Graph Transformation for Verification and Concurrency

(GT-VC 2005)

Satellite workshop of Concur 2005

Reiko Heckel, University of Leicester
Barbara König, University of Stuttgart
Arend Rensink, University of Twente
(organizers)

San Francisco, 22 August 2005

Contents

I Invited presentation 1

Formal Modeling via Executable Specifications in Rewriting Logic. *Carolyn Talcott, SRI International* 1

II Full papers 3

Bigraphical Semantics of Higher-order Mobile Embedded Resources with Local Names. *Mikkel Bundgaard and Thomas Hildebrandt* 3

Distributed Graph Traversals by Relabeling Systems with Applications. *Bilel Derbel and Mohamed Mosbah* 25

Graphical Verification of a Spatial Logic for the π-calculus. *Fabio Gadducci and Alberto Lluch-Lafuente* 41

Hoare vs Milner: comparing synchronizations in a graphical framework with mobility. *Ivan Lanese and Ugo Montanari* 59

III Work-in-progress 75

Extending C for Checking Shape Safety. *Mike Dodds and Detlef Plump* 75

Towards Attributed Graphs in Groove. *Harmen Kastenberg* 91

An integration tool for Stochastic Graph Transformation. *Sebastian Menge and Georgios Lajios* 99

Semi-local model of computations on graphs to break the local symmetry. *Dobieslaw Wroblewski* 105

Formal Modeling via Executable Specifications in Rewriting Logic

Carolyn Talcott, SRI International

SRI International

Abstract

The talk will begin with a brief introduction to Rewriting Logic and use of the Maude language. A case study based on modeling security aspects a remote service toolkit will be used to illustrate the approach to formal modeling and analysis in more detail.

(This page intentionally left blank)

Bigraphical Semantics of Higher-Order Mobile Embedded Resources with Local Names [1]

Mikkel Bundgaard [2] Thomas Hildebrandt [3]

Department of Theoretical Computer Science
IT University of Copenhagen
Denmark

Abstract

Bigraphs have been introduced with the aim to provide a topographical meta-model for mobile, distributed agents that can manipulate their own linkages and nested locations, generalising both characteristics of the π-calculus and the Mobile Ambients calculus. We give the first bigraphical presentation of a non-linear, higher-order process calculus with nested locations, non-linear active process mobility, and local names, the calculus of *Higher-Order Mobile Embedded Resources (Homer)*. The presentation is based on Milner's recent presentation of the λ-calculus in local bigraphs. The combination of non-linear active process mobility and local names requires a new definition of parametric reaction rules and a representation of the location of names. We suggest *localised bigraphs* as a generalisation of local bigraphs in which links can be further localised.

Key words: bigraphs, local names, non-linear process mobility

Introduction

The theory of *Bigraphical Reactive Systems* (BRS) [13] has been proposed as a topographical meta-model for mobile, distributed agents that can manipulate their own linkages and nested locations. A bigraph consists of two structures: the *place graph* and the *link graph*. The *place graph* is a tuple of unordered trees that represents the topology of the system. The roots of the trees are referred to as *regions* and the nodes are often referred to as *places* and may represent locations or other process constructors such as e.g. action prefixing. Some of the leaves may be *sites* (also referred to as holes) making the bigraph a (multi-hole) context. Each non-site place is typed with a *control* and has

[1] Funded by the Danish Research Agency grant: 2059-03-0031 (LaCoMoCo) and IT-Vest networking universities: National Teaching Network: Model-Based Design for Concurrency.
[2] Email: `mikkelbu@itu.dk`
[3] Email: `hilde@itu.dk`

a number of *ports* linked together by the link graph. The *link graph* represents the connectivity in the system, corresponding to shared names in the π-calculus. Free names are represented by links connected to a set of names in the (outer) *interface* of the bigraph.

In so-called *pure* bigraphs, the place and link graph can be considered to be orthogonal structures, since the nesting of the places and the connections of the links have no interrelationship. Pure bigraphs are sufficient to represent calculi such as the pure Mobile Ambient calculus. The orthogonality breaks when we move to so-called *binding* and *local* bigraphs. Binding bigraphs were introduced in [12] to capture the notions of binding and scope of names as found in the π-calculus. In binding bigraphs we allow for a node to have *binding ports*, and require that any other port linked to the same link as a binding port to be within the node of the binding port. In [15], Milner refines the definition of binding bigraphs into *local bigraphs*. In local bigraphs, the free names (i.e. names in the interface) are all explicitly located at the regions of the bigraph, the same name possibly located at several regions. Correspondingly, holes (i.e. sites) are explicitly annotated by a set of names connected to links. Local bigraphs are used to facilitate the presentation of the λ-calculus in [16], which demonstrates how higher-order processes (process passing) can be presented in the bigraphical framework using explicit substitutions.

In the present paper we give the first bigraphical presentation of the combination of active processes in nested locations as present in the Mobile Ambients, non-linear higher-order process passing (by explicit substitution) as present in the λ-calculus and local names as present in the π-calculus. It turns out that the combination of non-linear, active process mobility and local names needs special care, i.e. we can not simply combine the previous presentations of the Mobile Ambients, the λ-calculus, and the π-calculus.

We take as our starting point the calculus of (asynchronous) Higher-Order Mobile Embedded Resources (Homer) [9]. Homer is a pure higher-order calculus inspired by prior higher-order calculi such as Plain CHOCS [19] and HOπ [18], and can be regarded as an extension of the λ-calculus to contain nested, active locations and concurrent synchronisation over (nested) named channels. It is also a natural subclass of bigraphs for studying active, mobile processes in nested locations. Basically, asynchronous Homer has two constructors for located resources $\overline{\delta}\langle r\rangle$ (passive) and $\delta[r]$ (active) where δ is a sequence of names representing the address of the resource. These two constructors correspond respectively to a passive and an active bigraph control with ports connected to the links δ. The interactions are controlled by two corresponding constructors for moving located resources $\delta(x)\,.\,p$ (receive) and $\overline{\delta}(x)\,.\,p$ (take), denoting respectively the usual input-prefixed process waiting to receive a (passive) process on the channel δ, and an input action for taking an *active* process from location δ, in both cases substituting the moved resource in for x in p. We allow interactions with arbitrarily deeply nested, active processes by simply composing addresses. In the example below we send the resource r down to the nested address ab (composed of a and b), and it is received at the address b residing in the location a

$$\overline{ab}\langle r\rangle \mid a[b(x)\,.\,q \mid q'] \searrow a[\, q[r/x] \mid q'] \ . \tag{1}$$

Dually, we can also take up resources from nested locations as in

$$a[b[r] \mid p] \mid \overline{ab}(x)\,.\,q \searrow a[p] \mid q[r/x] \ . \tag{2}$$

As usual, we let $(n)p$ denote a process p in which the name n is local. With local names we also need to handle scope extension. For most of the process constructors scope extension is as expected, but when a resource is moved it may be necessary to extend the scope of a name through the boundary of a location, e.g. if the resource r contains the name n free, we will expect the reaction

$$a[(n)(b[r] \mid p)] \mid \overline{ab}(x)\,.\,q \searrow (n)(a[p] \mid q[r/x]) \ , \tag{3}$$

where we have *vertically*, through the location boundary, extended the scope of n to cover all possible occurrences of the name n. In the Mobile Ambients calculus vertical scope extension is performed in the structural congruence (along with the usual scope extension)

$$m[(n)p] \equiv (n)m[p] \ \text{, if } n \neq m \ . \tag{4}$$

However, as also discovered in [6] this rule is not sound when mobile processes may be copied. There exists several solutions to this problem, all of them exclude the vertical scope extension in the structural congruence (4), and instead extend the scope in the reaction relation. This extension is either done *eagerly*, meaning that we always extend the scope, or *if and only if* the name n is free in r. In Homer we have chosen the latter solution, which corresponds to the usual semantics of e.g. HOπ. Combined with nested locations it has the consequence that a context can test if a name is free in a process, and so for any non-trivial congruence related processes must have the same set of free names (see, e.g., [9] for a detailed discussion). It is sometimes useful, however, to be able to abstract from free, but non-accessible names, as e.g. in the *perfect firewall equation* $(n)(n[p]) \approx \mathbf{0}$, stating that the behaviour of a computing resource at a local location is unobservable. To facilitate this we type processes explicitly with a set of names $\tilde{n}$ containing the free names. The *typed* perfect firewall equation then becomes $(n)(n[p]) : \tilde{n} \approx \mathbf{0} : \tilde{n}$ for $fn(p)\backslash\{n\} \subseteq \tilde{n}$. Interestingly, it turns out that for this equation to hold we also need to explicitly annotate all located sub-resources with a type, which is done by extending the syntax to $\overline{\delta}\langle r\rangle_{\tilde{n}}$ and $\delta[r]_{\tilde{n}}$.

Related Work

The Homer calculus were introduced in [9] together with labelled transition bisimulation congruences, and an encoding in Homer of the synchronous π-calculus without summation was presented in [3,4]. Composite names in send and receive prefixes are also found in the π-calculus with polyadic synchronisation [5], however, the dual prefixes for active processes are not considered.

In [13,12] Jensen and Milner set up the basic theory of BRSs and exhibit a bigraphical presentation of the asynchronous π-calculus Aπ and prove that the derived LTS and its bisimilarity match closely the traditional LTS and bisimilarity of Aπ. Milner gives in [14] a bigraphical presentation of condition-event Petri nets and Jensen gives in his forthcoming thesis a presentation of the Mobile Ambient calculus [11]. Milner has refined the theory of binding bigraphs [15,16], to give a bigraphical presentation of a λ-calculus with explicit substitutions. Several aspects of the current paper are inspired by this presentation. Besides bigraphs there exist several graphical formalisms suitable for presenting calculi for concurrency and mobility: solo diagrams, synchronized hyperedge replacement, tile systems etc., see e.g. [2] for references.

Explicit substitutions have been widely applied in the setting of functional programming languages, primarily to bridge the gap between the abstract definition of a programming language and the concrete implementation. In the seminal work of Abadi et al. [1] on $\lambda\sigma$, a λ-calculus with explicit substitutions, the substitutions are propagated throughout the term and applied locally. The approach chosen in this paper differs from this solution, in the same way as Milner's λ-calculus did, since we also perform the substitution 'at a distance'. Explicit substitutions have also appeared in process calculi for concurrency and mobility. In particular the π-calculus has been augmented with explicit substitutions in several variants, e.g. using a global environment for the substitutions [8] or using De Bruijn indices and handling the name instantiation using a term rewrite system [10].

The paper is structured as follows: In Sec. 1.1 we briefly review the main concepts of local bigraphs, and in Sec. 1.2 we present the calculus Homerσ. Sec. 2 contains the presentation of Homerσ as a BRS, ending with the suggestion of localised bigraphs as a generalisation of local bigraphs in which links can be further localised.

1 Preliminaries

In this section we first briefly recall the main concepts of the theory of local bigraphs [15], and give a new definition of parametric reaction rules. We then present the asynchronous variant of the calculus Homer introduced in [9], but extended with explicit substitutions to present the higher-order process passing of Homer in the bigraphical framework.

1.1 Local Bigraphs

We refer the reader to [13] for the basic static and dynamic theory of (pure and binding) bigraphs and [15] and [16] for the remaining details about local bigraphs. In this paper we will primarily use a simple term language, introduced in the above mentioned papers, instead of the graphical representation of bigraphs. The term language consists of the following constructors: $h \mid\mid g$ and $h \mid g$ are the parallel product and prime parallel product of two bigraphs h and g, respectively. Whereas the prime parallel product merges the regions of two single-region (prime) bigraphs, the parallel product juxtaposes the regions. The closure constructor $/n \circ g$ is the bigraph g, where we have removed the outer name n by replacing the name with an edge in g.

The *outer face* of a local bigraph is a pair $\langle m, \overrightarrow{X} \rangle$, where m is the number of regions and $\overrightarrow{X}$ is a vector of length m, such that X_i is the set of local names attached to the $i'th$ region. Similarly, the *inner face* is a pair $\langle n, \overrightarrow{Y} \rangle$ where n is the number of sites, $|\overrightarrow{Y}| = n$ and Y_i is the local names attached to the $i'th$ site. We can compose two bigraphs H and G, if the outer face of G and inner face of H matches, resulting in the bigraph $H \circ G$, where the content of the regions of G have been inserted into the respective sites of H, and the links of corresponding local names have been fused together.

A bigraph *signature* $\mathcal{K}$ is a set of controls and provides for each control K a pair of finite ordinals, the number of binding and free ports, the *binding arity* h and the *free arity* k, written $K : h \to k$. It also determines which controls are atomic, and which of the non-atomic controls are active.

A *ground reaction rule* is a pair (r, r') of ground bigraphs (bigraphs with no holes) with the same outer face. Given a set of ground rules, the *reaction relation*, $\longrightarrow\!\!\!\triangleright$, is the least relation such that $D \circ r \longrightarrow\!\!\!\triangleright D \circ r'$ for each active context D and each ground rule (r, r'). Parametric reaction rules allow for the rules to contain parameters, that can be replicated, discarded, or just moved. A *parametric reaction rule* has a *redex* R and *reactum* R', and takes the form $(R : I \to K, R' : I' \to K, \eta)$, with inner faces $I = \langle m, \overrightarrow{X} \rangle$ and $I' = \langle m', \overrightarrow{X}' \rangle$, and $\eta : m' \to m$ is a map of ordinals, inducing the instantiation $\overline{\eta}$, defined below. For every parameter $d : I$ the parametric reaction rule generates a ground reaction rule $(R \circ d, R' \circ \overline{\eta}(d))$. Differently from the original definition in [15], we require that all outer names of a parameter are specified explicitly by the parametric reaction rule, to ensure that we handle scope extension properly. The *instantiation* maps a parameter for the redex to a parameter for the reactum and allows for the rules to replicate some of their parameters and discard others. More precisely, a ground bigraph $a : \langle m, \overrightarrow{X} \rangle$ with no closed links crossing regions can be factorised uniquely into prime bigraphs as $a = c_0 \mid\mid \cdots \mid\mid c_{m-1}$, with $c_i : X_i$. For a map $\eta : m' \to m$ we then define the instantiation $\overline{\eta}$ as

$$\overline{\eta}(a) : \langle m', \overrightarrow{X'} \rangle \stackrel{def}{=} c_{\eta(0)} \mid\mid \cdots \mid\mid c_{\eta(m'-1)} \text{ , where } X'_j \stackrel{def}{=} X_{\eta(j)} \text{ for all } j \in m'.$$

1.2 Higher-Order Mobile Embedded Resources

We assume an infinite set of *names* $\mathcal{N}$ ranged over by m and n, and let $\tilde{n}$ range over finite sets of names. We let γ range over (possibly empty) sequences

$$\frac{}{\tilde{x} \vdash \mathbf{0} : \tilde{n}} \qquad \frac{\tilde{x} \vdash p : \tilde{n}_1 \qquad \tilde{x} \vdash q : \tilde{n}_2}{\tilde{x} \vdash p \mid q : \tilde{n}_1 \cup \tilde{n}_2} \qquad \frac{}{\tilde{x} \vdash (-)_{\tilde{n}} : \tilde{n}}$$

$$\frac{}{\tilde{x}x \vdash x : \tilde{n}} \qquad \frac{\tilde{x}x \vdash p : \tilde{n} \qquad \vdash q : \tilde{m}}{\tilde{x} \vdash p[x := q : \tilde{m}] : \tilde{n} \cup \tilde{m}} \qquad \frac{\tilde{x} \vdash p : \tilde{n}n}{\tilde{x} \vdash (n)p : \tilde{n}}$$

$$\frac{\tilde{x}x \vdash p : \tilde{n}}{\tilde{x} \vdash \varphi(x) \,.\, p : \tilde{n} \cup fn(\varphi)} \qquad \frac{\tilde{x} \vdash r : \tilde{m}}{\tilde{x} \vdash \varphi[\![r]\!]_{\tilde{m}} : \tilde{m} \cup fn(\varphi)}$$

Table 1
Typing rules for Homerσ

of names, and let δ range over non-empty sequences of names, referred to as *addresses* and let $|\delta|$ denote the length of the address δ, also we let $\varphi ::= \delta \mid \overline{\delta}$. We assume an infinite set of *process variables* $\mathcal{V}$ ranged over by x and y, and let $\tilde{x}$ range over finite sets of variables. The set $\mathcal{P}$ of *process expressions* for the calculus Homerσ of (asynchronous) Higher-Order Mobile Embedded Resources with explicit substitutions is then defined as follows

$$\begin{array}{llllllll}
\textit{Processes:} & p,q,r & ::= & \mathbf{0} & \pi \,.\, p & p \mid q & (n)p & \\
 & & & p[x := q : \tilde{n}] & x & \overline{\delta}\langle r\rangle_{\tilde{n}} & \delta[r]_{\tilde{n}} & \\
\textit{Prefixes:} & \pi & ::= & \delta(x) & \overline{\delta}(x) & & &
\end{array}$$

The complementary actions $\overline{\delta}\langle r\rangle_{\tilde{n}}$ and $\delta(x)$ are the usual prefixes of Plain CHOCS [19] or HOπ, except that we allow sequences of names as addresses instead of only a name, and we explicit type the resource r. As described in the introduction, the actions $\delta[r]_{\tilde{n}}$ and $\overline{\delta}(x)$ are responsible for adding active process mobility to the calculus. We write $\varphi[\![r]\!]_{\tilde{n}}$ for $\delta[r]_{\tilde{n}}$ or $\overline{\delta}\langle r\rangle_{\tilde{n}}$. The process $p[x := q : \tilde{n}]$ is an explicit syntactic substitution, representing the process p in a context that can substitute q (of type $\tilde{n}$) in for x. The typing rules to be defined below ensures that q is closed and that the free names of q are contained in $\tilde{n}$. As usual, we let the restriction operator (n) bind the name n, and let the prefixes $\varphi(x)$ and $p[x := q : \tilde{n}]$ bind the variable x.

Contexts $\mathcal{C}$ are defined by taking the grammar for processes and augmenting it with a symbol called a *hole*, written $(-)_{\tilde{n}}$. Note that holes are typed, only a process with type $\tilde{n}$ can be placed in a hole $(-)_{\tilde{n}}$.

We define the valid typing judgements of the form $\tilde{x} \vdash p : \tilde{n}$ inductively by the rules in Tab. 1. From now on we will only consider well-typed processes. Note that a process p is well-typed with respect to a finite set of variables $\tilde{x}$ and names $\tilde{n}$, written $\tilde{x} \vdash p : \tilde{n}$, if and only if the free names (variables) of p are included in the set $\tilde{n}$ ($\tilde{x}$), and for every sub-term $\varphi[\![r]\!]_{\tilde{m}}$ and $q[x := r : \tilde{m}]$ in p we have that r can be typed with the type $\tilde{m}$. We define the free names and free variables as usual with the addition that the free names of $\varphi[\![r]\!]_{\tilde{n}}$ and $p[x := r : \tilde{n}]$ are defined as $fn(\varphi) \cup \tilde{n}$ and $fn(p) \cup \tilde{n}$, respectively.

We say that a process with no free variables is *closed* and let $\mathcal{P}\sigma_c$ denote the set of closed processes. We let $\mathcal{P}\sigma_{/\alpha}$ (and $\mathcal{P}\sigma_{c/\alpha}$) denote the set of α-equivalence classes of (closed) process expressions, and we consider processes up to α-equivalence. We omit trailing $\mathbf{0}$s, write $\vdash p : \tilde{n}$ for $\emptyset \vdash p : \tilde{n}$, and let prefixing and restriction be right associative and bind stronger than explicit substitution and let explicit substitution bind stronger than parallel composition. For a set of names $\tilde{n} = \{n_1, \ldots, n_k\}$ we let $(\tilde{n})p$ denote $(n_1) \cdots (n_k)p$. We write $\tilde{m}\tilde{n}$ for $\tilde{m} \cup \tilde{n}$, always assuming $\tilde{m} \cap \tilde{n} = \emptyset$.

1.3 Reaction Semantics

We provide Homerσ with a reaction semantics defined using structural congruence, evaluation contexts, and reaction rules. A binary relation $\mathcal{R}$ on well-typed processes is called *well-typed* if and only if it relates processes p and q with the same type $\tilde{n}$ $(\tilde{x})$, written $\tilde{x} \vdash p \mathrel{\mathcal{R}} q : \tilde{n}$. We will only consider well-typed relations in this paper. A relation $\mathcal{R}$ is called a *congruence* if and only if it is a well-typed equivalence relation and it satisfies that $\tilde{x} \vdash p \mathrel{\mathcal{R}} q : \tilde{n}$ implies $\tilde{x}' \vdash \mathcal{C}(p) \mathrel{\mathcal{R}} \mathcal{C}(q) : \tilde{n}'$ for all contexts $\mathcal{C}$.

Structural congruence $\equiv_\sigma$ is defined as the least congruence on well-typed processes relating $\tilde{x} \vdash p \equiv_\sigma q : \tilde{n}$, if $\tilde{x} \vdash p : \tilde{n}$, $\tilde{x} \vdash q : \tilde{n}$, and $p \equiv_\sigma q$ can be derived using the following rules

$$p \mid \mathbf{0} \equiv_\sigma p \qquad (p \mid p') \mid p'' \equiv_\sigma p \mid (p' \mid p'') \qquad p \mid q \equiv_\sigma q \mid p$$
$$(n)p \mid q \equiv_\sigma (n)(p \mid q), \text{ if } n \notin fn(q) \qquad \pi \,.\, (n)p \equiv_\sigma (n)\pi \,.\, p, \text{ if } n \notin fn(\pi)$$
$$(n)(m)p \equiv_\sigma (m)(n)p \qquad (n)p \equiv_\sigma p, \text{ if } n \notin fn(p)$$
$$(n)(p[x := r : \tilde{n}]) \equiv_\sigma (n)p[x := r : \tilde{n}], \text{ if } n \notin \tilde{n}$$

As Homerσ permits reactions arbitrarily deep in the location hierarchy and also permits reactions between a process and an arbitrarily deeply nested sub-resource, we define the concepts of evaluation and path contexts. An *evaluation context* $\mathcal{E}$ is a context with no free variables and whose hole is not guarded by a prefix, nor does it occur as the object of a send constructor

$$\mathcal{E} ::= (-)_{\tilde{n}} \;\mid\; \mathcal{E} \mid p \;\mid\; (n)\mathcal{E} \;\mid\; \delta[\mathcal{E}]_{\tilde{n}}, \text{ for } p \in \mathcal{P}\sigma_c \;.$$

We define a family of multi-hole *path contexts* $\mathcal{C}^{\tilde{n}}_{\gamma}$, indexed by a path address $\gamma \in \mathcal{N}^*$ and a set of names $\tilde{n}$, inductively in $\tilde{n}$ and γ

$$\mathcal{C}^{\emptyset}_{\epsilon} ::= (-)_{\tilde{n}} \quad \text{and} \quad \mathcal{C}^{\tilde{n}\tilde{m}}_{\delta\gamma} ::= \delta[(\tilde{n})(\mathcal{C}^{\tilde{m}}_{\gamma} \mid (-)_{\tilde{n}'})]_{\tilde{m}'} \;,$$

whenever $\tilde{n} \cap \gamma = \emptyset$. Note that the evaluation context $\delta[\mathcal{E}]_{\tilde{n}}$ enables internal reactions of active resources, and that for a path context $\mathcal{C}^{\tilde{n}}_{\gamma}$, the path address γ indicates the path under which the context's hole is found, and the set of names $\tilde{n}$ indicates the bound names of the hole. The side condition in

$$
\begin{array}{ll}
(\mathit{send}\sigma) & \vdash \overline{\gamma\delta}\langle r\rangle_{\tilde{n}} \mid \mathcal{C}^{\tilde{m}}_{\gamma}(\delta(x)\,.\,p, \overrightarrow{p}) \searrow_{\sigma} \tilde{n} \odot \mathcal{C}^{\tilde{m}}_{\gamma}(p[x := r : \tilde{n}], \overrightarrow{p}) : \tilde{n}' \ , \\
& \text{if } \tilde{m} \cap (\delta \cup \tilde{n}) = \emptyset \\
(\mathit{take}\sigma) & \vdash \mathcal{C}^{\tilde{m}}_{\gamma}(\delta[r]_{\tilde{n}}, \overrightarrow{p}) \mid \overline{\gamma\delta}(x)\,.\,p \searrow_{\sigma} (\tilde{n} \cap \tilde{m})\big(\tilde{n} \odot \mathcal{C}^{\tilde{m}}_{\gamma}(\mathbf{0}, \overrightarrow{p}) \mid p[x := r : \tilde{n}]\big) : \tilde{n}' \ , \\
& \text{if } \tilde{m} \cap (\delta \cup \mathit{fn}(p)) = \emptyset \\
(\mathit{apply}\sigma) & \vdash \mathcal{C}(x)[x := r : \tilde{n}] \searrow_{\sigma} \tilde{n} \odot \mathcal{C}(r)[x := r : \tilde{n}] : \tilde{n}' \ , \\
& \text{if } \mathcal{C} \text{ does not bind } x \text{ or the names in } \tilde{n} \\
(\mathit{garbage}\sigma) & \vdash p[x := q : \tilde{n}] \searrow_{\sigma} p : \tilde{n}' \ , \ \text{if } x \notin \mathit{fv}(p)
\end{array}
$$

Table 2
Reaction rules for Homerσ

the definition of path contexts ensures that none of the names in the path address of the hole are bound. The bound names ($\tilde{n}$) in the definition of path contexts are needed since the structural congruence does not permit vertical scope extension, as described in the introduction.

We handle the vertical scope extension and the update of the type annotation of a location using an *open* operator, defined on path contexts. We define an *open* operator on path contexts $\tilde{m} \odot \mathcal{C}^{\tilde{n}}_{\gamma}$ inductively by:

$$
\begin{aligned}
\tilde{m} \odot \mathcal{C}^{\emptyset}_{\epsilon} &= \mathcal{C}^{\emptyset}_{\epsilon} \\
\tilde{m} \odot \mathcal{C}^{\tilde{n}_1\tilde{n}_2}_{\delta\gamma} &= \delta[(\tilde{n}_1 \setminus \tilde{m})(\tilde{m} \odot \mathcal{C}^{\tilde{n}_2}_{\gamma} \mid (-)_{\tilde{n}'})]_{\tilde{m}' \cup \tilde{m}} \ ,
\end{aligned}
$$

if $\mathcal{C}^{\tilde{n}_1\tilde{n}_2}_{\delta\gamma} = \delta[(\tilde{n}_1)(\mathcal{C}^{\tilde{n}_2}_{\gamma} \mid (-)_{\tilde{n}'})]_{\tilde{m}'}$ and $\tilde{m} \cap \tilde{n}_1\tilde{n}_2 \cap \mathit{fn}(\mathcal{C}^{\tilde{n}_1\tilde{n}_2}_{\delta\gamma}) = \emptyset$. Intuitively, the open operator in $\tilde{m} \odot \mathcal{C}^{\tilde{n}}_{\gamma}$ removes the names $\tilde{m}$ from the bound names of the hole and adds them to the type annotation of the locations that are part of the address path. When applied in the reaction rule, the latter condition of the open operator can always be met by α-conversion, the condition ensures us that we can extend the scope by using the open operator and place the restriction at top level, without any name captures.

As for the structural congruence, we define the reaction relation for Homerσ, written $\searrow_{\sigma}$, as the least well-typed relation on well-typed closed processes satisfying the rules in Tab. 2 and closed under all evaluation contexts $\mathcal{E}$ and structural congruence.

The ($\mathit{send}\sigma$) rule expresses how a passive resource r is sent (down) to the (sub) location γ, where it is received at the address δ. The side conditions ensure the location path is not bound in the context and that no free names of r get bound during movement. The open operator only extends the type annotation of the locations constituting the location path and does not lift any restrictions. The ($\mathit{take}\sigma$) rule captures that a computing resource r is taken from the (sub) location γ, where it is running at the address δ. Again, the side conditions ensure that the location path is not bound in the context, and that no free name is bound, when we lift the restriction. It is possible that the open operator both lifts restrictions and extends the type annotation of the locations. The rule (*apply*σ) replaces one occurrence of the variable (arbitrarily deep in the context) with the content of the explicit substitution. Note that we overload the use of $\odot$ in (*apply*σ), applying the operator to a general context and not only a path context. However, the result of the operator is the same, it extends the type annotation of all the locations (and send constructors) containing this occurrence of the variable. The latter condition of the rule can always be satisfied using α-conversion of the context. The (*garbage*σ) rule is responsible for garbage collecting superfluous substitutions.

The types ensure that no names can disappear from the free names of a location or from top-level during reaction. Locations or send constructors in the process that receives a resource r can get their type annotation extended by the type of r that do not already appear in their annotation.

2 Bigraphical Semantics of Homerσ

In this section we give the bigraphical presentation of Homerσ as the BRS $\mathcal{H}$omerσ. First, we present the signature for $\mathcal{H}$omerσ, and give a fully compositional translation of Homerσ-terms into bigraphs. Second, we translate the path contexts and the reaction relation. An important criteria for the presentation is to show that there is a static and operational correspondence between Homerσ and its presentation as a BRS, meaning that structural congruence of Homerσ corresponds to graph isomorphism in the bigraphical presentation, and that reactions match.

The signature has controls **rece** and **take** representing the two input prefixes, and **send**, and **loca** representing the two kinds (passive and active) of located resources. Controls **var**, **sub**, and **def** represent a variable and the constructs for explicit substitutions, respectively. Finally, the signature also has atomic controls **tname** (abbreviation for **typename**) and **ann** (abbreviation for **annotation**) to represent the explicit type annotation of resource and send constructors. We will discuss this in more detail after having presented the reaction rules in the bigraphical framework. Note that since path addresses are represented with one port for each element in the sequence, we have an infinite family of controls indexed by the length of the address. In total, the signature for $\mathcal{H}$omerσ is defined as follows.

- The controls **var**: $0 \to 1$ and **tname**: $0 \to 1$ are atomic
- The families of controls: **rece**$_{|\delta|}$: $1 \to |\delta|$, **take**$_{|\delta|}$: $1 \to |\delta|$, and **send**$_{|\delta|}$: $0 \to |\delta|$ are all inactive
- The family of controls **loca**$_{|\delta|}$: $0 \to |\delta|$ is active
- The controls **def**: $0 \to 1$, **sub**: $1 \to 0$, and **ann**: $0 \to 0$ are inactive

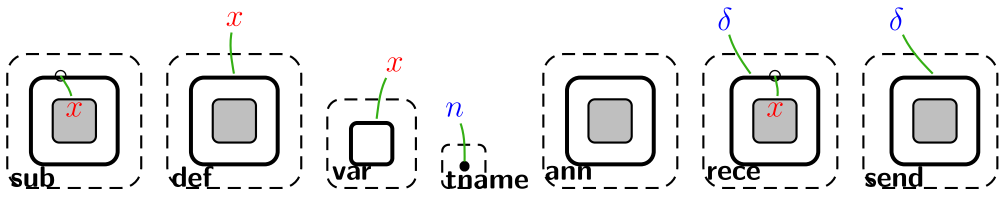

Figure 1. Ions and atoms for Homerσ

Note that we have no controls for restriction and the inactive process. This is to ensure the static correspondence, as stated in Thm. 2.2.

In Fig. 1 we depict the ions and the atoms used in the translation, we have left out the controls **take** and **loca** as they are similar to **rece** and **send**, respectively. We have chosen to depict the control **tname** as just a dot, •, in order to be able to distinguish graphically between **tname** and **var** controls. Following the convention of Milner [16], we write $\mathbf{var}_x$ and $\mathbf{tname}_n$ for the atoms, and we denote the ions as follows

$$\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_Z \quad \mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_Z \quad \mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_Z \quad \mathbf{rece}_{\delta(x)} \mathbin{\overline{\oplus}} \mathbf{id}_Z \quad \mathbf{send}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_Z \ .$$

We write the binding port names in parenthesis and last. We use the $\overline{\oplus}$ operator to extend a bigraph with an identity wiring, hereby extending the inner and outer face of the bigraph. So the ion $\mathbf{send}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_Z$ has Z as inner names and $Z \cup \delta$ as outer names.

2.1 The Translation

We have a fully compositional translation from Homerσ to bigraphs.

Definition 2.1 (Translation of Homerσ-terms into bigraphs) *We define the translation of a Homerσ-term p inductively in the inference of $\tilde{x} \vdash p : \tilde{n}$*

$$\begin{aligned}
[\![\tilde{x} \vdash \mathbf{0} : \tilde{n}]\!] &= \tilde{n} \mathbin{\overline{\oplus}} \tilde{x} \\
[\![\tilde{x} \vdash p \mid q : \tilde{n}_1 \cup \tilde{n}_2]\!] &= [\![\tilde{x} \vdash p : \tilde{n}_1]\!] \mid [\![\tilde{x} \vdash q : \tilde{n}_2]\!] \\
[\![\tilde{x} \vdash (n)p : \tilde{n}]\!] &= /n \circ ([\![\tilde{x} \vdash p : \tilde{n}n]\!]) \\
[\![\tilde{x}x \vdash x : \tilde{n}]\!] &= \mathbf{var}_x \mathbin{\overline{\oplus}} \tilde{n} \mathbin{\overline{\oplus}} \tilde{x} \\
[\![\tilde{x} \vdash p[x := r : \tilde{n}'] : \tilde{n} \cup \tilde{n}']\!] &= (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n},\tilde{x}})([\![\tilde{x}x \vdash p : \tilde{n}]\!] \mid \\
&\qquad (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})([\![\vdash r : \tilde{n}']\!] \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})[\![\tilde{n}']\!])) \\
[\![\tilde{x} \vdash \delta[r]_{\tilde{n}'} : \tilde{n}' \cup fn(\delta)]\!] &= (\mathbf{loca}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n},\tilde{x}})([\![\tilde{x} \vdash r : \tilde{n}']\!] \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})[\![\tilde{n}']\!]) \\
[\![\tilde{x} \vdash \overline{\delta}\langle r\rangle_{\tilde{n}'} : \tilde{n}' \cup fn(\delta)]\!] &= (\mathbf{send}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n},\tilde{x}})([\![\tilde{x} \vdash r : \tilde{n}']\!] \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})[\![\tilde{n}']\!]) \\
[\![\tilde{x} \vdash \delta(x) \,.\, p : \tilde{n} \cup fn(\delta)]\!] &= (\mathbf{rece}_{\delta(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n},\tilde{x}})[\![\tilde{x}x \vdash p : \tilde{n}]\!] \\
[\![\tilde{x} \vdash \overline{\delta}(x) \,.\, p : \tilde{n} \cup fn(\delta)]\!] &= (\mathbf{take}_{\delta(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n},\tilde{x}})[\![\tilde{x}x \vdash p : \tilde{n}]\!]
\end{aligned}$$

and we translate the type annotations as follows: $[\![\tilde{n}]\!] = \prod_{n \in \tilde{n}} \mathbf{tname}_n$.

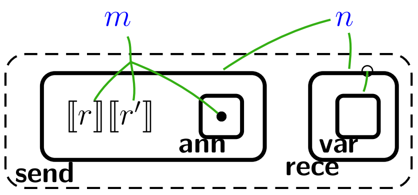

Figure 2. Example on translation of the term $\overline{n}\langle r \mid r' \rangle_{\{m\}} \mid n(x) \,.\, x$ into a bigraph

We represent **0** as an empty bigraph with the correct outer face, parallel composition is represented by the prime product, and we use a closure $/n$ to represent the restriction of the name n. A variable is represented as a node of control **var** which is connect to the name x. We represent the explicit substitutions in Homerσ in the same way as [16], except that we have augmented the explicit substitution with a type annotation.

The two constructors $\delta[r]_{\tilde{n}}$ and $\overline{\delta}\langle r \rangle_{\tilde{n}}$ are represented by a place with the corresponding control containing the representation of the resource r and the representation of the type annotation as a set of **tname** nodes enclosed by a place with control **ann**. The two prefixes $\delta(x) \,.\, p$ and $\overline{\delta}(x) \,.\, p$ are encoded straightforwardly by a node of the respective control, where the variable x is bound in the enclosed encoding of p, and we require that x and $\tilde{x}$ are disjoint. As an example on the translation from Homerσ-terms to bigraphs, we depict in Fig. 2 the result of the translation of $\overline{n}\langle r \mid r' \rangle_{\{m\}} \mid n(x) \,.\, x$. The static correspondence, stated by the theorem below, is proven in App. B.

Theorem 2.2 (Static correspondence) $\tilde{x} \vdash p \equiv_\sigma q : \tilde{n}$ *if and only if* $[\![\tilde{x} \vdash p : \tilde{n}]\!] = [\![\tilde{x} \vdash q : \tilde{n}]\!]$.

In order to present the reaction rules of Homerσ we first present the path contexts and the open operation. We define the translation of a path context $\mathcal{C}^{\tilde{n}}_{\gamma}$ into a bigraph of a certain form, called a *path bigraph*, inductively in the structure of $\mathcal{C}^{\tilde{n}}_{\gamma}$

$$\begin{aligned}
[\![\vdash \mathcal{C}^{\emptyset}_{\epsilon} : \tilde{n}'']\!] &= \mathsf{id}_{\tilde{n}''} \\
[\![\vdash \mathcal{C}^{\tilde{n}\tilde{m}}_{\delta\gamma} : \tilde{n}'']\!] &= (\mathsf{loca}_{\delta} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}''})(/\tilde{n} \circ ([\![\vdash \mathcal{C}^{\tilde{m}}_{\gamma} : \tilde{n}']\!] \mid \mathsf{id}_{\tilde{n}'}) \mid (\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{m}'})[\![\tilde{m}']\!])
\end{aligned}$$

if $\mathcal{C}^{\tilde{n}\tilde{m}}_{\delta\gamma} = \delta[(\tilde{n})(\mathcal{C}^{\tilde{m}}_{\gamma} \mid (-)_{\tilde{n}'})]_{\tilde{m}'}$. We let F, F' range over path bigraphs. And as for Homerσ we will sometimes use subscript to denote the address of the hole and superscript to denote the bound names of the hole. We define an *open operator* on path bigraphs, $\tilde{m} \odot_b F$, extending the type annotations with $\tilde{m}$

$$\begin{aligned}
\tilde{m} \odot_b \mathsf{id}_{\tilde{n}} &= \mathsf{id}_{\tilde{n}\cup\tilde{m}} \\
\tilde{m} \odot_b F &= (\mathsf{loca}_{\delta} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'',\tilde{m}})(/(\tilde{n} \setminus \tilde{m}) \circ ((\tilde{m} \odot_b [\![\vdash \mathcal{C}^{\tilde{m}}_{\gamma} : \tilde{n}']\!]) \mid \mathsf{id}_{\tilde{n}'}) \mid \\
&\qquad (\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{m}',\tilde{m}})[\![\tilde{m}' \cup \tilde{m}]\!])
\end{aligned}$$

if $F = (\mathbf{loca}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}''})(/\tilde{n} \circ (⟦\vdash \mathcal{C}^{\tilde{m}}_\gamma : \tilde{n}'⟧ \mid \mathbf{id}_{\tilde{n}'}) \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{m}'})⟦\tilde{m}'⟧)$. Note that we cannot just juxtaposition the type annotations as $⟦\tilde{m}'⟧ \mid ⟦\tilde{m}⟧$, since we represent the individual elements of the type annotations explicitly with one node per element in the annotation, as this would result in our annotations being multisets rather than sets. In App. A we present a sorting, which describes the bigraphs corresponding to Homerσ processes.

2.2 Reaction Rules of ´Homerσ

In this subsection we present the reaction rules of Homerσ.

Definition 2.3 (reaction rules of Homerσ) *We define the four reaction rules of ´Homerσ below*

Send:

$R = (\mathbf{send}_{\gamma\delta} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})\big(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})\big) \mid F_\gamma \circ (\mathbf{rece}_{\delta(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})$

$R' = (\tilde{n} \odot_b F_\gamma) \circ (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})(\mathbf{id}_{x\tilde{n}'} \mid (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})))$

$\eta = \{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1\}$

Take:

$R = F^{\tilde{m}}_\gamma \circ (\mathbf{loca}_\delta \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})) \mid (\mathbf{take}_{\gamma\delta(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})$

$R' = /(\tilde{m} \cap \tilde{n}) \circ ((\tilde{n} \odot_b F^{\tilde{m}}_\gamma) \circ \mathbf{0}) \mid (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})(\mathbf{id}_{x\tilde{n}'} \mid (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})))$

$\eta = \{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1\}$

Apply:

$R = (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})(\mathcal{C} \circ \mathbf{var}_x \mid (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})))$

$R' = (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})(\tilde{n} \odot_b \mathcal{C} \circ \mathbf{id}_{\tilde{n}} \mid (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})(\mathbf{id}_{\tilde{n}} \mid (\mathbf{ann} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})))$

$\eta = \{0, 1 \mapsto 0, 2 \mapsto 1\}$

Garbage:

$R = (\mathbf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}'})\big(\mathbf{id}_{\tilde{n}'} \mid (\mathbf{def}_x \mathbin{\overline{\oplus}} \mathbf{id}_{\tilde{n}})\big), \quad R' = \mathbf{id}_{\tilde{n}'}, \quad \eta = \{0 \mapsto 0\}$

In all the rules we have chosen to enumerate the holes from left to right in the terms representing the bigraphs, but omitting the last k holes in the $k+1$-hole path contexts F_γ and $F^{\tilde{m}}_\gamma$ on which the instantiation acts as the identity. In both the rules Send and Take the path bigraph F_γ does not bind the names in δ. In both rules the content of the **ann** node is used in the open operator, that is the set $\tilde{n}$. Both rules mimic their counterparts in Homerσ closely. Note that it is crucial that we have explicitly typed the parameters of the parametric reaction rule, and that we do not allow parameters to contain outer names not mentioned explicitly in the rules. In the rule Apply we utilise a general Homerσ context $\mathcal{C}$ satisfying the sorting requirement and that it does not close the variable-link x. The reaction rule Garbage, which discards the explicit substitution, is defined as in [16]. The proof of the operational correspondence, stated in the theorem below, is given in App. D.

Theorem 2.4 (Operational correspondence) *For every well-typed process $\vdash p : \tilde{n}$, we have*

$$\vdash p \searrow_\sigma p' : \tilde{n} \textit{ if and only if } ⟦\vdash p : \tilde{n}⟧ \longrightarrow ⟦\vdash p' : \tilde{n}⟧ \ .$$

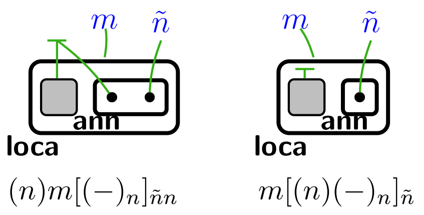

Figure 3. Location of a restriction

Now, let us take a closer look at the use of the type annotations. As mentioned in the introduction we have to be careful when combining local names and non-linear process passing. Since the two processes

$$(n)m[P] \quad \text{and} \quad m[(n)P] \qquad (\text{assuming } n \neq m) \tag{5}$$

are not structural congruent in general, they should not give rise to isomorphic bigraphs under the translation. If we consider our encoding without type annotations, then the two processes in (5) will give rise to isomorphic bigraphs, since we have no means to detect whether the closure occur outside or inside the location. In BRSs which copy parameters this would lead to the same kind of problems as mentioned in the introduction. In Fig. 3 we have illustrated how the type annotations helps us in distinguishing the two bigraphs. If the name appears in the type annotation, then the closure must be outside the location and every copy of the parameter will share this link. On the other hand, if the restricted name does not appear in the type annotation then every copy of the parameter will have a distinct link.

An immediate suggestion for an alternative to the type annotations is to represent name closures explicitly as a control with a binding port. However, then the usual scope condition would require the place with the binding port in the representation of $(n)p$ to be *around* the process p, which would break the usual structural congruence equalities such as $(n)(m)p \equiv_\sigma (m)(n)p$ and $(n)p \mid q \equiv_\sigma (n)(p \mid q)$, for $n \notin fn(q)$.

Recently Jensen and Milner have proposed a solution to the same problem of copying parameters with closed links unambiguously. In their solution they make use of an atomic **res** place for the restriction with a new kind of *outward-binding* port. The sole purpose of the **res** place is to facilitate this binding port, but contrary to the binding ports in normal binding bigraphs, this port does not bind inside the node, but instead it binds inside the parent node. Besides this change the port behaves as a traditional binding port. This explicit representation of restriction using one **res** place per restriction behaves well wrt the structural equalities above, but instead it breaks the equalities: $\pi\,.\,(n)p \equiv_\sigma (n)\pi\,.\,p$, if $n \notin fn(\pi)$ and $(n)p \equiv_\sigma p$, if $n \notin fn(p)$. More importantly, this solution does neither provide the desired *bisimulation* congruence. The typed perfect firewall equation $(n)(n[p]) : \tilde{n} \approx \mathbf{0} : \tilde{n}$ given in the introduction will only hold if $fn(p) \subseteq \{n\}$. The reason is that without the explicit localisation of links within active sub locations we loose local information

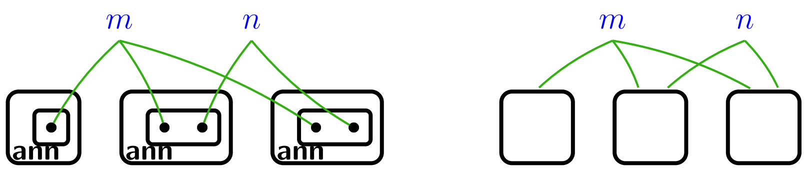

Figure 4. Original representation and using localised links

about the outer names of a process when we place it in a context.

2.3 Bigraphs with Localised Links

Since the type annotations in Homerσ are sets, we needed a way to associate an arbitrary number of names to a place in an *unordered* way. In the left-hand side of Fig. 4 we have sketched a situation where we have 3 places representing the Homerσ process $\delta[\mathbf{0}]_m \mid \delta[\mathbf{0}]_{m,n} \mid \delta[\mathbf{0}]_{m,n}$, where we have omitted the links δ. The solution used in this paper, and also used in the encoding of "The Game of Life" in [7], is to introduce an **ann** place as a child of the place, and let it contain one **tname** place per name that we want to associate with the grand-parent place.

The annotation of names to places suggests an extension to local bigraphs in which one can associate names directly to a place in an *unordered* way, as illustrated on the right-hand side of Fig. 4, which we will call *localised links*. A direct consequence of this extension will be that we can remove the controls **tname** and **ann** from the encoding and instead represent the type annotations directly using localised links.

We do not propose localised links as a replacement for traditional links, but rather as an extension to these, as we still also want to be able to connect links to ordered ports, e.g. when representing $m[p]_{\{m\}}$ the name m will both be connected to the port corresponding to the address of the location, and localised in the place because of the type annotation.

Formally, we suggest to introduce a new function to the definition of a local bigraph. For a local bigraph $G : \langle m, \overrightarrow{X} \rangle \to \langle n, \overrightarrow{Y} \rangle$ with the set of edges E and the set of places V, we let the function *localise* map edges and outer names to a set of places, $\mathit{localise} : E \uplus Y \to \mathtt{Pow}(V)$. We require that this map satisfies a scoping condition as for traditional links. We define the composition of two bigraphs

$$\begin{array}{ll} F : \langle m, \overrightarrow{X} \rangle \to \langle n, \overrightarrow{Y} \rangle & \text{with places } V\text{, edges } E\text{, and function } \mathit{localise} \\ G : \langle l, \overrightarrow{Z} \rangle \to \langle m, \overrightarrow{X} \rangle & \text{with places } V'\text{, edges } E'\text{, and function } \mathit{localise}' \end{array}$$

as usual for local bigraphs. The localisation function $\mathit{localise}'' : E \uplus E' \uplus Y \to \mathcal{P}(V) \uplus \mathcal{P}(V')$ for $F \circ G$ is defined as follows (using the link map, *link*, of F)

$$\mathit{localise}''(x) = \begin{array}{lll} \mathit{localise}'(x) & & \text{if } x \in E' \ , \\ \mathit{localise}(x) & {}_{x' \in X \text{ and } \mathit{link}(x')=x} \mathit{localise}'(x') & \text{if } x \in E \uplus Y \ . \end{array}$$

The locations of an edge in E' remain unchanged by the composition, whereas for a name in Y or an edge in E we might need to combine the locations of *localise* and *localise*$'$, if a name in X links to the name or edge, respectively.

3 Conclusions and Further Work

We have presented a higher-order calculus with non-linear active process mobility and local names, Homerσ as a bigraphical reactive system Ћomerσ. We prove that structural congruence of Homerσ corresponds to graph isomorphism in Ћomerσ and that there is a tight operational correspondence between the reaction relation of Homerσ and the reaction relation of Ћomerσ. The presentation highlights the importance of keeping explicit track of the free names of parameters in reaction rules of bigraphs. It also address the issue of localisation of names (links) which suggests an extension to local bigraphs called *bigraphs with localised links*.

Several interesting questions arise from the work done in this paper. First and foremost, we plan to examine the labelled transition bisimulation congruence derivable using the general theory of bigraphs and compare it to the labelled transition bisimulation congruences for Homer in [9]. In this process we plan to examine proof techniques known from calculi for concurrency and mobility in the setting of bigraphs. Especially we plan to investigate the notion of *up-to* proof techniques related to bisimulation equivalences in bigraphs. We would also plan to further examine the extension of localised links, both with respect to facilitate encodings as bigraphical reactive systems and with respect to the behavioural theory of bigraphical reactive systems, in particular if the extension retains relative pushouts.

Currently several proposals exists for expressing constraints on the possible nesting of nodes, the linkage between ports etc. It would be interesting to see whether the sorting presented in App. A can be expressed in these settings, and in particular if we can enforce a more strict control with the movement and locations of closed free links. Hence to capture some of the same information as the outward-binding node, but without introducing an explicit node representing the restriction.

Acknowledgement

The authors would like to thank the members of the *Bigraphical Programming Languages*-group at ITU for helpful discussions about how to encode certain properties and constructs in bigraphs. We would also like to thank Robin Milner and Ole Høgh Jensen for providing access to early drafts of their papers.

References

[1] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Levy, *Explicit substitutions*, Journal of Functional Programming **1** (1991), pp. 375–416.

[2] Bruni, R. and I. Lanese, *On graph(ic) encodings*, in: B. Koenig, U. Montanari and P. Gardner, editors, *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*, number 04241 in Dagstuhl Seminar Proceedings (2005).

[3] Bundgaard, M., T. Hildebrandt and J. C. Godskesen, *A CPS encoding of name-passing in higher-order mobile embedded resources*, in: J. Baeten and F. Corradini, editors, *Proceedings of the 11th International Workshop on Expressiveness in Concurrency (EXPRESS'04)*, Electronic Notes in Theoretical Computer Science **128** (2005), pp. 131–150.

[4] Bundgaard, M., T. Hildebrandt and J. C. Godskesen, *A CPS encoding of name-passing in higher-order mobile embedded resources*, Theoretical Computer Science (2005), accepted for publication in a special issue of TCS.

[5] Carbone, M. and S. Maffeis, *On the expressive power of polyadic synchronisation in π-calculus*, Nordic Journal of Computing **10** (2003), pp. 70–98.

[6] Castagna, G., J. Vitek and F. Z. Nardelli, *The Seal calculus* (2004), accepted for publication in *Information and Computation*.

[7] Debois, S. and T. C. Damgaard, *Bigraphs by example*, Technical Report TR-2005-61, IT University of Copenhagen (2005).

[8] Ferrari, G., U. Montanari and P. Quaglia, *A π-calculus with explicit substitutions*, Theoretical Computer Science **168** (1996), pp. 53–103.

[9] Hildebrandt, T., J. C. Godskesen and M. Bundgaard, *Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources*, Technical Report TR-2004-52, IT University of Copenhagen (2004).

[10] Hirschkoff, D., *Handling substitutions explicitely in the π-calculus*, in: *Proceedings of Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs (WESTAPP'99)*, 1999, pp. 28–43.
URL http://cermics.enpc.fr/~dh/sigma/full.ps.gz

[11] Jensen, O. H., "Mobile Processes in Bigraphs," Ph.D. thesis, Department of Computer Science, Aalborg University (2005), forthcoming.

[12] Jensen, O. H. and R. Milner, *Bigraphs and transitions*, in: *Proceedings of the 30rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03)* (2003), pp. 38–49.

[13] Jensen, O. H. and R. Milner, *Bigraphs and mobile processes (revised)*, Technical Report UCAM-CL-TR-580, University of Cambridge, Computer Laboratory (2004).
URL http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-580.pdf

[14] Milner, R., *Bigraphs for petri nets*, in: J. Desel, W. Reisig and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, Lecture Notes in Computer Science **3098** (2004), pp. 686–701.

[15] Milner, R., *Bigraphs whose names have multiple locality*, Technical Report UCAM-CL-TR-603, University of Cambridge, Computer Laboratory (2004). URL http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-603.pdf

[16] Milner, R., *Local bigraphs, confluence and λ-calculus* (2004), draft of October 31, 2004.

[17] Rose, K. H., *Explicit substitution — tutorial & survey*, Lecture Series LS-96-3, BRICS, Department of Computer Science, University of Aarhus (1996), v+150 pp.

[18] Sangiorgi, D., "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms," Ph.D. thesis, Department of Computer Science, University of Edinburgh (1992).

[19] Thomsen, B., *Plain CHOCS: A second generation calculus for higher order processes*, Acta Informatica **30** (1993), pp. 1–59.

A A Simple Sorting on Homerσ

In this appendix we present a simple sorting to ensure that we only work with a subset of ground bigraphs, that is the bigraphs that are 'correct' with respect to our encoding. The sorting introduces a requirement on the possible nesting of nodes and on how the linkage is performed, particularly that the sets of free names and variables are kept disjoint. We need some nomenclature to differentiate the different kinds of links and ports before stating the definition of the class of bigraphs that we are interested in. We have two kinds of ports: name- and variable-ports.

- The *name-ports* are the port of a **tname** node and all the free ports of a **rece**, **take**, **send**, or a **loca** node.
- The *variable-ports* are the free port of a **def** node or a **var** node or the binding port of a **sub**, **rece**, or a **take** node.

In the same way we define two kinds of links:

- A *name-link* is a link with only name-ports, and if free a name.
- A *variable-link* is a link with only variable-ports connected to it, and if free a variable name.

Definition A.1 (bigraphs good for Homerσ) *We define a sub-class $\mathcal{I}$ of ground bigraphs in ´Homerσ as the bigraphs that satisfy the following requirements*

- *We only allow name- and variable-links as links in the bigraph.*
- *A variable-link can be connected to any number of* **var***-ports.*

- · *If a variable-link is bound by either a* **rece**-*or a* **take**-*port, then it contains no* **def**-*ports.*
- · *If a variable-link is bound by a port on a* **sub**-*node v, then it also has one unique* **def**-*port, which resides on a child of v, and this is the only location where a* **def** *node can occur.*

- *A name-link can be connected to any number of name-ports.*
- *For every pair of distinct* **tname** *nodes enclosed in the same* **ann** *node their name-ports must be connected to distinct links.*
- *Every* **loca**, **send**, *and* **def** *node must contain an unique* **ann** *child node, and these are the only locations where* **ann** *nodes can occur.*
- *All* **tname** *nodes must be in a* **ann** *node and no other kind of nodes can reside here.*

We have introduced all the abovementioned restrictions to enforce that we only work with bigraphs, that have a structure corresponding to how we interpret Homerσ in bigraphs. In Homerσ the sets of names and variables are by definition disjoint, but since we use the links of bigraphs to encode both sets, we need some additional requirements to enforce the distinction in kinds of links.

The requirements enforce that a **loca** node and a **send** node contains unique **ann** node. We also require that **def** can only appear as a child of a **sub** node. Finally, we require that the **tname** nodes representing a type annotation only occur in a **ann** node and that they are unique, in the sense that they all are linked to different name-links.

Proposition A.2 (invariant) *The class of bigraphs $\mathcal{I}$ is preserved by the reaction relation $\twoheadrightarrow$ defined in Sec. 2.2 and contains all images of the translation given in Def. 2.1.*

B Static Correspondence

In this appendix we prove that two Homerσ-processes are structural congruent if and only if their image under the encoding are isomorphic. We prove each direction separately.

Proposition B.1 $\tilde{x} \vdash p \equiv_\sigma q : \tilde{n}$ *implies* $[\![\tilde{x} \vdash p : \tilde{n}]\!] = [\![\tilde{x} \vdash q : \tilde{n}]\!]$.

Proof Since the translation is compositional we can consider each of the axioms defining $\equiv_\sigma$ separately. We only present some of the cases

- Each of the axioms

  $$\tilde{x} \vdash p \mid \mathbf{0} \equiv_\sigma p : \tilde{n} \quad \tilde{x} \vdash (p \mid p') \mid p'' \equiv_\sigma p \mid (p' \mid p'') : \tilde{n} \quad \tilde{x} \vdash p \mid q \equiv_\sigma q \mid p : \tilde{n}$$

  follows directly from the translation, since we translate parallel composition in Homerσ as the prime product in bigraphs '$|$', which can be shown to be associative and commutative, and as we translate $\mathbf{0}$ into the unit for $|$.

- To prove the case for the axiom for reordering of restrictions

$$\tilde{x} \vdash (n)(m)p \equiv_\sigma (m)(n)p : \tilde{n}$$

  we show that the two bigraphs $[\![\tilde{x} \vdash (n)(m)p : \tilde{n}]\!]$ and $[\![\tilde{x} \vdash (m)(n)p : \tilde{n}]\!]$ can be constructed in the same manner (we assume that m and n are distinct and names of p). We construct $[\![\tilde{x} \vdash p : \tilde{n}nm]\!]$ and add two edges to its link graph e_m and e_n and make all points of m (n) point to e_m (e_n). Finally we remove the names m and n.

- The axiom for scope extension

$$\tilde{x} \vdash (n)p \mid q \equiv_\sigma (n)(p \mid q) : \tilde{n}, \text{ if } n \notin \mathit{fn}(q)$$

  can be proven in the same way. We construct the bigraphs $[\![\tilde{x} \vdash (n)p \mid q : \tilde{n}]\!]$ and $[\![\tilde{x} \vdash (n)(p \mid q) : \tilde{n}]\!]$ in the following way. Without loss of generality we assume that $\tilde{n} = \tilde{n}_1 \cup \tilde{n}_2$, where $\tilde{n}_1 n \tilde{x}$ and $\tilde{n}_2 \tilde{x}$ are the names in the outer face of $[\![\tilde{x} \vdash p : \tilde{n}_1 n]\!]$ and $[\![\tilde{x} \vdash q : \tilde{n}_2]\!]$, respectively. First we build $[\![\tilde{x} \vdash p : \tilde{n}_1 n]\!]$ and $[\![\tilde{x} \vdash q : \tilde{n}_2]\!]$ and combine them using the prime product, then we add one edge e_n to the link graph of this bigraph and make all points of the name n point to e_n. Since $n \notin \mathit{fn}(q)$ we only touch points in $[\![\tilde{x} \vdash p : \tilde{n}_1 n]\!]$. Finally we remove the name n.

- For the remaining cases we proceed in the same manner by exhibiting a constructing that forms both bigraphs.

□

Proposition B.2 *If* $[\![\tilde{x} \vdash p : \tilde{n}]\!] = [\![\tilde{x} \vdash q : \tilde{n}]\!]$ *then* $\tilde{x} \vdash p \equiv_\sigma q : \tilde{n}$.

From Prop. B.1 and Prop. B.2 it follows that two Homerσ-processes are structural congruent if and only if their image under the encoding are isomorphic.

Theorem B.3 (Static correspondence) $\tilde{x} \vdash p \equiv_\sigma q : \tilde{n}$ *if and only if* $[\![\tilde{x} \vdash p : \tilde{n}]\!] = [\![\tilde{x} \vdash q : \tilde{n}]\!]$.

C Mimicking Reactions

In this appendix we present how reactions in Homerσ are mimicked by the encoding as a BRS. We consider the following reactions, where we have omitted the top-level types.

$$\begin{gathered}
\overline{on}\langle r \mid r' \rangle_{\{m\}} \mid o[n(x) \,.\, x]_{\{n\}} \searrow_\sigma \\
o[x[x := (r \mid r') : \{m\}]]_{\{n,m\}} \searrow_\sigma \\
o[(r \mid r')[x := (r \mid r') : \{m\}]]_{\{n,m\}} \searrow_\sigma \\
o[r \mid r']_{\{n,m\}}
\end{gathered}$$

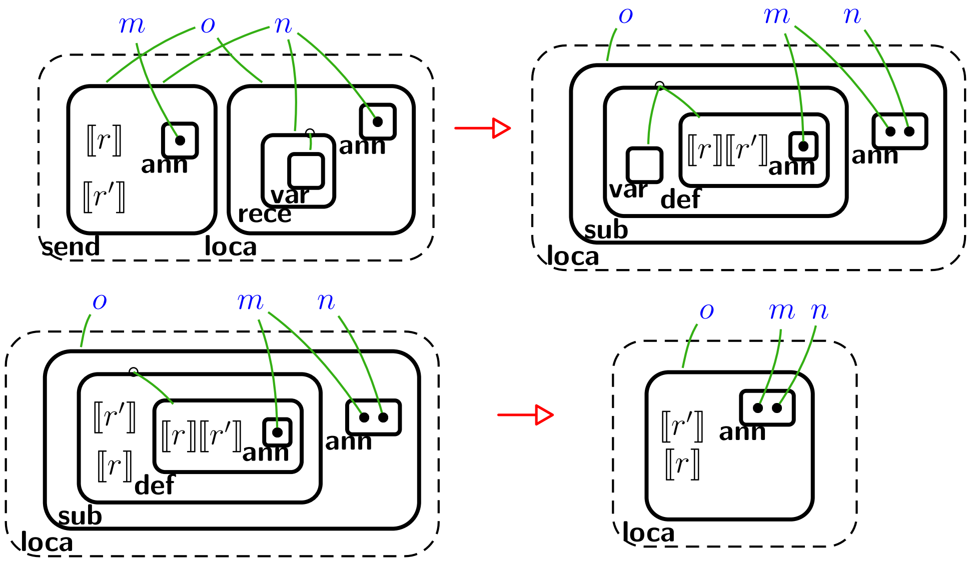

Figure C.1. Mimicking $\overline{on}\langle r \mid r'\rangle_{\{m\}} \mid o[n(x) \,.\, x]_{\{n\}} \searrow^*_\sigma o[r \mid r']_{\{n,m\}}$

using the rules, *sendσ*, *applyσ*, and *garbageσ*. In the second line we have the location o containing the process variable x enclosed in an explicit substitution, which can substitute $r \mid r'$ of type $\{m\}$ in for x. In bigraphs we have the matching sequence of reactions depicted in Fig. C.1. Note that we have chosen not to draw the possible free name m of r and r'.

D Operational Correspondence

In this appendix we prove the main theorem of the paper, the operational correspondence between reactions in Homerσ and reactions in its encoding as a BRS Ꞁomerσ. By inspecting the translation we can easy see that evaluation contexts in Homerσ are translated to active contexts, and conversely if the image under the translation is an active context then the preimage must have been an evaluation context.

We follow the same method as Jensen and Milner by first characterising the reactions in both Homerσ and Ꞁomerσ by the forms of the expressions involved. Then we use the definition of the translation to connect the characterisations. We only present two of the cases (*garbageσ*) and (*sendσ*) the remaining two are similar. Prop. D.1 and Prop. D.2 characterise the reaction relations $\searrow_\sigma$ and $\rightarrowtriangle$ (for the rules (*garbageσ*) and Garbage, respectively) in terms of the form of the processes and bigraphs.

Proposition D.1 *$\vdash p \searrow_\sigma p' : \tilde{n}$ by the rule (garbageσ) if and only if p and p' are of the forms*

$$\vdash p \equiv_\sigma \mathcal{E}(q[x := r : \tilde{n}']) : \tilde{n}$$
$$\vdash p' \equiv_\sigma \mathcal{E}(q) : \tilde{n} \;\; ,$$

if $x \notin fn(q)$ and for an evaluation context $\mathcal{E}$.

Proposition D.2 *$g \rightarrowtriangle g'$ by the rule Garbage if and only if g and g' are of the forms*

$$\begin{aligned} g \ &= E \circ ((\textsf{sub}_{(x)} \mathbin{\overline{\oplus}} \textsf{id}_{\tilde{n}})\ h \mid (\textsf{def}_x \mathbin{\overline{\oplus}} \textsf{id}_{\tilde{n}'})h'\ ) \\ g' &= E \circ h\ \ , \end{aligned}$$

if the outer face of h is $\tilde{n}$ and E is an active context.

Since the outer face of h is $\tilde{n}$, it means that h cannot be connected to the binder x in the surrounding **sub** control.

Lemma D.3 (operational correspondence on (*garbage*σ) and Garbage) *$\vdash p \searrow_\sigma p' : \tilde{n}$ by the rule (garbageσ) if and only if $[\![\vdash p : \tilde{n}]\!] \rightarrowtriangle [\![\vdash p' : \tilde{n}]\!]$ by the rule Garbage.*

Proof From Prop. D.1 we know that $\vdash p \searrow_\sigma p' : \tilde{n}$ if and only if p and p' have the forms

$$\begin{aligned} \vdash p\ &\equiv_\sigma\ \mathcal{E}(q[x := r : \tilde{n}']) : \tilde{n} \\ \vdash p' &\equiv_\sigma\ \mathcal{E}(q) : \tilde{n}\ \ , \end{aligned}$$

and $x \notin fn(q)$ and from α-conversion we can assume that all bound names are distinct and disjoint from the free names, and without loss of generality that the hole of $\mathcal{E}$ is annotated with the type $\tilde{n}''$. From the correspondence between structural congruence and graph isomorphism we have

$$\begin{aligned} [\![\vdash p : \tilde{n}]\!] \ &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ([\![\vdash q[x := r : \tilde{n}'] : \tilde{n}'']\!]) \\ &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ((\textsf{sub}_{(x)} \mathbin{\overline{\oplus}} \textsf{id}_{\tilde{n}''})([\![\vdash q : \tilde{n}'']\!] \mid (\textsf{def}_x \mathbin{\overline{\oplus}} \textsf{id}_{\tilde{n}'})h')) \\ [\![\vdash p' : \tilde{n}]\!] &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ([\![\vdash q : \tilde{n}'']\!])\ \ , \end{aligned}$$

since $x \notin fn(q)$ and letting $h' = [\![\vdash r : \tilde{n}']\!] \mid (\textsf{ann} \mathbin{\overline{\oplus}} \textsf{id}_{\tilde{n}'})[\![\tilde{n}']\!]$. By Prop. D.2 this holds if and only if $[\![\vdash p : \tilde{n}]\!] \rightarrowtriangle [\![\vdash p' : \tilde{n}]\!]$. □

We proceed in the same manner with the case for (*send*σ). Prop. D.4 and Prop. D.5 characterise the reaction relations $\searrow_\sigma$ and $\rightarrowtriangle$ (for the rules (*send*σ) and Send, respectively) in terms of the form of the processes and bigraphs.

Proposition D.4 *$\vdash p \searrow_\sigma\ \vdash p' : \tilde{n}$ by the rule (sendσ) if and only if p and p' are of the forms*

$$\begin{aligned} \vdash p\ &\equiv_\sigma\ \mathcal{E}(\overline{\gamma\delta}\langle r\rangle_{\tilde{n}'} \mid \mathcal{C}_\gamma^{\tilde{m}}(\delta(x)\,.\,q, \overrightarrow{q})) : \tilde{n} \\ \vdash p' &\equiv_\sigma\ \mathcal{E}(\tilde{n}' \odot \mathcal{C}_\gamma^{\tilde{m}}(q[x := r : \tilde{n}'], \overrightarrow{q})) : \tilde{n}\ \ , \end{aligned}$$

if $\tilde{m} \cap (\delta \cup \tilde{n}') = \emptyset$ and for an evaluation context $\mathcal{E}$.

Proposition D.5 *$g \rightarrowtriangle g'$ by the rule Send if and only if g and g' are of the* *forms*

$$\begin{aligned} g &= E \circ ((\mathsf{send}_{\gamma\delta} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})\ h \mid (\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})h' \ \mid F_\gamma \circ ((\mathsf{rece}_{\delta(x)} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'''})h''')) \\ g' &= E \circ ((\tilde{n}' \odot_b F_\gamma) \circ (\mathsf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}''' \cup \tilde{n}'})(\\ &\qquad h''' \mid (\mathsf{def}_x \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})(h \mid (\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})h'))) \ , \end{aligned}$$

if the outer face of h and h' are $\tilde{n}'$, of h''' is $\tilde{n}'''x$, E is an active context with inner face $\tilde{n}''$, and F_γ is a path bigraph with inner face $\tilde{n}'''$.

Note that we leave the last k holes in the $k+1$-hole path contexts F_γ unspecified.

Lemma D.6 (operational correspondence on (*sendσ*) and Send) *$\vdash p \searrow_\sigma p' : \tilde{n}$ by the rule (sendσ) if and only if $[\![\vdash p : \tilde{n}]\!] \twoheadrightarrow [\![\vdash p' : \tilde{n}]\!]$ by the rule Send.*

Proof From Prop. D.4 we know that $\vdash p \searrow_\sigma p' : \tilde{n}$ if and only if p and p' have the forms

$$\begin{aligned} \vdash p &\equiv_\sigma \ \mathcal{E}(\overline{\gamma\delta}\langle r\rangle_{\tilde{n}'} \mid \mathcal{C}^{\tilde{m}}_\gamma(\delta(x)\,.\,q, \overrightarrow{q})) : \tilde{n} \\ \vdash p' &\equiv_\sigma \ \mathcal{E}(\tilde{n}' \odot \mathcal{C}^{\tilde{m}}_\gamma(q[x := r : \tilde{n}'], \overrightarrow{q})) : \tilde{n} \ , \end{aligned}$$

if $\tilde{m} \cap (\delta \cup \tilde{n}') = \emptyset$ and from α-conversion we can assume that all bound names are distinct and disjoint from the free names, and without loss of generality that the hole of $\mathcal{E}$ is annotated with $\tilde{n}''$. From the correspondence between structural congruence and graph isomorphism we have

$$\begin{aligned} [\![\vdash p : \tilde{n}]\!] &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ([\![\vdash \overline{\gamma\delta}\langle r\rangle_{\tilde{n}'} \mid \mathcal{C}^{\tilde{m}}_\gamma(\delta(x)\,.\,q, \overrightarrow{q}) : \tilde{n}'']\!]) \\ &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ((\mathsf{send}_{\gamma\delta} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})([\![\vdash r : \tilde{n}']\!] \mid ((\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})[\![\tilde{n}']\!])) \mid \\ &\qquad [\![\vdash \mathcal{C}^{\tilde{m}}_\gamma : \tilde{n}'']\!] \circ (\mathsf{rece}_{\delta(x)} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'''})[\![x \vdash q : \tilde{n}''']\!]) \\ [\![\vdash p' : \tilde{n}]\!] &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ([\![\vdash \tilde{n}' \odot \mathcal{C}^{\tilde{m}}_\gamma(q[x := r : \tilde{n}'], \overrightarrow{q}) : \tilde{n}'']\!]) \\ &= [\![\vdash \mathcal{E} : \tilde{n}]\!] \circ ((\tilde{n}' \odot_b [\![\vdash \mathcal{C}^{\tilde{m}}_\gamma]\!]) \circ (\mathsf{sub}_{(x)} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}''' \cup \tilde{n}'}) \\ &\qquad ([\![x \vdash q : \tilde{n}''']\!] \mid (\mathsf{def}_x \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})([\![\vdash r : \tilde{n}']\!] \mid (\mathsf{ann} \mathbin{\overline{\oplus}} \mathsf{id}_{\tilde{n}'})[\![\tilde{n}']\!]))) \end{aligned}$$

By Prop. D.5 this holds if and only if $[\![\vdash p : \tilde{n}]\!] \twoheadrightarrow [\![\vdash p' : \tilde{n}]\!]$. □

Theorem D.7 (Operational correspondence) *For every well-typed process $\vdash p : \tilde{n}$, we have*

$$\vdash p \searrow_\sigma p' : \tilde{n} \text{ if and only if } [\![\vdash p : \tilde{n}]\!] \twoheadrightarrow [\![\vdash p' : \tilde{n}]\!] \ .$$

Distributed Graph Traversals by Relabelling Systems with Applications

Bilel Derbel [1]

LaBRI, Université Bordeaux 1
351, cours de la libération
33405 Bordeaux, France

Mohamed Mosbah [2]

LaBRI, Université Bordeaux 1
351, cours de la libération
33405 Bordeaux, France

Abstract

Graph traversals are in the basis of many distributed algorithms. In this paper, we use graph relabelling systems to encode two basic graph traversals which are the broadcast and the convergecast. This encoding allows us to derive formal, modular and simple encoding for many distributed graph algorithms. We illustrate this method by investigating the distributed computation of a breadth-first spanning tree and the distributed computation of a minimum spanning tree. Our formalism allows to focus on the correctness of a distributed algorithm rather than on the implementation and the communication details.

Key words: Distributed algorithms, Graph traversals, Relabelling systems.

1 Introduction

1.1 Motivation and contribution

Distributed algorithms are designed to run on networks consisting of interconnected autonomous entities of computations (processes) cooperating to solve given problems. Many of these algorithms, mainly dealing with the traversal of the network, appear as the compositions of some basic tasks. These basic tasks include the

[1] Email: `derbel@labri.fr`
[2] Email: `mosbah@labri.fr`

broadcasting or the propagation of information and the echo or convergecast algorithm [4,15,10,14]. For instance, in the message-passing scheme, a distributed computation of a spanning tree can be performed by broadcasting a message from an initial node to all other nodes of the graph; each node propagates the message to its neighbors upon receiving it. This simple algorithm may be described in some other way depending on the distributed model. In fact, in the distributed setting, many distributed algorithms are inherently dependent on the model one considers i.e., message-passing, shared memory, synchronous, asynchronous etc. A distributed algorithm which is designed and implemented in a given model becomes in general obsolete in another model. Even though it is possible, one has often to re-adapt or to re-encode the algorithm depending on the model specification.

In this context, graph relabelling systems and local computations [8,7,9] can be viewed as a tool which allows to encode distributed algorithms in a formal and unified way. In fact, a graph relabelling system is based on a set of relabelling rules which are executed locally and concurrently. These rules are described using mathematical and logic formulas which enables to derive formal and rigorous mathematical proofs of their correctness and by the same way to prove the correctness of an algorithm on a distributed system.

In this paper, we are interested in a high level encoding of some basic *Wave and graph traversal algorithms* which are in the basis of many sophisticated distributed algorithms. In particular, we show that by expressing the broadcast and the convergecast by graph relabelling systems, a large class of graph traversals can in turn be expressed by graph relabelling systems. The high-level encoding of such algorithms in form of graph relabelling systems allows to encode them and to prove them in a unified and simple way. Furthermore, we show that it is possible to combine these two subroutines to give a formal encoding for some basic applications which illustrate our approach.

First, we show how to encode the classical distributed layered breadth-first spanning (BFS for short) construction [5,15,13]. This algorithm involves the encoding of many iterations of a classical technique in distributed computing which is known as the “Propagation of Information with Feedback” (PIF for short) [14]. Even though our encoding is given in the special case of the BFS tree construction, it gives a general idea about how to design sophisticated algorithms based on the PIF technique.

Second, by using the convergecast as a building block, we give a general method to encode with relabelling system distributed algorithms for computing some particular (commutative and associative) global functions.

These two basic applications are then combined to derive the graph relabelling system that encodes the classical Prim’s distributed algorithm for computing a minimum spanning tree (MST for short) [5,15,13]. This example aims to show how to combine the basic graph traversals we have encoded in order to obtain a simple and formal encoding of more advanced algorithms.

1.2 Graph model and notations

In this section, we illustrate, in an intuitive way, the notion of graph relabelling systems by showing how some algorithms on networks of processors may be encoded within this framework [9]. As usual, such a network is represented by a graph $G = (V, E)$ whose nodes stand for processors and edges for (bidirectional) links between processors. We only consider undirected connected graphs without multiple edges and self-loops. At every time, each node and each edge is in some particular state encoded by a node or an edge label. According to its own state and to the states of its neighbors, each node may decide to realize an elementary *computation step*. After this step, the states of this node, of its neighbors and of the corresponding edges may have changed according to some specific *computation rules*. Let us recall that graph relabelling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labelling of its components (edges and/or nodes), the final labelling being the result,

(C2) they are local, that is, each relabelling changes only a connected subgraph of a fixed size in the underlying graph,

(C3) they are locally generated, that is, the applicability condition of the relabelling only depends on the local context of the relabelled subgraph.

A precise description and definition of local computations can be found in [7]. We recall here only the description of local computations and we explain the convention under which we will describe graph relabelling systems later. A relabelling system is a triple $\mathcal{R} = (\mathcal{L}, \mathcal{I}, \mathcal{P})$ where $\mathcal{L} = \mathcal{L}_v \cup \mathcal{L}_e$ a set of labels, $\mathcal{L}_v$ a set of node labels, $\mathcal{L}_v$ a set of edge labels, $\mathcal{I}$ a subset of $\mathcal{L}$ called the set of initial labels and P a finite set of relabelling rules.

For each relabelling rule, we will consider a generic star-graph of generic center v_0 and of generic set of nodes $B(v_0, 1)$ (star of radius 1 centered at v_0) and we will refer to a node $v \neq v_0$ of the star graph by writing $v \in B(v_0, 1)$. Within these conventions, each relabelling rule is described by its precondition and relabelling. If $\lambda(v)$ is the label of v in the precondition, then $\lambda'(v)$ will be its label in the relabelling. We will omit in the relabelling the description of labels that are not modified by the rule. This means that if $\lambda(v)$ is a label such that $\lambda'(v)$ is not explicitly described in the rule for a given v, then $\lambda'(v) = \lambda(v)$. The label of a node can be composed of k components (with k a given integer). In this case, we denote by $\mathcal{L}_v = \{L_1 \times L_2 \times ... \times L_k\}$ the set of labels where L_i ($1 \leq i \leq k$) is a set of possible values of the i^{th} component. For every node v, we denote by $\lambda(v).L_i$ the i^{th} component of the label of v. We adopt the same notations for edge labels. For instance, consider a node $v \in B(v_0, 1)$, then $\lambda(v_0, v)$ refers to the labels of the edge $e = (v_0, v)$ in the precondition and $\lambda'(v_0, v)$ will be its label in the relabelling. The preconditions and the relabelling are written as logic formulas. We use the logic symbols $\wedge$, $\vee$, $\exists$, $\exists!$ and $\forall$ to denote respectively the logic operators “and”, “or”, “it exist”, “it exists a unique” and “for all”. In the case of a weighted graph

$G_{\mathcal{W}}$, if $e = (u, v)$ is an edge then we will denote by $\mathcal{W}(u, v)$ the weight of e.

1.3 Summary

The paper is organized as follows. In Section 2, we give an encoding of the broadcast and convergecast process using graph relabelling systems. In Section 3, we give two basic applications which are the distributed layered construction of a BFS tree and the computation of global functions. As a combination of these two applications, in Section 4, we give a formal and detailed algorithm for computing a minimum spanning tree. Finally, in Section 5, we give some concluding remarks.

2 Building blocks

Many basic distributed algorithms can be described as a combination of many couples of broadcast and convergecast. The broadcast is usually used to deliver a given information (*e.g*: the value of a variable, the beginning of a new step in the algorithm) to all the nodes of the network. The convergecast is in general used to collect some information into one single node. This information is for example used to activate some treatment. In the next two subsections, we give the relabelling systems corresponding to the broadcast and convergecast operations. In this section, we do not care about the information to be broadcast or collected. We only give the intuitive method to do it using relabelling systems.

2.1 The broadcast technique

The broadcast operation can be defined as *the dissemination of some information from a source node to all nodes in the network*. It can be encoded with the relabelling system $\mathcal{R}_b = (\mathcal{L}_b, \mathcal{I}_b, \mathcal{P}_b)$ defined by: $\mathcal{L}_b = \{E\} \cup \{0, 1\}$, where $E \in \{A, S, O\}$, $\mathcal{I}_b = \{A, O\} \cup \{0\}$ and $\mathcal{P}_b = \{R_b^1, R_b^2\}$. Initially, one source node from whom the broadcast is initiated is labelled A and all other nodes are labelled O. All the edges of the graph are initially labelled 0. The label S encodes the fact that the broadcast process has reached some node.

R_b^1 : **Broadcast : initial step**

<u>*Precondition :*</u>
- $\lambda(v_0).E = A$

<u>*Relabelling :*</u>
- $\forall v \in B(v_0, 1)\ (\lambda(v).E = O \implies (\lambda'(v) := S, \lambda'(v, v_0) := 1))$

R_b^2 : **Broadcast**

<u>*Precondition :*</u>
- $\lambda(v_0).E = S$
- $\exists v \in B(v_0, 1)(\lambda(v).E = O)$

<u>*Relabelling :*</u>
- $\forall v \in B(v_0, 1)\ (\lambda(v).E = O \implies (\lambda'(v) := S, \lambda'(v, v_0) := 1))$

Rule R_b^1 (resp. R_b^2) can be applied as follows. If a star center v_0 is labelled A (resp. S) then for each node v in the star $B(v_0, 1)$, if v is labelled O then it becomes labelled S and the edge (v_0, v) becomes labelled 1. Note that as a basic application of the relabelling system $\mathcal{R}_b$, once the broadcast is terminated, we obtain a spanning tree by considering the edges with labels 1. Figure 1 shows an example of the broadcast algorithm using the relabelling system $\mathcal{R}_b$. Note that the rules can be applied by many nodes on distinct balls as far as their precondition states are at the same time satisfied.

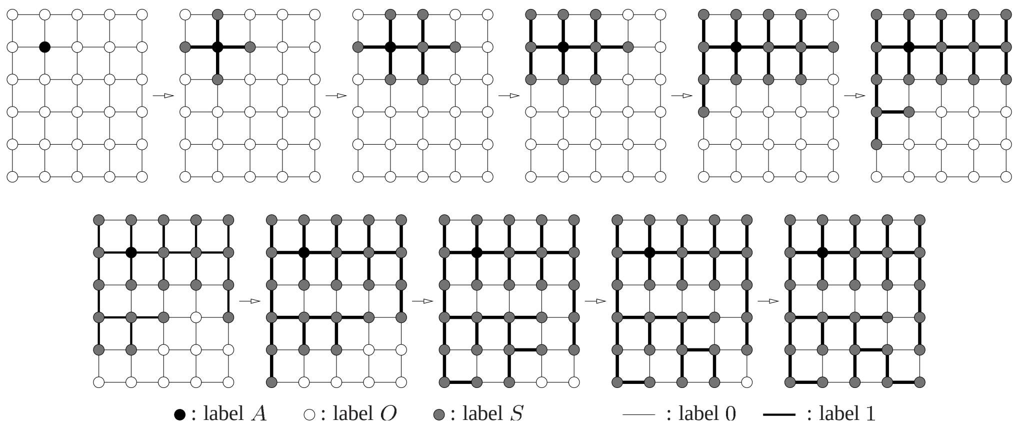

Fig. 1. An example of a broadcast using the relabelling system $\mathcal{R}_b$.

2.2 *The convergecast technique*

The convergecast operation consists in *collecting information upwards on a tree*. The most fundamental example is to let a source node, that has broadcast some information, detects that the broadcast has terminated. In fact, in order to detect the "broadcast termination", a convergecast process can be performed as follows. First, each leaf of the tree which has been reached by the broadcast sends an acknowledgment to its parent. Upon receipt of an acknowledgment from all its children, a node sends an acknowledgment to its parent and so on. When the source node receives an acknowledgment from all its children then the source node knows that the broadcast has reached all the nodes of the graph. This example can be generalized when we want to collect some other information in some root node.

We assume that we have a precomputed rooted spanning tree (Recall that the relabelling system $\mathcal{R}_b$ enables us to construct such a tree). Then, the convergecast operation can be encoded using the relabelling system $\mathcal{R}_c = (\mathcal{L}_c, \mathcal{I}_c, \mathcal{P}_c)$ defined by: $\mathcal{L}_c = \{E\} \cup \{0, 1\}$, where $E \in \{A, S, F, T\}$, $\mathcal{I}_c = \{A, S\} \cup \{0, 1\}$ and $\mathcal{P}_c = \{R_c^1, R_c^2\}$. Initially, the source node is labelled A, all other nodes are labelled S, an edge belonging to the spanning tree is labelled 1 and all other edges are labelled 0. If a node becomes labelled F then it has finished the convergecast. When the source node A becomes labelled T then the convergecast process is terminated. Figure 2 shows an example of the execution of the convergecast algorithm using the relabelling system $\mathcal{R}_c$.

R_c^1 : **A node becomes a leaf**

<u>*Precondition :*</u>
- $\lambda(v_0).E = S$
- $\exists!\, v_1 \in B(v_0,1)\ ((\lambda(v_1).E = S \vee \lambda(v_1).E = A) \wedge \lambda(v_0,v_1) = 1)$

<u>*Relabelling :*</u>
- $\lambda'(v_0).E := F$

R_c^2 : **Termination detection**

<u>*Precondition :*</u>
- $\lambda(v_0).E = A$
- $\forall v \in B(v_0,1)\ (\lambda(v).E = F)$

<u>*Relabelling :*</u>
- $\lambda'(v_0).E := T$

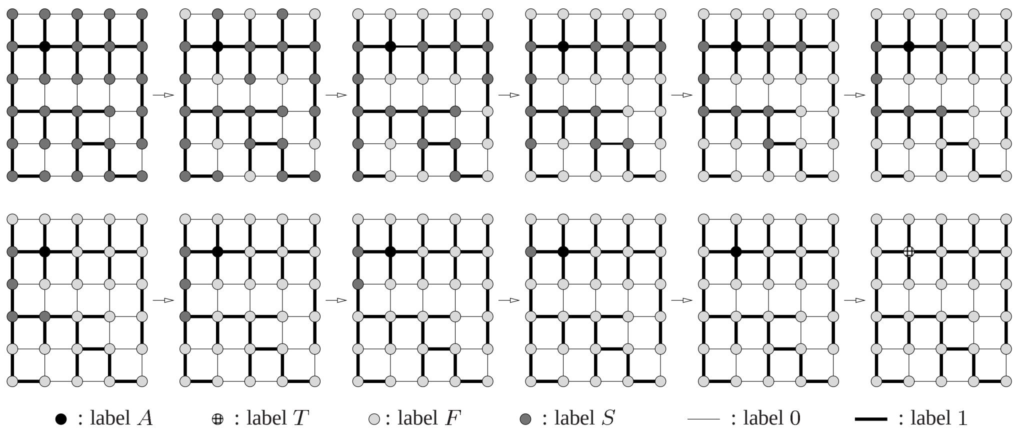

Fig. 2. An example of a convergecast using the relabelling system $\mathcal{R}_c$.

3 Two basic applications

3.1 Layered BFS tree construction

Many distributed algorithms can be implemented by performing as many as necessary phases of broadcast and convergecast. Each phase corresponds to a propagation of some information with feedback (PIF). The broadcast can be viewed as the beginning of a new stage of the algorithm and the convergecast corresponds to the termination of that stage. This technique is fundamental when designing distributed algorithms because, in the distributed setting, there is no centralized entity which supervises in a global way the execution of an algorithm. In the following, our main goal is to show how to encode by graph relabelling systems the Dijkstra's layered BFS tree [5,15,13] algorithm which is based on the PIF operation.

Recall that a BFS tree of a graph G with respect to a root node r is a spanning tree with the property that for every node v, the path leading from the root r to v in the tree is of the minimum (unweighted) length possible. One classical technique to construct a BFS tree begins by growing the tree from one pre-distinguished node. Then, it proceeds in many iterations by constructing the tree in a layered fashion beginning from the root downwards. At each iteration, the unprocessed nodes which are adjacent to some node marked as part of the tree are added. Once a layer is added, the construction of a new layer can begin. The main difficulty here is to begin adding the next layer only when the previous layer has been completely added.

The classical layered BFS tree construction can be encoded using the graph relabelling system $\mathcal{R}_t = (\mathcal{L}_t, \mathcal{I}_t, \mathcal{P}_t)$ defined by: $\mathcal{L}_t = \{E \times i\} \cup \{0, 1\}$, where $E \in \{A, S, F, T, O\}$ and $i \in \{-1, 1\}$; $\mathcal{I}_t = \{(O, -1), (A, -1)\} \cup \{0\}$ and $\mathcal{P}_t = \{R_t^1, R_t^2, R_t^3, R_t^4, R_t^5, R_t^6, R_t^7\}$.

In the remainder, by BFS tree, we mean the fragment which is being enlarged. Initially, a pre-distinguished node (the root) is labelled $(A, -1)$ i.e., active. All other nodes are labelled $(O, -1)$ i.e., outside the BFS tree. All edges are labelled 0. If an edge becomes labelled 1 then it is part of the tree. The A-labelled node acts as the initiator of a new iteration of the algorithm i.e., the construction of a new layer. The label $(S, 1)$ indicates that a node which is inside the fragment tree must broadcast some information (i.e., the construction of a new layer). In contrast, when a node is labelled $(S, -1)$ then it is waiting for the acknowledgment of its children. A node labelled F is a node that has finished the convergecast and is waiting for an order from its parent. Finally, when a node is labelled T then this node has locally terminated i.e., it can not contribute any more in the tree construction.

Rule R_t^1 initiates the computation of the BFS tree by adding the first layer. It also encodes the beginning of a new iteration of the algorithm. In fact, when the neighbors of the A-node become F (or T) labelled, then the A-node knows that a new layer has been added and the construction of a new layer can begin. Thus, the labels of all F-neighbors are set to $(S, 1)$ in order to begin the broadcast of this information up to the leaves of the BFS tree.

R_t^1 : **Beginning the construction of a new layer**

```
    Precondition :
    · $\lambda(v_0).E = A$                              /*the root node*/
    · $\forall v \in B(v_0,1)\ (\lambda(v).E \neq S)$   /*broadcast-convergecast finished*/
    · $\exists v_1 \in B(v_0,1)\ (\lambda(v_1).E \neq T)$  /*the computation is not over */
    Relabelling :
    · $\forall v \in B(v_0,1)\ (\lambda(v).E \neq T \implies (\lambda'(v) := (S,1), \lambda'(v_0,v) := 1))$
```

If an $(S, 1)$-labelled node u is in the interior of the BFS tree, then it just informs its children by setting their labels to $(S, 1)$ and it becomes $(S, -1)$ labelled (Rule R_t^2). Otherwise, if a u is a leaf, then either there exist some not yet marked O-neighbors in which case these nodes are added to the tree and u becomes F-labelled (Rule R_t^3), or there are no new nodes to add. In this case, u becomes T-labelled: terminated state (Rule R_t^4).

R_t^2 : **Broadcast**

Precondition :
· $\lambda(v_0) = (S, 1)$ `/*broadcast in progress*/`
· $\exists v_1 \in B(v_0, 1)$ $(\lambda(v_0, v_1) = 1 \wedge \lambda(v_1).E = F)$ `/*children are not informed*/`
Relabelling :
· $\lambda'(v_0) := (S, -1)$
· $\forall v \in B(v_0, 1)$ $((\lambda(v).E = F \wedge \lambda(v_0, v) = 1) \implies \lambda'(v) := (S, 1))$

R_t^3 : **Construction of the next layer**

Precondition :
· $\lambda(v_0) = (S, 1)$
· $\exists v_1 \in B(v_0, 1)$ $(\lambda(v_1).E = O)$ `/*some neighbors are not in the tree*/`
Relabelling :
· $\lambda'(v_0) := (F, -1)$
· $\forall v \in B(v_0, 1)$ $(\lambda(v).E := O \implies (\lambda(v) := (F, -1), \lambda'(v_0, v) := 1))$

R_t^4 : **No nodes to add to the next layer**

Precondition :
· $\lambda(v_0) = (S, 1)$
· $\forall v \in B(v_0, 1)(\lambda(v).E \neq O)$ `/*all neighbors are in the tree*/`
· $\exists!\, v_1 \in B(v_0, 1)$ $(\lambda(v_0, v_1) = 1)$ `/*`v_0` is a leaf*/`
Relabelling :
· $\lambda'(v_0) := (T, -1)$

After the broadcast step in Rule R_t^2, a node u which becomes $(S, -1)$-labelled waits for the acknowledgment of its children. When these children become F-labelled, then u knows that the new layer that corresponds to the last broadcast has been added by the leaves of the subtree rooted at it. Thus, it becomes $(F, -1)$-labelled in order to inform its parent (Rule R_t^5). Note that if all the children of u become T-labelled, then u knows that no new nodes can be added in the subtree rooted at it. Thus, it becomes T-labelled (Rule R_t^6).

R_t^5 : **Convergecast: Waiting for the next broadcast**

Precondition :
· $\lambda(v_0) = (S, -1)$
· $\exists!\, v_1 \in B(v_0, 1)$ $((\lambda(v_1).E = S \vee \lambda(v_1).E = A) \wedge \lambda(v_0, v_1) = 1)$
`/*all children have received an acknowledgment*/`
· $\exists v_2 \in B(v_0, 1)$ $(\lambda(v_2).E = F \wedge \lambda(v_0, v_2) = 1)$
`/*some children have not yet terminated the algorithm*/`
Relabelling :
· $\lambda'(v_0) := (F, -1)$

R_t^6 : **Convergecast: The tree construction is locally finished**

Precondition :
· $\lambda(v_0) = (S, -1)$
· $\exists!\, v_1 \in B(v_0, 1)\ ((\lambda(v_1).E = S \vee \lambda(v_1).E = A) \wedge \lambda(v_0, v_1) = 1)$
· $\forall v \in B(v_0, 1)\ ((v \neq v_1 \wedge \lambda(v_0, v) = 1) \implies \lambda(v).E = T)$
Relabelling :
· $\lambda'(v_0) := (T, -1)$

The construction of the BFS tree is terminated when all the neighbors of the A-labelled node become T-labelled. In fact, this means that there is no new layer to add: all the nodes of the graph are in the tree. In this case, the A-labelled node becomes T-labelled (Rule R_t^7). Note that at this stage of the algorithm, only the A-labelled node detects the global termination of the algorithm.

R_t^7 : **Termination detection**

Precondition :
· $\lambda(v_0).E = A$
· $\forall v \in B(v_0, 1)\ (\lambda(v).E = T)$
Relabelling :
· $\lambda'(v_0).E = T$

3.2 Global function computation

In many distributed algorithms, the convergecast and the broadcast are used in order to compute some functions of the graph. Suppose for instance that we want a source node to compute a global function $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$ where X_v is an input stored in each node v. Suppose that f verifies the following properties (we adopt the same notations as in [13] page 36) :

- f is well-defined for any subset of the inputs.
- f is associative and commutative.

In the following, we assume that we have a precomputed spanning tree (obtained for example by using the relabelling system $\mathcal{R}_t$). Such a function f, also called *semigroup function*, can be computed in a distributed manner by performing a convergecast process. In fact, $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$ can be computed using the relabelling system $\mathcal{R}_f = (\mathcal{L}_f, \mathcal{I}_f, \mathcal{P}_f)$ defined by: $\mathcal{L}_f = \{E \times X \times Y\} \cup \{0, 1\}$, where $E \in \{S, F, T\}$, X an input of the function f, $\mathcal{I}_f = \{S\} \cup \{0, 1\}$ and $\mathcal{P}_f = \{R_f^1, R_f^2\}$. Initially, all the nodes are labelled S and an edge with label 1 is part of the precomputed spanning tree.

By *local value of f* in rule R_f^1, we mean the value of f computed on the subtree T_v rooted at the node v that executes the rule R_f^1. The variable Y_1 in rule R_f^1 contains the value of f applied to all the entries X_{v_i} with $v_i \in T_v$ and $v_i \neq v$. The local value of f computed by v will enable the parent u of v to compute its own local value of f. Each time a node applies Rule R_f^1 it becomes F-labelled which means that it has finished to compute the local value of f. At the end of the process, only one node with label S remains. This node then applies rule R_f^2 and it computes the global value $f(X_{v_1}, X_{v_2}, ..., X_{v_n})$. Note that the convergecast process used here does not end at a pre-distinguished node but at some node which is elected at random depending on the algorithm execution. However, the two rules must be executed on disjoint stars that do not overlapp.

R_f^1 : **Convergecast: Computation of the local value of** f

Precondition :
- $\lambda(v_0).E = S$
- $\exists!\, v_1 \in B(v_0, 1)\ (\lambda(v_1).E = S \wedge \lambda(v_0, v_1) = 1)$

Relabelling :
- $Y_1 := f(\cup_{v_i \in B(v_0,1)(\lambda(v_i).E = F \wedge \lambda(v_0,v_i)=1)} \lambda(v_i).Y)$.
- $\lambda'(v_0).E := F$
- $\lambda'(v_0).Y := f(\lambda(v_0).X, Y_1)$

R_f^2 : **Convergecast: Global computation of** f **and termination detection**

Precondition :
- $\lambda(v_0).E = S$
- $\forall v \in B(v_0, 1)\ (\lambda(v).E = F)$

Relabelling :
- $Y_1 := f(\cup_{v_i \in B(v_0,1)(\lambda(v_0,v_i)=1)} \lambda(v_i).Y)$.
- $\lambda'(v_0).E := T$
- $\lambda'(v_0).Y := f(\lambda(v_0).X, Y_1)$

Using these two simple rules, we can encode in a formal way some classical distributed algorithms. For example, to compute the maximum of values stored by the network nodes, we just take $f := max$; to compute the sum over the node inputs, we take $f := +$. This method can also be used to derive distributed algorithms for computing some logical functions. For example, let X_v be a variable set to 1 if some predicate $Pred(v) = true$ and 0 otherwise. Suppose that we want to design a distributed algorithm to determine if the predicate $\exists v Pred(v)$ holds in the network (i.e., : there exists some node v with $Pred(v) = true$). This can be done by letting f be the logical or operator. The predicate $\forall v Pred(v)$ can also be computed distributively be setting $f := and$.

4 Distributed minimum spanning tree: Prim's algorithm

4.1 Preliminaries

In this section, we focus on the distributed construction of a minimum spanning tree. This problem is of special interest because the classical distributed algorithms for solving it use the basic techniques that we have described in previous sections as basic procedures. Our main motivation is to show that by using relabelling systems, we can design such a sophisticated algorithm in a detailed and comprehensive way.

```
Input: a weighted graph $G_{\mathcal{W}} = (V, E)$.
    Step 1: Initially, set $E_T$ (a set of edges) empty.
            Set $V_T$ (a set of nodes) empty.
    Step 2: Select an arbitrary node in $V$, and add it to $V_T$.
    Step 3: Find the lowest weight edge $e = (u, v)$ such that
            $u \in V_T$ but $v \notin V_T$. Add $v$ to $V_T$, and $e$ to $E_T$.
    Step 4: Repeat $Step3$ until $V_T$ equals $V$.
Output: A minimum spanning tree $T = (V, E_T)$.
```

Fig. 3. Prim's Algorithm

Recall that given a weighted graph $G_{\mathcal{W}}$, the MST problem consists in computing a spanning tree T such that the sum of the weights of the edges of T is the minimum over all possible spanning trees of $G_{\mathcal{W}}$. The problem has been heavily studied and many features of the MST problem, such as the distributed computability of such a tree or the time complexity for constructing it, were studied under many assumptions in past works. In this paper, we assume that the edge weights are unique, real and positive. Under this assumption, it is well known that there exists a unique minimum spanning tree of $G_{\mathcal{W}}$ (see [5,15,13] and references there).

One of the most basic algorithms for computing such a MST is the Prim's algorithm [5,15,13] (see Figure 3). Starting from one node, this algorithm consists in growing a fragment by adding at each iteration the minimum outgoing edge (MOE for short) to this fragment. The correctness of the algorithm relies on the fact that, at each iteration, the constructed fragment is part of the MST.

The classical distributed implementation of this algorithm consists of many phases, each one consists of two stages. In the first stage, the nodes in a fragment cooperate to compute the weight of the MOE. This is performed using a convergecast in the already computed fragment. The second stage consists in adding the MOE which is performed by broadcasting the weight of the MOE to all nodes in the fragment. When learning about the weight of the MOE, a node either adds the new edge to the fragment (if the MOE is incident to it) or re-initialize its state in order to begin another phase. The main difficulty here is to combine many broadcast and convergecast operations with the MOE computation. In the following, we give the relabelling system which encodes this MST algorithm.

By combining $\mathcal{R}_t$ and $\mathcal{R}_{f=min}$, Prim's algorithm can be encoded by the graph relabelling system $\mathcal{R}_m$=($\mathcal{L}_m$,$\mathcal{I}_m$,$\mathcal{P}_m$) defined by: $\mathcal{L}_m$=$\{E \times w_{subtree} \times w_{local} \times i\} \cup \{0,1\}$ where $E \in \{S, F, T, O\}$, $i \in \{-1, 1\}$ and $(w_{subtree}, w_{local}) \in \mathbb{R}^2_+ \cup \bot$; $\mathcal{I}_m = \{(O, \bot, \bot, -1), (S, \bot, \bot, -1)\} \cup \{0\}$ and $\mathcal{P}_m$=$\{R^1_m, R^2_m, R^3_m, R^4_m, R^5_m\}$. Note that, if the value of attribute $w_{subtree}$ (or w_{local}) is equal to $\bot$, then this value has not been set yet.

Initially, there is a distinguished node with label $(S, \bot, \bot, -1)$ which is the first node in the fragment. All other nodes are labelled $(O, \bot, \bot, -1)$. As for the relabelling system $\mathcal{R}_t$, if the value of the attribute i is equal to 1 (resp. -1) then an S-labelled node knows that it is in the broadcast (resp. convergecast) stage. At the beginning, all edges are labelled 0. If an edge becomes labelled 1 then it is part of the tree.

4.2 Computing the weight of the MOE: convergecast

The nodes of the fragment have to cooperate in order to compute the MOE i.e., convergecast from the leaves of the fragment up to an elected node (rule R_m^1). Each node must compute the attributes $w_{subtree}$ which is the weight of the minimum outgoing edge of the subtree rooted at it. Note that, during the convergecast, each node also stores the attribute w_{local} which is the weight of incident edges that connect it to nodes with label O. This will serve in the broadcast stage to find and to add the MOE of the whole fragment.

R_m^1 : **Computing the minimum outgoing edge**

```
Precondition :
· $\lambda(v_0) = (S, \perp, \perp, -1)$                    /*convergecast stage*/
· $\forall v \in B(v_0,1)((\lambda(v).E = S \wedge \lambda(v_0,v) = 1) \Rightarrow \lambda(v).i = -1)$
· $\exists!\, v_1 \in B(v_0,1)\ (\lambda(v_1).E = S \wedge \lambda(v_0,v_1) = 1)$
  /*all children have received an acknowledgment */
Relabelling :
· $w := min\{\{\mathcal{W}(v_0,v) \mid v \in B(v_0,1)(\lambda(v).E = O)\} \cup \{+\infty\}\}$ /*local MOE*/
· $w_{min} := min\{\{\lambda(v).w_{subtree} \mid v \in B(v_0,1)(\lambda(v_0,v) = 1 \wedge \lambda(v).E = F)\} \cup \{w\}\}$
  /*the MOE of the subtree rooted at $v_0$*/
· $(w_{min} = +\infty) \Rightarrow \lambda'(v_0).E := T$            /*local termination*/
· $(w_{min} \neq +\infty) \Rightarrow \lambda'(v_0) := (F, w_{min}, w, -1)$
```

At the end of the convergecast, the weight w_{min} of the MOE is computed at some elected node (rule R_m^2). This node sets its label to $(S, w_{min}, w, 1)$ in order to begin the broadcast phase. (Note also that rule R_m^2 also enables to initialize the MST construction).

R_m^2 : **Election of a node**

```
Precondition :
· $\lambda(v_0) = (S, \perp, \perp, -1)$
· $\forall v \in B(v_0,1)(\lambda(v_0,v) = 1 \implies \lambda(v).E \neq S)$
Relabelling :
· $w := min\{\{\mathcal{W}(v_0,v) \mid v \in B(v_0,1)(\lambda(v).E = O)\} \cup \{+\infty\}\}$
· $w_{min} := min\{\{\lambda(v).w_{subtree} \mid v \in B(v_0,1)(\lambda(v_0,v) = 1 \wedge \lambda(v).E = F)\} \cup \{w\}\}$
· $(w_{min} = +\infty) \Rightarrow \lambda'(v_0).E := T$
· $(w_{min} \neq +\infty) \Rightarrow \lambda'(v_0) := (S, w_{min}, w, 1)$
```

Rules R_m^1 and R_m^2 also allow to detect the termination of the MST construction. In fact, if the weight of the MOE is equal to $+\infty$ then there is no node with label O at the frontier of the fragment and thus all the nodes of the graph are in the fragment.

4.3 Finding and adding the MST: broadcast

At the end of the convergecast process (rule R_m^2), there is an elected node with label $(S, w, w', 1)$ that begins the broadcast (attribute i is equal to 1). Thus, a node u with label $(S, w, w', 1)$ first compares w and w'. If w=w' then the MOE is incident to u itself. Thus, the minimum edge is added and u sets its variable i to -1 in order to reinitialize the computation of another minimum edge (Rule R_m^4). Otherwise, if $w \neq w'$ then there must exist a neighbor with label $(F, w, w'', -1)$ from whom u has inherited its w value and the MOE must be in the subtree rooted at that neighbor. Thus, the F-labelled neighbor becomes $(S, w, w'', 1)$-labelled (Rule R_m^3). The other F-labelled children become $(S, \perp, \perp, 1)$-labelled in order re-initialize the computation of a new MOE (Rule R_m^5).

R_m^3 : **Broadcast the weight of the MOE**

Precondition :
- $\lambda(v_0) = (S, w, w', 1)$
- $w \neq w' \wedge (w \neq +\infty) \wedge (w \neq \perp)$
- $\exists!\, v_1 \in B(v_0, 1)\ (\lambda(v_0, v_1) = 1 \wedge \lambda(v_1).w_{subtree} = w)$
  `/*weights are unique*/`

Relabelling :
- $\lambda'(v_0) := (S, \perp, \perp, -1)$
- $\lambda'(v_1).E := S, \lambda'(v_1).i := 1$
- $\forall v \in B(v_0, 1)\ ((v \neq v_1 \wedge \lambda(v_0, v) = 1 \wedge \lambda(v).E = F) \Rightarrow \lambda'(v) := (S, \perp, \perp, 1))$

R_m^4 : **Adding the MOE**

Precondition :
- $\lambda(v_0) = (S, w, w, 1)$
- $(w \neq +\infty) \wedge (w \neq \perp)$
- $\exists!\, v_1 \in B(v_0, 1)\ (\lambda(v_0, v_1) = 1 \wedge \lambda(v_1).E = O \wedge \mathcal{W}(v_0, v_1) = w)$

Relabelling :
- $\lambda'(v_0) := (S, \perp, \perp, -1)$
- $\forall v \in B(v_0, 1)\ ((v \neq v_1 \wedge \lambda(v_0, v) = 1 \wedge \lambda(v).E = F) \Rightarrow \lambda'(v) := (S, \perp, \perp, 1))$
- $\lambda'(v_1) := (S, \perp, \perp, -1)$
- $\lambda'(v_0, v_1) := 1$

R_m^5 : **Reinitialization**

Precondition :
- $\lambda(v_0) = (S, \perp, \perp, 1)$ `/*the MOE is not in the subtree rooted at` v_0`*/`

Relabelling :
- $\lambda'(v_0) = (S, \perp, \perp, -1)$
- $\forall v \in B(v_0, 1)\ ((\lambda(v).E = F \wedge \lambda(v_0, v) = 1) \Rightarrow \lambda'(v) := (S, \perp, \perp, 1))$

5 Concluding Remarks

In this paper, we give a general technique that provides a modular construction of a large class of distributed computing algorithms. By exploiting this modular construction and the properties of graph relabelling systems, we obtain a general and a unified framework for expressing, proving and implementing distributed algorithms. The expressiveness has been clearly demonstrated by the numerous algorithms described in the paper. However, some other features concerning the proof techniques and the impelmentation issues remains to be studied in future work.

In fact, in order to prove the correctness of a graph relabelling system, that is the correctness of the algorithm encoded by such a system, it is useful to exhibit *(i)* some *invariant properties* associated with the system (*i.e.*, some properties of the graph labelling that is satisfied by the initial labelling and that is preserved by the application of every relabelling rule) and *(ii)* some properties of irreducible graphs [11]. The correctness of the algorithms given in this paper can be formally proven using that technique. Nevertheless, because our algorithms are clearly expressed as a combination of some few basic procedures, proving these basic procedures allows us to get basic building blocks which can be used as a bottelneck for proving the more sophisticated algorithms. Our aim is to give more generic algorithms allowing to automatically combine the basic techniques presented in this paper by using some logical functions $\mathcal{F}$ to be formally defined in future work. This will allow to automatically derive the relabelling system $\mathcal{A}_{\mathcal{F}}$ corresponding to a distributed algorithm A expressed in term of some basic procedures (convergcast, broadcast, PIF, etc.). By the same way, by using the properties of $\mathcal{F}$ together with the proofs of these basic procedures, we hope to develop new modular techniques that help proving the correcteness of the relabelling system $\mathcal{A}_{\mathcal{F}}$.

In addition to be formal, provable and tractable, the relabelling systems given in this paper can be translated in practical distributed algorithms in the message passing model. In fact, a new language called *Lidia* has been developped in [12] in order to automatically transform a given relabelling system in an executable distributed program using the *Visidia* [6,1,2] platform (i.e., a software tool for the simulation and the visualization of distributed algorithms in the message passing model). Furthermore, the distributed model studied in [3] combined with ideas from this paper will enable to translate a distributed algorithm expressed in a message passing model in a more formal and theoritical framework. This will enable to verify and to debug existing sophisticated algorithms which are in general hard to validate.

Acknowledgement

We would like to thank J. Chalopin and Y. Métivier for helpful remarks and suggestions.

References

[1] Bauderon, M., Y. Métivier, M. Mosbah and A. Sellami, *From local computations to asynchronous message passing systems*, Technical Report RR-1271-02, LaBRI

(2002).
URL http://www.labri.fr/visidia/

[2] Bauderon, M. and M. Mosbah, *A unified framework for designing, implementing and visualizing distributed algorithms*, International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02) (2002).
URL http://www.elsevier.nl/locate/entcs/volume72.html

[3] Chalopin, J. and Y. Métivier, *A bridge between the asynchronous message passing model and local computations in graphs*, 30th Int. Symp. on Mathematical Foundations of Computer Science **LNCS, to appear** (2005).

[4] Chang, E. J. H., *Echo algorithms: Depth parallel operations on general graphs.*, IEEE Trans. Software Eng. **8** (1982), pp. 391–401.

[5] Cormen, T. H., C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," MIT Press/McGraw-Hill, Cambridge, MA, 1990.

[6] Derbel, B. and M. Mosbah, *Distributing the execution of a distributed algorithm over a network*, 7th IEEE International Conference on Information Visualization, IV03-AGT. (16-18 July 2003, London), pp. 485–490.

[7] Godard, E., Y. Métivier and A. Muscholl, *Characterizations of classes of graphs recognizable by local computations*, Theory of Computing Systems **37:2** (2004), pp. 249–293.

[8] Litovsky, I., Y. Métivier and E. Sopena, *Different local controls for graph relabelling systems*, Math. Syst. Theory **28** (1995), pp. 41–65.

[9] Litovsky, I., Y. Métivier and E. Sopena, *Graph relabelling systems and distributed algorithms*, , **3**, H. Ehrig and H.J. Kreowski and U. Montanari and G. Rozenberg, World Scientific, 1999 pp. 1–56.

[10] Lynch, N. A., "Distributed Algorithms," Morgan Kaufmann Publishers, Inc., 1996.

[11] Métivier, Y., M. Mosbah and A. Sellami, *Proving distributed algorithms by graph relabeling systems: Examples of trees in networks with processor identities*, in: *Applied Graph Transformations*, Grenoble, 2002, pp. 45–57.

[12] Mosbah, M. and R. Ossamy, *A programming language for local computations in graphs: Computational completeness*, in: IEEE, editor, 5^{th} *Mexican Int. Conference in Computer Science Colima Mexico 20-24 September* (2004), pp. 12–19.

[13] Peleg, D., "Distributed Computing, A Locality-Sensitive Approach," SIAM Monographs on Discrete Mathematics and Applications, 2000.

[14] Segall, A., *Distributed network protocols.*, IEEE Transactions on Information Theory **29** (1983), pp. 23–34.

[15] Tel, G., "Introduction to distributed algorithms," Cambridge University Press, 2000.

(This page intentionally left blank)

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).

Key words: 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 (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.

[⋆] Work partly supported by the EU within the project HPRN-CT-2002-00275 SegraVis (*Syntactic and Semantic Integration of Visual Modelling Techniques*).

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

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

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

There are several approaches to the verification of spatial properties, on logics either for process calculi (see e.g. [2,4,3] 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 proposes our algorithm for the verification of (closed) spatial formulae, briefly discussing its computational costs. The final section outlines future research avenues. Due to space constraints, (sketches of) the proofs are included in an appendix.

2 The π-calculus and a Spatial Logic

2.1 Synchronous (finite) π-calculus

We now introduce the finite, sum-free fragment of 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*

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

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 (reduction semantics) *The* reduction relation *for processes is the 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*

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

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

The first rule denotes the communication between two processes: Process $\bar{a}c.Q$ 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. 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 (commitment semantics) *Let* $\Lambda = \{\tau\} \uplus \Delta$ *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*

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

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

Example 2.5 *Let us consider the process* ***race*** $\equiv (\nu a)\bar{b}a.\bar{a}a \mid b(d).\bar{d}c$*. 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*** $\xrightarrow{\tau} (\nu a)(\bar{a}a \mid \bar{a}c)$*. The residual process is deadlocked, since the restriction forbids* a *to be observed. Removing the restriction results in a process that may perform commitments* $\bar{a}a \mid \bar{a}c \xrightarrow{\bar{a}a} \bar{a}c$ *(*a *sent over* a*) and* $\bar{a}a \mid \bar{a}c \xrightarrow{\bar{a}c} \bar{a}a$ *(*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 $x, y, \ldots$, and let $\Xi = \Lambda \cup \{\overline{x}y, x(y) \mid x, y \in V\}$ be the set of* observables*, ranged over by ξ. A* spatial formula *is a term generated by the syntax*

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

where $\eta \in V \uplus \mathcal{N}$. We let $\phi, \phi_1, \ldots$ 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 $Иx.\phi$ 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 $\texttt{ffn}(\phi)$, is defined in the obvious way, since the only binding operators are the name quantifiers. A name is *fresh* with respect to 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*

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

$$[\![\phi_1|\phi_2]\!] = \{P \mid \exists P_1, P_2.P \equiv P_1|P_2 \textit{ and } P_1 \in [\![\phi_1]\!] \textit{ and } P_2 \in [\![\phi_2]\!]\}$$

$$[\![\langle\lambda\rangle\phi]\!] = \{P \mid \exists Q.P \xrightarrow{\lambda} Q \textit{ and } Q \in [\![\phi]\!]\}$$

In addition to the usual abbreviations, we shall use the hidden name quantifier ($\texttt{H}x.\phi \equiv Иx.x\circledR\phi$) 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*

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

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* $P \in \;_{a \notin (\mathtt{fn}(P) \cup \mathtt{ffn}(\phi))} [\![\phi\{^a/_x\}]\!]$.

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

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 has to be considered.

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 $\{a_1, \dots, a_n\} \subseteq \;_j \mathtt{fn}(P_j)$.*

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

Proposition 2.11 (revelation set) *Let P be a process and a®ϕ 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®ϕ holds in 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 $(\nu N)\mathcal{Q}$ 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 directly corresponds to 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 each edge $e \in E$, the arity of $l_e(e)$ is $(l_n(s(e)), l_n^*(t(e)))$, i.e., each edge preserves the arity of its label.*

With an abuse of notation, in the definition above we let l_n^* stand for the extension of the function l_n from nodes to strings of nodes; sometimes, we use l as a shorthand for l_n and l_e. In the following, we denote the components of a graph d by N_d, E_d, l_d, s_d and t_d, dropping the subscript whenever clear.

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 3.2 (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 $f_d : d \to d'$ 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 3.3 (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 clearly independent of the choice of the representative, up-to isomorphism. Moreover, the operators then become associative.

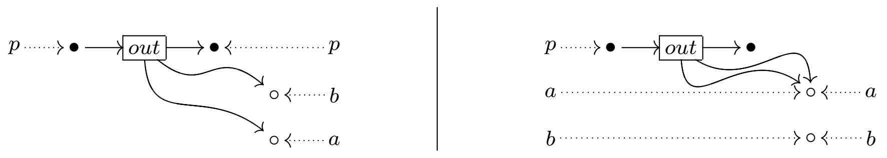

Fig. 1. Ranked graphs $out_{b,a}$ (left) and $\llbracket \bar{a}a \rrbracket \otimes id_{\{a,b\}}$ (right).

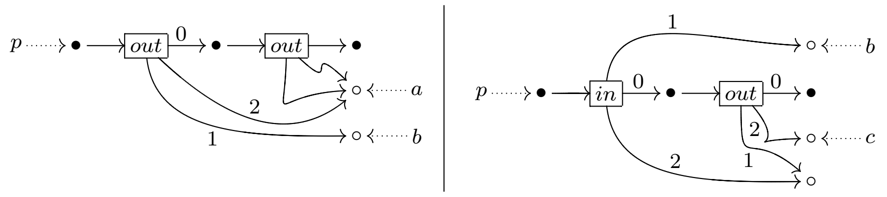

Fig. 2. Ranked graphs $out_{b,a} \circ (\llbracket \bar{a}a \rrbracket \otimes id_{\{a,b\}})$ (left) and $\llbracket b(d).\bar{d}c \rrbracket$ (right).

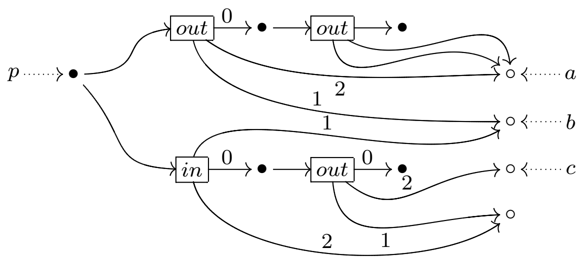

Fig. 3. The ranked graph $\llbracket \bar{b}a.\bar{a}a \rrbracket \otimes \llbracket b(d).\bar{d}c \rrbracket$.

Example 3.4 (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 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 occurrences of these operators are defined for the rank of the sub-expressions, according to Definition 3.3: Its rank is inductively computed and its *value* is the graph obtained by evaluating its operators.

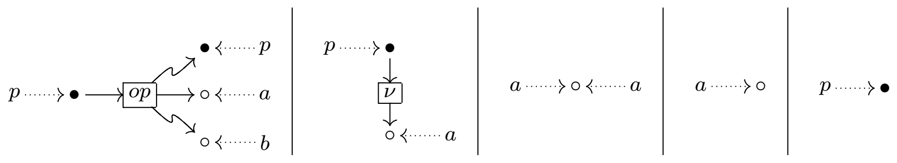

Fig. 4. Ranked graphs $op_{a,b}$ (for $op \in \{in, out\}$), ν_a, id_a, $\mathtt{0}_a$ and $\mathtt{0}_p$.

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 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 *in* and *out* simulate the input and output prefixes, respectively; and operator ν stands for restriction. Furthermore, please note that there is instead no explicit operator accounting 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 $\ _{x \in \Gamma}\, id_x$, for a set Γ of names (since the ordering is immaterial). Finally, 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*

$$\llbracket (\nu a)P \rrbracket = \begin{cases} \llbracket P \rrbracket & \text{if } a \notin \mathtt{fn}(P) \\ (\llbracket P \rrbracket \otimes \nu_a) \circ (\mathtt{0}_a \otimes id_{\mathtt{fn}(P)\setminus\{a\}}) & \text{otherwise} \end{cases}$$

$$\llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \otimes \llbracket Q \rrbracket$$

$$\llbracket \mathtt{0} \rrbracket = \mathtt{0}_p$$

$$\llbracket \overline{a}b.P \rrbracket = out_{a,b} \circ (\llbracket P \rrbracket \otimes id_{\{a,b\}})$$

$$\llbracket a(b).P \rrbracket = in_{a,b} \circ (\llbracket P \rrbracket \otimes id_{\{a,b\}}) \circ (\mathtt{0}_b \otimes id_{\mathtt{fn}(P)\setminus\{b\}})$$

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 $(\{p\}, \mathtt{fn}(P))$.

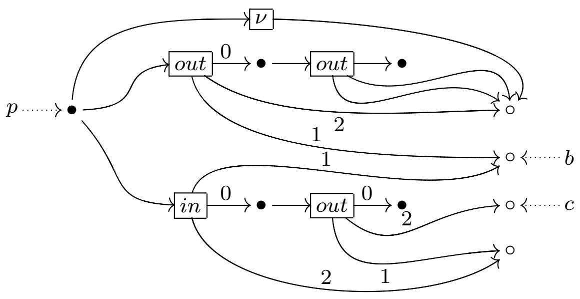

Fig. 5. The ranked graph $\llbracket (\nu a)\bar{b}a.\bar{a}a \mid b(d).\bar{d}c \rrbracket$.

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 graphical representation is depicted in Figure 2 (left)*

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

The denotation of $(\llbracket \bar{a}a \rrbracket \otimes \mathrm{id}_{\{a,b\}})$ *coincides with* $(out_{a,a} \otimes \mathrm{id}_{\{a,b\}}) \circ (\mathsf{0}_p \otimes id_{\{a,b\}})$*, 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 image of any process. Nevertheless, let us assume that we restrict our attention to processes verifying a simple 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 $P \in \llbracket \phi \rrbracket$, *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 checking `void` just consists on determining 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 $l_d(e) \neq \nu\}$;
    $R \leftarrow \{(n, e) \in N_d \times E_d \mid s_d(e) = r(p) \wedge l_d(e) = \nu \wedge t_d(e) = n\}$;
    foreach $E_1 \in 2^E$ 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 $(n, e) \in R$ 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 correspond 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 other 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 are defined in the appendix. They 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 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 let a be a name neither in d_v nor in $\texttt{ffn}(\phi)$, 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 $d_v \cup \texttt{ffn}(\phi)$;
    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.

```
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
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e1)[0]=t_d(e2)[0], t_d(e1)[2]=t_d(e2)[2]\}} \setminus \{e1, e2\}$;
                $G_1 \leftarrow gc(G_1)$;
                if eval$(G_1, \phi)$ 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
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e)[0]\}} \setminus \{e\}$;  $G_1 \leftarrow gc(G_1)$;
                if eval$(G_1, \phi)$ 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
                $G_1 \leftarrow G$;  $d_1 \leftarrow d_{\{r(p)=t_d(e)[0]\}} \setminus \{e\}$;  $d_{v_1} \leftarrow d_v \cup \{b\}$;
                $v_1 \leftarrow v \cup \{b \mapsto t_d(e)[2]\}$;  $G_1 \leftarrow gc(G_1)$;
                if eval$(G_1, \phi)$ 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.

Note that, even if not explicitly stated, the occurrence of labels as $x(x)$ in a formula is forbidden and the algorithm returns *false* whenever $a(a)$ is met.

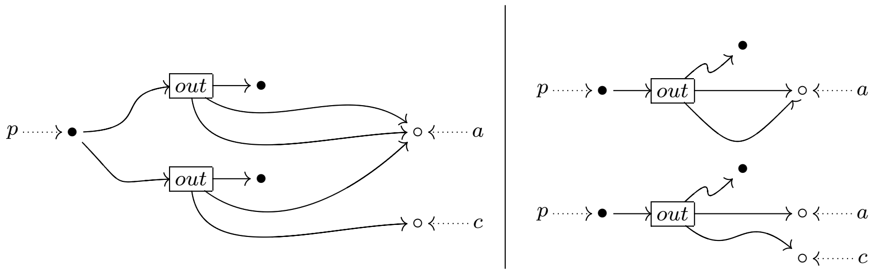

Fig. 6. The ranked graph $\llbracket \overline{a}a \mid \overline{a}c \rrbracket$ (left) and two sub-ranked graphs (right).

Checking Revelation ($a®\phi$). 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
        $G_1 \leftarrow G$;  $E_{d_1} \leftarrow E_d \setminus \{e\}$;  $d_{v_1} \leftarrow d_v \cup \{a\}$;  $v_1 \leftarrow v \cup \{a \mapsto t_d(e)[0]\}$;
        if eval$(G_1, \phi)$ then return true;
    return false;
```

Example 5.1 *Does* ***race*** $\equiv (\nu a)\overline{b}a.\overline{a}a \mid b(d).\overline{d}c$ *satisfy the property* ***crash*** $\equiv$ $\text{И}x.\exists y.\exists z.y \neq z \wedge \langle\tau\rangle(\langle\overline{x}y\rangle T \mid \langle\overline{x}z\rangle T)$*? The algorithm will first try and fix x as a fresh name (say a) and try to reveal it as one of the restricted names in* $\llbracket$***race***$\rrbracket$*. Thus, x is revealed as 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 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* eval$(\llbracket P \rrbracket, \phi) =$ 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 without major effects.

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 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 restricts to 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 main efforts are going to focus 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 and F. Trigueiro Ruiz *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 (2005).

[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 and G. Engels *et alii*, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, II: Applications, Languages and Tools*, Theoretical Computer Science **2**, 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*, University of Southampton Technical Reports **DSSE–TR–2003–2** (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.

A Technicalities and Proofs

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 $(\nu N)\mathcal{Q} \in [\![\phi]\!]$.*

Proof. By definition, a process $P \in [\![a®\phi]\!]$ iff $\exists P'.P \equiv (\nu a)P'$ and $P' \in [\![\phi]\!]$. Thus, this clearly implies that $a \notin \mathtt{fn}(P)$. So, let us choose such a process P', and let us assume that $a \notin \mathtt{fn}(P')$: Then, $P \equiv P'$. Otherwise, a normal form $(\nu M)\mathcal{Q}$ can be obtained for P', such that $a \notin M$, since structural congruence preserves free variables. And since structural congruence also preserve satisfiability, then $P \equiv (\nu a)(\nu M)\mathcal{Q}$ with $(\nu M)\mathcal{Q} \in [\![\phi]\!]$. □

Next we present the definition of ranked graph generated by a set of edges.

Definition A.1 (generated graph) *Let $G = \{p\} \overset{r}{\Rightarrow} d \overset{v}{\Leftarrow} d_v$ be a ranked graph, and $E \subseteq E_d$ a set of edges of its underlying graph with source $r(p)$. Then, the* sub-ranked graph *of G generated by E is the ranked graph $H = \{p\} \overset{r1}{\Rightarrow} d_1 \overset{v1}{\Leftarrow} d_{v_1}$, for d_1 the smallest graph containing E and $r(p)$ and satisfying*

- $\forall e \in E_d : e \in E_{d_1} \Rightarrow \forall i.t_d(e)[i] \in N_{d_1}$
- $\forall e \in E_d : s_d(e) \in N_{d_1} \setminus r(p) \Rightarrow e \in E_{d_1}$
- $\forall n \in d_v : v(n) \in N_{d_1} \Rightarrow n \in d_{v_1}$

where all the derived functions are obviously defined by restriction.

The definition is well-given, since H clearly is a ranked graph.

In order to show the correctness of the algorithms we need some additional lemmas. The first provides a set-theoretical characterization for those ranked graphs that are the encoding of a process (recall that $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ is not surjective, and there are some graphs of rank $(\{p\}, \Gamma)$ that are not image of any process.

Lemma A.2 (encoded process) *Let $G = \{p\} \Rightarrow d \Leftarrow \Gamma$ be a ranked graph. Then, there exists a process P with $\Gamma = \mathtt{fn}(P)$ such that $G = \lfloor\!\lfloor P \rfloor\!\rfloor$ iff*

(i) *d is a connected hyper-tree with $r(p)$ as root;*

(ii) *leaves of sort s_p are in the target on exactly one edge;*

(iii) *d has no useless restriction edge;*

(iv) *no variable in Γ is mapped to a bound node in d;*

(v) *no bound node in d is bounded more than once;*

(vi) *every free node of d is the image of a variable in Γ;*

(vii) *if e bounds a node name u and u is in the target of an edge e' then there is a path of length 1 or more from $s_d(e)$ to $s_d(e')$.*

where a node is bound if it is the first argument of an edge labeled by a restriction or the second argument of an input operator, and free otherwise.

Proof. Suppose $G = \llbracket P \rrbracket$ for some P. It is easy to see that the encoding only delivers ranked graphs that are connected trees and that satisfy exactly the conditions listed above.

If G satisfies the above conditions, then we define P as $[G]^g$

$$[G]^g = \begin{cases} (\nu t(e)[0])[G/\{e\}]^g & \text{if } \exists e \in E_d.l(e) = \nu \text{ and } r(p) = s(e) \\ \overline{t(e)[1]}t(e)[2].[G_1]^g \mid [G \setminus G_1]^g & \text{if } \exists e \in E_d.l(e) = out \text{ and } r(p) = s(e) \\ t(e)[1](t(e)[2]).[G_1]^g \mid [G \setminus G_1]^g & \text{if } \exists e \in E_d.l(e) = in \text{ and } r(p) = s(e) \\ \mathsf{0} & \text{otherwise} \end{cases}$$

where G_1 is the sub-ranked graph of G generated by a node e with $r(p) = t(e)[0]$ after its removal, i.e. the sub-ranked graph corresponding to the continuation of the operation represented by e. Clearly, the definition is ambiguous but all possibilities of choosing e deliver structurally congruent processes.

Moreover, it can be shown that $\llbracket \cdot \rrbracket$ and $[\cdot]^g$ are mutually inverse. □

The next lemma states that every ranked graph created during the evaluation of a closed formula corresponds to a process.

Lemma A.3 (algorithm sub-calls) *Let P be a process and ϕ a closed formula. Then, for every sub-call* eval(G, ϕ') *of* eval$(\llbracket P \rrbracket, \phi)$ *there exists a process Q such that $G = \llbracket Q \rrbracket$.*

The proof of the lemma is a straightforward check that every ranked graph considered during the evaluation satisfies the conditions of Lemma A.2.

Theorem 5.2 (algorithm soundness) *Let P be a process and ϕ a closed formula. Then, $P \in \llbracket \phi \rrbracket$ iff* eval$(\llbracket P \rrbracket, \phi) = $ true.

Proof. The proof is by induction on P and ϕ distinguishing the different cases for ϕ. Booleans and name equality are trivial.

If ϕ is `void` observe that if $P \in \llbracket \texttt{void} \rrbracket$ then $P \equiv \mathbf{0}$. Clearly, the set of edges in $\llbracket \mathbf{0} \rrbracket$ is empty. On the other hand, if we have a ranked graph $G = \llbracket P \rrbracket$ for some P such that E_d is empty, then P can only be the empty process and thus $P \in \llbracket \texttt{void} \rrbracket$.

If ϕ is $\phi_1 \mid \phi_2$ first observe that every edge in G is necessarily either in G_1 or G_2. By Lemma A.3, $G_1 = \llbracket Q \rrbracket$ and $G_2 = \llbracket R \rrbracket$ for some processes $Q, R \in \mathcal{P}$. It is easy to see that $G = G_1 \otimes G_2$, hence $P \equiv Q \mid R$. Applying induction we have that $eval(P, \phi_1 \mid \phi_2)$ implies $P \in \llbracket \phi_1 \mid \phi_2 \rrbracket$.

To show the opposite direction assume that $P \in [\![\phi_1 \mid \phi_2]\!]$. This implies that there are two processes Q, R such that $P \equiv Q \mid R$ with $Q \in [\![\phi_1]\!]$ and $R \in [\![\phi_2]\!]$. Suppose that $\mathit{eval}(P, \phi_1 \mid \phi_2)$ returns *false*. Since we assume the induction hypothesis to hold, the only possibility is that the pair of ranked graphs $\llfloor Q \rrfloor$, $\llfloor R \rrfloor$ is missed by the algorithm. Let E_Q,E_R respectively be the edges of the graphs of $\llfloor Q \rrfloor$ and $\llfloor R \rrfloor$. Since G can be seen as $\llfloor Q \rrfloor \otimes \llfloor R \rrfloor$, the ranked graphs generated by E_Q,E_R could not be missed. Thus, $\mathit{eval}(P, \phi_1 \mid \phi_2)$ returns *true*.

If ϕ is $a \circledR \phi_1$ we use Proposition 2.11. Since $d_v = \mathtt{fn}(P)$ checking whether $n \notin d_v$ and checking whether $n \notin \mathtt{fn}(P)$ is the same. By induction the first sub-call in the procedure is correct. Finally, observe that each of the ranked graphs G_1 considered by the algorithms is the encoding of one the processes $(\nu M)Q$ for $a \notin M$.

If ϕ is $\exists x.\phi_1$ observe that $d_v = \mathtt{fn}(P)$. Hence a sub-call $\mathit{eval}(G, \phi_1\{^a/_x\})$ returns *true* exactly when $P \in [\![\phi\{^a/_x\}]\!]$ for some $a \in \mathtt{fn}(P) \cup \mathtt{ffn}(\phi)$. Applying induction for the last call of the procedure and by Proposition 2.9 we obtain the desired result.

If ϕ is $\text{И}x.\phi_1$ we recall again that $d_v = \mathtt{fn}(P)$ and thus the sub-call $\mathit{eval}(G, \phi_1\{^a/_x\})$ returns *true* exactly when $P \in [\![\phi\{^a/_x\}]\!]$ for the fresh name a. By Proposition 2.9 we obtain the desired result.

If ϕ is $\langle\lambda\rangle\phi_1$ suppose that λ is τ and that $\mathit{eval}(\llfloor P \rrfloor, \phi)$ is true. This implies that the algorithm finds a ranked graph $\llfloor Q \rrfloor$ for some Q such that $\mathit{eval}(\llfloor Q \rrfloor, \phi_1)$. It is easy to see that this generation corresponds to a synchronization $P \xrightarrow{\tau} Q$. Hence, we have that $P \in [\![\phi]\!]$. The opposite direction is similar since the algorithm can not miss any synchronization. The other cases for λ are similar. $\square$

(This page intentionally left blank)

Hoare vs Milner: comparing synchronizations in a graphical framework with mobility [⋆]

Ivan Lanese [1] Ugo Montanari [2]

Computer Science Department, University of Pisa, Pisa, Italy

Abstract

We compare the expressive power of *Hoare* (i.e., CSP style) and *Milner* (i.e., CCS style) *synchronizations* for defining *graph transformations* in a framework where edges can perform actions on adjacent nodes to synchronize their evolutions. Furthermore, nodes can be communicated and merged. We show that the expressive powers of the two synchronization models are different, but no one is greater than the other. Finally, we show that in many interesting cases the behaviour of a synchronization model can be mimicked by the other one using suitable translations for the rewritten graphs.

Key words: graph transformations, Hoare synchronization, Milner synchronization, Synchronized Hyperedge Replacement, mobility, expressiveness.

1 Introduction

A fundamental aspect of many modern distributed systems is *synchronization*, i.e., how different components of the system can coordinate their behaviour in order to reach a common goal. Clearly, synchronization can be performed in different ways. This has emerged since the beginning of computational models for interacting systems: while CCS [11] used the so called *Milner synchronization*, where two processes interact by performing complementary actions, CSP [6] used *Hoare synchronization* where all the processes must synchronize by performing the same action.

We are interested in comparing these two synchronization models, but in a setting which is more complex than the original one. We work in the framework of *Synchronized Hyperedge Replacement* (SHR) [2,5,3], a *graph transformation* formalism aimed at representing distributed interacting systems. In

⋆ Research supported by the Project FET-GC II Sensoria.

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

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

particular, we model system components as hyperedges and communication channels as shared nodes. System evolution is specified by *productions*, i.e., rules that describe the evolution of single hyperedges. Productions are synchronized by performing *actions* on adjacent nodes, and a set of productions can be executed concurrently only if the actions performed on each node are compatible. Compatible here means that they must synchronize using a given synchronization model. While SHR can be used with any synchronization model [9], here we are interested in comparing the Hoare and the Milner models. In addition, we consider *mobility* of nodes: references to nodes can be sent together with actions, and when actions are synchronized corresponding nodes are merged. As far as Milner synchronization model is concerned, this is the style of mobility used in Fusion Calculus [12] as pointed out in [7].

We will compare these two synchronization models from the point of view of which classes of reconfigurations they can specify, in three important cases: (i) one-step reconfigurations, (ii) reconfigurations specified by maximal (i.e., where no transition is possible from the final graph) computations and (iii) reconfigurations specified by any possible computation.

We will prove the following original results:

(i) the expressiveness of Hoare and Milner synchronization models are incomparable for all the above defined classes of reconfigurations;

(ii) the expressiveness of Milner synchronization is greater than the one of Hoare synchronization for graphs with no interface to the environment where each node is shared by exactly two edges, since Milner synchronization is asymmetric;

(iii) Hoare synchronization can be implemented using Milner synchronization and a suitable translation for graphs;

(iv) the encoding approach used in proving (iii) can not be used in the opposite direction, since it would require to force interleaving in a distributed structure.

Structure of the paper.
§ 2 introduces Synchronized Hyperedge Replacement and the Hoare and Milner synchronization models. In § 3 we define the formal setting for comparing the models. The comparison is carried out in § 4. The case of closed graphs with nodes shared by exactly two edges is analyzed in § 5. § 6 deals with the problem of implementing one model using the other one. Finally, conclusions and traces for future work are presented in § 7.

2 Synchronized Hyperedge Replacement

In this section we present Synchronized Hyperedge Replacement (SHR) [2] and, in particular, *the Hoare and the Milner synchronization models*, but first we introduce some mathematical notation.

Mathematical notation. Given a syntactic structure t (e.g., a term, a set of terms, an equation), we denote with $t\sigma$ the application of substitution σ to t (in a capture-avoiding way if t contains binders). The operator $|-|$ computes the number of elements in a vector or in a set. Given a set S, $\wp(S)$ is its powerset and S^* is the set of strings on alphabet S. Given a function f, we denote with $\mathrm{dom}(f)$ its domain and with $f|_S$ its restriction to the new domain S. Finally, when we use set operators on functions and substitutions, we refer to their representation as sets of pairs.

SHR [2] is an approach to (hyper)graph transformation that defines *global transitions* using *local productions*. Productions define how a single (hyper)edge can be rewritten and the *conditions* that this rewriting imposes on adjacent nodes. Thus the global transition is obtained by applying in parallel different productions whose conditions are compatible. What exactly compatible means depends on which synchronization model is used. In this work we will use both the Hoare and the Milner synchronization models. The former requires that all the edges connected to a node execute the same action on it. The latter requires two edges to interact by performing complementary actions while the others stay idle. For a general definition of synchronization models see [9].

We use the extension of SHR with *mobility* [5,3], that allows edges to send node references together with actions, and nodes whose references are matched during synchronization are unified.

We will give a formal description of SHR as labelled transition system, but first of all we need an algebraic representation for graphs.

An edge is an atomic item with a label and with as many ordered tentacles as the rank $\mathrm{rank}(L)$ of its label L. A graph is composed by a set of nodes and a set of such edges, and each edge is connected, by its tentacles, to its attachment nodes. A graph is connected to its environment by an interface which is a subset of its nodes. Nodes in the interface are called free nodes, while other nodes are called bound (or restricted). We will consider graphs up to isomorphisms that preserve free nodes, labels of edges, and connections between edges and nodes. We denote with *Graphs* the set of such graphs.

Now, we present a definition of graphs as syntactic judgements, where nodes correspond to names, free nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where the x_i are arbitrary names and $\mathrm{rank}(L) = n$. Also, *nil* represents the graph with no edges, $|$ is the parallel composition of graphs (merging nodes with the same name) and νy is a declaration of a bound node y.

Definition 2.1 (Graphs as judgements) *Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A* judgement *is of the form $\Gamma \vdash G$ where:*

(i) *$\Gamma \subseteq \mathcal{N}$ is a finite set of names (the free nodes of the graph);*

(ii) *G is a term generated by the grammar*

$$\text{(AG1) } (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3) \quad \text{(AG2) } G_1|G_2 \equiv G_2|G_1 \quad \text{(AG3) } G|nil \equiv G$$
$$\text{(AG4) } \nu x\ \nu y\ G \equiv \nu y\ \nu x\ G \quad \text{(AG5) } \nu x\ G \equiv G \text{ if } x \notin \mathrm{fn}(G)$$
$$\text{(AG6) } \nu x\ G \equiv \nu y\ G\{y/x\} \text{ if } y \notin \mathrm{fn}(G)$$
$$\text{(AG7) } \nu x\ (G_1|G_2) \equiv (\nu x\ G_1)|G_2 \text{ if } x \notin \mathrm{fn}(G_2)$$

Table 1
Structural congruence for graph terms.

$G ::= L(\boldsymbol{x}) \mid G|G \mid \nu y\ G \mid nil$
where $\boldsymbol{x}$ is a vector of names, L is an edge label with $\mathrm{rank}(L) = |\boldsymbol{x}|$ *and y is a name.*

We define the restriction operator ν as a binder. We denote with fn *the function that given a term G returns the set* $\mathrm{fn}(G)$ *of free names in G. We demand that* $\mathrm{fn}(G) \subseteq \Gamma$.

When defining the interfaces, we use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$ and Γ_1, Γ_2 to denote the union of Γ_1 and Γ_2, assuming $\Gamma_1 \cap \Gamma_2 = \emptyset$.

Graph terms are considered up to the axioms of structural congruence in Table 1. As far as judgements are concerned, we define $\Gamma \vdash G \equiv \Gamma' \vdash G'$ iff $\Gamma = \Gamma'$ and $G \equiv G'$.

Axioms (AG1), (AG2) and (AG3) define respectively the associativity, commutativity and identity over *nil* for operation $|$. Axioms (AG4) and (AG5) state that nodes of a graph can be restricted only once and in any order. Axiom (AG6) defines α-conversion of a graph w.r.t its bound names. Axiom (AG7) defines the interaction between restriction and parallel composition.

Note that function fn is well-defined on equivalence classes.

Judgements up to structural axioms are isomorphic to graphs up to isomorphisms. For a formal statement of the correspondence see [4].

We present now the steps of an SHR computation.

Definition 2.2 (SHR transition) *Let Act be a set of actions, and given $a \in Act$ let* $\mathrm{ar}(a)$ *be its arity. A* SHR transition *is of the form:*

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$$

where $\Gamma \vdash G$ and $\Phi \vdash G'$ are judgements for graphs, $\Lambda : \Gamma \to (Act \times \mathcal{N}^*)$ is a total function and $\pi : \Gamma \to \Gamma$ is an idempotent substitution. Function Λ assigns to each node x the action $a \in Act$ and the vector $\boldsymbol{y}$ of node references exposed on x by the transition (in a more message-passing view, we say that node references are sent to x). If $\Lambda(x) = (a, \boldsymbol{y})$ then we define $\mathrm{act}_\Lambda(x) = a$ *and* $\mathrm{n}_\Lambda(x) = \boldsymbol{y}$. *We require that* $\mathrm{ar}(\mathrm{act}_\Lambda(x)) = |\,\mathrm{n}_\Lambda(x)|$.

We define:

- $\mathrm{n}(\Lambda) = \{z | \exists x. z \in \mathrm{n}_\Lambda(x)\}$ *set of exposed names;*

- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$ *set of exposed fresh names.*

Substitution π allows to merge nodes. Since π is idempotent, it maps every node into a standard representative of its equivalence class. We require that $\forall x \in \mathrm{n}(\Lambda).x\pi = x$, i.e., only references to representatives can be exposed. Furthermore we require $\Phi = \Gamma\pi \cup \Gamma_\Lambda$, namely free nodes are never erased ($\supseteq$) and new nodes are bound unless exposed ($\subseteq$).

Note that the set of free names Φ of the resulting graph is fully determined by Λ and π (since $\Gamma = \mathrm{dom}(\Lambda)$). When writing Λ as set of pairs we write the triple $(x, a, \boldsymbol{y})$ for the pair $(x, (a, \boldsymbol{y}))$.

SHR transitions are derived from basic productions using suitable sets of inference rules.

Definition 2.3 (Production)
A production *is an SHR transition of the form:*
$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$
where all x_i, $i = 1, \ldots, n$ are distinct.

We suppose to have for each edge label L of arity n a special idle production $x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda_\epsilon, id} x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n)$ where $\Lambda_\epsilon(x_i) = (\epsilon, \langle\rangle)$ for each i (ϵ is a special “idle” action with $\mathrm{ar}(\epsilon) = 0$). Idle productions are included in all sets of productions, which are also closed w.r.t. α-conversion of names in $\{x_1, \ldots, x_n\} \cup \Phi$.

We present now the set of inference rules for Hoare synchronization. The intuitive idea of Hoare synchronization is that all the edges connected to a node must expose the same action on that node.

Definition 2.4 (Rules for Hoare synchronization)

$$\textit{(par)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \qquad \Gamma' \vdash G_1' \xrightarrow{\Lambda',\pi'} \Phi' \vdash G_2'}{\Gamma, \Gamma' \vdash G_1|G_1' \xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'} \Phi, \Phi' \vdash G_2|G_2'}$$

where $(\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset$.

$$\textit{(merge)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma\sigma \vdash G_1\sigma \xrightarrow{\Lambda',\pi'} \Phi' \vdash G_2\sigma\rho}$$

where $\sigma : \Gamma \to \Gamma$ is an idempotent substitution and:

(i) $\forall x, y \in \Gamma.x\sigma = y\sigma \Rightarrow \mathrm{act}_\Lambda(x) = \mathrm{act}_\Lambda(y)$

(ii) $\rho = \mathrm{mgu}(\{(\mathrm{n}_\Lambda(x))\sigma = (\mathrm{n}_\Lambda(y))\sigma | x\sigma = y\sigma\} \cup \{x\sigma = y\sigma | x\pi = y\pi\})$ *where we choose names in $\Gamma\sigma$ as representatives whenever possible*

(iii) $\forall z \in \Gamma.\Lambda'(z\sigma) = (\Lambda(z))\sigma\rho$

(iv) $\pi' = \rho|_{\Gamma\sigma}$

$$(res)\quad \frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \Phi' \vdash \nu Z\ G_2}$$

where:

(v) $(\exists y \in \Gamma . x\pi = y\pi) \Rightarrow x\pi \neq x$

(vi) $Z = (\{x\} \cup \mathrm{n}(\Lambda)) \setminus \mathrm{n}(\Lambda|_\Gamma)$

$$(new)\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma, x \vdash G_1 \xrightarrow{\Lambda\cup\{(x,a,\boldsymbol{y})\},\pi} \Phi' \vdash G_2}$$

where $x \notin \Gamma \cup \Phi$ and $\boldsymbol{y} \cap (\Gamma \cup \Phi \cup \{x\}) = \emptyset$.

A transition is obtained by composing productions, which are first applied on disconnected edges. Composition is performed by merging nodes and thus connecting the edges. Finally, nodes can be bound. In particular, rule (par) deals with the composition of transitions which have disjoint sets of nodes and rule (merge) allows to merge nodes (note that σ is a projection into representatives of equivalence classes). Condition (i) requires that we have the same action on merged nodes. Condition (ii) defines the most general unifier ρ of the union of two sets of equations: the first set identifies (the representatives of) the tuples associated to nodes merged by σ, while the second set of equations adds previous merges traced by π. Thus ρ is the merge resulting from both π and σ. Note that (iii) Λ is updated with these merges and that (iv) π' is ρ restricted to the nodes of the graph which is the source of the transition. Rule (res) binds node x, guaranteeing that x is not a representative if it belongs to a non trivial equivalence class and binding also all the nodes that were extruded on node x in the starting transition. Rule (new) allows adding to the source graph an isolated node where arbitrary actions (with fresh names) are performed.

We write $\mathcal{P} \Vdash_H \Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ if $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ can be obtained from the productions in $\mathcal{P}$ using Hoare inference rules.

A similar set of rules can be defined also for Milner synchronization.

Definition 2.5 (Rules for Milner synchronization)

$$(par)\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \qquad \Gamma' \vdash G_1' \xrightarrow{\Lambda',\pi'} \Phi' \vdash G_2'}{\Gamma, \Gamma' \vdash G_1|G_1' \xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'} \Phi, \Phi' \vdash G_2|G_2'}$$

where $(\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset$.

$$(merge)\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma\sigma \vdash G_1\sigma \xrightarrow{\Lambda',\pi'} \Phi' \vdash \nu U\ G_2\sigma\rho}$$

where $\sigma : \Gamma \to \Gamma$ *is an idempotent substitution and:*

(i) $\forall x, y \in \Gamma. x\sigma = y\sigma \wedge \mathrm{act}_\Lambda(x) \neq \epsilon \wedge \mathrm{act}_\Lambda(y) \neq \epsilon \wedge x \neq y \Rightarrow$
$(\forall z \in \mathcal{N} \setminus \{x, y\}. z\sigma = x\sigma \Rightarrow \mathrm{act}_\Lambda(z) = \epsilon) \wedge$
$\mathrm{act}_\Lambda(x) = a \wedge \mathrm{act}_\Lambda(y) = \overline{a} \wedge a \neq \tau$

(ii) $\rho = \mathrm{mgu}(\{(\mathrm{n}_\Lambda(x))\sigma = (\mathrm{n}_\Lambda(y))\sigma | x\sigma = y\sigma\} \cup \{x\sigma = y\sigma | x\pi = y\pi\})$ *where we choose names in* $\Gamma\sigma$ *as representatives whenever possible*

(iii) $\Lambda'(z) = \begin{cases} (\tau, \langle\rangle) & \textit{if } x\sigma = y\sigma = z \wedge x \neq y \wedge \mathrm{act}_\Lambda(x), \mathrm{act}_\Lambda(y) \neq \epsilon \\ (\Lambda(x))\sigma\rho & \textit{if } x\sigma = z \wedge \mathrm{act}_\Lambda(x) \neq \epsilon \\ (\epsilon, \langle\rangle) & \textit{otherwise} \end{cases}$

(iv) $\pi' = \rho|_{\Gamma\sigma}$

(v) $U = (\Phi\sigma\rho) \setminus \Phi'$

$$\textit{(res)} \quad \frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \Phi' \vdash \nu Z\ G_2}$$

where:

(vi) $(\exists y \in \Gamma. x\pi = y\pi) \Rightarrow x\pi \neq x$

(vii) $\mathrm{act}_\Lambda(x) = \epsilon \vee \mathrm{act}_\Lambda(x) = \tau$

(viii) $Z = \{x\}$ *if* $x \notin \mathrm{n}(\Lambda|_\Gamma), Z = \emptyset$ *otherwise*

$$\textit{(new)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma, x \vdash G_1 \xrightarrow{\Lambda\cup\{(x,\epsilon,\langle\rangle)\},\pi} \Phi, x \vdash G_2}$$

where $x \notin \Gamma \cup \Phi$.

Rules for Milner synchronization suppose that actions can be normal actions a (representing input) or coactions $\overline{a}$ (representing "output"). We also assume $\overline{\overline{a}} = a$. Furthermore we have the two special actions ϵ and τ (completed synchronization) of arity 0.

Rules are similar to the ones for Hoare synchronization. The main differences are that in rule (merge) during action synchronization (i) we require to have (at most) two complementary non ϵ actions, and their composition is τ. Thus we may have to reintroduce restrictions (v) if some nodes were extruded by the synchronized actions. In rule (res), just nodes x where ϵ or τ actions are performed can be restricted, and since these actions have arity 0 only node x may have to be restricted in the final graph. Finally, in rule (new) only action ϵ is allowed on the newly created node.

We write $\mathcal{P} \Vdash_M \Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ if $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ can be obtained from the productions in $\mathcal{P}$ using Milner inference rules. We drop the subscript M or H from $\Vdash$ when we refer to an unspecified synchronization model.

A *SHR computation* is a sequence of SHR transitions such that for each i the final graph of transition i is the starting graph of transition $i+1$. A SHR computation is called *trivial* if the starting graph is equal to the final graph.

3 Expressiveness measures

We want to study the expressiveness of the Hoare and Milner synchronization models in the SHR framework. Different measures of expressiveness can be useful, according to which is the intended use of the model. In our case, we are mainly interested in using graph transformation to express reconfigurations of the topology of distributed systems, thus the main point is which is the class of reconfigurations that can be expressed by a set of productions together with a synchronization model.

Formally, we define *reconfigurations* as functions $r : \mathit{Graphs} \rightarrow \wp(\mathit{Graphs})$.

Intuitively, the *behaviour* of a set of productions $\mathcal{P}$ on a graph G w.r.t. a synchronization model S is the set of graphs that are the results of "suitable" computations starting from G. The choice of which computations are "suitable" determines the observable behaviour of the system.

Definition 3.1 (Behaviour function)
The function $C\text{-}\mathrm{behav}^S(\mathcal{P})(G)$ is the function that computes the set of graphs reachable from graph G using computations in the class C obtained from the productions in $\mathcal{P}$ using synchronization model S.

Thus we can say that the C_1-expressiveness of synchronization model S_1 is greater than the C_2-expressiveness of a synchronization model S_2, written as $(S_1, C_1) \geq (S_2, C_2)$ if there exists a function f from sets of productions to sets of productions such that for each set of productions $\mathcal{P}$ and for all graphs G we have that $C_2\text{-}\mathrm{behav}^{S_2}(\mathcal{P})(G) = C_1\text{-}\mathrm{behav}^{S_1}(f(\mathcal{P}))(G)$.

We will consider three different choices for C:

1 one-step computations;

max maximal computations (i.e., computations whose final state does not allow further non trivial transitions);

all all possible computations.

If a synchronization model S_2 is not as expressive as a synchronization model S_1, we can try to simulate reconfigurations of S_1 using reconfigurations of S_2 by translating the graph G (this will be done formally in § 5 and § 6).

4 The expressiveness of Hoare and Milner synchronizations are not comparable

In this section we show that the expressive power of Hoare and Milner synchronization models are different, but no one is greater than the other, independently of the class of computations used.

We first need an auxiliary definition.

Definition 4.1 (Monotonicity) *We define the following partial order on transitions:* $\Gamma \cup \Gamma' \vdash G_1|G' \xrightarrow{\Lambda',\pi} \Phi \cup \Gamma'\pi \vdash G_2|G'\pi$ *is greater than* $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi}$

$\Phi \vdash G_2$ iff $\Gamma' \vdash G'$ is a graph and $\Lambda = \Lambda'|_\Gamma$. A SHR system is monotone iff the set of derivable transitions is upward-closed (i.e., for each set of productions $\mathcal{P}$, if a transition is derivable, then all the greater transitions are derivable too).

Intuitively, in a monotone system we can always add to the graph an additional part which stays idle.

Proposition 4.2 *Milner SHR is monotone.*

Intuitively, this happens because Milner synchronization involves exactly two participants. The same is not true for Hoare synchronization, because there is a universal quantification on the participants connected to the node where the synchronization is performed.

Since the monotonicity property can be extended from transitions to general computations, each set of Milner computations must be upward-closed too. Using that, we can prove the following theorem.

Theorem 4.3 *$(Milner, C_1) \not\geq (Hoare, C_2)$ for each $C_1 \in \{1, all\}$ and each $C_2 \in \{1, max, all\}$.*

Proof. Let us consider the set of productions $\mathcal{P}$ generated by the only production:

$$x \vdash d(x) \xrightarrow{(x,a,\langle\rangle)} x \vdash d'(x)$$

For $C_2 \in \{1, all\}$ we have that:
C_2- behav$^H(\mathcal{P})(x \vdash d(x)) = \{x \vdash d(x), x \vdash d'(x)\}$
C_2- behav$^H(\mathcal{P})(x \vdash d(x)|d(x)) = \{x \vdash d(x)|d(x), x \vdash d'(x)|d'(x)\}$
while for $C_2 = max$ the behaviours do not contain the trivial reconfigurations (which are not maximal). Since, for each choice of C_2, behaviours are not monotone, the thesis follows from Proposition 4.2. □

The case of *max*-expressiveness requires a bit more work.

Theorem 4.4 *$(Milner, max) \not\geq (Hoare, C)$ for each $C \in \{1, max, all\}$.*

Proof. Let us consider the set of productions $\mathcal{P}$ generated by the only production:

$$x \vdash d(x) \xrightarrow{(x,a,\langle\rangle)} x \vdash nil$$

For each $C \in \{1, all\}$ we have that:
C- behav$^H(\mathcal{P})(x \vdash d(x)) = \{x \vdash d(x), x \vdash nil\}$
C- behav$^H(\mathcal{P})(x \vdash d(x)|d'(x)) = \{x \vdash d(x)|d'(x)\}$
while for $C = max$ we have not the trivial reconfiguration in the first case. Suppose that we can obtain this behaviour with Milner SHR. By monotonicity from the first case we have a transition from $x \vdash d(x)|d'(x)$ to $x \vdash d'(x)$. If $x \vdash d'(x)$ can not be rewritten then we have a contradiction, since it is not in the behaviour. Otherwise by monotonicity also $x \vdash d(x)|d'(x)$ can be rewritten and so it can not be in the behaviour for maximal computations, as it is. □

Now we consider the inverse problem, that is we prove that the expressiveness of Milner synchronization model can not be reached by Hoare SHR.

Notice that, if all nodes are free, Milner synchronization can not force productions to be executed together, i.e., each production can always be applied in isolation. Hence, restriction is fundamental for constraining the behaviour of components using Milner synchronization (and this does not surprise, since even in CCS restriction is necessary to reach Turing equivalence).

Notice that instead in Hoare SHR restriction just performs hiding of part of the observation, i.e., no transition can be forbidden by restriction. More formally, the following proposition holds.

Proposition 4.5 *Given a set of productions $\mathcal{P}$, if $\mathcal{P} \Vdash_H \Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ then $\mathcal{P} \Vdash_H \Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda',\pi'} \Phi' \vdash \nu Z\ G_2$ where Λ', π' and Φ' are subsets of Λ, π and Φ respectively, and $Z = \Phi \setminus \Phi'$.*

Theorem 4.6 $(Hoare, C_1) \not\geq (Milner, C_2)$ *for each* $C_1, C_2 \in \{1, max, all\}$.

Proof. Let us consider the set of productions $\mathcal{P}$ generated by the only production:

$$x \vdash d(x) \xrightarrow{(x,a,\langle\rangle)} x \vdash d'(x)$$

For each $C_2 \in \{1, all\}$ we have that:
C_2- behav$^M(\mathcal{P})(x \vdash d(x)) = \{x \vdash d(x), x \vdash d'(x)\}$
C_2- behav$^M(\mathcal{P})(\vdash \nu x\ d(x)) = \{\vdash \nu x\ d(x)\}$
while for $C_2 = max$ we have not the trivial reconfiguration in the first case. Since this behaviour does not satisfy Proposition 4.5, it can not be obtained by Hoare SHR (with any class of computations). □

5 Reconciling Hoare and Milner synchronizations

Until now we have shown that the expressiveness of Hoare and Milner SHR are quite different. We consider now a case where they become closer, i.e., when we consider closed graphs (i.e., graphs where all nodes are restricted) where each node is attached to exactly two tentacles. We call these graphs *closed 2-shared graphs*. Even if this case is quite simple, it shows some interesting features of the two synchronization models and it is a first step towards the more general results of next section.

The reconfigurations have to preserve the two invariants above. The invariant of having closed graphs is preserved automatically since new nodes are bound by default, and an extrusion can happen only if there is a free node on which it is performed.

The second condition is not preserved in general, but it can be enforced by constraining the allowed productions. Let us consider the application of a single production: when the rewritten edge is removed, all the nodes attached to it have one attached tentacle missing (two if the edge was connected two times to the same node). Thus when inserting the new graph, the same number of connections to those nodes must be provided, and two connections must be provided for each new node. Notice also that when merges are performed, two nodes with one connection each are merged into one with two connections, thus occurrences of nodes in Λ or in π count as new connections for that node.

Definition 5.1 *A production P is* connection-preserving *if for each node x the number of occurrences of x in the right hand side, plus the ones in Λ, plus the number of nodes that are merged with x by π equals the number of occurrences of x in the left hand side (that is, 1) if x occurs there, and it is 2 for new nodes.*

Proposition 5.2 *Let $\mathcal{P}$ be a set of connection-preserving productions and G a closed 2-shared graph. If $\mathcal{P} \Vdash G \xrightarrow{\Lambda,\pi} G'$ then G' is a closed 2-shared graph.*

Proof. By rule induction on the derivation. □

Thus from now on we consider only connection-preserving productions. We will show later that this kind of productions is expressive enough to simulate general Hoare transitions (via a translation of graphs).

Theorem 5.3 *For closed 2-shared graphs, $(Milner, C) \geq (Hoare, C)$ for each $C \in \{1, max, all\}$.*

Proof. The set of productions for Milner model can be obtained by replicating each Hoare production with all possible "orientations" of actions, i.e., any action a must be substituted by either a or $\overline{a}$ and a production is needed for each possible combination of choices. □

This proves that for closed 2-shared graphs, Hoare synchronization is equal to Milner synchronization where the distinction between actions and coactions is dropped. In that case, Milner synchronization is strictly more expressive than Hoare synchronization and the additional expressiveness is given exactly by the asymmetry, as shown by the following proposition (whether this result holds for *max* expressiveness is an open problem).

Proposition 5.4 *For closed 2-shared graphs, $(Hoare, C_1) \not\geq (Milner, C_2)$ for each $C_1, C_2 \in \{1, all\}$.*

Proof. Let us consider the graph $\vdash \nu x\ d(x)|d(x)$. This graph is symmetric. Using Hoare synchronization, for any choice of production for the left edge, the same production can always be applied in the same step also to the right edge, and the result is again a symmetric graph. Thus for each choice of productions, if a transition exists, then also a transition that preserves the symmetry exists.

Using Milner synchronization and the set of productions generated by:
$x \vdash d(x) \xrightarrow{(x,a,\langle\rangle)} x \vdash c(x)$
$x \vdash d(x) \xrightarrow{(x,\overline{a},\langle\rangle)} x \vdash d(x)$
we have just one non trivial allowed transition, with final graph $\vdash \nu x\ c(x)|d(x)$.

Notice that this is also the result of all the allowed non trivial computations. Since no symmetric graph is obtained, this reconfiguration can not be performed using Hoare SHR. $\square$

Thus we will consider a different form of simulation, that uses a translation for graphs. In particular, we define two functions $[\![-]\!], [\![-]\!]^{-1} : \mathit{Graphs} \to \mathit{Graphs}$ such that for each graph G we have $[\![[\![G]\!]]\!]^{-1} = G$ (but we may have $[\![[\![G]\!]^{-1}]\!] \neq G$).

We say that (C_1, S_1) can simulate (C_2, S_2) iff we have $C_2\text{-}\,\mathrm{behav}^{S_2}(\mathcal{P})(G) = [\![C_1\text{-}\,\mathrm{behav}^{S_1}(f(\mathcal{P}))([\![G]\!])]\!]^{-1}$, i.e., the result of a (C_2, S_2) reconfiguration can be obtained by translating the graph, reconfiguring it using (C_1, S_1) and translating it back again.

We use a translation based on the concept of amoeboid [8]: each node shared by n tentacles is translated into a graph called amoeboid with n external nodes. The inverse translation $[\![-]\!]^{-1}$ just removes the amoeboids and reinserts the nodes they stand for. In our case an amoeboid (which connects two nodes) for $[\![-]\!]$ is simply an edge $L(x, y)$, where L is a special label with productions of the following form for each action a of arity k:

$$x, y \vdash L(x, y) \xrightarrow{(x,a,\langle x_1,\ldots,x_k\rangle)(y,\overline{a},\langle y_1,\ldots,y_k\rangle)}$$
$$x, y, x_1, \ldots, x_k, y_1, \ldots, y_k \vdash L(x, y) | \coprod_{i=1,\ldots,k} L(x_i, y_i)$$

and where $\coprod_{i\in I} G_i$ is the parallel composition of graphs G_i for each $i \in I$. As far as $[\![-]\!]^{-1}$ is concerned, an amoeboid is any chain of such edges. The translation f of productions just drops π and connects each pair of nodes merged by π using an L edge.

The following theorem holds.

Theorem 5.5 *For each set generated by connection-preserving productions $\mathcal{P}$, each closed 2-shared graph G, and each class of computations $C \in \{1, max, all\}$ we have $C\text{-}\,\mathrm{behav}^M(\mathcal{P})(G) = [\![C\text{-}\,\mathrm{behav}^H(f(\mathcal{P}))([\![G]\!])]\!]^{-1}$.*

Proof. The result holds because the L edge allows on its nodes complementary synchronizations, and amoeboids to be merged are instead connected using L edges. The tricky part is that the chains of L edges that are created (and that are translated into nodes by the inverse translation) have always odd length, and this is exactly the condition required to have complementary actions on the two ends of the chain. Notice also that the productions for L edges are connection-preserving. $\square$

6 Dealing with general closed graphs

We want now to go back to the general case, at least as far as the number of tentacles attached to each node is concerned. In particular, we will show that by using a different kind of amoeboids, the general case can be reduced to the 2-shared one.

As far as Hoare synchronization is concerned, we want to use amoeboids that perform the broadcast of the action.

Those amoeboids are composed by edges H (for Hoare) of arity 3 and edges C (for closing) of arity 1 to deal with nodes with less than 3 attached tentacles. These edges have for each action (we consider as an example an action a of arity 2) productions of the form:

$$x,y,z \vdash H(x,y,z) \xrightarrow{(x,a,\langle x_1,x_2\rangle)(y,a,\langle y_1,y_2\rangle)(z,a,\langle z_1,z_2\rangle)} x,y,z,x_1,y_1,z_1,x_2,y_2,z_2 \vdash H(x,y,z)|H(x_1,y_1,z_1)|H(x_2,y_2,z_2)$$

$$x \vdash C(x) \xrightarrow{(x,a,\langle x_1,x_2\rangle)} x,x_1,x_2 \vdash C(x)|C(x_1)|C(x_2)$$

Such an amoeboid imposes the same action to be executed on each node and it creates a copy of itself for each set of corresponding names, that are in this way connected in the resulting graph.

An amoeboid used to connect a set of nodes S is any connected graph composed by H and C edges whose nodes in S are attached to just one tentacle while whose other nodes are 2-shared. Thus for each graph G, $[\![G]\!]$ is a 2 shared graph.

Analogously we have to translate productions in order to make them connection-preserving. This can be done by splitting nodes that are used too many times and connecting the different copies using H edges, while nodes that are used too few times must be closed using C edges. Also, π is dropped and the nodes to be merged are connected using amoeboids.

Example 6.1 Let us consider the following production, which is used in [5] to specify a reconfiguration from a ring graph to a star one:

$$x,y \vdash r(x,y) \xrightarrow{(x,r,\langle w\rangle)(y,r,\langle w\rangle)} x,y,w \vdash s(y,w)$$

In this production the name x is not used in the right hand side, whereas the name w is used 3 times (two times in Λ and one by edge s) while it does not occur in the left hand side. We can translate the production into:

$$x,y \vdash r(x,y) \xrightarrow{(x,r,\langle w_1\rangle)(y,r,\langle w_2\rangle)} x,y,w_1,w_2,w_3 \vdash C(x),s(y,w_3),H(w_1,w_2,w_3)$$

which is a connection-preserving production such that the inverse translation of the right hand side is the right hand side of the starting production (up to renaming of nodes).

By using for functions $[\![-]\!]$ and $[\![-]\!]^{-1}$ the new amoeboids, we have the following result.

Theorem 6.2 *For each set of productions $\mathcal{P}$, each closed graph G, and each class of computations $C \in \{1, max, all\}$ we have that $C\text{-}\mathrm{behav}^H(\mathcal{P})(G) = [\![C\text{-}\mathrm{behav}^H(f(\mathcal{P}))([\![G]\!])]\!]^{-1}$ where f performs the above described translation of productions.*

This result can be composed with Theorem 5.3 to get a translation from Hoare synchronization to Milner synchronization for any closed graph. To deal with general graphs, one just needs to trace which nodes are free. This can be done by adding to each amoeboid representing a free node an edge $ENV(x)$ representing a connection with the environment. Such an edge must allow any action and it must attach a copy of itself to each node it receives (to simulate the fact that a node sent on a free node is extruded), like the C edge does. Note that in this way we may get amoeboids with many connections to the environment. We can add productions to delete them if they are redundant, but there is no way to force these reconfigurations to be executed before the normal transitions.

Now we want to apply the same approach to Milner synchronization. Milner amoeboids are essentially routers that create a path from an action to the corresponding coaction.

We start by introducing an M (for Milner) edge of arity 3. We want the edge to perform complementary actions on any pair of its three attachment nodes. Thus we have a production of the form:

$$x, y, z \vdash M(x, y, z) \xrightarrow{(x,a,\langle s_1,s_2\rangle)(y,\overline{a},\langle s_1,s_2\rangle)(z,\epsilon,\langle\rangle)} x, y, z \vdash M(x, y, z)$$

Note that in Milner synchronization we always merge pairs of nodes, thus it is not necessary to replicate the amoeboid. We also have to use a different kind of edge for dealing with nodes shared by less than 3 tentacles, which we denote by I (for inactive). This edge has only the idle production. This guarantees that actions and coactions are performed only by edges from the original graph.

For productions we use the same kind of translation that we have used in the Hoare case, with the new edges for amoeboids.

However w.r.t. Hoare model we have here an additional problem: many independent synchronizations may be allowed inside an amoeboid during one transition, but this is not allowed in standard Milner synchronization. In particular, this occurs when the pairs of interacting nodes are connected by disjoint paths inside the amoeboid. Also, cycles in the amoeboid may cause new isolated nodes to be created, but these can be discarded by $[\![-]\!]^{-1}$.

Using the new definition for the translation functions, we have the following partial correctness result.

Theorem 6.3 *For each set of productions $\mathcal{P}$, each closed graph G, and each class of computations $C \in \{1, max, all\}$ we have that $C\text{-}\mathrm{behav}^M(\mathcal{P})(G) \subseteq [\![C\text{-}\mathrm{behav}^M(f(\mathcal{P}))([\![G]\!])]\!]^{-1}$ where f performs the usual translation of produc-*

tions into connection-preserving ones.

The other inclusion holds, e.g., for amoeboids connecting at most 3 nodes, since in that case we can have at most one synchronization. Notice that this theorem can be composed with Theorem 5.5 to have an implementation of Milner synchronization using Hoare synchronization. The composed translation has been used in [8] to map Fusion Calculus into logic programming.

We show now that the problem of guaranteeing interleaving inside amoeboids of the above seen kind can not be solved.

Theorem 6.4 *Let $\mathcal{G} \subseteq$ Graphs contain for each n at least a graph with n nodes in its interface and let it be closed w.r.t. composition of graphs by joining them via a node in the interface. Then the maximum k such that all $G \in \mathcal{G}$ allow only transitions where at most k actions on the interface are not ϵ, if it exists is 0.*

Proof. Suppose that such a k exists and it is not 0 and take a graph with more than k nodes in its interface, and a transition where k of the actions are not ϵ. Take a node where ϵ action is executed. By connecting two such graphs by merging these two nodes, we get a graph which allows at least $2k$ non ϵ actions on its interface. This gives a contradiction. □

This proves that we can not have a set of amoeboids for Milner synchronization (since this requires $k = 2$), since the closure property is needed to model mobility. Notice in fact that if we want to model reconfigurations without mobility we can use, e.g., amoeboids with a tree structure whose leaves are the interface and whose roots check that the resulting action is a τ. Using mobility, the tree shape can not be preserved.

Also in that case, free nodes can be managed using edges standing for connections to the environment.

7 Conclusion and future works

We have analyzed the expressive power of Hoare and Milner synchronizations in the SHR setting, proving that they are incomparable and that implementing one synchronization with the other is not a trivial task. Also, for Milner synchronization no fully satisfactory simulation can be obtained using the concept of amoeboid. Notice that no counterexample (but the last one) uses mobility, thus we have proved that the expressiveness is incomparable without mobility, and that adding mobility does not help to bridge the gap.

These results justify the idea of having different synchronization models available in the same framework in order to be able to use all of them without complex translations. Such an approach was used in process calculus ACP [1], and has been extended to deal with graph transformations and mobility in [9,10].

As future work we want to carry out a similar comparison among generic synchronization models as defined in [9]. Another issue is to consider not only the allowed reconfigurations, but also the labels of the transitions. Finally, the possibility of using maximal expressivity to break symmetry in Hoare synchronization must be further investigated (see discussion before Proposition 5.4).

References

[1] J.A. Bergstra, and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *Proc. of ICALP'84*, volume 172 of *LNCS*, pages 82–94. Springer, 1984.

[2] P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *Journal of the ACM*, 34(2):411–449, 1987.

[3] G. Ferrari, U. Montanari, and E. Tuosto. A LTS semantics of ambients via graph synchronization with mobility. In *Proc. of ICTCS'01*, volume 2202 of *LNCS*, pages 1–16. Springer, 2001.

[4] D. Hirsch. Graph transformation models for software architecture styles. PhD thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, U.B.A., 2003.

[5] D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*. Springer, 2001.

[6] C.A.R. Hoare. *CSP – Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[7] I. Lanese and U. Montanari. A graphical fusion calculus. In *Proc. of the Workshop of the COMETA Project on Computational Metamodels*, volume 104 of *ENTCS*, pages 199–215. Elsevier, 2004.

[8] I. Lanese and U. Montanari. Mapping fusion and synchronized hyperedge replacement into logic programming. *Theory and Practice of Logic Programming, Special Issue on Multiparadigm Languages and Constraint Programming*, 2004. To appear.

[9] I. Lanese and U. Montanari. Synchronization algebras with mobility for graph transformations. In *Proc. of FGUC'04 – Foundations of Global Ubiquitous Computing*, ENTCS, 2004. To appear.

[10] I. Lanese and E. Tuosto. Synchronized hyperedge replacement for heterogeneous systems. In *Proc. of COORDINATION'05*, volume 3454 of *LNCS*, pages 220–235. Springer, 2005.

[11] R. Milner. A calculus of communicating systems. In volume 92 of *LNCS*. Springer, 1989.

[12] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS '98*. IEEE, Computer Society Press, 1998.

Extending C for Checking Shape Safety

— Work in Progress —

Mike Dodds[1] and Detlef Plump[2]

The University of York, UK

Abstract

The project Safe Pointers by Graph Transformation at the University of York has developed a method for specifying the shape of pointer-data structures by graph reduction, and a static checking algorithm for proving the shape safety of graph transformation rules modelling operations on pointer structures. In this paper, we outline how to apply this approach to the C programming language. We extend ANSI C with so-called transformers which model graph transformation rules, and with shape specifications for pointer structures. For the resulting language C-GRS, we present both a translation to C and and an abstraction to graph transformation. Our main result is that the abstraction of transformers to graph transformation rules is correct in that the C code implementing transformers is compatible with the semantics of graph transformation.

Key words: Pointer programming; shape safety; C; graph transformation.

1 Introduction

Pointers in imperative programming languages are indispensable for the efficient implementation of many algorithms at both applications and systems level, but pointer programming is notoriously prone to undetected errors. This is because the type systems of current programming languages are too weak to detect ill-shaped pointer structures.

To improve this situation, the project Safe Pointers by Graph Transformation[3] (SPGT) at the University of York has developed a method to specify the intended shape of a family of pointer data-structures by *graph reduction specifications* (GRSs). A GRS consists of a signature of admissible node and edge labels, a set of graph reduction rules, and a so-called accepting graph.

[1] Email: `miked@cs.york.ac.uk`
[2] Email: `det@cs.york.ac.uk`
[3] `http://cs-people.bu.edu/bake/spgt/`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

The shape specified by a GRS contains all graphs that can be reduced to the accepting graph by some series of rule applications [1,3].

For example, Figure 1 shows a GRS for full binary trees with an auxiliary pointer. Tree nodes are either L-labelled leaves or B-labelled branch nodes with outgoing pointers l and r, and there is a unique R-labelled node with pointers *top* and *aux* which point to the root of the tree and to an arbitrary tree node, respectively. The accepting graph, *Acc*, is the smallest graph of this kind. The left reduction rule redirects the auxiliary pointer to the top of the tree (regardless of the labels of nodes 2 and 3), the right rule deletes two leaves and relabels their parent node as a leaf. Every full binary tree with an auxiliary pointer can be reduced to *Acc* by these two rules, but no other graph can be reduced to *Acc*.

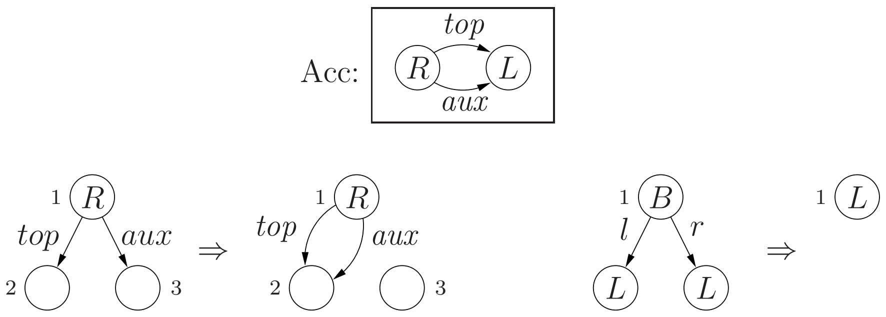

Fig. 1. Graph reduction specification of binary trees with an auxiliary pointer

Operations on pointer data-structures are also modelled by graph transformation rules. A static checking algorithm for proving that such operations are shape preserving is presented in [2] (generalizing a similar algorithm for context-free shapes given in [4,5]). Figure 6 shows an operation on the shape of Figure 1 that replaces a leaf destination of the auxiliary pointer with a branch node and two new leaves. This is an example of a shape preserving operation: when applied to a full binary tree with an auxiliary pointer, it will always produce a graph of the same shape.

In what follows, we outline how to apply the SPGT approach to the C programming language. The next section summarises how shapes are defined by graph reduction and sketches the checking algorithm for shape-preservation. Section 3 describes constructs which allow C programmers to write shape specifications and operations on shapes, Section 4 indicates how to translate the extended language—called C-GRS—to standard C, Section 5 discusses the correctness of an abstraction of C-GRS shape-specifications and operations to GRSs and graph transformation rules, and Section 6 concludes with a brief discussion of related work.

2 Safe Pointers by Graph Transformation

This sections summarises our method of specifying shapes [1,3] and briefly discusses the shape-checking method of [2].

A *graph* $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ consists of a finite set of nodes (or vertices) V_G, a finite set of edges E_G, functions $s_G, t_G\colon E_G \to V_G$ assigning a source and a target node to each edge, a partial node labelling function $l_G\colon V_G \to \mathcal{L}_V$, and an edge labelling function $m_G\colon E_G \to \mathcal{L}_E$. Graph G models a pointer-data structure by retaining only the pointer fields of records and abstracting from other values. Each node models a tagged record of pointers where the node label, drawn from the node-label alphabet $\mathcal{L}_V$, is the tag. Each edge leaving a node corresponds to a pointer field where the edge label, drawn from the edge-label alphabet $\mathcal{L}_E$, is the name of the pointer field. We use a function $\mathrm{type}\colon \mathcal{L}_V \to \wp(\mathcal{L}_E)$ to associate with each record tag its set of field names: if node v is labelled l and has an outgoing edge e, then the label of e must be in $\mathrm{type}(l)$ and no other edge leaving v must have this label. The triple $\Sigma = \langle \mathcal{L}_V, \mathcal{L}_E, \mathrm{type} \rangle$ is called a *signature* and graphs conforming to the above constraints are called *Σ-graphs*. A Σ-graph is *Σ-total* if every node v is labelled and for each label in $\mathrm{type}(l_G(v))$ there is an outgoing edge with that label. A *shape* is a set of Σ-total graphs. So shape members model pointer structures with no missing or dangling pointers.

A *graph morphism* $g\colon G \to H$ between Σ-graphs G and H consists of a node mapping $g_V\colon V_G \to V_H$ and an edge mapping $g_E\colon E_G \to E_H$ such that sources, targets and labels are preserved: $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g_V(v)) = l_G(v)$ for all nodes v where $l_G(v)$ is defined. Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. An *isomorphism* is a graph morphism that is injective and surjective in both components and maps unlabelled nodes to unlabelled nodes. If g is an isomorphism then G and H are *isomorphic*, denoted by $G \cong H$.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of three Σ-graphs L, K and R, and inclusions $K \to L$ and $K \to R$. Graph K is the *interface* of r. Intuitively, a rule deletes the nodes and edges in $L - K$, preserves those in K and allocates those in $R - K$. Our pictures of rules show only the left- and right-hand graphs, the interface always consists just of the numbered nodes of the left- and right-hand graphs. Σ-graphs in rules need not be Σ-total, they can contain nodes with an incomplete set of outgoing edges or unlabelled nodes with no outgoing edges. We refer to [1] for conditions on unlabelled nodes and outgoing edges in rules which ensure that rule applications preserve both Σ-graphs and Σ-total graphs. Rules satisfying these conditions are called Σ-total rules.

Graph G *directly derives* graph H through rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and injective morphism g, denoted by $G \Rightarrow_{r,g} H$ or $G \Rightarrow_r H$ or just $G \Rightarrow H$, if squares (1) and (2) in Figure 2 are natural pushouts. (See [6] for the definition of natural pushouts.)

Operationally, graph D is obtained from G by deleting the nodes and edges in $g(L) - g(K)$, and making each node unlabelled that is the image of an unlabelled node in K that is labelled in L. By the pushout property of square (1), deleted nodes cannot be incident to any edges in $G - (g(L) - g(K))$; this is called the *dangling condition*. Graph H is obtained from D by adding all items

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Fig. 2. A double-pushout diagram

in $R - K$, and labelling unlabelled nodes with the labels of their counterparts in R. We write $G \Rightarrow^*_{\mathcal{R}} H$ if there a sequence $G = G_0 \Rightarrow \ldots \Rightarrow G_n \cong H$, $n \geq 0$, where each direct derivation uses a rule from the set $\mathcal{R}$. If no graph can be directly derived from G through a rule in $\mathcal{R}$, we say that G is *$\mathcal{R}$-irreducible*.

A *graph reduction specification* $\mathcal{S} = \langle \Sigma, \mathcal{R}, Acc \rangle$ consists of a signature Σ, a set of Σ-total rules $\mathcal{R}$ and a Σ-total $\mathcal{R}$-irreducible *accepting graph Acc*. It defines the graph language $\mathrm{L}(\mathcal{S}) = \{G \mid G \Rightarrow^*_{\mathcal{R}} Acc\}$.

A GRS can be turned into an equivalent graph grammar by swapping left- and right-hand sides of the rules and using the accepting graph as a start graph. But we insist on the reduction-rule view as we usually impose conditions such as termination and closedness to ensure that shape membership can be efficiently checked (see below). In addition to the above definition, *nonterminal* labels can be allowed, see [1]. Because the rules in $\mathcal{R}$ are Σ-total, we have for every step $G \Rightarrow_{\mathcal{R}} H$ that G is a Σ-total if and only if H is Σ-total. So the graphs defined by GRSs are Σ-total and $\mathrm{L}(\mathcal{S})$ is a shape.

A GRS $\mathcal{S}$ is *polynomially terminating* if there is a polynomial p such that for every reduction $G_0 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ on Σ-total graphs, $n \leq p(|V_G| + |E_G|)$. It is *closed* if for all $G \in \mathrm{L}(\mathcal{S})$, $G \Rightarrow_{\mathcal{R}} H$ implies $H \in \mathrm{L}(\mathcal{S})$. A *polynomial* graph reduction specification, PGRS for short, is a polynomially terminating and closed GRS. Membership of PGRS shapes is decidable in polynomial time—see [1], where also sufficient conditions for closedness and polynomial termination are discussed.

Unrestricted GRSs are universally powerful in that they can define every recursively enumerable shape, but their membership problem is undecidable in general. The power of PGRSs goes beyond the reach of context-free graph grammars (used by Fradet and Le Métayer to specify shapes [4,5]). For example, [1] contains PGRSs for various forms of balanced trees, including red-black trees. Balance is known to be not context-free specifiable.

To illustrate the above notions, consider again the GRS of Figure 1. Its signature is given by $\mathcal{L}_V = \{R, L, B\}$, $\mathcal{L}_E = \{top, aux, l, r\}$, $\mathrm{type}(R) = \{top, aux\}$, $\mathrm{type}(B) = \{l, r\}$ and $\mathrm{type}(L) = \emptyset$. Every full binary tree with an auxiliary pointer can be reduced to the accepting graph: using the left rule in Figure 1, one first redirects the *aux*-edge to the target of the *top*-edge (if the *aux*-edge points to some other node), and then repeatedly applies the other rule which removes two leaves and relabels their parent node as a leaf. To see that the rules cannot reduce ill-shaped graphs to *Acc*, consider their inverses (which are obtained by swapping left- and right-hand sides):

these rules clearly preserve full binary trees with an auxiliary pointer which implies that the specified shape cannot contain other graphs. The GRS is polynomially terminating—actually linearly terminating—because for every step $G \Rightarrow H$ on Σ-graphs, the number of nodes without outgoing parallel edges is reduced. The GRS is also *non-overlapping*, meaning that for each pair of steps $H_1 \Leftarrow G \Rightarrow H_2$ on Σ-graphs, either $H_1 \cong H_2$ or there is a Σ-graph M such that $H_1 \Rightarrow M \Leftarrow H_2$. This property implies closedness and hence the GRS is a PGRS.

Operations on pointer structures—such as the replacement of a leaf in a tree shown in Figure 6—are also modelled by graph-transformation rules (which need not obey the restrictions of PGRSs). A graph-transformation rule r is *safe* with respect to a shape $\mathrm{L}(\mathcal{S})$ if for all G in $\mathrm{L}(\mathcal{S})$, $G \Rightarrow_r H$ implies $H \in \mathrm{L}(\mathcal{S})$.[4] The static checking algorithm for shape safety developed in the SPGT project is described in [2]. Briefly, given a graph-transformation rule r and a GRS $\mathcal{S}$, the algorithm constructs two *abstract reduction graphs* (ARGs) which represent all contexts of r's left- and right-hand side in members of $\mathrm{L}(\mathcal{S})$. The rule is safe if the right-hand ARG includes the left-hand ARG. Some ARGs are infinite and hence their construction does not terminate, but in many practical cases the algorithm produces finite ARGs representing all left- and right-hand contexts so that inclusion can be checked. The general shape-safety problem is undecidable even for context-free shapes [5] and hence every checking method is necessarily incomplete.

3 C-GRS – An Extension to C

The language C-GRS is a small extension to ANSI C which is intended to implement the approach to shape specification and shape checking described above. The main idea (adopted from [4]) is that pointers are only manipulated by *transformers* which correspond to graph transformation rules. For example, Figure 3 shows a C-GRS function which inserts an integer value `i` into a binary search tree `b` whose shape `bt` corresponds to the GRS of Figure 1. This function uses the transformer `bt_auxreset` to first move the auxiliary pointer to the root of the tree. Then the tree is traversed by repeatedly comparing the integer values in branch nodes (retrieved by `bt_getval`) with the integer `i` and following either the left or the right pointer, using the transformers `bt_goleft` and `bt_goright`. If the search ends at a leaf, then `bt_insert` transforms the leaf into a branch node and inserts `i` into that node. The definition of `bt_insert` is shown on the right of Figure 4.

[4] For simplicity, this paper assumes that rules have the same input and output shape. The shape-checking method of [2] can handle shape-changing rules, too.

```
bt *insert(int i, bt *b) {
  int t;
  bt_auxreset(b);
  while ( bt_getval(b, &t) ) {
    if ( t == i ) return b;
    else if ( t > i ) bt_goleft(b);
    else bt_goright(b);
  }
  bt_insert(b, &i);
  return(b);
}
```

Fig. 3. C-GRS function to insert a value into a binary search tree

3.1 Shapes

Nodes in a C-GRS shape are similar to C structures. In addition to values of normal C data-types, nodes can contain pointers to nodes, declared by the keyword `edge`. Unlike C pointers, edges are defined without stating the type of the objects they are pointing to—edges can point to every (non-root) node of the given shape. For example, the type of a branch node of the binary-tree shape is declared as follows:

```
nodetype branchnode {
  edge l, r;
  int val;
}
```

The collection of node-type definitions of a C-GRS shape declaration corresponds to a GRS signature. Shape declarations also contain transformers, described below, which correspond to the reduction rules of a GRS. The accepting graph of a C-GRS shape is defined after the keyword `accept`, using the same syntax as for the left- and right-hand sides of transformers. Figure 4 shows the declaration of the shape `bt` which corresponds to the GRS of Figure 1 (where the node types `btroot`, `branchnode` and `leafnode` correspond to the node labels R, B and L.) Note that nodes in a C-GRS shape can contain values such as the integer `val` which do not occur in the graphs specified by a GRS.

3.2 Transformers

Transformers are the mechanism by which pointer data-structures are manipulated in C-GRS programs. To ensure shape safety, all manipulations of shape members must be written as transformers. Like graph transformation rules, transformers consist of a left- and right-hand graph. For example, consider the transformer `bt_insert` of Figure 4 which replaces a leaf with a value-carrying branch. To apply this transformer to a tree, its left-hand node `rt` must match the source node of the auxiliary pointer `aux` in the tree and node `n1` must

```
shape bt {
  signature {
    nodetype btroot {
      edge top, aux;
    }
    nodetype branchnode {
      edge l, r;
      int val;
    }
    nodetype leafnode {}
  }
  accept {
    root btroot rt;
    leafnode leaf;
    rt.top => leaf;
    rt.aux => leaf;
  }
  rules {
    moveaux2root;
    branch2leaf;
  }
}
```

```
transformer
bt_insert( bt *tree,
           int *inval ) {
  left (rt, n1) {
    root btroot rt;
    leafnode n1;
    rt.aux => n1;
  }
  right (rt, n1, l1, l2) {
    branchnode n1;
    leafnode l1, l2;
    rt.aux => n1;
    n1.l => l1;
    n1.r => l2;
    n1.val = *inval;
  }
}
```

Fig. 4. Left: shape declaration corresponding to the GRS of Figure 1. Right: transformer replacing a leaf with a branch and inserting a value

match a leaf in the tree that is pointed to by the auxiliary pointer.

The constituent nodes of the left- and right-hand sides of a transformer are declared in a list, as follows:

```
left(rt,n1)
```

Nodes are assigned a type (or *tag* in the terminology of Section 2) using a syntax similar to C variable declarations:

```
btroot rt;
leafnode n1;
```

It is important to note that transformers can alter node types. One can also declare nodes without assigning a type, these nodes correspond to unlabelled nodes in graph transformation rules and will match nodes of every type.

The target of the `aux`-edge leaving `rt` is specified as follows:

```
rt.aux => n1;
```

Edges must point to nodes, null edges are not allowed. The right-hand side of `bt_insert` allocates the leaves `l1` and `l2` as children of `n1`. The types of the new leaves are assigned in the same way as shown for the left-hand side. Leaf `n1` is retyped as a branch node by the following assignment on the right-hand side:

```
branchnode n1;
```

Transformers can overwrite the values held in nodes or return them through transformer parameters. Arguments to a transformer are passed by reference to ensure that several values can be manipulated simultaneously, as C makes it difficult for a function to return several values. The transformer `bt_insert`, for instance, inserts an integer value into the branchnode on its right-hand side as follows:

```
n1.val = *inval;
```

A transformer such as `bt_insert` is used in the same way as an ordinary C function, as shown in the search-tree insertion example of Figure 3.

The EBNF syntax definition of C-GRS is given in Appendix A, together with some (but not all) context conditions. The definition extends the syntax of ANCI C by adding transformer and shape declarations to the categories fun-def and type-def.

3.3 Rootedness

Adding graph transformation rules to C presents two problems. Rules can be applied in a graph wherever their left-hand sides match, which does not fit with C's deterministic world. Moreover, the search for a match of a (fixed) rule requires polynomial time which is too expensive. We solve both problems by requiring that C-GRS shape-structures and left-hand sides of transformers contain unique *roots* and that all nodes in the left-hand sides of transformers are reachable from the roots.

Roots are distinguished nodes which can be declared with the keyword `root` in a shape declaration. For example, the node `rt` in the declaration of `bt` in Figure 4 with its outgoing edges `top` and `aux` is a unique entry point for every binary-tree structure. In general, we require that every shape structure has at least one root and that different roots in the same structure must have different node types. The same applies to the left-hand sides of transformers where, in addition, each node must be reachable from some root by a directed path of edges. Also, transformers must not delete or add root nodes.

Under these conditions the application of a transformer to a shape structure is a deterministic process: the roots of the left-hand side occur in unique places in the structure and whenever a node of the left-hand side has been matched, it is checked if all its outgoing edges are among the outgoing edges of the corresponding node in the structure. The matching of the transformer fails as soon as one of the edge comparisons fails. The matching is successful if all edges of the left-hand side have been found, if the sharing of target nodes in the left-hand side corresponds to the sharing in the structure, and if the dangling condition for direct derivations (see Section 2) is satisfied.

It is not difficult to see that for a fixed transformer, the matching process requires only constant time. This is because every member of a shape comes with a fixed selection of roots (which can be found in constant time) and because the number of outgoing edges of each node is bounded (as shape structures correspond to Σ-total graphs).

4 Translating C-GRS to C

For execution, C-GRS is translated into ANSI C by the translation function $\mathcal{C}$ given in Appendix B (shape translation) and Appendix C (transformer translation). Only shapes and transformers result in modified code, the C portion of a C-GRS program is left unmodified by the translation.

The translation of shapes transforms node types into C structures with the same name. All non-root structures of a shape `S` are wrapped into a single C union `S_node` and edges become pointers to `S_node`. This also allows to retype nodes in-place in memory. For example, the type `btroot` from the shape `bt` of Figure 4 is translated into the following C structure:

```
struct btroot {
  bt_node  *top, *aux;
}
```

Appendix C shows the translation of transformers into C functions which can be applied to shape members. The application of such a function proceeds in two major phases: first, the transformer's left-hand side is matched against nodes in the shape member, and second the image of the left-hand side is transformed into the right-hand side by deleting and adding nodes and altering the contents of preserved nodes.

The matching phase of a transformer uses *matching variables* which correspond to nodes in the left-hand graph. These variables hold pointers to nodes in the shape member such that a variable holds a pointer to a node if and only if the left-hand node corresponding to the variable has been matched with the shape-member node.

Root nodes correspond to pointer fields in the C structure `S` representing a shape member, so they can be easily assigned to matching variables. As only one root of each type can exist in a shape member, the fields are distinctly named after the types of the root nodes. For example, the following code in the translation of the transformer `bt_insert`,

```
rt = tree->btroot;
```

assigns the pointer to the root of a binary-tree to the matching variable `rt`, where `tree` is the parameter of `bt_insert` holding a pointer to the tree.

When a root has been matched, the matching process proceeds by following the edges outgoing from matched nodes. To keep the description of the translation simple, we assume that edge statements are ordered in a way such that no non-root node occurs as the source of an edge before it occurs as a target of some edge. This ensures that nodes are matched in a correct order.

```
if ( ! typeeq( (rt->btroot).aux, "leafnode") )
  return False;
else if (n1 == NULL) n1 = (rt->btroot).aux;
else if (n1 != (rt->btroot).aux) return False;
```

Fig. 5. Matching code for the edge `aux` in the left-hand side of `bt_insert`

As all nodes in the left-hand side of a transformer are reachable from some root, each node will eventually be matched. For example, Figure 5 shows the matching code produced for the edge `aux` from `rt` to `n1` in the left-hand side of `bt_insert`. This code first checks that the node found by following the `aux`-edge is a leaf. Then, if the matching variable `n1` is null, it is assigned a pointer to the target of the `aux`-edge. This gives `n1` an initial value if it is the first time it has been reached. If `n1` already holds a non-null value, then node `n1` on the left-hand side of the transformer has more than one incoming edge and the code checks that the pointers `n1` and `aux` point to the same node. If any of the checks fail, the transformer function returns `False` without modifying the shape member.

Once all nodes of the left-hand side of a transformer have been matched, the system performs two more checks. First, it checks by comparison of pointer values that each pair of distinct transformer nodes has been matched with distinct nodes in the shape member. Then the dangling condition for deleted nodes (see Section 2) is checked by reference counting, using the `indegree` field of deleted nodes. Failure of the dangling condition is treated in the same way as failure in the above cases.

If the matching process has been successful, the image of the transformer's left-hand side in the shape member is modified to the right-hand side by deleting, allocating and retyping nodes, and recreating edges. Nodes are managed using the normal C memory allocation functions. Edges are recreated by assigning new values to pointer fields in nodes. For example, the edge `aux` in the right-hand side is recreated by

```
(rt->btroot).aux = n1;
```

where `rt` and `n1` are the matching variables for the nodes of the same names.

After the C-GRS code has been translated by $\mathcal{C}$, the resulting C code can be compiled and executed in the normal way.

5 Abstracting C-GRS to Graph Transformation

Our main aim in adding shapes and transformers to C is to make it possible to statically check the shape safety of graph transformation rules corresponding to transformers, using the algorithm described in [2]. We denote by $\mathcal{G}$ the function which abstracts C-GRS shapes and transformers to GRSs and graph transformation rules, respectively. The form of C-GRS shapes and transformers is intentionally very close to GRSs and graph transformation rules, so $\mathcal{G}$'s straightforward definition is omitted from this paper. As an example, Figure 6 shows the graph transformation rule produced by applying $\mathcal{G}$ to the transformer `bt_insert` of Figure 4. The node labels R, B and L stand for `btroot`, `branchnode` and `leafnode`, respectively.

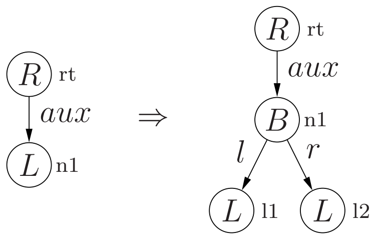

Fig. 6. Graph transformation rule produced from `bt_insert` by the abstraction $\mathcal{G}$

The translation $\mathcal{G}$ maps C-GRS shape declarations to GRSs whose shapes consist of graphs which model pointer data-structures by abstracting from non-pointer values. Accordingly, graph transformation rules produced from transformers only model structural modifications of pointer structures and ignore value changing operations. For instance, $\mathcal{G}$ forgets the integer held in node `n1` when abstracting the transformer `bt_insert` to the rule in Figure 6.

To analyse the correctness of the translation $\mathcal{C}$ with respect to the graph model given by $\mathcal{G}$, we fix a few notions. By a *pointer structure* we mean a set of individual records ('structures' in C's terminology) in a C program-state such that all pointers in the records point to records in the set. A pointer structure is *consistent* with a signature $\Sigma = \langle \mathcal{L}_V, \mathcal{L}_E, \mathrm{type} \rangle$ if each record contains a field `type` holding a value $l \in \mathcal{L}_V$ such that $\mathrm{type}(l)$ consists of the names of the pointer fields in the record. We denote by α_Σ the function that abstracts (in the obvious way) pointer structures consistent with Σ to Σ-total graphs.

Using these notions, we say that the translation $\mathcal{C}$ is *correct* with respect to $\mathcal{G}$ if for every transformer F and every pointer structure S that is consistent with F's signature Σ,

$$\mathcal{G}[\![F]\!](\alpha_\Sigma(S)) = \alpha_\Sigma(\mathcal{C}[\![F]\!](S)).$$

In other words, the diagram of Figure 7 has to commute. Here we assume that a failed application of the graph-transformation rule $\mathcal{G}[\![F]\!]$ returns the input graph unmodified.

$$\begin{tikzcd}
G \arrow[r, "\mathcal{G}[\![F]\!]"] & G' & \Sigma\text{-total graphs} \\
S \arrow[u, "\alpha_\Sigma"] \arrow[r, "\mathcal{C}[\![F]\!]"] \arrow[ur, phantom, "="] & S' \arrow[u, "\alpha_\Sigma"] & \text{C pointer-structures}
\end{tikzcd}$$

Fig. 7. Correctness of the translation $\mathcal{C}$

Suppose that this correctness property holds and that all pointer manipulations in a C-GRS program P happen through applications of transformers to pointer structures that correspond to members of the shapes associated with the transformers. Then we can check that P is shape safe by checking the corresponding graph-transformation rules produced by $\mathcal{G}$.

To show that the diagram of Figure 7 commutes for every transformer F, we first show that $\mathcal{G}[\![F]\!]$ and $\mathcal{C}[\![F]\!]$ select corresponding graph elements in their matching phases. This can be proved by induction on the size of the left-hand side of F:

(i) Roots are correctly matched, as both graphs and pointer structures can only contain a single instance of a particular root.

(ii) The children of correctly matched nodes are correctly matched, as in both the graph and the pointer structure they are connected to their parent by a distinctly-labelled edge.

The same kind of argument shows that in the case of a matching failure, it fails for corresponding nodes processed by $\mathcal{G}[\![F]\!]$ and $\mathcal{C}[\![F]\!]$. Similarly, $\mathcal{G}[\![F]\!]$ violates the dangling condition for a deleted node if and only if the C code in $\mathcal{C}[\![F]\!]$ checking the dangling condition reports failure for the C record corresponding to that node. The proof of correctness is completed by showing that corresponding right-hand modifications are performed by $\mathcal{G}[\![F]\!]$ and $\mathcal{C}[\![F]\!]$.

6 Related Work

Our language C-GRS is similar to Fradet's and Le Métayer's Shape-C [4], the main difference is that Shape-C is restricted to shapes specified by context-free graph grammars. The graph reduction specifications incorporated in C-GRS—even when restricted to polynomial GRSs—allow programmers to specify non-context-free data structures such as grids and various forms of balanced trees. In addition, shapes defined by polynomial GRSs come with an efficient membership test which can be used for testing and debugging shape specifications. Shapes defined by context-free graph grammars, on the other hand, are known to have an NP-complete membership problem.

Graph types [8] are spanning trees with additional pointers defined by path expressions; they form the basis of *pointer assertion logic* [10], a monadic second-order logic for expressing properties of pointer structures in program annotations. This requires programmers to use quite a sophisticated logic, but the formalism is still too weak to express some important properties such as balance in trees. These drawbacks also apply to the TVLA system [9] which demands that data structures and the effects of program statements are expressed in three-valued logic with transitive closure. TVLA employs the shape analysis method of [12] to verify invariants.

Separation logic [7,11] extends classical Hoare-style program verification so that specifications and proofs can deal with properties of linked data structures. The logic allows the heap to be divided into regions for which different logical formulas hold, making it possible to reason locally about pointers. But so far there seems to be no automatic verification method for separation logic.

Acknowledgement. We would like to thank Adam Bakewell and the anonymous referees for comments that helped to improve this paper.

References

[1] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction*, Mathematical Structures in Computer Science. To appear. Preliminary version available as Technical Report YCS-2003-367, University of York, 2003.

[2] Bakewell, A., D. Plump and C. Runciman, *Checking the shape safety of pointer manipulations*, in: *Int. Seminar on Relational Methods in Computer Science (RelMiCS 7), Revised Selected Papers*, Lecture Notes in Computer Science **3051** (2004), pp. 48–61.

[3] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction*, in: *Int. Workshop Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, Lecture Notes in Computer Science **3062** (2004), pp. 30–44.

[4] Fradet, P. and D. Le Métayer, *Shape types*, in: *Proc. Principles of Programming Languages (POPL '97)* (1997), pp. 27–39.

[5] Fradet, P. and D. Le Métayer, *Structured Gamma*, Science of Computer Programming **31** (1998), pp. 263–289.

[6] Habel, A. and D. Plump, *Relabelling in graph transformation*, in: *Proc. International Conference on Graph Transformation (ICGT 2002)*, Lecture Notes in Computer Science **2505** (2002), pp. 135–147.

[7] Ishtiaq, S. and P. W. O'Hearn, *BI as an assertion language for mutable data structures*, in: *Proc. Principles of Programming Languages (POPL '01)* (2001), pp. 14–26.

[8] Klarlund, N. and M. Schwartzbach, *Graph types*, in: *Proc. Principles of Programming Languages (POPL '93)* (1993), pp. 196–205.

[9] Lev-Ami, T. and M. Sagiv, *TVLA: A system for implementing static analyses*, in: *Proc. Static Analysis (SAS '00)*, Lecture Notes in Computer Science **1824** (2000), pp. 280–301.

[10] Møller, A. and M. I. Schwartzbach, *The pointer assertion logic engine*, in: *Proc. Programming Language Design and Implementation (PLDI '01)* (2001), pp. 221–231.

[11] Reynolds, J., *Separation logic: A logic for shared mutable data structures*, in: *Proc. Logic in Computer Science (LICS '02)* (2002), pp. 55–74.

[12] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM Transactions on Programming Languages and Systems **24** (2002), pp. 217–298.

Appendix

A EBNF Syntax of C-GRS

```
                         transformer trid ( sid *id, [tid *id]* ) {
     fun-def    ::=        left ([nid]*) { [node-dec;]+ [left-graph;]* }
                           right ([nid]*) { [node-dec;]* [right-graph;]* }
                         }
                  |      ...

     node-dec   ::=      root ntid nid | ntid nid [, nid]*
   left-graph   ::=      nid.ed => nid
  right-graph   ::=      nid.ed => nid | nid.id = id | id = nid.id

                         shape sid {
                           signature { [node-def;]+ }
     type-def   ::=        accept { [node-dec;]+ [nid.ed => nid;]* }
                           rules { [trid;]* }
                         }
                  |      ...
     node-def   ::=      nodetype ntid { [node-cont;]+ }
    node-cont   ::=      edge ed [, ed]* | struct-decl-cont
```

- *id* and *tid* stand for identifiers of C variables and C types, respectively. *nid*, *ntid*, *sid*, *trid* and *tid* stand for identifiers of nodes, node types, shapes and transformers, respectively. *ed* stands for edge labels.
- struct-decl-cont corresponds to the statements that can be part of a C structure-declaration.

Context Conditions

- Root nodes must not be deleted by a transformer.
- On both sides of a transformer, all nodes must be reachable from some root node.
- Nodes that are created or retyped on the right-hand side of a transformer must have all the edges for their type declared.
- All rules of a shape must be defined as transformers.

B Shape Translation

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l}
\texttt{shape}\ \mathrm{S}\ \{\\
\quad\texttt{signature}\ \{\ \mathrm{N}_1\texttt{;}\ldots\mathrm{N}_n\texttt{;}\ \}\\
\quad\texttt{accept}\ \{\ \mathrm{A}_1\texttt{;}\ldots\mathrm{A}_n\texttt{;}\ \}\\
\quad\texttt{rules}\ \{\ \mathrm{P}_1\texttt{;}\ldots\mathrm{P}_n\texttt{;}\ \}\\
\}
\end{array}\right]\!\!\right]
=
\begin{array}{l}
[\ \mathcal{N}[\![\ \mathrm{N}_i\ ]\!]\ ]_{i=1,\ldots,n}\\
\\
\texttt{typedef struct}\ \mathrm{S}\ \{\\
\quad[\ \mathrm{S\_node}\ \texttt{*}r\texttt{;}\ ]_{r\in R}\\
\}\\
\\
\texttt{typedef struct}\ \mathrm{S\_node}\ \{\\
\quad\texttt{char *type;}\\
\quad\texttt{int indegree;}\\
\quad\texttt{union}\ \{\\
\quad\quad[\ \texttt{struct}\ d;\ ]_{d\in D}\\
\quad\}\ \texttt{node;}\\
\}\\
\\
\mathrm{S}\ \texttt{*}\ \texttt{newgraph\_}\mathrm{S}\ \texttt{()}\ \{\\
\quad\mathrm{S}\ \texttt{*new;}\\
\quad\texttt{new = malloc(sizeof(}\ \mathrm{S}\texttt{));}\\
\quad[\ \mathcal{I}[\![\ \mathrm{A}_i\ ]\!]\ ]_{i=1,\ldots,n}\\
\quad\texttt{return new;}\\
\}
\end{array}
$$

where:
D = types of nodes defined in $\{\mathrm{N}_1, \ldots, \mathrm{N}_n\}$
R = types of root nodes declared in $\{\mathrm{A}_1, \ldots, \mathrm{A}_n\}$

$$\mathcal{N}[\![\ \texttt{nodetype}\ \mathrm{N}\ \{\ \mathrm{C}\ \}\ ]\!] \quad = \quad \texttt{struct}\ \mathrm{N}\ \{\ \mathcal{T}[\![\ \mathrm{C}\ ]\!]\ \}$$

$$\mathcal{T}[\![\ \texttt{edge}\ \mathrm{E}_1,\ldots,\mathrm{E}_n\ ]\!] \quad = \quad [\ \mathrm{S\_node}\ \texttt{*}\mathrm{E}_i\texttt{;}]_{i=1,\ldots,n}$$

$$\mathcal{T}[\![\ \mathrm{C}\ ]\!] \quad = \quad \mathrm{C}$$

$$\mathcal{I}[\![\ \texttt{root}\ \mathrm{T}\ \mathrm{V}\ ]\!] \quad = \quad \begin{array}{l}\texttt{new.}\mathrm{V}\ \texttt{= createnode(}\mathrm{T}\texttt{);}\\ \text{define } t_a(\mathrm{V}) = \mathrm{T}\end{array}$$

$$\mathcal{I}[\![\ \mathrm{T}\ \mathrm{V}_1,\ldots,\mathrm{V}_n\ ]\!] \quad = \quad \begin{array}{l}[\ \mathrm{V}_i\ \texttt{= createnode(}\mathrm{T}\texttt{);}\ ]_{i=1,\ldots,n}\\ \text{define } t_a(\mathrm{V}_i) = \mathrm{T}, \text{ for } i = 1, \ldots, n\end{array}$$

$$\mathcal{I}[\![\ \mathrm{S.E}\ \texttt{=>}\ \mathrm{T}\ ]\!] \quad = \quad \texttt{(}\mathrm{S}\texttt{->}t_a(S)\texttt{).}\mathrm{E}\ \texttt{=}\ \mathrm{T};$$

C Transformer Translation

$\mathcal{C}\left[\!\left[\begin{array}{l}\texttt{transformer}\ \mathrm{F}\ (\mathrm{S}\ {*}\mathrm{G};\ \mathrm{A})\\ \quad\texttt{left}(\mathrm{N}_l)\{\ \mathrm{L}_1;\ldots\mathrm{L}_n;\ \}\\ \quad\texttt{right}(\mathrm{N}_r)\{\ \mathrm{R}_1;\ldots\mathrm{R}_n;\ \}\\ \}\end{array}\right]\!\right] =$

```
bool F (S *G; A) {
   [ $\mathcal{L}[\![\mathrm{L_i}]\!]$ ]$_{i=1,\ldots,n}$
   [ $x$ != $y$; ]$_{\{x,y\}\in Pairs}$
   [ if ( $d$.indegree != $C(d)$ )
            return False; ]$_{d\in Delete}$
   [ $n$.indegree
        = $n$.indegree - $C(n)$; ]$_{n\in N_l}$
   [ retypenode($p$, $t_r(p)$); ]$_{p\in Retype}$
   [ $a$ = createnode($t_r(a)$) ); ]$_{a\in Allocate}$
   [ $\mathcal{R}[\![\mathrm{R_i}]\!]$ ]$_{i=1,\ldots,n}$
   [ deletenode($d$); ]$_{d\in Delete}$
   return True;
}
```

where:
$C(i) = \#\{(s,e) \mid (s.e \texttt{ => } i) \in \{\mathrm{L}_1,\ldots,\mathrm{L}_n\}\}$
$Delete = \mathrm{N_l} - \mathrm{N_r}$
$Allocate = \mathrm{N_r} - \mathrm{N_l}$
$Retype = \{p \in \mathrm{N_l} \cap \mathrm{N_r} \mid \mathrm{t_l(p)} \neq \mathrm{t_r(p)}\}$
$Pairs = \{\{x,y\} \subseteq \mathrm{N_l} \mid \mathrm{x} \neq \mathrm{y}\}$

$\mathcal{L}[\![\ \texttt{root}\ \mathrm{T\ V}\ ]\!] =$

```
S_node *V;
V = G->T;
define $t_l(V) = \mathrm{T}$
```

$\mathcal{L}[\![\ \mathrm{T\ V_1,\ldots,V_n}\ ]\!] =$

```
[ S_node V$_i$; V$_i$ = NULL ]$_{i=1,\ldots,n}$
define $t_l(\mathrm{V}_i) = \mathrm{T}$, for $i = 1,\ldots,n$
```

$\mathcal{L}[\![\ \mathrm{S.E}\ \texttt{=>}\ \mathrm{T}\ ]\!] =$

```
if ( !  typeeq((S->$t_l(S)$).E, $t_l(T)$) )
  return False;
else if (T == NULL)
  T = (S->$t_l(S)$).E;
else if ( T == (S->$t_l(S)$).E )
  return False;
```

$\mathcal{R}[\![\ \texttt{root}\ \mathrm{T\ V}\ ]\!] =$ define $t_r(\mathrm{V}) = \mathrm{T}$

$\mathcal{R}[\![\ \mathrm{T\ V_1,\ldots,V_n}\ ]\!] =$ define $t_r(\mathrm{V}_i) = \mathrm{T}$, for $i = 1,\ldots,n$

$\mathcal{R}[\![\ \mathrm{S.V = X}\ ]\!] =$ `(S->`$t_r(S)$`).V = X;`

$\mathcal{R}[\![\ \mathrm{X = S.V}\ ]\!] =$ `X = (S->`$t_r(S)$`).V;`

$\mathcal{R}[\![\ \mathrm{S.E}\ \texttt{=>}\ \mathrm{T}\ ]\!] =$

```
(S->$t_r(S)$).E = T;
T.indegree = T.indegree + 1;
```

Towards Attributed Graphs in Groove

Work in Progress

Harmen Kastenberg [1]

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
h.kastenberg@cs.utwente.nl

Abstract

Graphs are a very expressive formalism for system modeling, especially when attributes are allowed. Our research is mainly focused on the use of graphs for system verification.

Up to now, there are two main different approaches of modeling (typed) attributed graphs and specifying their transformation. Here we report preliminary results of our investigation on a third approach. In our approach we couple a graph to a data signature that consists of unary operations only. Therefore, we transform arbitrary signatures into a structure comparable to what is called a graph structure signature in the literature, and arbitrary algebras into the corresponding algebra graph.

Key words: attributed graphs, graph transformation, algebra graph, signature structure

1 Introduction

Representing (parts of) software systems (or their states) as graphs, has proven to be a very powerful approach for specifying program structure and verifying its behavior. In the Groove project [10] we aim at the use of graphs for verifying object oriented systems. Since the state of such systems is determined on basis of the occurring objects and the values of their attributes, it is necessary to extend the Groove Tool to support the use of attributed graphs.

Although we want to stay as close as possible to currently available theory about modeling attributed graphs and specifying their transformation [7,1,6], we believe there is a simpler, more intuitive way of specifying attributed graph transformations in the context of our tool. In our investigation we focus on

[1] The author is employed in the GROOVE project funded by the Dutch NWO (project number 500.19205).

minimizing tool implementation efforts and keeping transformation specification as straightforward as possible. This means that instead of both changing the graphical representation of graphs in our tool (e.g. to more UML-like structures as used in [11]) and extending the underlying tool engine to support attribution, we combine the latter with introducing some notational conventions.

In order to support graph attribution in our tool we need to introduce data type signatures and couple those to ordinary graphs, as currently available. This coupling can be established by using special edges connecting nodes from the graph-part to nodes from the data-part representing attribute values. The difference with other approaches is that in our approach the data-part is based on data type signatures consisting of unary operations only. Therefore, we transform arbitrary signatures into a structure that is comparable to what is called a graph structure signature in the literature [5] and arbitrary algebras into the corresponding algebra graph. The algebra graph contains all necessary information about the data types that are supported and provides the semantics of the operations of each of the data types.

In this report we focus on how to model and transform attributed graphs in our tool, instead of focusing on the transformation of data type signatures, although some details of this transformation will be mentioned. In the future we will work on a more precise functor specification of this transformation in order to specify the exact relation to other approaches.

This paper is structured as follows. First we give some definitions of concepts used in the rest of this work and give some insight in how we transform arbitrary signatures into the structure we prefer. Then we show how we model attributed graphs and how we specify their transformation by means of a simple example. Thereafter, we list the advantages of our approach one by one. We conclude with a short note on related work and some comments on the restrictions of our approach.

2 Signature Structure and Attributed Graphs

In this section we define the notion of *signatures* and *attributed graphs* as they are generally accepted. We also show how we transform arbitrary signatures into signatures with unary operations only and how this is done for a small example.

Traditionally (e.g. [4]), signatures are defined as follows.

Definition 2.1 (signature) A *signature* $SIG = \langle s_1, \ldots, s_n; op_1, \ldots, op_m \rangle$ consists of sorts s_i $(1 \leq i \leq n)$ and constant and operation symbols op_j $(1 \leq j \leq m)$. □

We transform arbitrary signatures of the above form into signatures with unary operations only (this structure is comparable to what is called *graph structure signature* in [5]):

$SIG' = \langle s_1, \ldots, s_n, op'_1, \ldots, op'_m; proj_{1,1}, \ldots, proj_{1,a_1+1}, \ldots, proj_{m,1}, \ldots, proj_{m,a_m+1} \rangle$, such that op'_j is the sort-counterpart of the original SIG-operation op_j. When $op_j(x_1, \ldots, x_{a_j}) = r$, $proj_{j,i}$ projects op'_j on its i^{th} component and $proj_{j,a_j+1}$ projects op'_j on its last component, being the result of op_j (a_j is the arity of op_j and $1 \leq i \leq a_j$).

Example 2.2 (integer algebra) In order to specify an integer algebra with only the add-operation we start with the following signature:

$SIGINT = \langle \mathtt{int}; + \rangle$,

where `int` is the set of integer values and $+ : \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int}$.

In a $SIGINT$-algebra A the $+$-operation could then be partially specified as $+ = \{\langle (1,2), 3 \rangle, \langle (2,3), 5 \rangle\}$.

The transformed signature $SIGINT'$ then has the following structure:

$SIGINT' = \langle \mathtt{int}, +'; \mathtt{arg0}, \mathtt{arg1}, \mathtt{result} \rangle$,

where $+' : \mathtt{int} \times \mathtt{int} \times \mathtt{int}$ is the sort representing the original $+$-operation and the operations `arg0`, `arg1`, and `result` (all of type $+' \rightarrow \mathtt{int}$) are projections that correspond to the $+'$-sort.

In the $SIGINT'$-algebra A' the $+'$-sort would contain the tuples $\langle 1, 2, 3 \rangle$ and $\langle 2, 3, 5 \rangle$ and the projections would look as follows:

$$\begin{array}{lll} \mathtt{arg0}(\langle 1,2,3 \rangle) = 1 & \mathtt{arg1}(\langle 1,2,3 \rangle) = 2 & \mathtt{result}(\langle 1,2,3 \rangle) = 3 \\ \mathtt{arg0}(\langle 2,3,5 \rangle) = 2 & \mathtt{arg1}(\langle 2,3,5 \rangle) = 3 & \mathtt{result}(\langle 2,3,5 \rangle) = 5 \end{array}$$

□

Attributed graphs generally consist of two parts: a graph-part and a data-part.

Definition 2.3 (attributed graph) Consider a data signature $DSIG = \langle S, OP \rangle$ with attribute value sorts S and a graph $G = \langle V, E \rangle$. An *attributed graph* $AG = \langle G, D \rangle$ consists of a graph G and a $DSIG$-algebra D such that $\uplus_{s \in S} D_s \subseteq G_V$. □

When applying the described transformation to the data signature $DSIG$ from Definition 2.3, there exists a straightforward way of visually modeling the transformation of (the data-part of) attributed graphs in which the constants and operations are part of the *algebra graph*. The algebra operations to be applied are represented by nodes, labeled with the operation-symbols, being connected to the operands on which they will be applied and the resulting value.

The algebra graph can be looked upon as being a *bipartite graph* in which the nodes representing the instances of the algebra operations (with arity > 0) form one set and the nodes representing the constant data values form the other disjoint set. Moreover, the edges of the algebra graph all have the same direction, namely from the set of algebra operations to the set of constant data values. Fig. 2.1 shows part of the algebra graph of the $INTSIG'$-algebra A' from Example 2.2 as a bipartite graph. The right subset contains the instances of the algebra operations; the left subset contains the constant data values. This bipartitioning property of the algebra graph will later on play an important role when discussing the finiteness of attributed graphs in our approach.

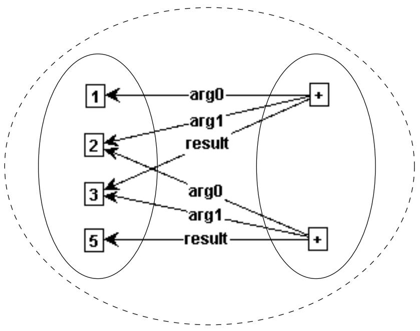

Figure 2.1. Bipartitioning of the algebra graph.

3 Transformation Specification

We have just explained how we model attributed graphs theoretically. In this section we will explain how we model and transform attributed graphs in a visual way by means of an example, focusing on how to change attribute values. The example, inspired by [5], is a graphical representation of method signatures in which a method is identified by its name and its ordered parameters.

3.1 Attributed Graphs

In Groove the attribute values are each represented by a single node and the names of the attributes are represented by the labels on the edges connecting them to the graph-part. An example method signature can then look as shown in Fig. 3.1.

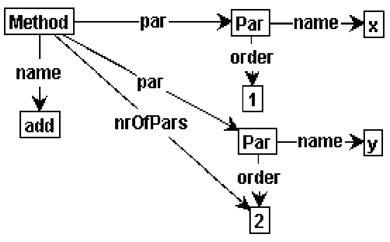

Figure 3.1. Graphical representation of the method signature `add(x,y)`.

3.2 Changing Attribute Values

Specifying the transformation of attributed graphs basically consists of two parts: specifying (1) graph-structure changes and (2) attribute value changes.

The first part is performed by *graph rewriting*, while the second part involves *term graph rewriting* [8]. Here we focus on the second part. Fig. 3.2 shows a transformation rule, using the single pushout approach [3], which adds a parameter to a method signature.

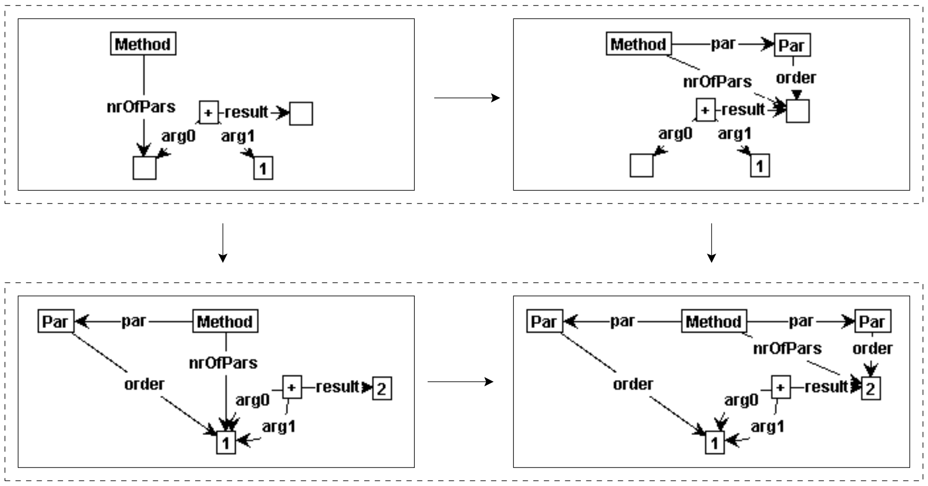

Figure 3.2. Rule application for adding a parameter to a method signature.

In this rule we specify how to add a new parameter to the method signature. [2] This involves the calculation of the new value of the nrOfPars-attribute and the creation of a new parameter which gets this new value as its order-attribute. The part of this rule that specifies the attribute value change in this rule consists of four nodes and three edges connecting them. One node represents the operation, two nodes represent the operands on which the operation must be applied and one node represents the result. Note that two nodes in the transformation rule (upper row in the figure) representing constant data values are left unlabeled. The value of the unlabeled operand can be determined after matching the rule's left-hand-side in the source graph. The result of applying the algebra operation on the operands is then determined by the algebra graph.

Note that we assume that the algebra elements (operations and constants) are *always* present. Of course, this is practically not possible because this would imply infinitely large graphs. In a tool implementation this could be resolved by only including those attribute-values of the algebra graph that are directly reachable from the graph-part. Combining the facts that the graph-part only refers to nodes from the algebra graph representing constant data values and that those nodes do not have outgoing edges (remember the bipartitioning property of the algebra graph discussed in Sect. 2) then implies the inclusion of a finite subgraph of the algebra graph in any attributed graph. Another point of attention is the fact that constant data values are represented

[2] In the given rule specification we have left out the parts involving the creation of the name-attribute because of implementation issues.

by *unique* nodes. This becomes clear from the rule application part of Fig. 3.2 where the node representing the integer-value 1 is both the first and second operand of the +-operation. The uniqueness of algebra operations is determined by their operands, i.e. algebra operations having distinct operands are represented by distinct nodes labeled with the operation symbol being connected to the nodes representing the operands and the corresponding unique result [8].

4 Advantages

We have shown how we transform arbitrary signatures in a signature structure with unary operations only and how we use the latter to specify the transformation of attributed graphs. Here we discuss a number of advantages of this approach, some of which are mainly related to tool implementation issues.

4.1 Variables

Normally, specifying the transformation of attributed graph requires the introduction of a set of variables when changing data values. In that case the transformation process involves activities such as assigning the actual value of the attributes to those variables, calculating the new value by applying the algebraic operation and assigning that new value to the original attribute. In our approach we do not introduce such a set of variables. We, instead, re-use ordinary rule nodes to stand for data values. When these nodes are matched in the source-graph, we can easily obtain the data value it stands for. This reduces the efforts needed for implementing attribution support in our tool.

4.2 Graphical Representation

The way we specify the transformation of attributed graphs graphically is closely related to the underlying theory. The part of the transformation rule shown in Fig. 3.2 that specifies the attribute value change, could also be viewed as being a hyperedge, having the nodes representing the constant data values as its endpoints and the operation symbol as its label. The list of endpoints then is an element of the sort corresponding to that operation. The labels of each tentacle represent the corresponding projection functions. Since our tool does not (yet) support hypergraphs, the algebra operation to be applied is represented by a distinct node having one outgoing edge for each tentacle of the corresponding hyperedge.

4.3 Changing Semantics

A third advantage of our approach is the separation of the use of algebra operations in transformation rules and their semantics. Since the semantics of the algebra operations are enclosed in the algebra graph, operation semantics can be changed by changing the algebra graph, or better stated, by changing the algebra from which the algebra graph is derived. Fortunately, this has no effect on the transformation rules themselves, because they do not refer to the algebraic semantics of the used operations: the nodes representing the result of applying algebra operations are left unlabeled. Actually, other approaches may be using the same idea, but to our best knowledge this has never been stated explicitly.

5 Conclusion

A number of approaches to transform attributed graphs have been developed [7,6,5]. They all distinguish between the graph-part and the data-part of attributed graphs, but describe different ways of connecting these two parts together. In this report we focussed on how to specify attribute-value changes. In the literature, two main different approaches appear for this: relabeling attribute-nodes (see e.g. [7,9]) and reconnecting graph-nodes to the new attribute-nodes (see e.g. [6]). Our work is based on the second approach and differs from [6] in the way of specifying the actual attribute value change, since we store the semantics of the algebra operations in, what we call, the algebra graph by means of hyperedge-like structures. This way of modeling operation application is comparable to what is called a graph structure signature in [5]. This graphical structure binds every operation to the corresponding projection functions. In contrast to [5], our approach neither allows edge attribution nor typing.

Further work on this subject consists firstly of specifying the exact relation between our way of modeling attributed graphs and the other approaches (functor specification) and secondly of finishing the implementation of the tool concerning attribution support.

Acknowledgements

We would like to thank the anonymous referees for their detailed comments and constructive suggestions.

References

[1] Berthold, M. R., I. Fischer and M. Koch, *Attributed Graph Transformation with Partial Attribution*, in: H. Ehrig and G. Taentzer, editors, *Proceedings Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems* (2000), pp. 171–178.

[2] Ehrig, H., G. Engels, F. Parisi-Presicce and G. Rozenberg, editors, “Proceedings of the 2^{nd} International Conference on Graph Transformation,” Lecture Notes in Computer Science **3256**, Springer, 2004.

[3] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison with Double Pushout Approach*, in: G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, World Scientific, 1997 pp. 247–312.

[4] Ehrig, H. and B. Mahr, "Fundamentals of Algebraic Specification 1: Equations and Initial Semantics," Monographs on Theoretical Computer Science **6**, Springer-Verlag, 1985.

[5] Ehrig, H., U. Prange and G. Taentzer, *Fundamentel Theory for Typed Attributed Graph Transformation*, in: Ehrig et al. [2] pp. 161–177.

[6] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of Typed Attributed Graph Transformation Systems*, in: A. Corradini, H. Ehrig, H. J. Kreowski and G. Rozenberg, editors, *Proceedings of the* 1*st* *International Conference on Graph Transformations*, Lecture Notes in Computer Science **2505**, Springer, 2002 pp. 161–176.

[7] Löwe, M., M. Korff and A. Wagner, *An Algebraic Framework for the Transformation of Attributed Graphs*, in: *Term Graph Rewriting: Theory and Practice*, John Wiley and Sons Ltd., 1993 pp. 185–199.

[8] Plump, D., *Term Graph Rewriting*, in: H. Ehrig, G. Engels, H. J. Kreowski and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages and Tools*, World Scientific, 1999 pp. 3–61.

[9] Plump, D. and S. Steinert, *Towards Graph Programs for Graph Algorithms*, in: Ehrig et al. [2], pp. 128–143.

[10] Rensink, A., *The GROOVE Simulator: A Tool for State Space Generation*, in: J. L. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[11] Taentzer, G., *AGG: The Attributed Graph Grammar System* (2005), http://tfs.cs.tu-berlin.de/agg/.

A Framework for Stochastic System Modelling and Analysis

Work in Progress

Sebastian Menge [1]   Georgios Lajios [2]

Software Technology
University of Dortmund
Germany

Abstract

Stochastic Graph Transformation combines the benefits of graphical modelling with stochastic analysis techniques. In this paper we report on our framework SMA for Stochastic Modelling and Analysis, and SGT*, a tool which uses the framework for Stochastic Graph Transformation.

Key words: graph transformation, stochastic analysis, model checking, tool support

1 Introduction

In distributed and mobile systems with volatile bandwidth and fragile connectivity, non-functional aspects such as performance and reliability become more and more important. To analyse such properties, stochastic methods are required. At the same time such systems are characterized by a high degree of architectural reconfiguration. This gave rise to the notion of *Stochastic Graph Transformation* [HLM04], which combines the benefits of using graph transformation for system modelling with the power of stochastic analysis as known from areas such as queueing theory, Markov theory, or recently probabilistic model checking.

While this combination is conceptually beneficial, it is still difficult to integrate graphical modelling with stochastic analysis when it comes to tool support. Though many tools are available to meet the requirements of either

[1] Email: `sebastian.menge@uni-dortmund.de`
[2] Email: `georgios.lajios@uni-dortmund.de`

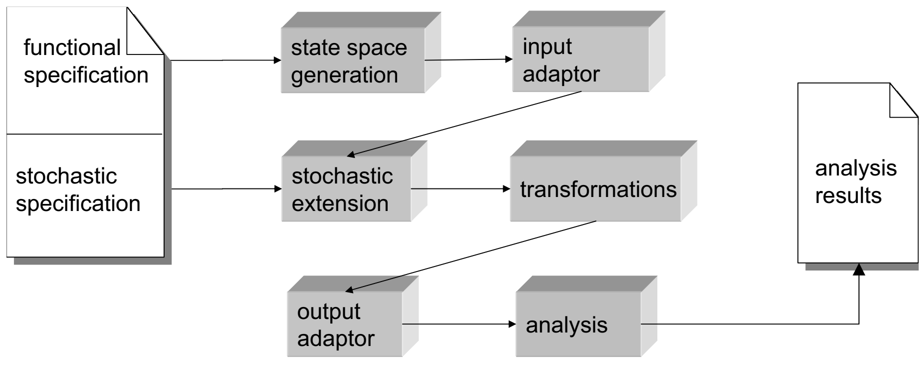

Fig. 1. the architecture of the framework

graphical modelling or stochastic analysis, there is still lack of tool support to combine both aspects. Thus, to analyse case studies for stochastic graph transformation, there was the need to develop something to bridge the gap between intuitive modelling and good analysis capabilities.

When developing the tool, we did not want to restrict ourselves to a specific approach, but wanted to retain flexibility both in the modelling and the analysis paradigm. Therefore, we decided to build a framework to accomodate the integration of different approaches to stochastic modelling and analysis.

In this paper, we present such a framework. Section 2 presents the overall architecture and main ideas, while Section 3 discusses how we used the framework for stochastic graph transformation. Section 4 concludes the paper and presents further ideas.

2 A Framework for Stochastic Modelling and Analysis

The main aim of the SMA framework is to keep modelling and analysis of stochastic systems separate but to allow tight integration of specific tools at the same time.

Since we want to reuse the existing powerful tools when investigating stochastic systems, we have to adapt to these tools and map between the different kinds of models they deal with. Furthermore, we want to perform non-trivial transformations on the models, such as merging modular specifications [HLM05a], minimizing the state space etc. To meet these requirements, we use *pipes and filters* [SG96] as main architectural style: Each pipe represents a model and each filter transforms data from one representation into another. Because the filters are independent of each other, we are able to reuse the adaptors and transformations in many different combinations.

Figure 1 depicts the architecture of the SMA framework. The system specification is twofold: we separate functional and stochastic specification. Using the functional specification, the first step is *state space generation*. Because this is done by external tools, the *input adaptor* transforms the state space representation into our own data structures. In the next step, we extend the state space with the stochastic parts of the specification. The resulting model could subsequently be *transformed*. This is an important step, because most often, the input transition system is not in a form that is easy to analyse. At last, the model is exported through an *output adaptor* and eventually *analysed*.

Thus, we focus on two key aspects: The extension of a state-based system to obtain some kind of stochastic model (such as a Discrete or Continuous Time Markov Chain) and the transformation of this model such that it is easier to analyse. For the modelling and analysis part we are able to reuse existing tools.

The separation of functional and stochastic aspects in the specification is most reasonable since many of the existing approaches to stochastic system modelling extend existing formalisms with stochastic information. Examples are stochastic Petri nets [KBD$^+$94] or stochastic process algebras [BH01]. Since both rely on labelled transition systems it would be possible to build up a tool chain to investigate such systems using our framework.

3 Stochastic Graph Transformation with SGT*

Since our focus is on stochastic graph transformation (SGT), we now illustrate the framework by discussing SGT*, our tool to analyse SGT systems along with a simple example.

A SGT System consists of a Graph Transformation System [Roz97] together with a mapping which associates with each rule a positive real number, the rate of the exponentially distributed application delay of the transformation. We showed in [HLM04] how this leads to a Continuous Time Markov Chain (CTMC).

As a proof of concept, we modelled and analysed an example situation in mobile communications with SGT*. Given a fixed network of base stations, a mobile device can connect to one of them in order to make a call, and disconnect afterwards. A station may be broken with a certain probability, and will then be repaired. The actual state of a station (broken or not broken) is expressed by a boolean attribute, whose value switches between true and false, stochastically triggered by rules *fail* and *repair*. The configuration of such a mobile network can easily and intuitively be represented as an attributed graph. The corresponding rules are shown in Fig. 2 and 3.

Apart from the graph grammar, the specification of a SGT System consists of a table containing the rates of the exponential distribution[3] associated

[3] We assume all tranformations to be exponentially distributed. While this is standard for the reliability of hardware components [Kec93], also call attempt rates und call holding

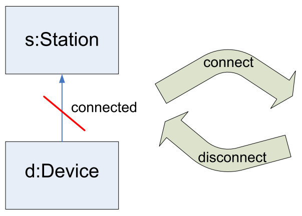

Fig. 2. *connect* and *disconnect*

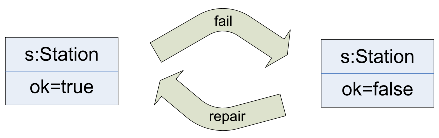

Fig. 3. *fail* and *repair*

rule name p	rate $\rho(p)$	rule name p	rate $\rho(p)$
repair	500	connect	10000
fail	1	disconnect	10000

Fig. 4. Rates associated with the rules

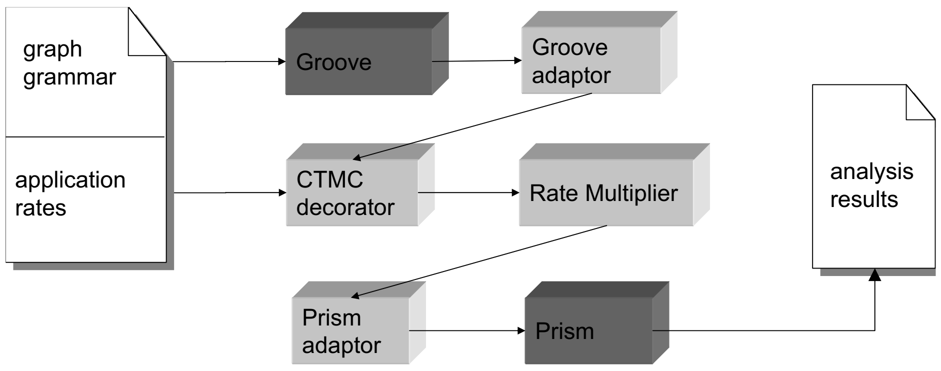

Fig. 5. SGT*

with the rules (Fig. 4). We generate the labelled transition system defined by the graph grammar with GROOVE [Ren04] and use SGT* to combine the GROOVE output with the rates, yielding a CTMC, exported in PRISM [KNP02] format. PRISM is a stochastic model checking tool which allows for a variety of stochastic models and logics, including CTMCs and Continuous Stochastic Logic (CSL).

As Fig. 5 shows, SGT* consists of different components: The GROOVE adaptor to understand GROOVE's representation of a labelled transition system, the *CTMC decorator* which maps the application rates of the transformation rules to the corresponding transitions, a so-called *Rate Multiplier* to replace multiple transitions with identical label, source and target with one transition and the multiplied rate (see [HLM05b]), and the PRISM adaptor to generate a CTMC in the PRISM language.

times are often modelled in this manner (see the discussion in [FCL98]).

We emphasize, that the transformation-step (multiplication of the rates), can be easily extended with additional transformations. This could be used to cope with parameterized rules, prioritized rules, typing information (Groove has no typing-concept)

4 Conclusion and Perspectives

In this paper, we presented the Sma framework and SGT*, our tool for stochastic graph transformation. By using the pipes and filters architecture we gain a lot of flexibility. First, we can easily replace modelling and analysis tools, for example there are other probabilistic model checkers besides Prism or interesting specification languages like PEPA[GH94]. Second, we could also model more complex systems. For example, if a rule is associated with an Erlang distribution, SGT* can be extended by a filter which introduces virtual states into the CTMC in order to simulate the Erlang distribution. Every probability density function with rational Laplace transform can be treated in that manner [Cox55]. At last, apart from CTMCs, we plan to support other stochastic models like Discrete Time Markov Chains or Hybrid Systems [MP03].

References

[BH01] Ed Brinksma and Holger Hermanns. Process algebra and markov chains. In J.-P. Katoen E. Brinksma, H. Hermanns, editor, *FMPA 2000*, number 2090 in LNCS, pages 183–231. Springer, 2001.

[Cox55] D.R. Cox. A use of complex probabilities in the theory of stochastic processes. *Proc. Camb. Phil. Soc., 51, 1955, pp. 313-319*, 51:313–319, 1955.

[FCL98] Yuguang Fang, Imrich Chlamtac, and Yi-Bing Lin. Channel occupancy times and handoff rate for mobile computing and pcs networks. *IEEE Trans. Comput.*, 47(6):679–692, 1998.

[GH94] Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Computer Performance Evaluation*, pages 353–368, 1994.

[HLM04] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic graph transformation systems. In Hartmut Ehrig, Gregor Engels, and Francesco Parisi-Presicce, editors, *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy*, volume 3256 of *LNCS*, pages 210–225. Springer, 2004.

[HLM05a] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Modulare Analyse Stochastischer Graphtransformationssysteme. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 141–152. GI, 2005. ISBN 3-88579-393-8.

[HLM05b] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic Graph Transformation Systems. Technical Report 154, Lehrstuhl fr Softwaretechnologie, Uni-Dortmund, Dortmund, Germany, March 2005.

[KBD$^+$94] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and Giuseppe Conte. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Kec93] Dimitri Kececioglu. *Reliability Engineering Handbook*, volume 1. Prentice Hall, 1993.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[MP03] O. Maler and A. Pnueli, editors. *Hybrid Systems: Computation and Control: 6th International Workshop, HSCC 2003.* LNCS 2623. Springer, 2003.

[Ren04] A. Rensink. The GROOVE simulator: A tool for state space generation. In J.L. Pfaltz, M. Nagl, and B. Bhlen, editors, *Applications of Graph Transformation with Industrial Relevance Proc. 2nd Intl. Workshop AGTIVE'03, Charlottesville, USA, 2003*, volume 3062 of *LNCS*. Springer, 2004.

[Roz97] G. Rozenberg, editor. *Handbook on Graph Grammars: Foundations*, volume 1. World Scientific, Singapore, 1997.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Semi-local model of computations on graphs to break the local symmetry

Work in Progress

Dobiesław Wróblewski [1,2]

Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland

Abstract

We consider finite connected undirected graphs without self-loops as a model of computer networks. The nodes of the graph represent computers or processors, while the edges of the graph correspond to the links between them. We present a model of distributed computations, called semi-local. This extension of the classical local model breaks the local symmetry. As a result, many useful tasks become deterministically solvable in every network assuming a very small initial knowledge about its graph representation. One of these tasks is a creation of a token in an arbitrary anonymous ring – an example of election of a leader. A semi-local solution to this problem is presented.

Key words: Transformations and refinements, verification and analysis, local computations, graph relabelling systems, election, anonymous graphs, rings, token ring networks.

1 Introduction, Related Work

A finite connected labelled graph is a natural model of a computer network. Its nodes represent computers or processors, its edges stand for communication links, and its labelling represents the network state. The labelled graph is called anonymous, if its labelling is uniform. A series of transformations of graph labelling is a model of a computation in the network.

Different models of distributed computations in undirected graphs were presented ([1,2,3,6]). They are called local models of computations. Among these models, the one presented in [3] has the most computational power – if

[1] The author acknowledges support from the Ministry of Scientific Research and Information Technology grant No 3 T11C 006 27 for the years 2004-2005.
[2] Email: `wrobldob@wars.ipipan.waw.pl`

a certain computational task is proved not to be solvable in this model, it is not solvable in the other ones, either. We refer to the model presented in [3] as to the (classical) local model.

Certain tasks are not solvable in this model. The most important example is the election problem in anonymous graphs of arbitrary structure [3]. The weakness of the local model comes from the symmetry of certain types of anonymous graphs. Such graphs are locally indistinguishable from other, not isomorphic ones.

The semi-local model of computations is the least known extension of the classical local model that breaks the local symmetry. As a result, all problems solvable with global methods are also solvable semi-locally [5]. This includes the election problem. In [5] we present a semi-local election protocol for anonymous graphs of arbitrary unknown size and structure. The main drawback of the protocol is the complexity of its definition.

In this paper we present another practical application of the same idea: a semi-local solution to the well-known problem of creation of a unique token in anonymous ring of arbitrary size. This problem is an example of the election task and is proved to be solvable locally only for rings with a priori known prime size [4]. Although the algorithm presented in [5] might be applied in this case without any modifications, we decided to define a new optimised protocol for rings by applying the general idea used in the universal protocol. The new protocol is very simple and readable when defined as a relabelling system. The protocol presented in [5] in turn, was defined in a different formalism and only shown to be definable in terms of relabelling systems (the actual definition was skipped due to its expected complexity and unreadability). Before we define the protocol, we briefly present the semi-local model and compare it with the classical local one.

Standard mathematical notation is used through the paper. The reader is assumed to be familiar with basic notions from graph theory. By convention, we use bold fonts to denote labelled graphs.

The paper is organised as follows. Section 2 introduces the semi-local model of computations. Section 3 defines a semi-local token creation protocol for rings. Then come the conclusions, including a discussion on complexity of the defined algorithm and prospects for further research.

2 Locality and semi-locality

Graph transformations are represented as binary relations in the set of labelled graphs. We say that a transformation T is a *relabelling* if it changes only the labelling, i.e. for all $(\mathbf{G}, \mathbf{G}') \in T$ the underlying graphs of $\mathbf{G}$ and $\mathbf{G}'$ are equal[3].

[3] The requirement of equality (not just isomorphism) has its practical explanation. The underlying graph models the network and the physical structure of the network remains *the*

We say that a relabelling T is *local* in $\mathbf{H}$ iff for all $(\mathbf{G}, \mathbf{G}') \in T$ such that $\mathbf{H}$ is a subgraph of $\mathbf{G}$:

(a) the labelling does not change outside $\mathbf{H}$, and

(b) the change does not depend on the structure or labelling of the graph outside $\mathbf{H}$.

$\mathbf{H}$ is called a locality region of T. Note that if T is local in $\mathbf{H}$, it is also local in every $\mathbf{H}'$ such that $\mathbf{H} \subseteq \mathbf{H}' \subseteq \mathbf{G}$. The minimum locality region of T is denoted as $reg(T)$.

Distributed computations are modelled by sequences of local relabellings. However, the relabellings whose locality regions do not intersect might be applied concurrently.

In the classical *local model* it is required that in every sequence of relabellings, all transformations are local in balls of radius 1[4], i.e. the subgraphs consisting of some node linked with its neighbours (see Fig. 1).

More formally, in the classical local model, for every sequence of labelled graphs $(\mathbf{G}_1, \mathbf{G}_2, ...)$ such that for each $i \in \mathbb{N}$ $(\mathbf{G}_i, \mathbf{G}_{i+1}) \in T_i$ (where T_i is a relabelling), for all $j \in \mathbb{N}$ we have:

$$reg(T_j) \subseteq \mathbf{B}(v_j),$$

where v_j is a node of $\mathbf{G}_j$.

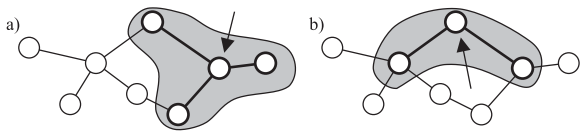

Fig. 1. Two successive relabellings in the local model. Locality regions are indicated with grey background, their centres are pointed with arrows.

In the semi-local model we employ the fact (ignored in the local model) that a distributed protocol might gather a structural knowledge about the network in every step. Namely, if some step of the protocol is a local transformation in a ball $\mathbf{B}(v)$ centred in some node v, we assume that the structure of $\mathbf{B}(v)$ is recognised. Now, take any node $w \in \mathbf{B}(v)$. In the local model, the next transformation (the next step of the protocol) might be local in $\mathbf{B}(w)$. This means, however, that the previously gathered knowledge of the structure of $\mathbf{B}(v)$ would be ignored despite the fact that $w \in \mathbf{B}(v)$. Why not use $\mathbf{B}(v) \cup \mathbf{B}(w)$ as the new locality region?

same after the change of its logical state.

[4] More generally, in balls of some a priori chosen radius $k \in \mathbb{N}$ (a k-local model).

Thus, in the *semi-local* model we allow that in every sequence of relabellings, each transformation is local in some ball of radius 1 [5], or in some connected subgraph that is a sum of such a ball and some locality regions used in the preceding transformations (see Fig. 2).

More formally, in the semi-local model, for every sequence of labelled graphs $(\mathbf{G}_1, \mathbf{G}_2, ...)$ such that for each $i \in \mathbb{N}$ $(\mathbf{G}_i, \mathbf{G}_{i+1}) \in T_i$ (where T_i is a relabelling), for all $j \in \mathbb{N}$ we have:

$$reg(T_j) \subseteq \mathbf{B}(v_j), \text{ or}$$
$$reg(T_j) \text{ is a connected subgraph of } [reg(T_1) \cup ... \cup reg(T_{j-1})] \cup \mathbf{B}(v_j)$$

where v_j is a node of $\mathbf{G}_j$.

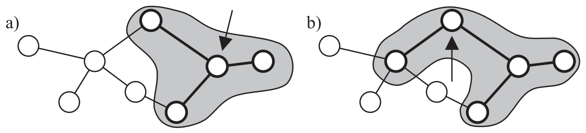

Fig. 2. Two successive relabellings in the semi-local model (compare Fig. 1).

This means that the initial locality regions are balls of radius 1, and then they might grow using local methods (by adding balls of radius 1). Thus, semi-local process still conforms with the intuitive meaning of a local computation, but it is capable of solving all tasks solvable with global methods. Next section provides a representative example.

3 Semi-local creation of a token in a ring

The simplest symmetric network architecture is modelled by a ring – a connected graph in which every node v has exactly two neighbours (let us call them $left(v)$ and $right(v)$. We assume that $left(right(v)) = right(left(v)) = v$ for every node v [6]. Our task is to define a semi-local protocol which starts with an anonymous ring and transforms its initial uniform labelling into such a labelling in which exactly one node is labelled differently than the rest. This node will be given the token. Such a task is a typical example of a leader election and the result labelling breaks the initial symmetry.

The idea of our protocol is quite simple. Let a *group* be a connected subgraph of a ring. The global state of the protocol is a set of groups numbered with non-zero natural numbers. Every node can belong to at most two groups, and it can be *left border*, *interior*, or *right border* of any group it belongs to. If a node does not belong to any group, we call it a *free* node. Initially, the set of groups is empty, thus every node is *free*. In the subsequent steps, groups

[5] More generally, a ball of some a priori chosen radius $k \in \mathbb{N}$ (a k-semi-local model).
[6] This global assumption simplifies our algorithm. However, it can be easily avoided: the definition of the algorithm would be approximately two times longer.

are created (from triples of *free* nodes), extended (by *free* nodes adjacent with *border* nodes) or merged (when two different groups have the same *border* node). The product of each creation, extension or merge is numbered in such a way that any two incident groups have different numbers. After a series of extensions and merges, all nodes belong to the same group and exactly one node is its *left* and *right border*. This node is selected and gets the token.

Let R be any finite ring. R is fixed till the end of Section 3. The set of R's nodes is denoted as V. The local states of nodes are described by the labelling functions $l, i, r : V \to \mathbb{N}$ and $t : V \to \{0, 1\}$ where for each $v \in V$:

- $l(v)$ / $i(v)$ / $r(v)$ – a number of a group for which v is *right border* / *interior* / *left border*, respectively[7] ; they are all 0 for *free* nodes; initially 0,
- $t(v)$ – the indicator of the presence of the token in v; it is 1 if v has the token, otherwise 0; initially 0.

The labelled graph (R, l, i, r, t) is denoted $\mathbf{R}$, the initial labelling is anonymous.

Let $v \in V$. The list of protocol transformations follows. The symbols l, i, r, t denote the labelling before the transformation, whereas the primed symbols l', i', r', t' denote its result.

- If a node v is *free*, let $w = left(v)$ and $x = right(v)$.
  A new group is created from w, v, x, namely:
  $l(v) = i(v) = r(v) = 0 \wedge g_{\max} = max(l(w), r(x)) \wedge$
  $r'(w) = i'(v) = l'(x) = 1 + g_{\max}$ (see Fig. 3a).
- If a node v is *left border* of some group $\mathbf{G}$ and is not a *right border* of any other group[8] , then let $w = left(v)$ and let x be the other *border* node of $\mathbf{G}$. The group $\mathbf{G}$ is extended by w, namely:
  $l(v) = i(v) = 0 \wedge r(v) > 0 \wedge g_{\max} = max(l(w), r(v), r(x)) \wedge$
  $r'(w) = i'(v) = l'(x) = 1 + g_{\max} \wedge r'(v) = 0 \wedge$
  $\forall y \in \mathbf{G} - \{v, x\}\ i'(y) = 1 + g_{\max}$ (see Fig. 3b).
- If a node v is *right border* of some group $\mathbf{G}$ and *left border* some other group $\mathbf{H}$, then let w be the other *border* node of $\mathbf{G}$, and x be the other *border* node of $\mathbf{H}$. The groups $\mathbf{G}$ and $\mathbf{H}$ are merged, namely:
  $l(v) > 0 \wedge i(v) = 0 \wedge r(v) > 0 \wedge g_{\max} = max(l(w), l(v), r(v), r(x)) \wedge$
  $r'(w) = i'(v) = l'(x) = 1 + g_{\max} \wedge l'(v) = r'(v) = 0 \wedge$
  $\forall y \in (\mathbf{G} \cup \mathbf{H}) - \{w, v, x\}\ i'(y) = 1 + g_{\max}$ (see Fig. 3c).
- If a node v is *right border* and *left border* of the same group $\mathbf{G}$, and it does not have a token yet, then it is given the token, namely:

[7] Note that the symbol $l(v)$ corresponds to the text "*right border*". Intuitively speaking, $l(v)$ denotes the number of the group that spans from v to the left (i.e. in the direction pointed by v's *left* neighbour). This means that v is *right border* of the group numbered $l(v)$. The situation is symmetrical for the symbol $r(v)$.
[8] The situation in which v is *right border* and *not* a *left border* is symmetrical.

$t(v) = 0 \wedge l(v) > 0 \wedge i(v) = 0 \wedge r(v) > 0 \wedge l(v) = r(v) \wedge$
$t'(v) = 1$.

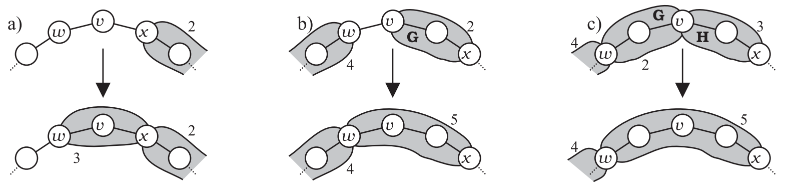

Fig. 3. Examples of a) creation, b) extension and c) merging of groups. The groups are indicated with grey background, their numbers are placed nearby.

The example of a full run of the defined protocol is depicted in Fig. 4

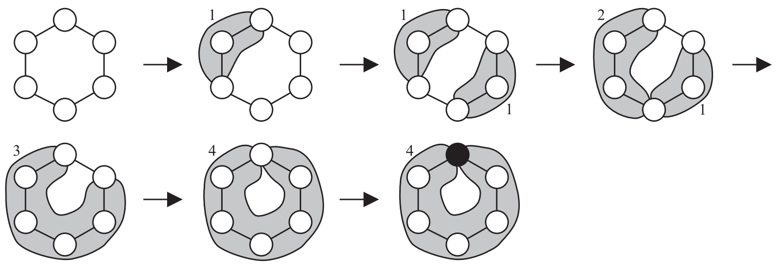

Fig. 4. An example of a run of the algorithm. The selected node that receives the token is indicated with black background.

The scope of this paper does not allow for a detailed discussion of the properties of the defined protocol. Instead, we present the most important properties in the form of the following theorem.

Theorem 3.1 *The defined protocol is semi-local and creates a unique token in R using exactly $|V|$ transformations.*

Proof. The protocol is semi-local because every transformation is a local relabelling either

- in the ball of radius 1 centred in a *free* node, or
- in a group summed with the ball of radius 1 centred in a node that is *left border* of the group and is not a *right border* of any other group, or
- in two different groups whose intersection is a node that is *right border* of the first group and *left border* of the latter, or
- in a single node that is *right border* and *left border* of the same group,

and every group is a locality region used in some previous transformation.

Every run of the protocol uses $|V|$ transformations because:

- every transformation requires a node v such that $i(v) = 0$ and $t(v) = 0$; after the transformation one of these labels changes to non-zero value, but for v only,

- as long as there is a node v such that $i(v) = 0$ and $t(v) = 0$, a transformation might be performed,

and in the initial configuration $i(v) = 0$ and $t(v) = 0$ for all $v \in V$.

All groups created by the transformations of the protocol are given different numbers if they intersect. Thus, if for some node v we have $l(v) = r(v) > 0$, then v is is *right border* and *left border* of the same group. This means that the group contains all nodes of the ring, so for all nodes $w \neq v$ we have $i(w) > 0$, thus only v might be given the token.

On the other hand, subsequent transformations increase the number of nodes w for which $i(w) > 0$. At the same time the appropriate groups are created, extended or merged. As soon as for all $w \neq v$ we have $i(w) > 0$, all nodes belong to the same group. This means that v will be given the token.

□

4 Conclusions

The semi-local model of computations makes several useful tasks deterministically solvable without using global transformations and with employment of very little knowledge about the graph that models the network. A solution to a representative problem was presented.

Future work will include detailed discussion of the properties of the defined protocol, including its complexity measured as the number of actual changes of individual labels. We currently estimate it to be $O(|V|^2)$.

However, our main focus is to define a self-stabilising version of the protocol. We believe that the protocol for rings is a good starting point, because it is by far less complicated than the universal protocol defined in [5]. On the other hand, we hope that achieving self-stabilisation for rings will be easy to generalise for the universal case.

References

[1] Angluin, D., *Local and Global Properties in Networks of Processors*, Proc. of the 12th Symposium on Theory of Computing (1980), 82–93.

[2] Chalopin, J., Y. Métivier, and W. Zielonka, *Election, naming and cellular edge local computations*, Proc. of ICGT'04, LNCS **3256** (2004) 242-256.

[3] Godard, E., and Y. Métivier, *A characterization of families of graphs in which election is possible*, LNCS **2303** (2002), 159–171.

[4] Mazurkiewicz, A., *Solvability of the Asynchronous Ranking Problem*, Information Processing Letters **28/5** (1988), 221–224.

[5] Wróblewski, D., *Universal Semi-local Election Protocol Using Forward Links*, Fundamenta Informaticae **67/1-3** (2005), 287-301.

[6] Yamashita M., and T. Kameda, *Computing on anonymous networks: Part I - characterizing the solvable cases*, IEEE Transactions on Parallel and Distributed Systems **7/1** (1996) 69–89.