of spatial formulae into formulae of a temporal graph logic that is going to appear elsewhere.

The present proposal focuses on the finite fragment of the π-calculus. We are currently investigating how to generalize our approach in order to include recursive specifications, and thus considering the full spatial logic of [2]. The original graphical encoding of [8] already considers recursive processes, hence our efforts are going to be on extending the algorithm. Finally, we are planning an implementation of our approach, possibly by integrating it in existing tools for the analysis of graphically designed systems, such as [9,12].

References

[1] Baldan, P., A. Corradini and B. Köenig, *A static analysis technique for graph transformation systems*, in: K. Larsen and M. Nielsen, editors, *Concurrency Theory*, Lect. Notes in Comp. Sci. **2154** (2001), pp. 381–395.

[2] Caires, L., *Behavioral and spatial observations in a logic for the π-calculus*, in: I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, Lect. Notes in Comp. Sci. **2987** (2004), pp. 72–87.

[3] Caires, L. and L. Cardelli, *A spatial logic for concurrency (part I)*, Information and Computation **186** (2003), pp. 194–235.

[4] Caires, L. and L. Cardelli, *A spatial logic for concurrency – II*, Theor. Comp. Sci. **322** (2004), pp. 517–565.

[5] Cardelli, L., P. Gardner and G. Ghelli, *A spatial logic for querying graphs*, in: P. Widmayer *et alii*, editors, *Automata, Languages and Programming*, Lect. Notes in Comp. Sci. **2380** (2002), pp. 597–610.

[6] Cardelli, L., P. Gardner and G. Ghelli, *Manipulating trees with hidden labels*, in: A. Gordon, editor, *Foundations of Software Science and Computation Structures*, Lect. Notes in Comp. Sci. **2620** (2003), pp. 216–232.

[7] Corradini, A. and F. Gadducci, *An algebraic presentation of term graphs, via gs-monoidal categories*, Applied Categorical Structures **7** (1999), pp. 299–331.

[8] Gadducci, F., *Term graph rewriting and the π-calculus*, in: A. Ohori, editor, *Programming Languages and Semantics*, Lect. Notes in Comp. Sci. **2895** (2003), pp. 37–54.

[9] Kozioura, V. and B. König, *AUGUR: An unfolding-based verification tool for GTS*, available at http://www.fmi.uni-stuttgart.de/szs/tools/augur.

[10] Milner, R., *Bigraphical reactive systems*, in: K. Larsen and M. Nielsen, editors, *Concurrency Theory*, Lect. Notes in Comp. Sci. **2154** (2001), pp. 16–35.

[11] Plump, D., *Term graph rewriting*, in: H. Ehrig *et alii*, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, II: Applications, Languages and Tools*, World Scientific, 1999, pp. 3–61.

[12] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfaltz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, Lect. Notes in Comp. Sci. **3062** (2003), pp. 479–485, tool available at http://sourceforge.net/projects/groove.

[13] Rensink, A., *Towards model checking graph grammars*, in: M. Leuschel, S. Gruner and S. Lo Presti, editors, *Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, School of Electronics and Computer Science, University of Southampton (2003), pp. 150–160.

[14] Reynolds, J., *Separation logic: A logic for shared mutable data structures*, in: *Logic in Computer Science* (2002), pp. 55–74.

[15] Torres Vieira, H. and L. Caires, *The spatial logic model checker user's manual*, Technical Report TR-DI/FCT/UNL-03/2004, Faculty of Science and Technology, New University of Lisbon (2004).

[16] Varró, D., *Automated formal verification of visual modeling languages by model checking*, Software and Systems Modeling **3** (2004), pp. 85–113.