To appear in *Information and Computation*

Tile Formats for Located and Mobile Systems*

GianLuigi Ferrari
Dipartimento di Informatica, Corso Italia 40, Università di Pisa
56125, Pisa Italy
E-mail: giangi@di.unipi.it
and
Ugo Montanari
Dipartimento di Informatica, Corso Italia 40, Università di Pisa
56125, Pisa Italy
E-mail: ugo@di.unipi.it

Abstract

Standard SOS formats are limited in their ability to define the operational semantics of process calculi with concurrency, causality and mobility, and with bound names and name generation mechanisms. In this paper we describe a general approach, based on the tile model, to the definition of the operational semantics of process calculi. By providing tile systems for located CCS and asynchronous π-calculus we demonstrate that the proposed approach is more suited than SOS to provide a uniform treatment of concurrency and mobility within a compositional framework.

1 Introduction

Process calculi are usually equipped with notions of operational semantics based on Plotkin's *Structural Operational Semantics* (SOS) [54] and of abstract semantics based on observed actions and bisimilarity. Since De Simone's work on the expressiveness of SCCS-MEIJE [26], several works have dealt with the abstract semantics of process calculi in terms of *meta-theoretic properties* of the SOS method, and have helped to explain some of the most important constructions in process semantics. Examples of these kinds of meta-theoretic

*Research supported in part by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9633363, and by the Information Technology Promotion Agency, Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization). Also research supported in part by U.S. Army contract DABT63-96-C-0096 (DARPA); CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione*; and Esprit Working Groups *CONFER2* and *COORDINA*. Research carried out in part while the second author was on leave at Computer Science Laboratory, SRI International, Menlo Park, USA, and visiting scholar at Stanford University.

results which hold for all calculi whose operational semantics is given by inference rules that fit a certain format (i.e., they satisfy certain syntactic constraints) can be found in [2, 1, 3, 10, 36, 37, 9, 7, 60].

Although powerful and elegant, the meta-theory of SOS formats is designed for labelled transition systems whose states are closed terms and whose labels are uninterpreted actions. These actions are intended to represent the abstract activities processes may do. Since SOS formats only deal with closed terms, the SOS meta-theory cannot be applied to calculi with binding mechanisms (e.g. mobile and higher order process calculi [47, 59]). Moreover, it is often useful to have SOS semantics based on more informative actions. For instance, concurrent abstract semantics is often defined by decorating actions with causality links or with abstract locations and by introducing specialized versions of bisimulation [24, 23, 13, 41, 18, 52, 49]. It is still debatable whether these approaches are natural, or rather whether they are an *ad hoc* addition to the more basic interleaving semantics.

The aim of this paper is to study more general formats than those based on SOS, for which the definition of calculi with mobility, causality and locality becomes natural and uniform. We introduce an approach based on the *tile model* [31, 32, 33], and show that this approach is flexible and general enough to describe the observational semantics of asynchronous π-calculus and CCS with locality.

The tile model relies on certain rewrite rules with side effects, called *tiles*, which are reminiscent both of SOS context systems [43] and of rewriting logic rules [45]. Although tiles can be considered as a generalization of SOS inference rules, tile logic introduces two innovative aspects. First, tiles can be composed in a more general form than SOS rules[1], thus allowing for the definition of open, partially specified systems. Second, the underlying algebraic structure is not limited to ordinary terms. It is the explicit use of a richer algebraic structure which allows us to model concurrent and mobile systems within a compositional framework.

We now briefly introduce the tile model. A tile has the form

$$s \xrightarrow[b]{a} s'$$

and states that the *initial configuration* s of the system evolves to the *final configuration* s' producing an *effect* b. However, s is generally *open* (not closed) and the rewrite step is only possible if the subcomponents of s also evolve producing the *trigger* a. Both trigger and effect are called *observations*, and model the interaction during a computation between the system being described and its environment. More precisely, both system configurations are equipped with an *input* and an *output interface*, and the trigger simply describes the evolution of the input interface during the rewriting; likewise for the effect. It is useful to visualize a tile as a two-dimensional structure (see Fig. 1), where the horizontal dimension corresponds to the extension of the system, while the vertical dimension corresponds to the extension of the computation. We should also imagine a third dimension (the thickness of the tile), which models parallelism: configurations, observations, interfaces and tiles themselves are all assumed to consist of several components in parallel. The initial configuration of a tile α can also be called $\mathtt{north}(\alpha)$, and likewise $\mathtt{south}(\alpha)$, $\mathtt{west}(\alpha)$ and $\mathtt{east}(\alpha)$.

[1] Larsen and Xinxin's contexts [43] are analogous, in this respect, but their algebraic structure is limited to terms and not axiomatized.

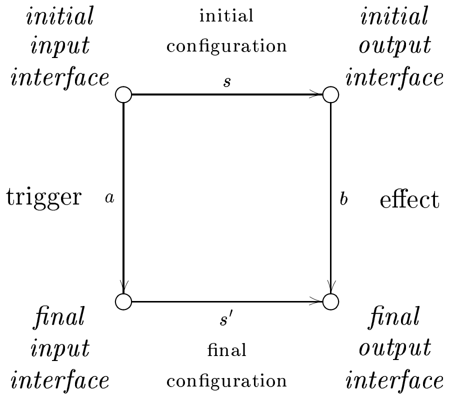


Figure 1: A tile.

Both configurations and observations are assumed to be equipped with parallel and sequential composition (represented by the infix operators $\otimes$ and ; respectively). These allow us to build larger components, extended horizontally for the configurations and vertically for the observations. The tiles themselves possess three operations of composition[2]: parallel ($\_\otimes\_$), horizontal ($\_*\_$), and vertical composition ($\_\cdot\_$). If we consider tiles as logical sequents, it is natural to define the three operations via inference rules called *composition* rules.

The operation of parallel composition is self explanatory. Vertical composition models sequential composition of transitions and computations. Horizontal composition corresponds to rewriting synchronization: the effect of the first tile acts as a trigger of the second tile, and the resulting tile expresses the synchronized behavior of both. Computing in a *tile logic* consists of starting from a set of basic tiles called *rewrite rules* (and from a set of *auxiliary* tiles which depend on the version of the tile model at hand), and applying the composition rules in all possible ways.

A tile logic can also be seen as a double category [27], and tiles themselves as double cells. The categorical interpretation [17, 31, 32] is useful since it makes the model more general (configurations and observations can be arrows of any category), allows for universal constructions (e.g. a tile logic is the double category freely generated by its rewrite rules), and suggests analogies with fruitful concepts of algebraic semantics, such as institutions. However, the tile model is presented here in a purely logical form.

One of the contributions of this paper is the innovative use of *gs-graphs* (a generalization of term graphs) as the underlying algebraic structure for tiles. Term graphs [6] (over a certain signature) are like terms, but two term graphs can explicitly share some of their subterms. Hence, copying the shared subterms to make the two terms disjoint is generally not allowed in a term graph, since this would yield a different term graph. Gs-graphs generalize term graphs in that they are equipped with a hyper-signature where each operator transforms a tuple of input data into a tuple of output data (rather than just into a single datum).

Gs-graphs have several advantages over ordinary syntactic terms. Gs-graphs are useful

[2]Tiles are also equipped with proof terms which distinguish between sequents with the same configurations and observations which are derived in different ways. Suitable axioms for normalizing proof terms are also provided [17, 31, 32, 33].

structures for modeling configurations of distributed systems and their partial ordering observations, since they are equipped with operations of sequential composition and of parallel composition (which models independent juxtaposition), and they can share subcomponents. Sharing is used within configurations for modeling the parallel composition $(\_ \mid \_)$ of process calculi, which in this context means sharing *the same location*. Within observations, sharing expresses the fact that two events share *the same cause*, or, equivalently, that the same location has two different sub-locations. For instance, the process $a.nil \mid b.nil$ is represented by the cogs-graph[3] $G = \{a(e)!, b(e)!\}$. The shared name e represents the common location (the only one in this case) of the two components $a(e)!$ and $b(e)!$. Component $a(e)!$ has one leaf but no root, since the *discharger* operator ! disposes of the result of the subterm $a(e)$. Thus both components $a(e)!$ and $b(e)!$ are arrows from the natural number 0 (i.e. zero roots) to the natural number 1 (i.e. one leaf). Cogs-graphs initiating from 0 represent closed agents, and in fact the discharger operator represents the agent nil. Notice also that names only have a local meaning, i.e. in $G = \{a(e)!, b(e)!\}$ name e represents the only existing leaf. In other words also $\{a(e')!, b(e')!\}$ denotes the same cogs-graph G. Only the *ordering* of names is meaningful.

Gs-graphs are also suitable for representing configurations with shared and possibly bound names. In fact, bound names can be simply modeled as different, shared subterms which consist of the same constant. In the case of (co)gs-graphs, the handling of bound names (with the associated mechanisms for alpha conversion and for generating new names in mobile calculi) can be fully delegated to the underlying logic, while explicit mechanisms are needed in the case of terms.

We show that by taking (co)gs-graphs as the underlying structure for configurations and observations, we obtain a uniform treatment of concurrency and mobility within a compositional framework. In this paper, as case studies, we provide tile systems for located CCS [13] and for asynchronous π-calculus [12, 40]. In both cases we prove that tile bisimulation semantics is equivalent to the ordinary bisimulation semantics for these calculi.

Here we show a few examples of rewrite rules for our calculi. We are aware that they cannot be fully understood at this stage, since we have not explained most of the notations yet, but we feel that it is helpful to have a preview of the main components of our formalism, i.e. tile rewrite rules, which would otherwise be encountered much later in the paper. An example of a rewrite rule for located CCS is:

$$(\mathtt{Prefix}_\lambda) \quad e' := \lambda(e) \xrightarrow[e':=\lambda(e),e:=e]{e':=e'} e' := e', e!.$$

It represents the firing of a prefix λ and corresponds to the following SOS axiom for located CCS:

$$\lambda.p \xrightarrow[l]{\lambda} l :: p.$$

The rule $\mathtt{Prefix}_\lambda$ rewrites the initial configuration consisting of the cogs-graph $\{e' := \lambda(e)\}$ into the final configuration $\{e' := e', e!\}$. The initial input interface and the final input interface are represented by the name e', and the trigger by the identity gs-graph $\{e' := e'\}$. This means that the rule applies to an open process, where the missing subprocess is

[3]Cogs-graphs are like gs-graphs. While gs-graphs are oriented from leaves to roots, cogs-graphs are visited from roots to leaves.

represented by the name e', but this subprocess is only required to produce a trivial (identity) trigger $\{e' := e'\}$. The initial output interface is represented by name e and the final output interface by names e, e'. The effect of rule $\mathtt{Prefix}_\lambda$ is the gs-graph $\{e' := \lambda(e), e := e\}$. Thus the rule creates a new location e' in its effect, which however turns out to be the location (name) of the missing subprocess. The name e representing the location of the initial process is kept in the final configuration (via the operator $\_!$) since it could be shared by other processes. The tile relevant for process $\lambda.p$ is the horizontal composition $id_{[\![p]\!]} * \mathtt{Prefix}_\lambda$, where $id_{[\![p]\!]}$ is the identity tile of the configuration associated with process p. Notice that the tile is deterministic, while the SOS axiom yields an infinite branching since l can be any location.

In the case of asynchronous π-calculus, we distinguish two kinds of tiles: *activity* tiles and *coordination* tiles. Activity tiles describe the evolution of processes without considering the operational context where processes live. As a consequence, they have no triggers, but only effects. Coordination tiles, on the other hand, define the behaviour of coordinators. They determine the evolution of processes inside their operational environment by taking coordinating actions in response to actions of processes. For asynchronous π-calculus, coordination tiles take care also of *creating* and *sharing names*.

As an example of an activity tile, let us consider the `Input` tile which models a message entering the system:

$$(\mathtt{Input})\ e!, a! \xrightarrow[e'b := input(e,a), a := a]{} M(e', a, b).$$

Notice that the `Input` rule restricts messages entering the system to those messages whose subject is a name which is already known (the argument a of *input*). As happens in [4], in this paper we consider the so-called *ground semantics*, where the formal parameter of the input prefix is treated as a fresh name. The effect is the graph $\{e'b := input(e, a), a := a\}$, which creates a new name b (the object). The effect replaces e in the output interface with a new event e'. Messages are represented by means of a symbol M of the signature, with three variables: the current event, the subject and the object.

Notice that names (e.g. e, e', a and b) have a purely local scope. The only name sharing mechanism between two configurations or tiles is via explicit composition. For instance, the fact that the object is new can be determined from the above rule purely by inspection.

As an example of a coordination rule, let us consider the `Open` rule which models the extrusion of a name while a message is leaving the system.

$$(\mathtt{Open})\ ea := ea, b := \nu \xrightarrow[e'b := Output(e,a), a := a]{e' := output(e,a,b), ab := ab} e'ab := e'ab$$

The trigger is the graph $\{e' := output(e, a, b), ab := ab\}$, and the application of the rule causes the transformation of the initial configuration $\{ea := ea, b := \nu\}$ into the final configuration $\{e'ab := e'ab\}$ (where the restriction is removed) and the creation of the effect $\{e'b := Output(e, a), a := a\}$. Notice that the effect creates a new name, which is clearly "the same" name b which occurred restricted in the initial configuration.

1.1 Outline of the Paper

The paper is organized as follows. Section 2 introduces gs-graphs and their various representations. Section 3 presents the tile model, in the simplified version needed in the paper, and tile bisimulation. Section 4 defines the tile system for located CCS. We show that the equality of computations specified by commuting diamonds (see e.g. [28, 14]) holds in the tile system, and that tile bisimilarity yields, in the weak case, the same equivalence on processes as ordinary location bisimilarity. Section 5 defines a tile system for asynchronous π-calculus and proves the equivalence with ordinary semantics. Preliminary versions of some sections have appeared in [29, 30]. In this paper, the notation is unified and simplified, and most proofs are given in more detail.

2 Gs-Graphs

In this section we introduce *gs-graphs*, the data structure we use to represent the configurations and the observations of our tile systems. We will introduce three different notations for gs-graphs to fulfill different needs:

(i) a basic definition, where a gs-graph is a multiset of variable assignments (up-to alpha conversion of the variables). Operations of sequential and parallel composition and atomic constants are also defined;

(ii) an axiomatic definition (taken from [20, 21]) of the above algebra;

(iii) a compact representation which closely corresponds to the basic definition.

Axiomatic definition (ii) is needed as a basis of the version of the tile model presented in the paper. Moreover, the algebraic structure will be essential in the proofs. The concrete version of the algebra presented in (i) is useful to provide both a more concrete, comprehensible definition of gs-graphs and the starting point for representation (iii), which is the one actually used in rewrite rules and examples. Gs-graphs presented in the basic definition would be too cumbersome, on the other hand operations are not easily defined on compact representations. Although the axiomatic description of gs-graphs has been already presented in [20, 21], the basic and compact definitions of gs-graphs are an original contribution of this paper (a preliminary presentation appeared in [30]).

A gs-graph is very similar to a tuple of typed terms with variables, equipped with an environment which orders the variables and assigns types to them. There are two differences:

1. gs-graphs can explicitly share some subgraphs. In other words, copying shared subgraphs of a gs-graph is not allowed, since it would produce a different gs-graph. This is the case of *term graphs* [6] too, which are in fact a special case of gs-graphs;

2. gs-graphs are different from term graphs (and terms) in that they are equipped with hyper-signatures, which can contain symbols returning not only one result, but possibly any number of them, including zero. The latter case accounts for *open graphs* [50], a compositional version of ordinary graphs.

Note that ordinary tuples of terms are commonly presented using notations analogous to (i)–(iii) above. In particular, Lawvere's axiomatization of terms [44] is essentially the same as the axiomatization of gs-graphs, except for two additional naturality axioms for terms, which mean that shared subterms can be copied and unused subterms can be discarded.

Thus, to familiarize the reader with our notations, we first introduce them informally on terms.

2.1 Terms

The usual way of presenting tuples of typed terms with variables is via syntactic judgments of the form:

$$\Gamma \triangleright t : \sigma$$

where Γ is an environment, and σ is a type on a type system equipped with product types and with a first order signature Σ. In this subsection we will not give formal definitions, but we will rely on a descriptive running example:

$$x : s_2, y : s_1 \triangleright \langle x, f(x), h(g(f(x)))\rangle : s_2 s_1 s_2$$

where s_1 and s_2 are basic types.

Concrete and *abstract* judgments differ since the latter are defined up to alpha conversion on the environment Γ. An abstract judgment $T = \Gamma \triangleright t : \sigma$ can be considered as an arrow $T : \sigma' \to \sigma$ from $\sigma' = [\![\Gamma]\!]$ to σ, where for instance $[\![x : s_2, y : s_1]\!] = s_2\, s_1$. The elementary judgments are:

- *generators*

$$f = x_1 : s_1, \ldots, x_k : s_k \triangleright f(x_1, \ldots, x_k) : s \text{ with } f \in \Sigma_{s_1,\ldots s_k,s}$$

- *identities*

$$id_s = x : s \triangleright \langle x\rangle : s, id_{\texttt{unit}} = \triangleright\langle\rangle : \texttt{unit},$$

- *permutations*

$$\rho_{s,s'} = x : s, y : s' \triangleright \langle y, x\rangle : s's,$$

- *duplicators*

$$\nabla_s = x : s \triangleright \langle x, x\rangle : ss,$$

- *dischargers*

$$!_s = x : s \triangleright \langle\rangle : \texttt{unit}.$$

where `unit` is the type of the empty tuple.

Two judgments can be composed in parallel (via concatenation of environments and types) and sequentially (via substitution, if the target type of the first coincides with the source type of the second). For instance, we have:

$$id_s \otimes (x : s \triangleright \langle g(x)\rangle : s') = x : s, y : s \triangleright \langle x, g(y)\rangle : ss'$$

$$\nabla_s; (x : s, y : s \triangleright \langle f(x, y)\rangle : s') = x : s \triangleright \langle f(x, x)\rangle : s'.$$

The algebra of judgments (with the above constants and operations) has been axiomatized by Lawvere [44]. The axiomatization consists of that of gs-monoidal theories[4] (see the Appendix), with the following two additional axioms:

$$T; \nabla_s \quad = \quad \nabla_\sigma ; T \otimes T$$

$$T; !_s \quad = \quad !_\sigma$$

where $T : \sigma \to s$, and judgments ∇_σ and $!_\sigma$ are the obvious generalizations of ∇_s and $!_s$ to a generic product type σ.

Notice that in the first axiom the judgment T, which is shared in the left member, is duplicated in the right member, whereas in the second axiom it is erased in the right member.

Since gs-graphs have explicit sharing of subterms, an ordinary syntactic judgment is not adequate to introduce the notation we need. Our running example can be transformed as follows:

$$x : s_2, y : s_1, u : s_2 \ ; \ v : s_1, w : s_2 \rhd \{u := x, v := f(x), z := g(v), w := h(z)\}$$

where the body of the judgment is now a set of variable assignments, and two environments declare leaf and root variables. However, explicit variable declarations would make our notation cumbersome. We thus assume a predefined global environment:

$$\Gamma_0 = \ldots, u : s_2, \ldots, x : s_2, \ldots, y : s_1, \ldots, v : s_1, \ldots, w : s_2, \ldots, z : s, \ldots$$

which assigns (unique) types to all variables and orders them. Once we have eliminated the environment, we might hope to distinguish leaf and root variables by the fact that they are the minima and maxima of the partial ordering implicit in the set of assignments. However, it is clear that the set:

$$\{u := x, v := f(x), z := g(v), w := h(z)\}$$

would not be enough, since the leaf variable y (which does not appear in the set) and the root variable v would be lost. We thus introduce additional assignments $x := y$ in order to make clear which are the roots, and a new kind of assignment $x!$. The latter assignment is reminiscent of the discharger judgment $!_s$ and should be understood as an assignment to an empty list of variables. Hence, our running example becomes:

$$\{u := x, z_1 := f(x), z_2 := g(z_1), z_3 := h(z_2), v := z_1, w := z_3, x!, y!, z_1!, z_2!, z_3!\}.$$

Here, assignments of the form $x := y$ have been introduced for all the root variables, while all the other variables appear in assignments of the form $x!$. The notation above corresponds to our basic definition of gs-graphs, and has the advantage of being a normal form (i.e. two judgments coincide if they have the same form) and of allowing an easy definition of parallel and sequential composition[5].

[4] Lawvere's theories only require an ordinary signature, rather than a hyper-signature as gs-monoidal theories.

[5] This would not be the case if assignments of the form $x!$ were introduced only for leaf variables not appearing in any assignment. For instance, $\nabla_s; !_s \otimes id_s = id_s$ would be represented as $\{u := x, v := x\}; \{y!, w := z\} = \{x!, w := x\}$, where the right member is not in normal form.

The compact form we now present is aimed at being shorter and at introducing as few variables as possible. In fact, in the calculi discussed in this paper, variables have semantic meanings and useless duplications would make the development less comprehensible. First, we eliminate useless assignments of the form $x!$:

$$\{u := x, z_1 := f(x), z_2 := g(z_1), z_3 := h(z_2), v := z_1, w := z_3, y!\}.$$

We also eliminate assignments of the form $x := y$ when y is a maximal element of the partial ordering and it is not a leaf variable. To avoid changing the root variables (and thus possibly their ordering), we also replace y for x in all the other assignments. Our example becomes:

$$\{u := x, z_1 := f(x), z_2 := g(z_1), w := h(z_2), v := z_1, y!\}.$$

We now notice that the internal variable z_2 is only used once. Thus it can be safely replaced by its value:

$$\{u := x, z_1 := f(x), w := h(g(z_1)), v := z_1, y!\}.$$

Finally, we introduce assignments of the form $x := x$ to mark root variables without introducing redundant variables. The compact form of our example thus becomes:

$$\{u := x, v := f(x), w := h(g(v)), v := v, y!\}.$$

2.2 Gs-graphs: Basic Definition

Term graphs (over a certain signature) are like terms, but two term graphs can explicitly share some of their subterms. Gs-graphs provide a smooth generalization of term graphs in that they are equipped with a hyper-signature where each operator transforms a tuple of input data into a tuple of output data (rather than just into a single datum).

Definition 2.1 *(many-sorted hyper-signatures)*

Given a set S of sorts, a *hyper-signature* (hereafter a *signature*) Σ is a family $\{\Sigma_{u,v}\}$, with $u, v \in S^*$, of sets of *operators*. $\square$

Intuitively, an operator $f \in \Sigma_{u,v}$, $u = s_1 \ldots s_h$ and $v = s'_1 \ldots s'_k$ tells us how an h-tuple of input data of sorts $s_1, \ldots, s_h$ is transformed into a k-tuple of output data of sorts $s'_1, \ldots, s'_k$. Notice that ordinary (not hyper) signatures have $\Sigma_{u,v} = \emptyset$ for $v \notin S$. We now introduce gs-graphs.

Definition 2.2 *(gs-graphs)*

Let Σ be a signature. We consider the extended signature Σ' where $\Sigma'_{s,\epsilon} = \{!_s\} \cup \Sigma_{s,\epsilon}$, with $s \in S$; and where $\Sigma'_{u,v} = \Sigma_{u,v}$, with $u \notin S$, or $v \neq \epsilon$, where $!_s$ are special postfix operators[6]. Furthermore, let V be a totally ordered (by $\leq$) set of *names*, a name being denoted by n or similar letters. Names are sorted (we write $n : s$), and there is an infinite number of names for every sort.

A *concrete gs-graph* G consists of

[6]We will often overload operator $!_s$ by writing $!$ instead of $!_s$.

- a *multiset* of *proper assignments* of the form $n'_1 \dots n'_k := f(n_1, \dots, n_h)$, where $f \in \Sigma_{u,v}$, being $u = s_1 \dots s_h$ and $v = s'_1 \dots s'_k$, with $n_i : s_i,\ i = 1 \dots h$ and $n'_j : s'_j,\ j = 1 \dots k$. When $f \in \Sigma_{\epsilon,v}$, the assignments will be written as $n'_1 \dots n'_{|v|} := f$, while when $f \in \Sigma_{u,\epsilon}$ we will simply write $f(n_1, \dots, n_{|u|})$;

- a *set* of *auxiliary assignments* of the form $n := n'$ or of the form $n!$.

When a name appears in the left (right) member of an assignment we say it is *assigned* (*used*). We require that every name is assigned at most once. Thus every proper assignment with $f \in \Sigma_{u,v}$, $v \neq \epsilon$, must appear in G with a multiplicity of at most one. Furthermore, if we let $n' \sqsubseteq n$ whenever n is assigned and n' is used in the same assignment, then relation $\sqsubseteq$ must be acyclic. Names not assigned in G are called *leaves*, while names n assigned in $n := n'$ are called *roots* and must be maximal elements of relation $\sqsubseteq$. Finally, an auxiliary assignment of the form $n!$ must be present if n occurs in G, but must not be present if n is a root[7].

Given a concrete gs-graph G, $\texttt{leaves}(G)$ (or simply `leaves`) is the list of its leaves ordered according to $\leq$. Similarly $\texttt{roots}(G)$ (or simply `roots`) is the list of roots.

An *abstract gs-graph* (or simply a *gs-graph*) is a concrete gs-graph defined up to isomorphism, where two concrete gs-graphs G and G' are isomorphic if G' can be obtained from G by applying to it an injective substitution of names which respects the sorts of the names and the total ordering $\leq$ of leaves and roots. The set of all gs-graphs on a signature Σ is called GS_Σ.

Given a gs-graph G, let u and v be the lists of sorts of its lists of leaves `leaves` and of roots `roots`, respectively[8]. Thus G can be seen as an arrow of *type* $u \to v$. We write $G : u \to v$ and call u and v the *source* and *target* of G respectively. □

Example 2.3 Let $s_1, s_2 \in S$; $f \in \Sigma_{s_1,s_2}$; $d \in \Sigma_{s_1 s_2, s_2 s_1 s_2}$; $p \in \Sigma_{s_2,\epsilon}$; $n_1, n_2, n_4, n_5, n_8, n_{10}, n_{11} : s_1$; and $n_3, n_6, n_7, n_9 : s_2$, with $n_1 \leq \dots \leq n_{11}$. Thus the following multiset G of assignments

$$\begin{array}{ll} \text{Proper:} & n_2 := f(n_1),\ n_3 n_5 n_6 := d(n_1, n_2),\ p(n_3),\ p(n_3) \\ \text{Auxiliary:} & n_4 := n_1,\ n_7 := n_6,\ n_8 := n_1,\ n_{11} := n_{10}, \\ & n_1!,\ n_2!,\ n_3!,\ n_5!,\ n_6!,\ n_9!,\ n_{10}! \end{array}$$

is a concrete gs-graph. Its lists of leaves and roots are

$$\texttt{leaves} = (n_1, n_9, n_{10}) \quad \texttt{roots} = (n_4, n_7, n_8, n_{11})$$

and its type is

$$G : s_1 s_2 s_1 \to s_1 s_2 s_1 s_1.$$

□

The following gs-graphs which are called *elementary* are particularly interesting. They are defined for all $s, s' \in S$.
generators: for every $f \in \Sigma_{u,v}$

$$f : u \to v = \{n'_1 \dots n'_{|v|} := f(n_1, \dots, n_{|u|});\}$$

[7]However a name n can only be used in an assignment $n!$.

[8]Notice that u and v do not depend on the choice of a concrete gs-graph for G.

identities: id_s and id_ϵ

$$id_s : s \to s = \{n_2 := n_1, n_1!\} \qquad id_\epsilon : \epsilon \to \epsilon = \emptyset;$$

permutations: $\rho_{s,s'}$

$$\rho_{s,s'} : ss' \to s's = \{n_4 := n_1, n_3 := n_2, n_1!, n_2!\};$$

duplicators: ∇_s

$$\nabla_s : s \to ss = \{n_2 := n_1, n_3 := n_1, n_1!\};$$

dischargers: $!_s$

$$!_s : s \to \epsilon = \{n!\}.$$

We now introduce two operations on gs-graphs. The *sequential composition* of two gs-graphs is obtained by gluing the list of roots of the first graph with the list of variables of the second, and it is defined only if their types are equal. The *parallel composition* instead is always defined, and it is a sort of disjoint union where variable and root lists are concatenated.

Definition 2.4 *(sequential and parallel composition of gs-graphs)*

Given two gs-graphs $G_1 : u \to v$ and $G_2 : v \to w$, let us take two instances in their isomorphism classes such that $\mathtt{roots}(G_1) = \mathtt{leaves}(G_2)$ and that no other names are shared between G_1 and G_2. Furthermore, let A be the set of assignments in G_1 of the form $n := n'$, and let σ be the corresponding name substitution. The *sequential composition* of G_1 and G_2 is the gs-graph

$$G_1; G_2 : u \to w = (G_1 \setminus A) \cup G_2\sigma.$$

Notice that we have $\mathtt{leaves}(G_1; G_2) = \mathtt{leaves}(G_1)$ and $\mathtt{roots}(G_1; G_2) = \mathtt{roots}(G_2)$.

Given two gs-graphs $G_1 : u_1 \to v_1$ and $G_2 : u_2 \to v_2$, let us take two instances in their isomorphism classes such that no names are shared between G_1 and G_2; and for every name n_1 in G_1 and n_2 in G_2 we have $n_1 \le n_2$. The *parallel composition* of G_1 and G_2 is the gs-graph

$$G_1 \otimes G_2 : u_1u_2 \to v_1v_2 = G_1 \cup G_2.$$

Notice that we have $\mathtt{leaves}(G_1 \otimes G_2) = \mathtt{leaves}(G_1)\mathtt{leaves}(G_2)$ and $\mathtt{roots}(G_1 \otimes G_2) = \mathtt{roots}(G_1)\mathtt{roots}(G_2)$. □

We close this section by stating some basic properties of gs-graphs. These results (and their proofs) are immediate extensions of Corradini and Gadducci's results for term-graphs [20, 21].

Proposition 2.5 (decomposition of gs-graphs)

Every gs-graph can be obtained by evaluating some expression containing only elementary gs-graphs as constants, and sequential and parallel composition as operators. □

For instance the gs-graph in Example 2.3 can be represented as

$$G = (\nabla_{s_1}; \nabla_{s_1} \otimes \nabla_{s_1}; id_{s_1} \otimes (id_{s_1} \otimes f; d; (\nabla_{s_2}; p \otimes p) \otimes !_{s_1} \otimes id_{s_2}) \otimes id_{s_1}) \otimes !_{s_2} \otimes id_{s_1}.$$

The following proposition gives a characterization of gs-graphs as *gs-monoidal theories.* A gs-monoidal theory is a logical theory similar to, but weaker than, the algebraic (Lawvere) theories [44] of terms and substitutions we mentioned in Section 2.1. The differences are the extended signature and the missing naturality axioms for duplicators and dischargers. Gs-monoidal theories are briefly presented in the Appendix.

It is immediate to see that elementary gs-graphs and the operations of sequential and parallel composition of gs-graphs are the "concrete" counterparts of the corresponding structures of gs-monoidal theories. For instance, the gs-monoidal counterpart of name sharing is provided by the absence of naturality of duplicator ∇.

Proposition 2.6 (characterization of gs-graphs)

The gs-graphs on the signature Σ *are the arrows of the* gs-monoidal theory $\mathbf{GS}(\Sigma)$ *generated by* Σ. $\square$

2.3 Compact gs-graphs

A more compact representation of concrete gs-graphs can be provided. The compact representation provides a useful way to concisely write complex gs-graphs in a linear form.

The compact representation of a gs-graph is obtained by the following sequence of steps.

- (i) An auxiliary assignment $n!$, where n is used in some other assignment, is omitted;
- (ii) an auxiliary assignment $n := n'$ is omitted, and n is replaced by n' in the other assignments, where n' is a maximal element, except for n, of the partial ordering $\sqsubseteq$, provided that n' is not a leaf. After the first two steps, the roots of the gs-graph are the maximal names of relation $\sqsubseteq$;
- (iii) a proper assignment $n' := f(n_1, \ldots, n_h)$, with $f \in \Sigma_{u,s}$, $s \in S$, where n' is used linearly in exactly another assignment, is deleted after replacing the only other occurrence of n' with $f(n_1, \ldots, n_h)$;
- (iv) if n' is not a leaf, an assignment $n := n'$ is replaced by $n := n$ and n is replaced for n' in the remaining assignments. If n' is a leaf, and there is no root n'' with $n \leq n'' \leq n'$ or $n' \leq n'' \leq n$, assignment $n := n'$ is replaced by $n' := n'$, provided that the assignment $n' := n'$ has not already appeared in the representation;
- (v) sets of assignments $n_1 := n'_1, \ldots, n_h := n'_h$ are written as multiple assignments $n_1 \ldots n_h := n'_1 \ldots n'_h$.

The first two steps delete all the auxiliary assignments except those that are needed to mark names which are roots but not maximals, or names which are maximals but not roots. Step (iii) creates *nested* assignments of the form

$$n'_1 \ldots n'_k := f(C_1(n_1, \ldots, n_m), \ldots, C_h(n_1, \ldots, n_m))$$

where $C_i, i = 1, \ldots, h$, are suitable contexts on $\{\Sigma_{u,s}\}, s \in S$. This allows us to include the ordinary representation of terms in the concise notation. Finally, step (iv) avoids the introduction of useless names, a convenient feature whenever names have a mnemonic meaning.

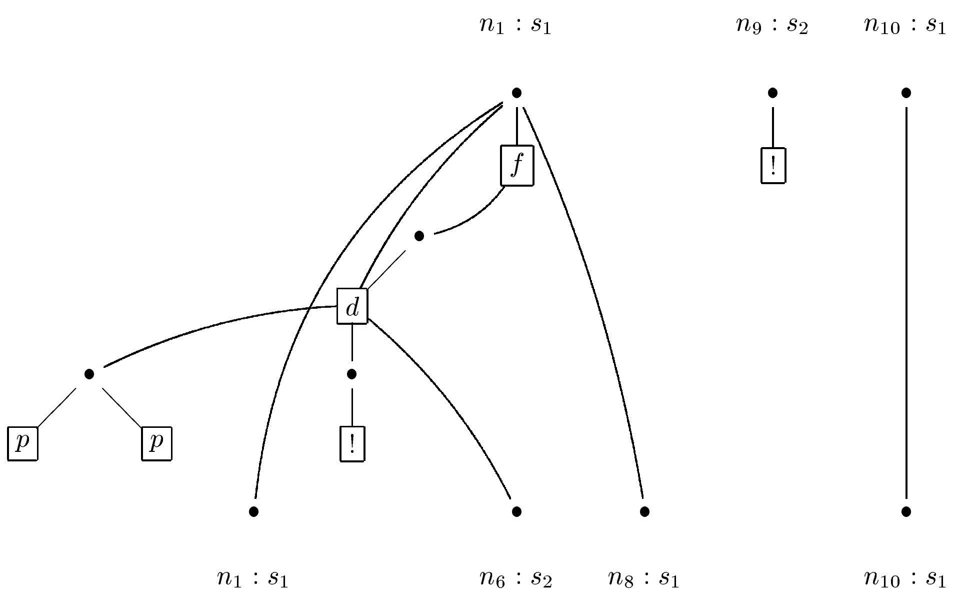


Figure 2: Wire-and-box diagrams for gs-graphs

For instance, the compact representation for the gs-graph of the Example 2.3 is $G = \{n_3 n_5 n_6 := d(n_1, f(n_1)),\ p(n_3),\ p(n_3),\ n_1 n_8 n_{10} := n_1 n_1 n_{10},\ n_5!,\ n_9!\}$. Notice that the assignment $n_8 := n_1$ cannot be replaced by $n_1 := n_1$ because this assignment is already present in the representation. A concrete gs-graph can be reconstructed from its compact representation, since the above steps can all be easily reversed. Notice that when checking for equality of gs-graphs, it is necessary to revert to the basic definition, since a concrete gs-graph may have more than one compact representation: for instance $\{n_1 := n_2,\ n_3 := n_2,\ n_2!\}$ can be represented as both $\{n_2\, n_3 := n_2\, n_2\}$ and $\{n_1\, n_2 := n_2\, n_2\}$.

We can represent gs-graphs by *wire-and-box* diagrams: names are represented by dots, (dots must respect the ordering over names) while their occurrences are represented by wires leaving them, and the operator symbols are denoted by boxes labeled with the name of the operator. For instance, the gs-graph of Example 2.3 is illustrated by the diagram in Figure 2. Notice that the partial ordering of compact gs-graphs grows downward in the wire-and-box representation. Finally, wire overlapping, wire duplication and wire termination are *auxiliary*: they are the basic components of any wire-and-box diagrams independently of the operators.

Figure 3 illustrates the wire-and-box representation of elementary gs-graphs.

2.4 CoGs-Graphs

Wire-and-box diagrams describing gs-graphs are read from top to bottom. However, such diagrams can be equivalently read bottom up. Reading bottom up leads to the notion of *cogs-graphs*. Cogs-graphs are gs-graphs where the arrows of types, and the order of factors in sequential composition, are all reversed. In particular, the arrows of generators must also be reversed, and $\nabla_s : ss \to s$ and $!_s : 0 \to s$. However, assignments in the basic and compact

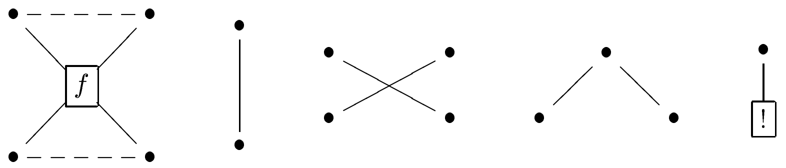


Figure 3: Generators, identities, permutations, duplicators and dischargers

definitions still have the same polarity since we want to keep the property that a name is never assigned twice. Thus also the notions of roots and leaves are preserved, i.e. the arrows of types go from leaves to roots in gs-graphs, and from roots to leaves in cogs-graphs.

The set of all cogs-graphs over a signature Σ is called $CoGS_\Sigma$.

For instance the gs-graph of Example 2.3 can be described by the following expression

$$G : s_1 s_2 s_1 \to s_1 s_2 s_1 s_1 =$$
$$(\nabla_{s_1}; \nabla_{s_1} \otimes \nabla_{s_1}; id_{s_1} \otimes (f \otimes id_{s_1}; d; (\nabla_{s_2}; p \otimes p) \otimes !_{s_1} \otimes id_{s_2}) \otimes id_{s_1}) \otimes !_{s_2} \otimes id_{s_1}$$

while, given the signature Σ' with $f \in \Sigma'_{s_2,s_1}$, $d \in \Sigma'_{s_2 s_1 s_2, s_1 s_2}$ and $p \in \Sigma'_{\epsilon,s_2}$, the graph in $CoGS_{\Sigma'}$ obtained by reading bottom up is represented by the expression

$$G' : s_1 s_2 s_1 s_1 \to s_1 s_2 s_1 =$$
$$((id_{s_1} \otimes (((p \otimes p; \nabla_{s_2}) \otimes !_{s_1} \otimes id_{s_2}); d; f \otimes id_{s_1}) \otimes id_{s_1}); \nabla_{s_1} \otimes \nabla_{s_1}; \nabla_{s_1}) \otimes !_{s_2} \otimes id_{s_1}.$$

3 The Tile Model

We now describe the basic features of the tile model, in the version where observations are gs-graphs and configurations are cogs-graphs. The presentation of the tile model follows [32], however the tile sequents we have here (the *flat* sequents) are not decorated by proof terms. Hereafter we will simply call them tile sequents.

3.1 Tile Logic

Definition 3.1 *(tile sequent, tile rewrite system)*

Let Σ_H and Σ_V be two (many sorted) signatures, called the *horizontal* and the *vertical* signature respectively, on the same set of sorts S.

A Σ_H-Σ_V *tile sequent* is a quadruple $g \xrightarrow[b]{a} h$, where $g : u \to w$ and $h : y \to z$ are cogs-graphs on Σ_H, while $a : u \to y$ and $b : w \to z$ are gs-graphs on Σ_V. Graphs g, h, a and b are called the *initial configuration*, the *final configuration*, the *trigger*, and the *effect* of the tile, respectively. Trigger and effect are called observations. Sort strings u, w, y and z are called the *initial input interface*, the *initial output interface*, the *final input interface,* and the *final output interface*, respectively.

A *tile rewrite system* (TRS) $\mathcal{R}$ is a triple $\langle \Sigma_H, \Sigma_V, R \rangle$, where R is a set of Σ_H-Σ_V sequents called *rewrite rules.* □

A TRS $\mathcal{R}$ can be considered as a logical theory, and new sequents can be derived from it via certain inference rules.

Definition 3.2 *(tile logic)*

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, R \rangle$ be a TRS. Then we say that $\mathcal{R}$ *entails* the class $\mathbf{R}$ of the tile sequents $g \xrightarrow[b]{a} h$ obtained by finitely many applications of the following inference rules

basic rules:

$$(generators)\ \frac{g \xrightarrow[b]{a} h \in R}{g \xrightarrow[b]{a} h \in \mathbf{R}}$$

$$(h\text{-}refl)\ \frac{g : u \to w \in CoGS_{\Sigma_H}}{id_g = g \xrightarrow[id_w]{id_u} g \in \mathbf{R}} \qquad (v\text{-}refl)\ \frac{a : u \to w \in GS_{\Sigma_V}}{id_a = id_u \xrightarrow[a]{a} id_w \in \mathbf{R}};$$

composition rules:

$$(p\text{-}comp)\ \frac{\alpha = g \xrightarrow[b]{a} h, \alpha' = g' \xrightarrow[b']{a'} h' \in \mathbf{R}}{\alpha \otimes \alpha' = g \otimes g' \xrightarrow[b \otimes b']{a \otimes a'} h \otimes h' \in \mathbf{R}}$$

$$(h\text{-}comp)\ \frac{\alpha = g \xrightarrow[c]{a} h, \alpha' = g' \xrightarrow[b]{c} h' \in \mathbf{R}}{\alpha * \alpha' = g; g' \xrightarrow[b]{a} h; h' \in \mathbf{R}}$$

$$(v\text{-}comp)\ \frac{\alpha = g \xrightarrow[b]{a} k, \alpha' = k \xrightarrow[b']{a'} h \in \mathbf{R}}{\alpha \cdot \alpha' = g \xrightarrow[b;b']{a;a'} h \in \mathbf{R}};$$

auxiliary rules: (permutations) $u, w \in S$ implies

$$\rho^{0,0}_{u,w} = \rho_{u,w} \xrightarrow[id_{wu}]{\rho_{u,w}} id_{wu} \in \mathbf{R} \qquad \rho^{0,1}_{u,w} = \rho_{w,u} \xrightarrow[\rho_{u,w}]{id_{wu}} id_{wu} \in \mathbf{R}$$

$$\rho^{1,0}_{u,w} = id_{wu} \xrightarrow[id_{wu}]{\rho_{w,u}} \rho_{u,w} \in \mathbf{R} \qquad \rho^{1,1}_{u,w} = id_{wu} \xrightarrow[\rho_{w,u}]{id_{wu}} \rho_{w,u} \in \mathbf{R}.$$

□

Basic rules provide the sequents corresponding to rewrite rules, together with suitable identity tiles, whose intuitive meaning is that an element of $CoGS_{\Sigma_H}$ can be rewritten to itself using only trivial triggers and effects. Likewise for GS_{Σ_V}. Composition rules provide all the possible ways in which sequents can be composed.

Auxiliary rules provide all the consistent rearrangements of the interfaces. For instance, a tile denoted by $\rho^{0,1}_{u,w}$ consists of a horizontal permutation on the *initial configuration* (notice the character 0 as the *first* upper index) of the tile, and of the inverse permutation on the *effect observation* (notice the character 1 as the *second* upper index). The remaining sides are identities, likewise for the other permutation tiles. The role of permutation tiles is to permute the names on one vertex of the tile (the initial output interface in the example), yet still maintaining the same connections between the adjacent gs-graphs. For instance, given any tile $\alpha = g \xrightarrow[b]{a} h$ with g and b having u and w as target and source respectively, the composition $(id_s * \rho^{0,1}_{w,u}) \cdot (\alpha * id_b)$ produces the tile $\alpha' = g; \rho_{u,w} \xrightarrow[\rho_{w,u};b]{a} h$.

Here two permutation graphs have been introduced, but the connections between the two original horizontal and vertical graphs, represented by the composition $g; \rho_{u,w}; \rho_{w,u}; b$, are still the same as $g; b$, since $\rho_{u,w}; \rho_{w,u} = id_{uw}$.

The presence of permutation graphs in both horizontal and vertical structures and the possibility of consistent rearrangements as allowed by permutation tiles is a main difference from the formulation of tile logic presented in [32], where permutation graphs were not allowed in observations. Notice also that while duplicators and dischargers appear in both horizontal and vertical graphs, no auxiliary tiles are available for them. The choice of which rearrangements are allowed between horizontal and vertical structures is an essential design parameter. A detailed treatment of this topic can be found in [17].

It can be seen that permutation tiles $\rho^{0,1}_{u,w}$, $\rho^{1,0}_{u,w}$, and $\rho^{1,1}_{u,w}$ can all be obtained from $\rho^{0,0}_{u,w}$ by horizontal and vertical compositions with identity tiles.

It is also easy to see that, by horizontal and vertical composition of the auxiliary tiles, one can obtain permutation tiles $\rho_1 \xrightarrow[\rho_2]{\rho_3} \rho_4$ with arbitrary permutation graphs ρ_1, ρ_2, ρ_3 and ρ_4 on the four sides, provided that $\rho_1; \rho_2 = \rho_3; \rho_4$.

Especially interesting are those tiles obtained by composing vertically, horizontally and in parallel only h-refl tiles and permutation tiles. These correspond to computation steps which freely rearrange the orderings of roots and leaves in the horizontal graphs, but keep track of them via corresponding permutations as triggers and observations. Hereafter, these tiles will be called *perm tiles*.

Let α be a tile. Then, $\texttt{north}(\alpha)$ is defined to be the initial configuration of α; $\texttt{south}(\alpha)$ the final configuration of α; $\texttt{west}(\alpha)$ the trigger of α; and $\texttt{east}(\alpha)$ the effect of α.

3.2 Tile Bisimulation

It is straightforward to define the notion of bisimilarity in the framework of the tile model. Indeed, a tile rewrite system $\mathcal{R} = \langle \Sigma_H, \Sigma_V, R \rangle$ naturally induces a labeled transition system whose states are the horizontal graphs, whose labels are pairs of vertical graphs (triggers and effects), and whose transition relation is given by the set of tiles entailed by $\mathcal{R}$.

Hence, we can directly apply the definition of bisimulation to the transition system induced by a tile rewrite system. This leads to the notion of tile bisimilarity.

Definition 3.3 *(tile bisimilarity).* Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, R \rangle$ be a TRS. A symmetric equivalence relation $\equiv_b \subseteq CoGS_{\Sigma_H} \times CoGS_{\Sigma_H}$ is a *tile bisimulation* for $\mathcal{R}$ if, whenever $g \equiv_b h$ for generic g, h elements of $CoGS_{\Sigma_H}$, then for any sequent $\alpha = g \xrightarrow[b]{a} g'$ entailed by $\mathcal{R}$ there exists a corresponding one $\beta = h \xrightarrow[b]{a} h'$ with $g' \equiv_b h'$. The maximal tile bisimulation equivalence is called *tile bisimilarity*, and denoted by $\sim_t$. □

When triggers are empty (i.e. they are the gs-graph id_ϵ), tiles represent computations of closed systems and their bisimulations rely only on effects. However, the definition above allows us to use directly the standard notion of bisimulation for open (partially specified) systems. This differs from the approach followed by Rensink [56], where generalizations of bisimulation and transition systems are introduced to deal with open systems.

4 Process Calculi: Concurrent and Located Semantics

Process calculi are usually equipped with notions of operational semantics based on transition systems and of abstract semantics based on observed actions and bisimilarity. Sometimes it is useful to consider concurrency as a primitive notion, rather than to reduce it to nondeterminism via interleaving. For this purpose, ordinary transition systems have been extended in the literature in several ways. From an operational point of view, certain *commuting diamonds* are introduced (see e.g. [28, 14]), whose role is to define as concurrent those pairs of events which can occur in any order. Concurrent abstract semantics is defined instead by decorating actions with causality links or with abstract locations and possibly by introducing specialized versions of bisimulation [24, 23, 13, 41, 18, 52, 49]. However, while the concurrent semantics of process calculi has been given much attention in the past few years, it is still debatable whether the existing approaches are natural, or rather whether they are an *ad hoc* addition to the more basic interleaving semantics. We believe a more natural treatment of concurrency is possible, as we feel has been achieved (at least from an operational point of view) for other models of computations, like Petri nets [25] and term [45, 42], graph [19], and term graph [21] rewriting, where axioms generating commuting diamonds are automatically imposed by the framework of definition.

The aim of this section is to propose a treatment of concurrent process calculi where the same operational and abstract concurrent semantics described in the literature now descend from general, uniform notions.

4.1 A Simple Process Calculus

There are many concurrent models for process calculi. Some of them focus on the *operational* aspects, defining certain *concurrent machines* for the calculi. Other models are equipped with notions of observation that can capture causal dependencies or localities, and define *abstract semantics*, usually via bisimulation. In this section we try to combine both aspects by considering a simple process calculus (basically a subset of Milner's CCS [46]) with locations. We equip the calculus with both the concurrent operational and the abstract location semantics. This simple process calculus is close to those presented in the literature [28, 14, 13, 41, 18, 49].

The basic idea of location semantics [13] is to associate a different location with each sequential component of a process, in order to allow the external observer to see an action together with the location where it takes place. Hence, processes $a.b.nil + b.a.nil$ and $a.nil \mid b.nil$ are distinguished, since the second process can perform a and b in different locations, while the first process cannot.

To define the concurrent operational semantics, we follow the approach of [28] which associates an n-ary operator to each SOS rule with n-premises, and then imposes certain χ *axioms* on the resulting algebra of transitions and computations. Concurrent computations are equivalence classes in this formal system. In [35] it is proved that when applied to full CCS, the same equivalence is induced by mapping CCS into Petri nets. It is usually conjectured that the same equivalence can also be derived by following the approach based on proved transition systems and residuals [14].

We first show a strong version of the operational semantics, where locations are visible also in the case of synchronization. We then present a different synchronization rule, which hides locations. Besides being closer to the tile version, this presentation of the locality transition system allows us to avoid defining two different kinds of transitions (i.e. *standard* and *location* transitions) as in [13].

Let Δ be the alphabet for basic actions and $\overline{\Delta}$ the alphabet of complementary actions ($\Delta = \overline{\overline{\Delta}}$ and $\Delta \cap \overline{\Delta} = \emptyset$); the set $\Lambda = \Delta \cup \overline{\Delta}$ will be ranged over by λ. Let $\tau \notin \Lambda$ be an invisible action, and let $\Lambda \cup \{\tau\}$ (ranged over by μ) be the set of actions. Let *Loc* be a totally ordered (by $<$) denumerable set of *locations* (ranged over by l). Finally, ℓ will be used to denote either a location or the distinguished symbol ϵ, u will be used to indicate a sequence of locations (ϵ being the null string), and k to denote a tuple of sequences.

We distinguish between located processes $p, p', \ldots$, and ground processes (i.e. processes without locations) $g, g', \ldots$. The syntax of the calculus is defined by the following grammar.

$$
\begin{array}{lcl}
p & ::= & g \quad | \quad l :: p \quad | \quad p \mid p \\
g & ::= & nil \quad | \quad \lambda.g \quad | \quad g + g \quad | \quad g \mid g.
\end{array}
$$

The calculus does not include restriction, although there is no problem with it. Recursion could also be handled by introducing *activity* rules (as done in Section 5). Finally, since our main concern here is generating and handling new locations, τ prefixes are not considered (although they are necessary to provide axiomatizations of location based behavioral equivalences).

We use $loc(p)$ and $loc(k)$ to indicate the set of location names occurring in process p or in the tuple k of strings. Notice that $loc(g) = \emptyset$ for each ground process g. Throughout the paper we assume that a newly generated location is different from all the locations already present in the same process. This restriction does not appear in [13]. In [41], however, it has been pointed out that no discriminating power is added if we are allowed to choose a location twice in a computation and that our definition is equivalent to the one in [13].

4.1.1 Operational Semantics

Table 1 shows the inference rules for the *strong located transition systems.* Labels of transitions consist of actions and of strings of locations, denoted by u. A synchronization transition is labelled by two strings. The generic denotation is k. In lk, location l is concatenated with each string in k. The *bound* location(s) of a transition is (are) the last location(s) of its string(s).

We can briefly comment on the rules of Table 1. Rule (Act) generates a name for a new location, hence we have a transition for each location (infinite branching). In rule (Loc) the side condition $l \notin loc(k)$ ensures that only fresh names are generated; a similar interpretation applies to the side conditions of rules (Comp) and (Synch).

When it is clear from the context we will use $t : p \xrightarrow{w} p'$ to denote a transition in the strong located transition system.

Example 4.1 As an example of application of the rules, we have that the proof term

$$[a, l_4, g_1 >\rfloor g_2 \mid l_3 :: ([\overline{a}, l_1, g_4 >\rfloor g_5 \rfloor l_2 :: g_3)$$

$$
\begin{array}{ll}
(Act) & [\lambda, l, g >: \lambda.g \xrightarrow[l]{\lambda} l :: g \\[2ex]
(Sum) & \dfrac{t : g \xrightarrow[k]{\mu} p}{t <+g' : g + g' \xrightarrow[k]{\mu} p} \qquad\qquad \dfrac{t : g \xrightarrow[k]{\mu} p}{g' +> t : g' + g \xrightarrow[k]{\mu} p} \\[3ex]
(Loc) & \dfrac{t : p \xrightarrow[k]{\mu} p', l \notin loc(k)}{l :: t : l :: p \xrightarrow[lk]{\mu} l :: p'} \\[3ex]
(Comp) & \dfrac{t : p_1 \xrightarrow[k]{\mu} p_1', loc(k) \cap loc(p) = \emptyset}{t \rfloor p : p_1 \mid p \xrightarrow[k]{\mu} p_1' \mid p} \qquad \dfrac{t : p_1 \xrightarrow[k]{\mu} p_1', loc(k) \cap loc(p) = \emptyset}{p \lfloor t : p \mid p_1 \xrightarrow[k]{\mu} p \mid p_1'}
\end{array}
$$

$$
(Synch)\ \frac{t_1 : p_1 \xrightarrow[u_1]{\lambda} p_1', t_2 : p_2 \xrightarrow[u_2]{\overline{\lambda}} p_2', loc(u_1) \cap loc(p_2') = \emptyset = loc(u_2) \cap loc(p_1')}{t_1 \mid t_2 : p_1 \mid p_2 \xrightarrow[u_1, u_2]{\tau} p_1' \mid p_2'}
$$

Table 1: Strong located transition system

describes the proof of the transition

$$
a.g_1 \mid g_2 \mid l_3 :: (\overline{a}.g_4 \mid g_5 \mid l_2 :: g_3) \xrightarrow[l_4, l_3 l_1]{\tau} l_4 :: g_1 \mid g_2 \mid l_3 :: (l_1 :: g_4 \mid g_5 \mid l_2 :: g_3).
$$

This will be the running example we will use throughout the paper to explain most of our constructions. $\square$

As another example, the synchronization of process $l_1 :: a.nil \mid l_2 :: l_3 :: \overline{a}.nil$ is described by the following transition:

$$
\begin{array}{l}
l_1 :: [a, l_4, nil > \mid l_2 :: l_3 :: [\overline{a}, l_5, nil >: \\
\qquad\qquad l_1 :: a.nil \mid l_2 :: l_3 :: \overline{a}.nil \quad \xrightarrow[l_1 l_4, l_2 l_3 l_5]{\tau} l_1 :: l_4 :: nil \mid l_2 :: l_3 :: l_5 :: nil.
\end{array}
$$

We can now introduce the concurrency relation χ.

Definition 4.2 *(concurrency relation χ)*

Let $(-\text{ then }-\chi-\text{ then }-)$ be a quaternary relation on transition proof terms, defined as the least commutative[9] relation generated by the structural rules of Table 2, with $t : p \xrightarrow[k]{\mu} q$, $t_i : p_i \xrightarrow[k_i]{\mu_i} q_i$, $t_i' : p_i' \xrightarrow[k_i']{\mu_i'} q_i'$. $\square$

The concurrency relation χ identifies the *diamonds* in the strong located transition system. The axiom defines the basic diamonds, while the inductive rules reproduce the diamonds in all possible contexts. For instance, for the process $l_1 :: a.nil \mid l_2 :: l_3 :: \overline{a}.nil$ we have:

[9] Namely, $(t_1 \text{ then } t_2\, \chi\, t_3 \text{ then } t_4)$ iff $(t_3 \text{ then } t_4\, \chi\, t_1 \text{ then } t_2)$.

$$t_1 \rfloor p_2 \text{ then } q_1 \lfloor t_2 \,\chi\, p_1 \lfloor t_2 \text{ then } t_1 \rfloor q_2 \qquad \frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{l :: t_1 \text{ then } l :: t_2 \,\chi\, l :: t_3 \text{ then } l :: t_4}$$

$$\frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{t_1 < +g \text{ then } t_2 \,\chi\, t_3 < +g \text{ then } t_4} \qquad \frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{g+ > t_1 \text{ then } t_2 \,\chi\, g+ > t_3 \text{ then } t_4}$$

$$\frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{t_1 \rfloor p \text{ then } t_2 \rfloor p \,\chi\, t_3 \rfloor p \text{ then } t_4 \rfloor p} \qquad \frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{p \lfloor t_1 \text{ then } p \lfloor t_2 \,\chi\, p \lfloor t_3 \text{ then } p \lfloor t_4}$$

$$\frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{t_1 \mid t \text{ then } t_2 \rfloor q \,\chi\, t_3 \rfloor p \text{ then } t_4 \mid t} \qquad \frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4}{t \mid t_1 \text{ then } q \lfloor t_2 \,\chi\, p \lfloor t_3 \text{ then } t \mid t_4}$$

$$\frac{t_1 \text{ then } t_2 \,\chi\, t_3 \text{ then } t_4, t_1' \text{ then } t_2' \,\chi\, t_3' \text{ then } t_4'}{t_1 \mid t_1' \text{ then } t_2 \mid t_2' \,\chi\, t_3 \mid t_3' \text{ then } t_4 \mid t_4'}$$

Table 2: The Concurrency Relation

$$\begin{gathered} l_1 :: [a, l_4, nil >\rfloor l_2 :: l_3 :: \overline{a}.nil \text{ then } l_1 :: l_4 :: nil \lfloor l_2 :: l_3 :: [\overline{a}, l_5, nil > \\ \chi \\ l_1 :: a.nil \lfloor l_2 :: l_3 :: [\overline{a}, l_5, nil > \text{ then } l_1 :: [a, l_4, nil >\rfloor l_2 :: l_3 :: l_5 :: nil. \end{gathered}$$

We now introduce the notion of *model of computation* and then the model of computation associated with the location semantics. Computations are sequences of transitions and a model of computation is a transitively closed transition system together with an equivalence relation on computations, where equivalent computations are meant to execute the same event in different orders. In the located model of computation, equivalent computations are determined by the diamonds of the χ relation.

Definition 4.3 A model of computation is a structure $\mathcal{M} = (S, C, \partial_0, \partial_1, id, ;, \equiv)$ where

- S is a set of *states;*
- C is a set of *computations*;
- ∂_0, ∂_1 are functions mapping computations into states; $s_1 = \partial_0(c)$ ($s_2 = \partial_1(c)$) is the initial (final) state of computation c. We write $c : s_1 \Longrightarrow s_2$ for $\partial_0(c) = s_1$, and $\partial_1(c) = s_2$;
- $id : S \rightarrow C$, where $id(s) : s \Longrightarrow s$ is the *identity* computation associated with state s. We write $id(s)$ as id_s;
- ; is an associative partial function with
$$\frac{c : s_1 \Longrightarrow s_2, c' : s_2 \Longrightarrow s_3}{c; c' : s_1 \Longrightarrow s_3} \qquad \frac{c : s_1 \Longrightarrow s_2}{id_{s_1}; c = c = c; id_{s_2}}\ ;$$
- $\equiv$ is an equivalence relation on computations respecting ;, i.e. $c_1 \equiv c_1'$ and $c_2 \equiv c_2'$ imply $c_1; c_2 \equiv c_1'; c_2'$.

The *strong located* model of computation $\mathcal{L}_S$ is the freely generated model of computation where states are located (or ground) processes, every transition in the strong located transition system is a computation, and t_1 then $t_2 \,\chi\, t_3$ then t_4 implies $t_1; t_2 \equiv t_3; t_4$. $\Box$

4.1.2 Abstract Semantics

The inference rule for the abstract semantics is obtained by replacing rule $(Synch)$ in Table 1 with the following rule:

$$(Synch')\frac{t_1 : p_1 \xrightarrow[u_1 l_1]{\lambda} p_1', t_2 : p_2 \xrightarrow[u_2 l_2]{\overline{\lambda}} p_2'}{t_1 \mid t_2 : p_1 \mid p_2 \xrightarrow{\tau} d(p_1', l_1) \mid d(p_2', l_2)}$$

where $d(p, l)$ deletes l in p, i.e. replaces $l :: p'$ with p' in p. Notice that now in the rules tuple k of strings of locations may be empty, and in that case lk is also empty.

Using $(Synch')$, our running example (Example 4.1) becomes:

$$a.g_1 \mid g_2 \mid l_3 :: (\overline{a}.g_4 \mid g_5 \mid l_2 :: g_3) \xrightarrow{\tau} g_1 \mid g_2 \mid l_3 :: (g_4 \mid g_5 \mid l_2 :: g_3).$$

as defined in [13].

We now introduce the notion of location bisimilarity. We adopt the standard notation for weak transitions: $\stackrel{\epsilon}{\Longrightarrow} = (\xrightarrow{\tau})^*$ and $\underset{u}{\stackrel{\lambda}{\Longrightarrow}} = \stackrel{\epsilon}{\Longrightarrow}\xrightarrow[u]{\lambda}\stackrel{\epsilon}{\Longrightarrow}$. Weak transitions define a *weak located transition system.*

Definition 4.4 *(location bisimilarity)*

A symmetric relation $\mathcal{R}$ is a *location bisimulation* if $q_1 \mathcal{R} q_2$ implies:

- for each $q_1 \underset{ul}{\stackrel{\lambda}{\Longrightarrow}} q_1'$, with $loc(l) \cap loc(q_2) = \emptyset$, there exists some $q_2 \underset{ul}{\stackrel{\lambda}{\Longrightarrow}} q_2'$ with $q_1' \mathcal{R} q_2'$;
- for each $q_1 \stackrel{\epsilon}{\Longrightarrow} q_2'$ there exists some $q_2 \stackrel{\epsilon}{\Longrightarrow} q_2'$ with $q_1' \mathcal{R} q_2'$.

Two processes q and q' are *location bisimilar* $(q \approx_l q')$ if $q \mathcal{R} q'$ for some location bisimulation $\mathcal{R}$. $\Box$

It has been noticed [49] that the whole list of locations does not need to be observed, but it is enough to truncate it by keeping the two rightmost locations. More precisely, $tr(ul_1 l_2)$ returns $l_1 l_2$, while $tr(l) = l$.

Definition 4.5 *(incremental location bisimilarity)*

A symmetric relation $\mathcal{R}$ is an *incremental location bisimulation* if $p_1 \mathcal{R} p_2$ implies:

- for each $p_1 \underset{ul}{\stackrel{\lambda}{\Longrightarrow}} p_1'$, with $loc(l) \cap loc(p_2) = \emptyset$, there exists some $p_2 \underset{vl}{\stackrel{\lambda}{\Longrightarrow}} p_2'$ with $tr(ul) = tr(vl)$ and $p_1' \mathcal{R} p_2'$;
- for each $p_1 \stackrel{\epsilon}{\Longrightarrow} p_2'$ there exists some $p_2 \stackrel{\epsilon}{\Longrightarrow} p_2'$ with $p_1' \mathcal{R} p_2'$.

Two processes p and p' are *incremental location bisimilar* $(p \approx_L p')$ if $p \mathcal{R} p'$ for some incremental location bisimulation $\mathcal{R}$. $\Box$

In [49] it has been shown that incremental location bisimilarity coincides with location bisimilarity for ground processes. Notice, however, that the two notions do *not* coincide on located processes. For instance, given $p_1 = l_1 :: a.nil|l_2 :: b.nil$ and $p_2 = l_1 :: (a.nil|l_2 :: b.nil)$ we have that p_1 is incremental location bisimilar to p_2 but p_1 is not location bisimilar to p_2. We will implement on tiles the latter notion of incremental bisimilarity.

Given a computation $c : p \Longrightarrow q$ of the strong located model of computation $\mathcal{L}_S$, it is possible to derive a weak computation $\mathtt{wkc}(c) : p \Longrightarrow q'$ by replacing $(Synch)$ with $(Synch')$ in every transition of c. It is easy to see that the sequence $\mathtt{wkl}(w)$ of labels of $\mathtt{wkc}(c)$ depends only on the sequence of labels w of c, and not on c itself. In fact, $\mathtt{wkl}(w)$ replaces every label $\xrightarrow[u_1 l_1' l_1, u_2 l_2' l_2]{\tau}$ in w with $\xrightarrow{\tau}$, and substitutes $l_1' l_1$ with l_1' and $l_2' l_1$ with l_2' in all the following labels of w. Here, l_1' and/or l_2' can be missing, and in this case l_1 and l_2 are simply erased. Similarly, it is easy to see that q' depends only on q and w, i.e. we can write $q' = \mathtt{wkp}(q, w)$. In fact, for every label $\xrightarrow[u_1 l_1, u_2 l_2]{\tau}$ in w, function $\mathtt{wkp}(q, w)$ erases locations l_1 and l_2 in q.

4.2 Tiles for Concurrent and Located Calculi

We show how the framework provided by the tile model can be applied to provide a natural concurrent and located semantics for the process calculus presented in the previous section. We now introduce the components of the tile rewrite system, i.e. horizontal signature, vertical signature, and rewrite rules. In this case both the horizontal signature and the vertical signature are one-sorted, hence a rank function will be used to specify the input and output arities of each operator.

4.2.1 Horizontal Structure

The symbols of the signature Σ_H, and their arities, are as follows

$$\begin{array}{ll} \lambda : 1 \to 1 & (Prefix) \\ + : 2 \to 1 & (Choice) \\ \kappa : 1 \to 0 & (Eraser). \end{array}$$

Configurations $s, s', s_1, \ldots$ of the tile rewrite system are cogs-graphs over the signature Σ_H. The operator κ will be used for making inactive the processes refused in a choice.

Before presenting the translation function $[\![\_]\!]$ that maps processes p into configurations, it is convenient to introduce a structural congruence $\cong$ on processes. It is the smallest congruence, such that

1. $|$ is associative, commutative, and nil is its identity,
2. $l :: (p \mid q) \cong l :: p \mid l :: q$, $\epsilon :: g \cong g$,
3. $l :: (p \mid l' :: p') \cong l :: p \mid l' :: p'$.

Note that law 3 allows location prefixes to be pulled into the outermost position. A located agent thus becomes a flat record structure, where the locations are the fields, and the ground processes the values. For example, if $p = l_1 :: (l_7 :: a.nil \mid l_2 :: b.nil)$ then

$$\begin{array}{rcl} p = l_1 :: (l_7 :: a.nil \mid l_2 :: b.nil) & \cong & l_1 :: (l_7 :: a.nil) \mid l_2 :: b.nil \\ & \cong & l_1 :: (nil \mid l_7 :: a.nil) \mid l_2 :: b.nil \\ & \cong & l_1 :: nil \mid l_7 :: a.nil \mid l_2 :: b.nil \\ & \cong & l_1 :: nil \mid l_2 :: b.nil \mid l_7 :: a.nil \\ & \cong & nil \mid l_1 :: nil \mid l_2 :: b.nil \mid l_7 :: a.nil \end{array}$$

These structural axioms (originally introduced in [49]) formalize the idea (also exploited in [18]) that in a location semantics processes can be viewed as collections of sequential subprocesses running on different locations. This leads to the following definition.

Definition 4.6 A located process p is in *normal form* if it can be written as

$$p = g_0 \mid l_1 :: g_1 \mid \ldots \mid l_n :: g_n$$

with $l_i \neq l_j$, $i \neq j$, $i, j = 1, \ldots, n$ and $l_1 < l_2 < \ldots < l_n$.

Here we assume that $\mid$ associates to the left and that $g_0, \ldots, g_n$ are normal forms, e.g. they are obtained by replacing $g \mid nil$ with g and by lexicographically ordering g_1 and g_2 in $g_1 \mid g_2$. □

It is immediate to see that every class of the structural congruence $\cong$ contains exactly one process in normal form.

For instance, the normal form of the located processes

$$a.g_1 \mid g_2 \mid l_3 :: (\overline{a}.g_4 \mid g_5 \mid l_2 :: g_3)$$

of our running example (Example 4.1) is

$$(a.g_1 \mid g_2) \mid l_2 :: g_3 \mid l_3 :: (\overline{a}.g_4 \mid g_5)$$

where $l_2 < l_3$. Furthermore, the labelled transition describing the synchronization becomes

$$(a.g_1 \mid g_2) \mid l_2 :: g_3 \mid l_3 :: (\overline{a}.g_4 \mid g_5) \xrightarrow[l_4, l_3 l_1]{\tau} g_2 \mid l_1 :: g_4 \mid l_2 :: g_3 \mid l_3 :: g_5 \mid l_4 :: g_1$$

where $l_1 < \ldots < l_4$.

We are now ready to introduce the function mapping processes in normal forms into horizontal graphs.

Definition 4.7 *(from located processes to configurations)*

Let p be a located process in normal form. Then $[\![p]\!] : 0 \to |loc(p)| + 1$ is the configuration corresponding to process p, where function $[\![\_]\!]$ is inductively defined as follows.

$$\begin{array}{l} [\![nil]\!] = !_1 \\ [\![\lambda.g]\!] = [\![g]\!]; \lambda \\ [\![g_1 + g_2]\!] = [\![g_1]\!] \otimes [\![g_1]\!]; +, \\ [\![g_1 \mid g_2]\!] = [\![g_1]\!] \otimes [\![g_1]\!]; \nabla_1, \\ [\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_n :: g_n]\!] = [\![g_0]\!] \otimes [\![g_1]\!] \otimes \ldots \otimes [\![g_n]\!] \end{array}$$

□

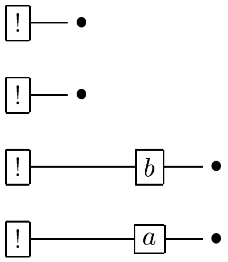


Figure 4: The configuration of process $nil \mid l_1 :: nil \mid l_5 :: b.nil \mid l_7 :: a.nil$

Hereafter, we will consider $[\![p]\!]$ as being extended also to any located process p not in normal form. We have $[\![p]\!] = [\![p']\!]$, where p' is the unique process in normal form structurally equivalent to p.

Some examples should make the mapping clear. We start with ground processes.

$$[\![a.b.nil]\!] = b(a(e))!$$
$$[\![a.nil \mid b.nil]\!] = a(e)!, b(e)!$$
$$[\![(a.nil \mid b.nil) \mid (c.nil + d.nil)]\!] = a(e)!, b(e)!, c(e_1)!, d(e_2)!, e_1 e_2 = +(e).$$

Considering now a located process $p = l_1 :: (l_7 :: a.nil \mid l_5 :: b.nil)$ we have the normal form $p' = nil \mid l_1 :: nil \mid l_5 :: b.nil \mid l_7 :: a.nil$ and the expression:

$$[\![p]\!] = !_1 \otimes !_1 \otimes (!_1; b) \otimes (!_1; a) : 0 \to 4,$$

or the cogs-graph:

$$[\![p]\!] = e_0!, e_1!, b(e_2)!, a(e_3)!,$$

or the wire-and-box diagram of Figure 4.

The initial configuration of our running example is

$$([\![g_1]\!]; a \otimes [\![g_2]\!]); \nabla \otimes [\![g_3]\!] \otimes ([\![g_4]\!]; \overline{a} \otimes [\![g_5]\!]); \nabla.$$

Proposition 4.8 *Let P be the subset of cogs-graphs which consists of all the cogs-graphs of type $0 \to n$, with $n > 0$, over the signature Σ_H, but without the eraser operator κ. Also, let G be the subset $\{s \mid s : 0 \to 1 \in P\}$. Function $[\![\_]\!]$ maps bijectively ground processes to G, and surjectively located processes to P.*

Proof. (Outline)

We define $[\![\_]\!]^{-1}$ inductively as follows, exploiting the decomposition property of gs-graphs (Proposition 2.5).

- $[\![!]\!]^{-1} = nil$,
- $[\![s_1 \otimes s_2; +]\!]^{-1} = [\![s_1]\!]^{-1} + [\![s_2]\!]^{-1}$ with $s_i : 0 \to 1, i = 1, 2$,
- $[\![s_1 \otimes s_2; \nabla]\!]^{-1} = [\![s_1]\!]^{-1} \mid [\![s_2]\!]^{-1}$ with $s_i : 0 \to 1, i = 1, 2$, and $[\![s_1]\!]^{-1}$, $[\![s_2]\!]^{-1}$ lexicographically ordered.

- $[\![s_0 \otimes s_1 \otimes \ldots \otimes s_n]\!]^{-1} = \{g_0 \mid l_1 :: g_1 \mid \ldots \mid l_n :: g_n$ such that $g_0 = [\![s_0]\!]^{-1}, g_1 = [\![s_1]\!]^{-1}, \ldots, g_n = [\![s_n]\!]^{-1}, l_1 < \ldots < l_n\}$, with $s_i : 0 \to 1, i = 0, \ldots, n$.

In fact, there is no elementary cogs-graph $s : n \to m$ with $m > 1$. Thus, whenever we have $s' : 0 \to m$ with $m > 1$ we can decompose s' uniquely into the parallel composition of m graphs $s_i : 0 \to 1$, with $i = 1, \ldots, m$, using the last rule. Furthermore, given $s'' : 0 \to 1$ we can decompose it inductively starting from the leaf using the other rules. □

4.2.2 Vertical Structure

The symbols of Σ_V are as follows

$$\begin{array}{ll} \lambda : 1 \to 1 & (\mathit{Action}) \\ T : 2 \to 2 & (\mathit{Synchronization}). \end{array}$$

Observations $o, o', o_1, \ldots$ of the tile rewrite system are gs-graphs over the signature Σ_V.

In the vertical graphs that we will generate with our rules, the discharger operator will never appear, since we will never deallocate a location. Also, the duplicator operator will not appear freely, but rather only within certain derived operators called *action operators* which we will introduce below. They correspond, to a large extent, to the actions of the located calculus, but where the creation of a new location associated with every visible action is structurally implemented via the duplicator operator.

The first action operator, called λ-operator, corresponds to action λ:

$$\xrightarrow{\lambda} : \ 1 \to 2$$

$$\xrightarrow{\lambda} = \nabla_1 \ ; \ id_1 \otimes \lambda.$$

When in compact form, the corresponding graph will be:

$$e' := \lambda(e), e := e$$

with $e < e'$. Here e' is the new sub-location created by the occurrence of the action λ, while e is the present location.

We have two action operators for synchronization. The first, called *τ_s-operator*, works on subprocesses located at different locations, while the second, called *τ-twin* or *τ_t-operator*, works within the same location.

$$\xrightarrow{\tau_s} : \ 2 \to 4$$

$$\xrightarrow{\tau_s} = \nabla_1 \otimes \nabla_1 \ ; \ id_1 \otimes T \otimes id_1 \ ; \ id_1 \otimes \rho_{2,1}.$$

In compact form:

$$\xrightarrow{\tau_s} = e_1' e_2' := T(e_1, e_2), e_1, e_2 := e_1, e_2$$

with $e_1 < e_2 < e_1' < e_2'$

$$\xrightarrow{\tau_t} : \ 1 \to 3$$

$$\xrightarrow{\tau_t} = \nabla_1 \ ; \ id_1 \otimes (\nabla_1 \ ; \ T)$$

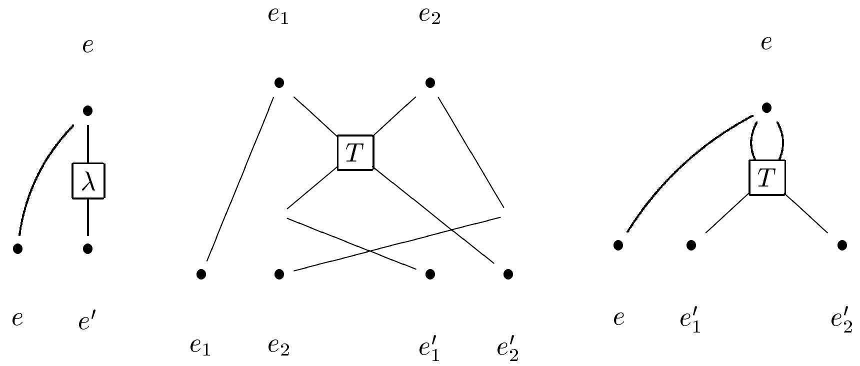


Figure 5: Action operators: wire-and-box diagrams

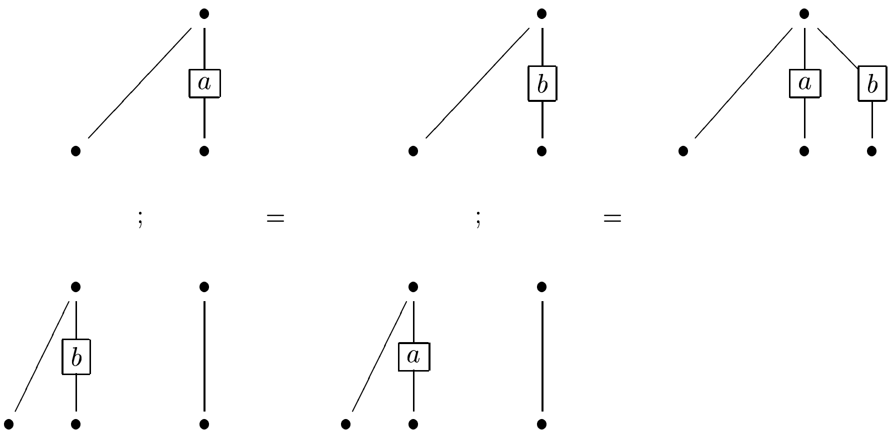


Figure 6: Concurrency between action operators

In compact form:

$$\xrightarrow{\tau_t} = e'_1 e'_2 := T(e,e), e := e$$

with $e < e'_1 < e'_2$.

The wire-and-box diagrams of the three action operators are shown in Figure 5.

All the vertical graphs of our tiles will be generated by combining the above three operators. More precisely, we call *located observations* all the gs-graphs generated via parallel and sequential compositions starting from action operators and permutation graphs. Roughly speaking, a located observation is a partial ordering of events, where each event is an occurrence of an action operator. Moreover, note that located observations are graphs where leaf locations are never erased (there is no discharger) or duplicated (there is no duplicator) but only *read*, i.e. used (but not consumed!) to produce other locations. Thus, the only causal dependencies (as specified by the partial ordering associated with the gs-graph) are between the producer of a certain location and all its users, while any pairs of users of the same location, or of different locations, are independent, i.e. concurrent.

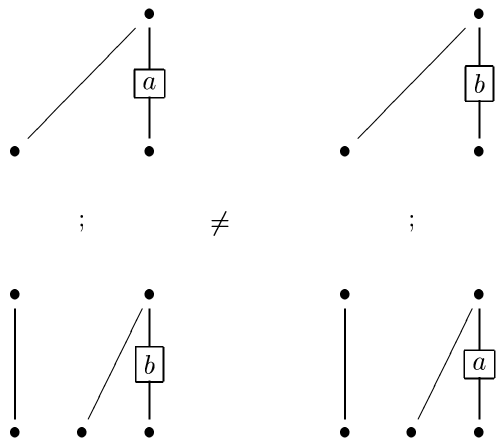


Figure 7: Causality between action operators

For instance we have:

$$\xrightarrow{a}\ ;\ \xrightarrow{b} \otimes id_1 = \xrightarrow{b}\ ;\ \xrightarrow{a} \otimes id_1,$$

i.e. two actions occurring in two sub-locations of the same location are concurrent (see Figure 6). Instead we have:

$$\xrightarrow{a}\ ;\ id_1 \otimes \xrightarrow{b} \neq \xrightarrow{b}\ ;\ id_1 \otimes \xrightarrow{a},$$

i.e. an action occurring in the location created by another action is causally dependent from that action (see Figure 7).

Since no location is erased, all the locations which are leaves of a located observation appear also (and only once) as roots. If they appear as roots in the same ordering they had as leaves, then the location observation is called *ordered*. Thus any location observation can be decomposed uniquely as the sequential composition of a permutation graph and of an ordered located observation. If an ordered location observation contains exactly one action operator, it is called *atomic*. Notice that all the action operators are atomic observations.

Extending the results of [25], which apply to symmetric monoidal theories (i.e. gs-monoidal theories without dischargers and duplicators, see the Appendix) it is interesting to study standard representations for located observations. An expression for an ordered located observation is in *standard form* if it is the sequential compositions of atomic observations.

Proposition 4.9 *Given an ordered located observation O, let ξ be any total ordering of its proper assignments which is a linearization of the partial ordering $\sqsubseteq^*$, with $n'_1, \ldots, n'_k := f(n_1, \ldots, n_h) \sqsubseteq m'_1 \ldots m'_i := g(m_1, \ldots, m_j)$ iff $\{n'_1, \ldots, n'_k\} \cap \{m_1, \ldots, m_j\} \neq \emptyset$. Then O can be obtained as the value of a unique*[10] *expression in standard form, where the atomic observations have action operators corresponding to the proper assignments of ξ and are sequentially composed according to ξ. Conversely, given an expression in standard form, the total ordering of its action operators is a linearization of the partial ordering of its corresponding proper assignments.*

[10]The expression is unique if we assume normal expressions for atomic graphs.

$$(\texttt{Prefix}_\lambda) \quad e' := \lambda(e) \xrightarrow[e':=\lambda(e),e:=e]{e':=e'} e' := e', e!$$

$$(\texttt{Suml}_\lambda) \quad e_1e_2 := +(e) \xrightarrow[e':=\lambda(e),e_1:=e]{e':=\lambda(e_1),e_1e_2:=e_1e_2} e'e_1 := e'e_1, e_2 := \kappa$$

$$(\texttt{Sumr}_\lambda) \quad e_1e_2 := +(e) \xrightarrow[e':=\lambda(e),e_2:=e]{e':=\lambda(e_2),e_1e_2:=e_1e_2} e'e_2 := e'e_2, e_1 := \kappa$$

$$(\texttt{Suml}_\texttt{T}) \quad e_1e_2 := +(e) \xrightarrow[e'_1e''_1:=T(e,e),e_1:=e]{e'_1e''_1:=T(e_1,e_1),e_1e_2:=e_1e_2} e_1e'_1e''_1 := e_1e'_1e''_1, e_2 := \kappa$$

$$(\texttt{Sumr}_\texttt{T}) \quad e_1e_2 := +(e) \xrightarrow[e'_2e''_2:=T(e,e),e_2:=e]{e'_2e''_2:=T(e_2,e_2),e_1e_2:=e_1e_2} e_2e'_2e''_2 := e_2e'_2e''_2, e_1 := \kappa$$

$$(\texttt{Synch}_\lambda) \quad e_1e_2 := e_1e_2 \xrightarrow[e'_1e'_2:=T(e_1,e_2),e_1e_2:=e_1e_2]{e_1e_2:=e_1e_2,e'_1:=\lambda(e_1),e'_2:=\overline{\lambda}(e_2)} e_1e'_1e_2e'_2 := e_1e'_1e_2e'_2$$

$$(\texttt{Twin}) \quad e_1e_2 := ee \xrightarrow[e'_1e'_2:=T(e,e),e:=e]{e'_1e'_2:=T(e_1,e_2),e_1e_2:=e_1e_2} e_1e_2e'_1e'_2 := eee'_1e'_2$$

$$(\texttt{Comp}_\lambda) \quad ee_1 := ee \xrightarrow[e:=e,e':=\lambda(e)]{ee_1:=ee_1,e':=\lambda(e)} ee'e_1 := ee'e$$

$$(\texttt{TwinComp}) \quad e_1e := ee \xrightarrow[e'_1e'_2:=T(e,e),e:=e]{e'_1e'_2:=T(e,e),ee_1:=ee_1} e_1e'_1e'_2e := ee'_1e'_2e$$

Table 3: Strong Rewrite System

Proof. The first part is proved by induction on linearizations. The property is obvious if there is only one proper assignment, since in that case the given ordered located observation is already atomic. Given an additional assignment, the expression in normal form, which exists according to the induction hypothesis, can be sequentially composed with an atomic observation whose operator and causal dependencies are uniquely determined by the additional assignment, and where the ordering of the roots is uniquely determined by the position(s) of the location(s) produced by the additional assignment within the roots of O. The second part is obvious. □

4.2.3 Strong Rewrite Rules

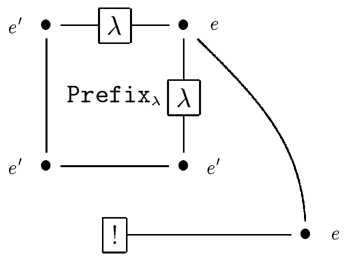
The rewrite rules are shown in Table 3. Notice that roots (leaves) of cogs-graphs representing configurations belong to input (output) interfaces of the tiles, while variables (roots) of gs-graphs representing observations belong to initial (final) interfaces of the tiles.

Hereafter, the antecedent (trigger) a of a tile $s \xrightarrow[b]{a} s'$ will be omitted when it is the empty graph.

We can now comment on the definition of the rules. We start by considering the prefix rule. The rule $\texttt{Prefix}_\lambda$ rewrites the initial configuration consisting of the cogs-graph $\{e' :=$

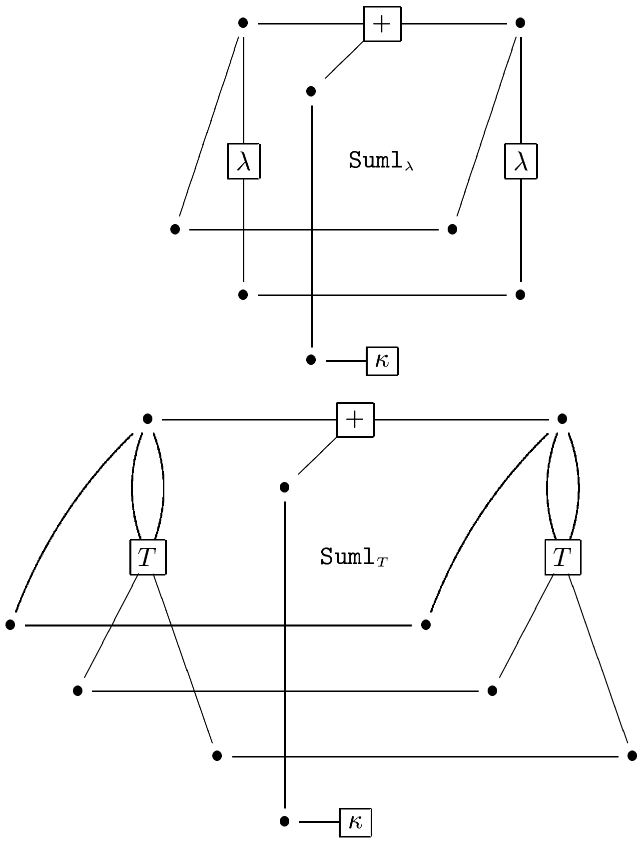
$\lambda(e)\}$ into the final configuration $\{e' := e', e!\}$. The initial input interface and the final input interface are represented by the name e' and the trigger by the identity gs-graph $\{e' := e'\}$. This means that the rule applies to an open process, where the missing subprocess is represented by the name e', but this subprocess is only required to produce a trivial (identity) trigger $\{e' := e'\}$. The initial output interface is represented by name e and the final output interface by names e, e'. The effect of rule $\texttt{Prefix}_\lambda$ is the action operator $\xrightarrow{\lambda}$, i.e. the gs-graph $\{e' := \lambda(e), e := e\}$. Thus the rule creates a new location e' in its effect, which however turns out to be the location (name) of the missing subprocess. The name e representing the location of the initial process is kept in the final configuration (via the operator $\_!$) since it could be shared by other processes (see the comment on rule $\texttt{Comp}_\lambda$).

From the tile $\texttt{Prefix}_\lambda$ in sequent form it is possible to derive a wire-and-box diagram for the whole tile. This diagram is simply obtained by gluing together the wire-and-box diagrams for the $\texttt{north}(\texttt{Prefix}_\lambda)$, $\texttt{south}(\texttt{Prefix}_\lambda)$, $\texttt{west}(\texttt{Prefix}_\lambda)$ and $\texttt{east}(\texttt{Prefix}_\lambda)$ graphs of the tile, via the shared interfaces. In this diagram we keep the names, in order to make the correspondence with the sequent form easier, but all the relevant information is completely represented by the wiring. In fact, according to our definition of gs-graphs, names are alpha convertible, but at least an ordering of leaves and roots must be provided, since it is essential in computing the sequential composition of two gs-graphs. Thus it would appear to be necessary to specify the ordering of names in all the interfaces of our tiles. For instance, what is the ordering between e' and e in the final output interface of rule $\texttt{Prefix}_\lambda$? However it is easy to see that name ordering is immaterial for rules. In fact, given a rule α it is always possible to obtain tiles with different orderings of names in the interfaces by composing α with suitable auxiliary (permutation) tiles. Thus any renaming in the rules of a tile rewrite system $\mathcal{R}$ does not change the tiles entailed by $\mathcal{R}$.

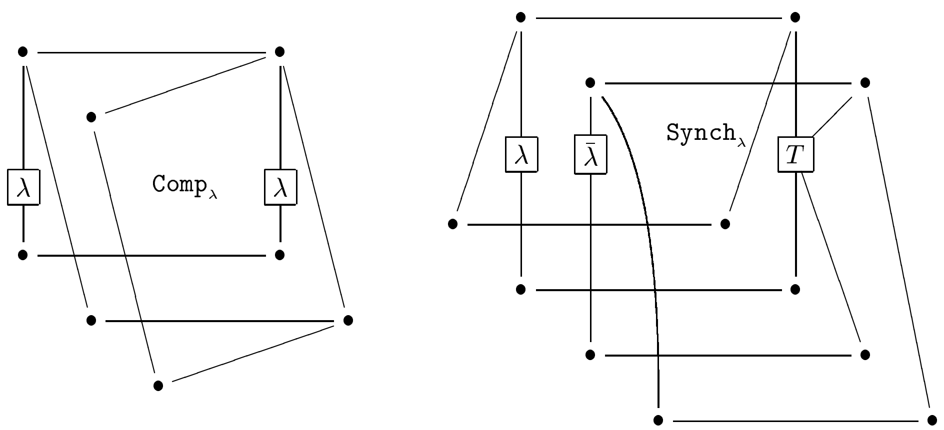


Below we comment directly on the diagram version of rules. The diagram version of rule $\texttt{Prefix}_\lambda$ tells us in direct, visual form that the horizontal constructor λ is erased from the initial configuration, but at the same time the vertical action operator $\xrightarrow{\lambda}$ is created in the effect. Also the role of the two locations involved is immediately evident, simply by following the wires.

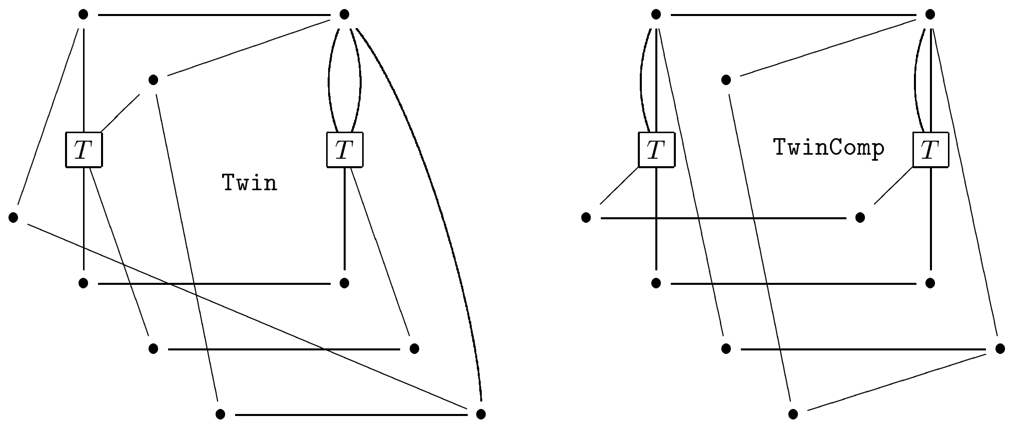
To describe the basic behaviour of the choice module, two rules for each choice are needed. Whenever the *left* (*right*) process makes a move, the *right* (*left*) process is discharged via the κ module. Moreover, there is a forward propagation of the locations of the process that makes the choice. Refused alternatives will appear as inactive factors in configurations. However, the move can be either an action move or a synchronization. Rule $\texttt{Suml}_\lambda$ takes care of handling λ actions, while rule $\texttt{Suml}_\texttt{T}$ takes care of synchronizations (at one location only). Symmetrically for $\texttt{Sumr}_\lambda$ and $\texttt{Sumr}_\texttt{T}$. Hereafter, we pictorially illustrate only the rewrite rules $\texttt{Suml}_\lambda$ and $\texttt{Suml}_\texttt{T}$; the remaining rules are represented by similar wire-and-box diagrams.



The rule ($\texttt{Comp}_\lambda$) basically describes the asynchronous evolution of parallel processes. The rule tells us that if two processes share the same location and one of them makes a move then its subprocesses will be placed on the new location, whereas the other process will remain linked to the old location. Notice that we do not need to distinguish between left or right moves: they can be obtained from each other by applying the auxiliary rules. Rule $\texttt{Synch}_\lambda$ accounts for synchronizations at different locations. The resulting effect tells us which the new locations are and how the old locations are maintained.



To complete the treatment of synchronization two more rules are needed. Rule Twin tells us how synchronization behaves in the presence of sharing. Namely, if the two synchronizing processes are allocated on the same location, then only a single copy of the old location is propagated forward. Notice that the horizontal composition of rule $\mathtt{Synch}_\lambda$ with rule `Twin` allows for synchronizations at one location only, like $a.nil \mid \overline{a}.nil \xrightarrow{\tau} nil \mid nil$. Rule `TwinComp` takes care of composition for twin T-moves.



The following results highlight some interesting properties of the strong tile rewrite system.

Proposition 4.10 *Any horizontal cogs-graph $s : 0 \to m$, with $m > 0$, can be decomposed as $s = s' \otimes s''$ with*

$$s'' = s_1; \kappa \otimes \ldots \otimes s_n; \kappa, \ \ n = 0, 1, \ldots$$

where $s', s_1, \ldots, s_n$ do not contain the constructor κ. Furthermore, any tile α with $\mathtt{north}(\alpha) = s$ can be decomposed as $\alpha = \alpha' \otimes id_{s''}$, where $\mathtt{north}(\alpha') = s'$.

Proof. If $t : 0 \to 0$, it is easy to prove using the naturality axioms that for any t_1 and t_2 we have

$$\begin{aligned} t \otimes t_1 &= t_1 \otimes t \\ t \otimes t_1; t_2 &= (t_1; t_2) \otimes t. \end{aligned}$$

We could reach the same conclusion by considering the basic definition of gs-graphs as well.

Thus the "erased" components $s_i; \kappa$ can always be composed in parallel with the components not containing κ. Also, since there is no rule α in the strong tile rewrite system with κ in $\mathtt{north}(\alpha)$, only perm tiles can be applied to the erased components. But it is easy to see that, due to the laws above, in this case perm tiles are just identities. □

Proposition 4.10 shows that the components of the form $s = [\![p]\!]; \kappa : 0 \to 0$, which are created by the `Sum` rules, behave like garbage which is accumulated during the computation, but which does not interfere with the execution of the rest. Thus, hereafter, we will consider configurations defined up to multiplicative ($\otimes$) factors of the above form[11].

Definition 4.11 A derivable tile is *closed* if and only if the trigger is the empty observation. □

[11] A way to make this equivalence more formal would be to introduce axioms for κ similar to those for the discharger ! in an algebraic theory: $s : m \to n$ implies $s; \kappa^n = \kappa^m$, where $\kappa^0 = id_\epsilon$ and $\kappa^{n+1} = \kappa^n \otimes \kappa$. With these axioms it would be possible to get rid of the garbage (and only of it).

Proposition 4.12 *Any derivable tile having a permutation graph as an observation is a perm tile.*

Proof. No rewrite rule or v-refl rule, with parameter a that is not an identity, can be used in a derivation without yielding in the resulting tile an observation different from a permutation graph. $\square$

Proposition 4.13 *Any derivable tile having a located observation as a trigger has a located observation as an effect, and vice versa.*

Proof. By induction on the structure of derivations in the tile logic. All rewrite rules enjoy the required property: a trigger located observation is rewritten into an effect located observation.

The crucial point of the induction base is rule v-refl. In this case the application of the rule must be avoided for those vertical graphs a which are not located observations. In fact, no such rule can be used in a derivation without yielding in the resulting tile a trigger which is not a located observation. Finally, all composition rules transform located observations into located observations. $\square$

Corollary 4.14 *Any closed derivable tile has a located observation as effect.*

Proof. Immediate, since the trigger of a close tile is a (trivial) located observation. $\square$

Proposition 4.15 *Let α be any derivable tile with $a = a_1; a_2$ (or $a = a_1 \otimes a_2$) as effect, where a_1, a_2 and a are all located. Then α can be decomposed as $\alpha = \alpha_1 \cdot \alpha_2$ ($\alpha = \alpha_1 \otimes \alpha_2$), where a_1 and a_2 are the effects of α_1 and α_2, respectively.*

Proof. By induction on the proof of α. The base cases are easy since the property is obvious for identity and auxiliary tiles, and true for rewrite rules, since they all have effects which cannot be decomposed in two non trivial located observations. Vertical and parallel composition maintains the property, since any decomposition of the effect of the composed tile can be seen as the composition of decomposed effects for the component tiles. Finally, horizontal composition also maintains the property, since any decomposition of the right tile would produce a decomposition of its trigger, i.e. of the effect of the left tile, and thus a decomposition of the whole left tile for the inductive hypothesis. $\square$

Putting all the above properties together we have the following result.

Definition 4.16 An *atomic tile* is any derivable tile with an atomic effect. $\square$

Proposition 4.17 *Given a closed derivable tile and any linearization ξ of its (located) effect, it can be decomposed as the vertical composition of a perm tile and of a sequence of closed atomic tiles corresponding to the linearization.*

Proof. By definition of ordered observation, any location observation can be decomposed sequentially into a permutation graph and an ordered located observation. But by Proposition 4.9 the ordered located observation can be decomposed into atomic observations according to ξ. By Proposition 4.15, it is thus possible to decompose in the same way the given tile. The tile with the permutation graph as observation is a perm tile by Proposition 4.12, while the remaining tiles are atomic by definition. □

We can now introduce the strong tile transition system.

Definition 4.18 The *strong tile transition system* has closed horizontal cogs-graphs $s : 0 \to 1$ as states, and atomic and perm tiles as transitions, labeled by their effects. □

4.2.4 Abstract Semantics

We want to handle weak location bisimulation with the uniform notion of tile bisimulation. For this purpose, we need to add rewrite rules that can transform effects with T observations into wire identities.

More precisely, we extend the horizontal signature with an additional symbol:

$$F : 1 \to 1 \ (Filter).$$

We also extend the vertical signature with the following symbols (one for each visible action λ)

$$\lambda^\dagger : 1 \to 1.$$

Notice that action operators $\xrightarrow{\lambda^\dagger}$ are introduced too. Finally, we add the following additional *weak* rules (one for each action operator).

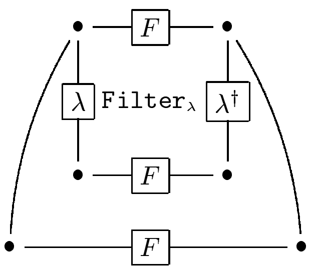
$$(\texttt{Filter}_\lambda) \qquad e_1 := F(e) \xrightarrow[e := e, e' := \lambda^\dagger(e)]{e_1 := e_1, e_1' := \lambda(e_1)} e_1 := F(e), e_1' := F(e')$$

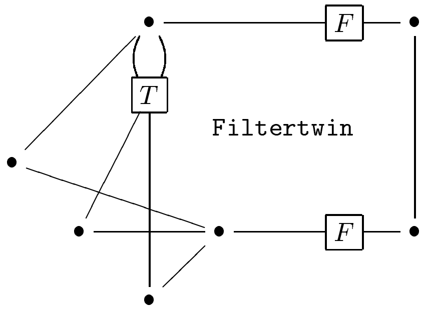
$$(\texttt{Filtertwin}) \qquad e' := F(e) \xrightarrow[e := e]{e_1 e_2 := T(e,e), e' := e'} e' e_1 e_2 := e' e' e', e' := F(e)$$

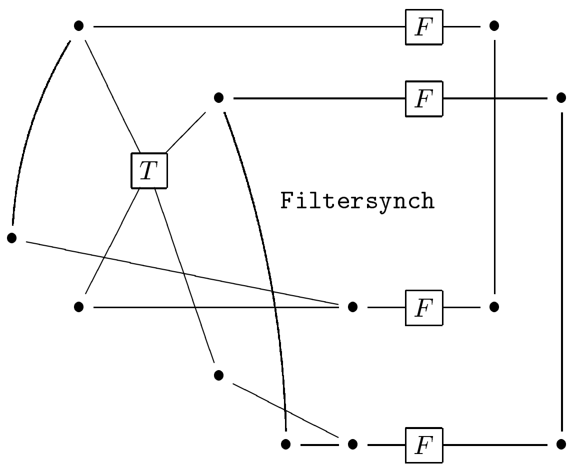
$$(\texttt{Filtersynch}) \quad e_1' := F(e_1), e_2' := F(e_2) \xrightarrow[e_1 e_2 := e_1 e_2]{e_1'' e_2'' := T(e_1', e_2'), e_1' e_2' := e_1' e_2'} G$$

where $G = \{e_1'\, e_1''\, e_2'\, e_2'' \ := e_1'\, e_1'\, e_2'\, e_2', \ \ e_1' := F(e_1), \ e_2' := F(e_2)\}$.

The wire-and-box representation of the Filter rewrite rules is given below.







The filters work as follows. If we start from a configuration with a filter for every name, action operators $\xrightarrow{\lambda}$ are rewritten into action operators $\xrightarrow{\lambda^{\dagger}}$, which can no longer be synchronized. Action operators $\xrightarrow{\tau_s}$ and $\xrightarrow{\tau_t}$, on the other hand, are transformed into wire identities. Notice that in the `Filtersynch` case the new names generated by the transition are merged with the names before the transition, i.e. eliminated.

Hereafter, we will use F^n to indicate the n-th product $F \otimes \ldots \otimes F$.

Definition 4.19 The *weak rewrite system* consists of the strong rules and of the above three weak rules.
Location bisimilarity is defined for the weak rewrite system as tile bisimilarity. □

Lemma 4.20 *Given a located observation $o : n \to m$ there is one and only one tile derivable in the weak rewrite system with*

$$F^n \xrightarrow[o']{o} \sigma; F^{m-2k},$$

where k is the number of occurrences of T in o and $\sigma : m \to m - 2k$ is a surjective co-substitution, i.e. a cogs-graph composed of ∇ and permutations. Furthermore, o' is also a located observation.

Proof. In the weak tile rewrite system each action operator has a unique weak (filter) rewrite rule. Then, rules (`Filtersynch`) and (`Filtertwin`) rewrite the action operators $\xrightarrow{\tau_s}$, $\xrightarrow{\tau_t}$ into wire identities, and add ∇'s to the south side of the tile but respect the ordering structure of the located observation. Hence, the overall effect is that o' is a located observation too. Finally, in the rule `Filter`$_\lambda$ filters reproduce themselves on every newly generated name. □

Derivable tiles of the form

$$F^n \xrightarrow[o']{o} \sigma; F^{m-2k}$$

are called *filter* tiles. Let `flto`(_) be the function from located observations to located observations defined as `flto`$(o) = o'$. Moreover, let `flts`(_) be the function from located observations to (surjective) co-substitutions defined as `flto`$(o) = \sigma$.

As a direct application of Lemma 4.20 the following property holds.

Proposition 4.21 *Given a closed tile*

$$\alpha = s_1 \underset{o}{\rightarrow} s_2$$

derivable in the strong rewrite system, there is a tile

$$\alpha' = s_1; F \otimes \ldots \otimes F \xrightarrow[\texttt{flto}(o)]{} s_2; \texttt{flts}(o); F \otimes \ldots \otimes F$$

derivable in the weak rewrite system. Conversely, given a closed tile

$$\alpha' = s_1; F \otimes \ldots \otimes F \underset{o'}{\rightarrow} s_2; \sigma; F \otimes \ldots \otimes F$$

derivable in the weak rewrite system, there is a closed tile

$$\alpha = s_1 \underset{o}{\rightarrow} s_2$$

derivable in the strong rewrite system with $\texttt{flto}(o) = o'$, *and* $\texttt{flts}(o) = \sigma$. □

Definition 4.22 The *weak tile rewrite system* has cogs-graphs of the form $s; F^n$ with $s : 0 \to n$, $n \geq 1$, as states; and perm tiles and atomic tiles horizontally composed with filter tiles as observations. □

Exploiting Proposition 4.17 and Proposition 4.21 the following property holds.

Proposition 4.23 *Every closed tile derivable in the weak rewriting system can be decomposed in the vertical composition of one perm tile and of atomic tiles, each horizontally composed with a filter tile and permutation tiles.*

4.3 Comparing SOS and Tile Semantics

This section presents the correspondence between the tile semantics and both the concurrent operational semantics and the abstract location semantics.

4.3.1 Operational Semantics

We introduce a translation function which maps transition proof terms into tiles. For this purpose, we exploit the structural congruence of processes. Thus in this section we introduce an intermediate transition system where states are processes up to structural congruence, and where transition proof terms are handled in a similar way.

Analogously to the structural laws on processes, we impose the following laws on transitions.

- Commutativity of parallel composition

$$\begin{aligned} p\lfloor t &\cong t\rfloor p, \\ t_1 \mid t_2 &\cong t_2 \mid t_1, \end{aligned}$$

- Associativity of parallel composition[12]

$$\begin{aligned} (t\rfloor p_1)\rfloor p_2 &\cong t\rfloor (p_1 \mid p_2), \\ (t_1 \mid t_2)\rfloor p &\cong t_1 \mid (t_2\rfloor p), \end{aligned}$$

[12] Note that laws $(p_1 \mid p_2)\lfloor t \cong p_1\lfloor(p_2\lfloor t)$, $(p_1\lfloor t)\rfloor p_2 \cong p_1\lfloor(t\rfloor p_2)$, $(p\lfloor t_1) \mid t_2 \cong p\lfloor(t_1 \mid t_2)$, and $(t_1\rfloor p) \mid t_2 \cong t_1 \mid (p\rfloor t_2)$ are derivable by commutativity and the other associativity axioms.

- Identity of parallel composition,

$$t \rfloor nil \;\cong\; t$$

- Locations

$$\begin{array}{rcl}
\epsilon :: t & \cong & t \\
\ell :: (t \rfloor p) & \cong & \ell :: t \rfloor \ell :: p \\
\ell :: (t_1 \mid t_2) & \cong & \ell :: t_1 \mid \ell :: t_2 \\
\ell :: (t \rfloor \ell' :: p) & \cong & \ell :: t \rfloor \ell' :: p \\
\ell :: (p \mid \ell' :: t) & \cong & \ell :: p \mid \ell' :: t \\
\ell :: (t \mid \ell' :: t') & \cong & \ell :: t \mid \ell' :: t'
\end{array}$$

As an application of the previous laws we show how to derive the law $(p_1 \lfloor t) \rfloor p_2 \cong p_1 \lfloor (t \rfloor p_2)$.

$$\begin{array}{rcl}
(p_1 \lfloor t) \rfloor p_2 & \cong & (t \rfloor p_1) \rfloor p_2 \\
& \cong & t \rfloor (p_1 \mid p_2) \\
& \cong & (t \rfloor (p_2 \mid p_1) \\
& \cong & (t \rfloor p_2) \rfloor p_1 \\
& \cong & p_1 \lfloor (t \rfloor p_2)
\end{array}$$

By inspecting the axioms, equivalent transitions clearly have the same labels and structurally equivalent source and target states.

As for processes, we can now define a normal form for proof terms.

Definition 4.24 A *transition proof term* t is in normal form if it can be written as

$$t = g_0 \lfloor l_1 :: g_1 \lfloor \ldots \lfloor l_i :: t_i \rfloor \ldots \rfloor l_n :: g_n,$$

with $t_i : g_i \xrightarrow[k]{\mu} p_i$, or as

$$t = g_0 \lfloor l_1 :: g_1 \lfloor \ldots \lfloor l_i :: t_i \rfloor \ldots \mid l_j :: t_j \rfloor \ldots \rfloor l_n :: g_n,$$

with $t_i : g_i \xrightarrow[k]{\lambda} p_i$, $t_j : g_j \xrightarrow[k']{\overline{\lambda}} p_j$, where $l_h \neq l_k$, $h, k = 1, \ldots, n$, $h \neq k$ and $l_1 < l_2 \ldots < l_n$. $\square$

In the above normal forms for proof terms, operators $\rfloor$, $\lfloor$, and $\mid$ associate to the left, and t_i, t_j are transitions of ground processes, normalized in some fixed way using the above laws.

As for located processes, every equivalence class of transition proof terms clearly contains exactly one transition in normal form.

Hereafter, for the sake of simplicity, we will use the notations

$$g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_n :: g_n,$$

and

$$g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_j :: t_j \mid \ldots \mid l_n :: g_n,$$

to represent normal forms.

In our running example, the proof term

$$([a, l_4, g_1 >\rfloor g_2) \mid l_2 :: g_3 \mid l_3 :: ([\overline{a}, l_1, g_4 >\rfloor g_5)$$

where $l_1 < \ldots < l_4$, is the normal form for the transition

$$(a.g_1 \mid g_2) \mid l_2 :: g_3 \mid l_3 :: (\overline{a}.g_4 \mid g_5) \xrightarrow[l_4, l_3 l_1]{\tau} g_2 \mid l_1 :: g_4 \mid l_2 :: g_3 \mid l_3 :: g_5 \mid l_4 :: g_1.$$

The next property tells us that the normalized transition system does not add extra behaviours.

Proposition 4.25 *Let p be a located process. Then for every transition $t : p \xrightarrow[k]{\mu} q$ we have* $\mathtt{norm}(t) : \mathtt{norm}(p) \xrightarrow[k]{\mu} q'$ *with* $\mathtt{norm}(q) = \mathtt{norm}(q')$. *Furthermore, for every transition* $t' : \mathtt{norm}(p) \xrightarrow[k]{\mu} q'$ *there is a transition* $t : p \xrightarrow[k]{\mu} q$ *with* $t' = \mathtt{norm}(t)$ *and* $\mathtt{norm}(q) = \mathtt{norm}(q')$.

Proof. (Outline) Given a transition (proof term) t for a process p, and a proof that $p' = \mathtt{norm}(p)$, it is possible to build the normalized proof term $\mathtt{norm}(t)$ by applying to transition proof terms the corresponding axioms. Also, it is easy to see that every axiom respects the transition preserving property above. Similarly, given a transition t' for p' and a proof for $p' = \mathtt{norm}(p)$, it is possible to build a proof t for p such that $t' = \mathtt{norm}(t)$. $\square$

One further step consists of making labels of transitions more informative. Instead of $p \xrightarrow[k]{\mu} q$ we introduce transitions

$$p \overset{\omega}{\longmapsto} q \overset{def}{=} p \underset{loc(p), tr(k)}{\overset{\mu}{\longmapsto}} q,$$

exposing also the locations of the (source) agent and truncating strings of locations in k. It is easy to see that neither of these modifications changes bisimilarity (see Definition 4.5).

We can now introduce our notion of *intermediate transition system.*

Definition 4.26 The *intermediate transition system* has as states (transition proof terms) the states (the transition proof terms) of the strong located transition system up to structural congruence $\cong$.

Also, $[t]_\simeq : [p]_\simeq \underset{loc(p), tr(k)}{\overset{\mu}{\longmapsto}} [q]_\simeq$ if and only if $t : p \xrightarrow[k]{\mu} q$.

Hereafter, we will write $[t]$ and $[p]$ instead of $[t]_\simeq$ and $[p]_\simeq$. Moreover, we overload the symbol ω to denote sequences of labels of transitions. Also, given the sequence w of labels of a sequence of transitions of the strong located transition system (starting from process p), we define $\mathtt{addloc}(w, loc(p))$ to be the sequence of labels of the corresponding sequence of transitions of the intermediate transition system, i.e.

$$\mathtt{addloc}(\xrightarrow[k]{\mu}, loc(p)) = \underset{loc(p), tr(k)}{\overset{\mu}{\longmapsto}}.$$

$\square$

Notice that, when applied to a sequence of labels w, $\mathtt{addloc}(w, loc(p))$ adds $loc(p)$ to the first transition, but must add $loc(p) \cup \{l\}$ to the second transition, if l is the location generated by the first transition, and so on. However, it is immediate to see that `addloc(_)` only depends on its arguments, and not on other information contained in the sequence of transitions.

The useful fact about the intermediate transition system is that its elements, besides being equivalence classes, can be inductively defined using normal forms.

We now map the labels of the intermediate transition system into ordered located observations.

Definition 4.27 The mapping $[\![\omega]\!]$ is inductively defined as follows:

$$[\![\underset{\{l_1,\ldots,l_n\},l_i l}{\overset{\lambda}{\longmapsto}}]\!] = \{l := \lambda(l_i), l_0 l_1 \ldots l_n := l_0 l_1 \ldots l_n\}$$

$$[\![\underset{\{l_1,\ldots,l_n\},l_i l_1,l_j l_2}{\overset{\tau}{\longmapsto}}]\!] = \{l_1 l_2 := T(l_i, l_j), l_0 l_1 \ldots l_n := l_0 l_1 \ldots l_n\}$$

$$[\![\overset{\omega_1}{\longmapsto}\overset{\omega_2}{\longmapsto}]\!] = [\![\overset{\omega_1}{\longmapsto}]\!] ; [\![\overset{\omega_2}{\longmapsto}]\!].$$

□

In the representations of gs-graphs, we assume that names are ordered as locations, i.e., $l_0 < l_1 < \ldots < l_n$. Notice that in the definition above we may have $l_i = l_j$ (and thus the $\xrightarrow{\tau_t}$ action operator will be generated). Also, we allow $l_i = l_0$, or $l_j = l_0$ (or both), in which case $l_0 l$ is just l. Similarly for $l_0 l_1$, $l_0 l_2$.

For instance, let us consider our running example. We have:

$$[\![\underset{\{l_2,l_3\},l_4,l_3 l_1}{\overset{\tau}{\longmapsto}}]\!] = \{l_4 l_1 := T(l_0, l_3), l_0 l_2 l_3 := l_0 l_2 l_3\}.$$

Proposition 4.28 $[\![\omega]\!]$ *is ordered located.*

Proof. By construction, the mapping of the label of a single transition is ordered located. The sequential composition of ordered located observations is still an ordered located observation. □

We are now ready to introduce the translation function mapping proof terms of the intermediate transition system into tiles.

Definition 4.29 Let t be a proof term in the intermediate transition system. Then $[\![t]\!]$ is the tile defined by the following clauses.

$$[\![[\lambda, l, g >]\!] = id_{[\![g]\!]} * Prefix_\lambda$$

$$[\![t < +g]\!] = \begin{cases} [\![t]\!] \otimes id_{[\![g]\!]} * Suml_\lambda \text{ if } \mathtt{east}([\![t]\!]) \ni \lambda \\ [\![t]\!] \otimes id_{[\![g]\!]} * Suml_T \text{ if } \mathtt{east}([\![t]\!]) \ni T \end{cases}$$

and symmetrically for $[\![g < +t]\!]$

$$[\![t \rfloor g]\!] = \begin{cases} [\![t]\!] \otimes id_{[\![g]\!]} * Comp_\lambda \text{ if } \mathtt{east}([\![t]\!]) \ni \lambda \\ [\![t]\!] \otimes id_{[\![g]\!]} * TwinComp \text{ if } \mathtt{east}([\![t]\!]) \ni T \end{cases}$$

$$[\![t_1 \mid t_2]\!] = [\![t_1]\!] \otimes [\![t_2]\!] * Synch_\lambda * Twin \text{ if } \mathtt{east}([\![t_1]\!]) \ni \lambda$$

$$[\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_n :: g_n]\!] = id_{[\![g_0]\!]} \otimes id_{[\![g_1]\!]} \otimes \ldots \otimes [\![t_i]\!] \otimes \ldots \otimes id_{[\![g_n]\!]} * \theta$$

where $t_i : g_i \stackrel{\omega}{\longmapsto} p_i$, and θ is a tile defined as follows:

- $\theta = id_n \xrightarrow[id_i \otimes \mathtt{east}([\![t_i]\!]) \otimes id_{n-i};\sigma]{id_i \otimes \mathtt{east}([\![t_i]\!]) \otimes id_{n-i}} \sigma$

  and σ is the unique permutation graph which makes the equality true (up to inactive configurations):

$$\mathtt{south}([\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_n :: g_n]\!]) = [\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: p_i \mid \ldots \mid l_n :: g_n]\!]$$

$$\begin{aligned}&[\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_j :: t_j \mid \ldots\ldots \mid l_n :: g_n]\!] = \\ &\qquad id_{[\![g_0]\!]} \otimes id_{[\![g_1]\!]} \otimes \ldots \otimes [\![t_i]\!] \otimes \ldots \otimes [\![t_j]\!] \otimes \ldots id_{[\![g_n]\!]} * \theta_1 * Synch_\lambda \otimes id_{n-1} * \theta_2\end{aligned}$$

where $t_i : g_i \stackrel{\omega_i}{\longmapsto} p_i$, $t_j : g_j \stackrel{\omega_j}{\longmapsto} p_j$, and θ_1, θ_2 are the tiles defined as follows:

- $\theta_1 = \rho \xrightarrow[\mathtt{east}([\![t_i]\!]) \otimes \mathtt{east}([\![t_j]\!]) \otimes id_{n-1}]{id_i \otimes \mathtt{east}([\![t_i]\!]) \otimes id_{j-i-1} \otimes \mathtt{east}([\![t_j]\!]) \otimes id_{n-j}} \sigma_1$

  with ρ the permutation graph which connects the i-th and the j-th root to the first and second leaf, and leaves the respective order of all the other wires unmodified. Instead, σ_1 is the unique permutation graph such that:

$$id_i \otimes \mathtt{east}([\![t_i]\!]) \otimes id_{j-i-1} \otimes \mathtt{east}([\![t_j]\!]) \otimes id_{n-j}; \sigma_1 = \rho; \mathtt{east}([\![t_i]\!]) \otimes \mathtt{east}([\![t_j]\!]) \otimes id_{n-1};$$

- $\theta_2 = \rho^{-1} \xrightarrow[\rho;\mathtt{east}(Synch_\lambda) \otimes id_{n-1};\sigma_2]{\mathtt{east}(Synch_\lambda) \otimes id_{n-1}} \sigma_2$

  with σ_2 the unique permutation graph which makes the equality true (up to inactive configurations):

$$\begin{aligned}&\mathtt{south}([\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_j :: t_j \mid \ldots\ldots \mid l_n :: g_n]\!]) = \\ &[\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: p_i \mid \ldots \mid l_j :: p_j \mid \ldots \mid l_n :: g_n]\!].\end{aligned}$$

□

Intuitively, permutation graph σ (σ_2) of Definition 4.29 brings the new location (the two new locations) generated by the transition(s) in the right order in the final configuration with respect to the total ordering on locations. This is why permutation graphs σ and σ_2 are unique.

Notice that tiles θ_1, θ_2 above are derivable in the strong rewrite system since they can be vertically decomposed as follows:

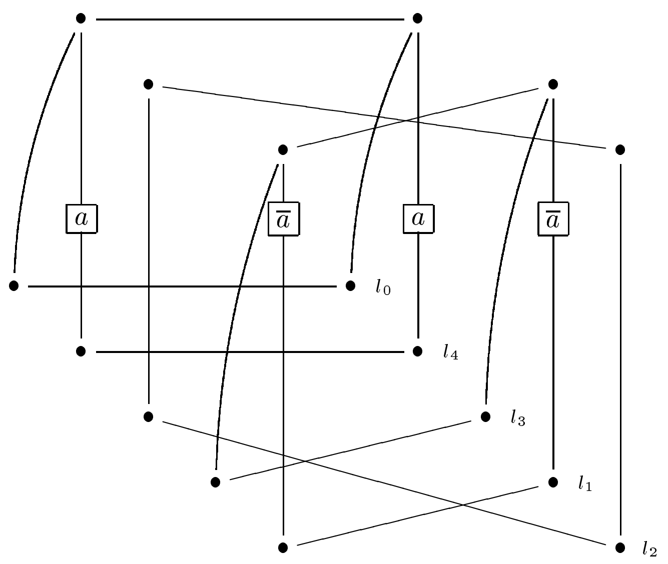
$$\begin{aligned}\theta_1 \quad = \quad & \rho \xrightarrow[id_{n+1}]{\rho} id_{n+1} \\ & \cdot\mathtt{east}([\![t_i]\!]) \otimes \mathtt{east}([\![t_j]\!]) \otimes id_{n-1} \\ & \cdot id_{n+1} \xrightarrow[id_{n+1}]{\sigma_1^{-1}} \sigma_1\end{aligned}$$

$$\begin{aligned}\theta_2 \quad = \quad & \rho^{-1} \xrightarrow[\rho]{id_{n+1}} id_{n+1} \\ & \cdot\mathtt{east}(Synch_\lambda) \otimes id_{n-1} \\ & \cdot id_{n+3} \xrightarrow[\sigma_2]{id_{n+3}} \sigma_2\end{aligned}$$

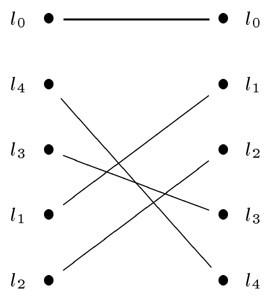
To clarify the definition of the mapping $[\![\_]\!]$, we consider our running example. We have that $([\![[a, l_4, g_1 >\rfloor g_2) \mid l_2 :: g_3 \mid l_3 :: ([\overline{a}, l_1, g_4 >\rfloor g_5]\!])$ is the tile:

$$\begin{pmatrix} (id_{[\![g_1]\!]} * \mathtt{Prefix_a}) \otimes id_{[\![g_2]\!]}) * Comp_a \otimes \\ id_{[\![g_3]\!]} \otimes \\ (id_{[\![g_4]\!]} * \mathtt{Prefix_{\overline{a}}}) \otimes id_{[\![g_5]\!]}) * Comp_{\overline{a}} \end{pmatrix} \quad * \quad \theta_1 * \mathtt{Synch_a} \otimes id_1 * \theta_2$$

The tile θ_1, which brings the locations involved in the synchronization close to each other, is illustrated below.



The permutation graph σ_2 which reorders the locations generated by the synchronization in order to match the total ordering on locations is shown below.



Proposition 4.30 *$t : p_1 \overset{\omega}{\longmapsto} p_2$ implies $[\![t]\!] = [\![p_1]\!] \underset{[\![\omega]\!]}{\longrightarrow} s$, where $s = [\![p_2]\!]$ up to parallel composition with some inactive configuration s'.*

Proof. By (standard) induction on the structure of proof terms. We show a few cases.

- Take α to be the tile $[\![[\lambda, l, g >]\!] = id_{[\![g]\!]} * Prefix_\lambda$. Then we have:

  1. $\mathtt{north}(\alpha) = [\![g]\!]; \lambda = [\![\lambda.g]\!]$,
  2. $\mathtt{south}(\alpha) = !_1 \otimes [\![g]\!] = [\![nil \mid l :: g]\!]$,
  3. $\mathtt{west}(\alpha) = \emptyset$, as $[\![g]\!]$ is a horizontal graph from 0 to 1.

4. $\mathtt{east}(\alpha) = \xrightarrow{\lambda} = [\![\overset{\lambda}{\underset{\emptyset,l}{\longmapsto}}]\!]$.

- Let α be the tile $[\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: t_i \mid \ldots \mid l_n :: g_n]\!]$. Then, we have

  1. $\mathtt{north}(\alpha) = ([\![g_0]\!] \otimes [\![g_1]\!] \otimes \ldots \otimes \mathtt{north}(t_i) \otimes \ldots \otimes [\![g_n]\!]); \mathtt{north}(\theta)$. By the inductive hypothesis and from the fact that $\mathtt{north}(\theta) = id_n$ we have: $\mathtt{north}(\alpha) = ([\![g_0]\!] \otimes [\![g_1]\!] \otimes \ldots \otimes [\![g_i]\!] \otimes \ldots \otimes [\![g_n]\!])$ which is by definition $[\![g_0 \mid l_1 :: g_1 \mid \ldots \mid l_i :: g_i \mid \ldots \mid l_n :: g_n]\!]$.
  2. $\mathtt{south}(\alpha) = ([\![g_0]\!] \otimes [\![g_1]\!] \otimes \ldots \otimes \mathtt{south}(t_i) \otimes \ldots \otimes [\![g_n]\!]); \mathtt{south}(\theta)$. By the inductive hypothesis and from the fact that $\mathtt{south}(\theta) = \sigma$ we have that $\mathtt{south}(\alpha) = [\![p_2]\!]$ up to some inactive configuration.
  3. $\mathtt{west}(\alpha) = \emptyset$ by the inductive hypothesis.
  4. $\mathtt{east}(\alpha) = \mathtt{east}(\theta) = id_i \otimes \mathtt{east}([\![t_i]\!]) \otimes id_{n-i}; \sigma$. Then, by the inductive hypothesis and by the property of σ we have that $\mathtt{east}(\alpha) = [\![\omega]\!]$.

$\square$

Lemma 4.31 *A diamond* $(t_1$ *then* $t_2 \,\chi\, t_3$ *then* $t_4)$ *implies* $[\![t_1]\!] \cdot [\![t_2]\!] = [\![t_3]\!] \cdot [\![t_4]\!]$.

Proof. The crucial point of the tile semantics is that the only causal dependencies specified by the tiles in the target of the mapping $[\![\_]\!]$ are between the producer of a certain location and all its users, while any pair of users of the same location, or of different locations, is independent. This is what the χ relation expresses. Consequently, the proof is basically a routine induction on the clauses of the χ relation. $\square$

Let function $[\![\_]\!]$ be extended to computations via vertical composition (i.e. $[\![c_1; c_2]\!] = [\![c_1]\!] \cdot [\![c_2]\!]$).

Proposition 4.32 $c_1 \equiv c_2$ *in* $\mathcal{L}_S$ *implies* $[\![c_1 = c_2]\!]$.

Proof. Immediate from the previous lemma. $\square$

Proposition 4.33 *Assume that* α *is a derivable closed tile such that* $s = \mathtt{north}(\alpha)$ *(up to inactive configurations) and* $o = \mathtt{east}(\alpha)$ *is an ordered located observation. Then, for any process* p *of the intermediate transition system such that* $[\![p]\!] = s$*, and any sequence of labels* ω *such that* $[\![\omega]\!] = o$*, a sequence of transitions* c *exists such that* $c : p \overset{\omega}{\longmapsto} q$ *with* $[\![c]\!] = \alpha$.

Proof. We first assume that α is atomic, in which case c is a transition. Recall that, by Proposition 4.8, $[\![s]\!]^{-1}$ is a set of processes which only differ with regard to the names of locations (but not their ordering). Also, once p is fixed, all the sequences ω of labels such that $[\![\omega]\!] = o$ will differ only in terms of the bound locations (but not in relation to its ordering among the other locations).

As locations are just placeholders in the located semantics, it is enough to extract one proof term t from the proof of α since all the other located processes and observations will have a proof term different from t only by an injective substitution of locations, which preserves the ordering.

We first handle the case where the action operator is $\xrightarrow{\lambda}$. Here we have a property similar to Proposition 4.15, but now on the horizontal side. Namely, given any atomic tile α and any decomposition $s = s_1; s_2$ ($s = s_1 \otimes s_2$) of its north side, we have $\alpha = \alpha_1 * \alpha_2$ ($\alpha = \alpha_1 \otimes \alpha_2$) with $\texttt{north}(\alpha_1) = s_1$, $\texttt{north}(\alpha_2) = s_2$. This property is due to the fact that, if we exclude rules $\texttt{Synch}_\lambda$, $\texttt{Twin}$ and $\texttt{TwinComp}$, which cannot be used when the action operator is $\xrightarrow{\lambda}$, all the other rules have an elementary cogs-graph as the north side. Therefore, the north side of α can be decomposed as in the proof of Proposition 4.8, and for every step we have a unique rule which applies.

When the action operator is $\xrightarrow{\tau_t}$ ($\xrightarrow{\tau_s}$), it can be seen that the proof of α can be reshaped as the composition

$$\alpha = \theta_1 \cdot ((\alpha_1 \otimes \alpha_2 * \alpha_3) \otimes I) \cdot \theta_2$$

where:

- (i) θ_1 is a perm tile which brings in the first position (first and second positions) the wire (wires) where the action operator $\xrightarrow{\tau_t}$ ($\xrightarrow{\tau_s}$) is attached;
- (ii) α_1 and α_2 are atomic tiles having action operators $\xrightarrow{\lambda}$ and $\xrightarrow{\overline{\lambda}}$ as the east side;
- (iii) α_3 is an atomic tile having $\xrightarrow{\lambda} \otimes \xrightarrow{\overline{\lambda}}$ as the west side, $\xrightarrow{\tau_t}$ ($\xrightarrow{\tau_s}$) as the east side, and a cogs-graph without generators as the north side;
- (iv) I is an h-refl tile;
- (v) θ_2 is a perm tile which brings back the wires attached to the $\xrightarrow{\tau_t}$ ($\xrightarrow{\tau_s}$) operator to the original position (so α has an ordered located observation), and brings the new wires generated by α to their correct positions (as specified in α).

It is now easy to see that a proof term exists for

$$\alpha' = (\alpha_1 \otimes \alpha_2 * \alpha_3) \otimes I$$

since we already have proof terms t_1 and t_2 with $[\![t_1]\!] = \alpha_1$ and $[\![t_2]\!] = \alpha_2$ by the first case we considered, and

$$\begin{aligned}
&\alpha' = [\![t_1 \mid l_1 :: t_2 \mid l_2 :: g_2 \mid \ldots \mid l_n :: g_n]\!] = \\
&[\![t_1]\!] \otimes [\![t_2]\!] \otimes [\![g_2]\!] \otimes \ldots \otimes [\![g_n]\!] * (\texttt{Synch}_\lambda \otimes id_{n-1}) = \\
&([\![t_1]\!] \otimes [\![t_2]\!] * \texttt{Synch}_\lambda) \otimes [\![g_2]\!] \otimes \ldots \otimes [\![g_n]\!]
\end{aligned}$$

(or $\alpha_1 \otimes \alpha_2 * \alpha_3 = [\![t_1 \mid t_2]\!] = [\![t_1]\!] \otimes [\![t_2]\!] * \texttt{Synch}_\lambda * \texttt{Twin}$; and
$[\![t^i \rfloor g^i]\!] = [\![t^i]\!] \otimes [\![g^i]\!] * \texttt{TwinComp}$
with $t^1 = t_1 \mid t_2$, $t^{i+1} = t^i \rfloor g^i$, $i = 1, \ldots n$, $n = 0, 1, \ldots$; and
$\alpha' = [\![t^{n+1} \mid l_1 :: g_1 \mid \ldots \mid l_m :: g_m]\!] = [\![t^{n+1}]\!] \otimes [\![g_1]\!] \otimes \ldots \otimes [\![g_m]\!]$
with $m = 0, 1, \ldots$).
Now a proof term for

$$\alpha = \theta_1 \cdot \alpha' \cdot \theta_2$$

exists provided that it exists for α'. In fact it is easy to see that if $\alpha' = [\![t]\!]$, then $\alpha = [\![t\sigma]\!]$ here σ is an injective location substitution (including location l_0) corresponding in the obvious way to the perm tile θ_2.

Finally, since we found $t : p \overset{\omega}{\longmapsto} q$ such that $[\![t]\!] = \alpha$, we can use Proposition 4.17 to derive that $\texttt{north}(\alpha) = [\![p]\!]$, and $\texttt{east}(\alpha) = [\![o]\!]$.

If α is not atomic, then the sequence of labels ω such that $o = [\![\omega]\!]$ may also differ with regard to the ordering of the actions. But Proposition 4.17 ensures that α can be decomposed into the vertical composition of a sequence of atomic tiles for every linearization ξ of the action operators in its effect. Then, for each atomic tile we can iterate the construction above. $\square$

4.3.2 Abstract Semantics

We now prove that location bisimilarity and tile location bisimilarity coincide for ground agents.

Lemma 4.34 *Let p be a located process and let w be a sequence of labels. We have*

(i) $[\![\mathtt{addloc}(\mathtt{wkl}(w), L)]\!]^\dagger = \mathtt{flto}([\![\mathtt{addloc}(w, L)]\!])$,

where L is a set of locations which does not contain any bound location of w and contains all the other locations in it, and function $(\_)^\dagger$ transforms each label λ in $\lambda^\dagger$;

(ii) $[\![\mathtt{wkp}(p, w)]\!] = [\![p]\!]; \mathtt{flts}([\![\mathtt{addloc}(w, loc(p) - \mathtt{bloc}(w))]\!])$,

with $loc(w) \subseteq loc(p)$, and where $\mathtt{bloc}(w)$ are the bound locations of w.

Proof. The lemma states that first abstracting and then mapping the tiles, or first mapping the tiles and then abstracting, one gets the same ordered observation and final configuration. The proof is by induction on the length of the sequence of labels w. The basic step is trivial and the inductive step follows directly from the definitions. $\square$

Proposition 4.35 (tile semantics is correct) *Given two ground agents g_1 and g_2, we have $g_1 \approx_L g_2$ iff $[\![g_1]\!]; F \sim_t [\![g_2]\!]; F$.*

Proof.

We start by proving the *if part*. Given a bisimulation S for the weak tile transition system we prove that if we take $p_1 R p_2$ whenever $loc(p_1) = loc(p_2)$ and $([\![p_1]\!]; F \otimes \ldots \otimes F)\ S\ ([\![p_2]\!]; F \otimes \ldots \otimes F)$, then R is also a bisimulation. In fact, given a pair $p_1 R p_2$ and a transition in the weak transition system

$$p_1 \xrightarrow{\mathtt{wkl}(w)} \mathtt{wkp}(p_1', w),$$

there is a transition in the ordinary transition system

$$p_1 \xrightarrow{w} p_1'$$

by definition of weak transition. Then, by Proposition 4.25 there is a sequence of transitions in the intermediate transition system

$$[p_1] \overset{\mathtt{addloc}(w, loc(p_1))}{\longmapsto} [p_1']$$

By Proposition 4.30, there is also a tile in the strong tile rewrite system

$$[\![p_1]\!] \xrightarrow[{[\![\mathtt{addloc}(w, loc(p_1))]\!]}]{} [\![p_1']\!].$$

Then, by Proposition 4.21, there is a tile in the weak tile rewrite system

$$[\![p_1]\!]; F \otimes \ldots \otimes F \xrightarrow[\mathtt{flto}([\![\mathtt{addloc}(w, loc(p_1))]\!])]{} [\![p_1']\!]; \mathtt{flts}([\![\mathtt{addloc}(w, loc(p_1))]\!]); F \otimes \ldots \otimes F.$$

But by hypothesis

$$([\![p_1]\!]; F \otimes \ldots \otimes F) \; S \; ([\![p_2]\!]; F \otimes \ldots \otimes F),$$

thus, there is a tile in the weak tile rewrite system

$$[\![p_2]\!]; F \otimes \ldots \otimes F \xrightarrow[\mathtt{flto}([\![\mathtt{addloc}(w, loc(p_1))]\!])]{} s; F \otimes \ldots \otimes F$$

with

$$([\![p_1']\!]; \mathtt{flts}([\![\mathtt{addloc}(w, loc(p_1))]\!]); F \otimes \ldots \otimes F) \; S \; (s; F \otimes \ldots \otimes F).$$

Then, by Proposition 4.21, there is a tile in the strong tile rewrite system

$$[\![p_2]\!] \xrightarrow[o]{} s'$$

with

$$\mathtt{flto}(o) = \mathtt{flto}([\![\mathtt{addloc}(w, loc(p_1))]\!]) \text{ and } s = s'; \mathtt{flts}(o).$$

Furthermore, by Proposition 4.33, there is a sequence of transitions in the intermediate transition system:

$$q \stackrel{\omega}{\longmapsto} q'$$

with $[\![q']\!] = s'$, for every process q such that $[\![q]\!] = [\![p_2]\!]$, and for every ω such that $[\![\omega]\!] = o$.

Also, by Proposition 4.25 there is a sequence of transitions in the strong located transition system

$$p' \xrightarrow{w'} p_2'$$

where $[p'] = q$, $[p_2'] = q'$ and $\omega = \mathtt{addloc}(w', loc(p'))$, for every process p' with $[p'] = q$. Thus in particular we can take $p' = p_2$.

We can now move to the weak located transition system. We have the sequence of transitions:

$$p_2 \xrightarrow{\mathtt{wkl}(w')} \mathtt{wkp}(p_2', w').$$

We now show that this sequence is in fact a single transition and that it can match the original transition

$$p_1 \xrightarrow{\mathtt{wkl}(w)} \mathtt{wkp}(p_1', w)$$

according to the Definition 4.5 of incremental location bisimilarity.

In fact by construction we have

$$\mathtt{flto}([\![\mathtt{addloc}(w, L)]\!]) = \mathtt{flto}([\![\mathtt{addloc}(w', L)]\!])$$

with $L = loc(p_1) = loc(p_2)$. By Lemma 4.34 we can conclude

$$[\![\mathtt{addloc}(\mathtt{wkl}(w), L)]\!]^\dagger = [\![\mathtt{addloc}(\mathtt{wkl}(w'), L)]\!]^\dagger.$$

Hence, we can also conclude

$$[\![\mathtt{addloc}(\mathtt{wkl}(w), L)]\!] = [\![\mathtt{addloc}(\mathtt{wkl}(w'), L)]\!].$$

Here, $\texttt{wkl}(w)$ is a single visible label. Thus, by Proposition 4.28, its mapping is an atomic observation. Thus, also $\texttt{wkl}(w')$ is a single visible action label. In conclusion we can write

$$\texttt{wkl}(w) = \underset{u_1}{\overset{\lambda_1}{\Longrightarrow}} \text{ and } \texttt{wkl}(w') = \underset{u_2}{\overset{\lambda_2}{\Longrightarrow}} .$$

Thus our equation becomes

$$[\![\underset{L,tr(u_1)}{\overset{\lambda_1}{\longmapsto}}]\!] = [\![\underset{L,tr(u_2)}{\overset{\lambda_2}{\longmapsto}}]\!].$$

It is now easy to see that we have $\lambda_1 = \lambda_2$ and $tr(u_1) = tr(u_2)$, as required by bisimilarity. The first equality is immediate, while we can write the second as $l_1' l_1 = l_2' l_2$ where, as usual, l_1' and l_2' can be the empty location l_0. We have that $l_1' = l_2'$, since the initial locations L are the same and the equality of the atomic observations implies that the leaf of the action operator is the same. We also have $l_1 = l_2$, since the equality of the atomic observations ensures that the root generated by the transition has the same ordering in both cases among the pre-existing roots, while the degree of freedom we have in choosing ω above (proved by Proposition 4.33) allows us to match in the response the location generated in the first move.

With regard to the states $\texttt{wkp}(p_1', w)$ and $\texttt{wkp}(p_2', w')$ that we reach after the moves, we have the following property by construction:

$$([\![p_1']\!]; \texttt{flts}([\![\texttt{addloc}(w, loc(p_1))]\!]); F \otimes \ldots \otimes F) \; S \; ([\![p_2']\!]; \texttt{flts}([\![\texttt{addloc}(w', loc(p_2))]\!]); F \otimes \ldots \otimes F)$$

which, by Lemma 4.34, becomes

$$([\![\texttt{wkp}(p_1', w)]\!]; F \otimes \ldots \otimes F) \; S \; ([\![\texttt{wkp}(p_2', w')]\!]; F \otimes \ldots \otimes F).$$

Thus by construction we have

$$\texttt{wkp}(p_1', w) \; R \; \texttt{wkp}(p_2', w'),$$

which is what is required by bisimilarity. However, we still have to check that $loc(\texttt{wkp}(p_1', w)) = loc(\texttt{wkp}(p_2', w'))$. But this can be definitely achieved by matching the generated location as we mentioned above.

We now show the `only if` part.

We prove that if we take $([\![p_1]\!]; F \otimes \ldots \otimes F; \rho) \, S \, ([\![p_2]\!]; F \otimes \ldots \otimes F; \rho)$ for every permutation ρ whenever $p_1 \approx_L p_2$ and $loc(p_1) = loc(p_2)$, then S is a bisimulation. This is enough since $loc(g_1) = loc(g_2) = \emptyset$.

Given a tile derivable in the weak tile system whose initial configuration is $[\![p_1]\!]; F \otimes \ldots \otimes F$, it can be decomposed, by Proposition 4.23, into the vertical composition of a perm tile and some atomic tiles sequentially composed with filter tiles and permutation tiles. Thus assuming as usual the shorter moves, if $([\![p_1]\!]; F \otimes \ldots \otimes F; \rho) \; S \; ([\![p_2]\!]; F \otimes \ldots \otimes F; \rho)$ and $[\![p_1]\!]; F \otimes \ldots \otimes F$ takes a perm tile, also $[\![p_2]\!]; F \otimes \ldots \otimes F$ can take a tile with the same effect and for the resulting configurations we have $([\![p_1]\!]; F \otimes \ldots \otimes F; \rho') \; S \; ([\![p_2]\!]; F \otimes \ldots \otimes F; \rho')$ by construction.

If $([\![p_1]\!]; F \otimes \ldots \otimes F) \, S \, ([\![p_2]\!]; F \otimes \ldots \otimes F)$ and $[\![p_1]\!]; F \otimes \ldots \otimes F$ takes an atomic tile, then it is possible to prove in a way similar to the `if part` that a transition with a corresponding observation exists for p_1 in the weak transition system, that thus also p_2 has a transition with the same truncated observation, and finally that the desired tile for $[\![p_2]\!]; F \otimes \ldots \otimes F$ can be built from this transition in the weak tile system.

The case $([\![p_1]\!]; F \otimes \ldots \otimes F; \rho)\, S\, ([\![p_2]\!]; F \otimes \ldots \otimes F; \rho)$ is already subsumed by the previous case, since if $p_1 \approx_L p_2$ we can find p'_1 and p'_2 with $p'_1 \approx_L p'_2$ and $[\![p_i]\!]; F \otimes \ldots F; \rho = [\![p'_i]\!]; F \otimes \ldots \otimes F$, for $i = 1, 2$. In fact it is easy to see that $[\![p_i]\!]; F \otimes \ldots \otimes F; \rho = [\![p_i]\!]; \rho; F \otimes \ldots \otimes F$ by the naturality axiom of permutation graphs. Notice now that if $p_1 \approx_L p_2$, then for every injective substitution σ of locations (including the empty location ϵ) we have $p_1\sigma \approx_L p_2\sigma$. But then a substitution σ can easily be found such that $[\![p_i\sigma]\!] = [\![p_i]\!]; \rho$. Thus we have $p'_i = p_i\sigma$. $\square$

5 A Tile-Based Coordination View of Asynchronous π-calculus

The π-calculus [47] is one of the best studied examples of *mobile* process calculi, namely calculi in which the communication topology among processes can dynamically evolve when computation progresses. In π-calculus, mobility is achieved via the communication of *names* rather than *processes*, as happens in so-called *higher order* process calculi (e.g. [59]). The π-calculus has been used to model objects [62], higher order communication [57] and also higher order configuration languages [55]. Asynchronous π-calculus [12, 40] is a variant of π-calculus where emission of messages is non-blocking.

This section presents a tile-based bisimulation semantics for asynchronous π-calculus, which we prove to be equivalent to the ordinary bisimulation semantics. Unlike the tile rewrite system presented in the previous section, two kinds of tiles are provided: *activity tiles* and *coordination tiles*. Activity tiles describe the evolution of processes, and they do not consider the operational context where processes live. They thus have no (i.e. empty) triggers, but only effects. Instead, coordination tiles define the behaviour of coordinators: they determine the evolution of processes inside their operational environment by taking coordinating actions in response to actions of other processes.

5.1 Asynchronous π-calculus

We assume an infinite set of channel names $a, b, c \ldots$ and, following [4], we distinguish between processes or agents $p, q, \ldots$, sequential processes or guards $g, h, \ldots$ and messages m. The syntax of the calculus is specified by the following grammar

$$\begin{array}{rcl} p & ::= & m \quad | \quad g \quad | \quad p \,|\, p \quad | \quad \nu a\, p \\ m & ::= & \overline{a}b \\ g & ::= & \mathtt{0} \quad | \quad a(b).p \quad | \quad g_1 + g_2 \quad | \quad !g. \end{array}$$

In $a(b).p$ and $\nu b p$ the occurrences of b are binding with scope p. We assume the standard notions of free names, bound names and substitution, and we denote with $\mathtt{fn}(p)$ (resp. $\mathtt{bn}(p)$) the names free (resp. bound) in p. Finally, we use $\mathtt{n}(p)$ to denote the set of *names* of p ($\mathtt{n}(p) = \mathtt{fn}(p) \cup \mathtt{bn}(p)$).

Processes and guards are defined up to a *structural congruence* $\equiv$. This is the smallest congruence such that:

1. processes and guards which differ by α-conversion are equivalent;
2. $|$ is associative and commutative, and $\mathtt{0}$ is its identity;
3. if $a \notin \mathtt{fn}(p)$ then $p \mid (\nu a\ q) \equiv \nu a(p \mid q)$;
4. $\nu a\ \nu b\ p \equiv \nu b\ \nu a\ p$;

Notice that using the laws of structural congruence $\equiv$ any process p can be transformed into a *normal form*

$$p \equiv \nu a_1 \ldots \nu a_k (g_1 \mid \ldots \mid g_n \mid \overline{b_1} c_1 \mid \ldots \mid \overline{b_m} c_m).$$

The actions α are specified as follows

$$\alpha \quad ::= \quad \tau \quad | \quad \overline{a}b \quad | \quad \overline{a}(b) \quad | \quad ab$$

The notions of free names $\mathtt{fn}(\alpha)$, bound names $\mathtt{bn}(\alpha)$ and names $\mathtt{n}(\alpha)$ are naturally extended to actions α.

On the grounds of the results of Amadio, Castellani and Sangiorgi [4] we introduce a presentation of the operational semantics of asynchronous π-calculus which will be easier to compare with the tile semantics. Following [4], this transition system will be called the *direct* transition system. It distinguishes two kinds of input transitions.

- Transitions $\xrightarrow{ab}_0$ are used to create contexts of the form $[\_] \mid \overline{a}b$.
- Transitions $\xrightarrow{ab}_1$ are only used in communications.

A generic transition in the transition system will be denoted by $\xrightarrow{\alpha}$. Table 4 illustrates the structural rules of the operational semantics.

Notice that the structural rule

$$(close) \quad \frac{p \xrightarrow{\overline{a}(b)} p' q \xrightarrow{ab}_1 q' b \notin \mathtt{fn}(q)}{p \mid q \xrightarrow{\tau} \nu b\,(p' \mid q')}$$

is not needed since all restrictions can be brought outside the parallel composition of processes using the laws of structural congruence.

We now present the bisimilarity semantics for asynchronous π-calculus.

Definition 5.1 A symmetric relation S on processes is an *$o\tau$-bisimulation* if pSq, $p \xrightarrow{\alpha} p'$, α is not an input action, and $\mathtt{bn}(\alpha) \cap \mathtt{fn}(q) = \emptyset$ implies $q \xrightarrow{\alpha} q'$ and $p'Sq'$. ☐

Definition 5.2 A *direct ground bisimulation* is an $o\tau$-bisimulation S such that whenever pSq and $p \xrightarrow{ab}_0 p'$ and $a \in \mathtt{fn}(p \mid q)$, $b \notin \mathtt{fn}(p \mid q)$, then $q \xrightarrow{ab}_0 q'$ and $p'Sq'$. Let $\sim_D$ be the largest direct ground bisimulation. ☐

The remarkable property of ground bisimulation is that the formal parameter of the input prefix is treated as a fresh name, and the subject of the input prefix is a name that is already known.

Exploiting the results of [4] it is immediate to prove the coincidence of $\sim_D$ with the bisimulation proposed by Honda and Tokoro [40].

$$
\begin{array}{llll}
(in_0) & p \xrightarrow{ab}_0 p \mid \overline{a}b & (in_1) & a(b).p \xrightarrow{ac}_1 p[c/b] \\[2ex]
(out) & \overline{a}b \xrightarrow{\overline{a}b} \mathsf{0} & (rep) & \dfrac{g \xrightarrow{\alpha} p}{!g \xrightarrow{\alpha} p \mid !g} \\[3ex]
(open) & \dfrac{p \xrightarrow{\overline{a}b} p' \quad a \neq b}{\nu b\, p \xrightarrow{\overline{a}(b)} p'} & (\nu) & \dfrac{p \xrightarrow{\alpha} p' \quad a \notin n(\alpha)}{\nu a\, p \xrightarrow{\alpha} \nu a\, p'} \\[3ex]
(synch) & \dfrac{p \xrightarrow{\overline{a}b} p' \quad q \xrightarrow{ab}_1 q'}{p \mid q \xrightarrow{\tau} p' \mid q'} & (comp) & \dfrac{p \xrightarrow{\alpha} p', \mathtt{bn}(\alpha) \cap \mathtt{fn}(q) = \emptyset}{p \mid q \xrightarrow{\alpha} p' \mid q} \\[3ex]
(suml) & \dfrac{g \xrightarrow{\alpha} p}{g + h \xrightarrow{\alpha} p} & (sumr) & \dfrac{g \xrightarrow{\alpha} p}{h + g \xrightarrow{\alpha} p} \\[3ex]
(cong) & \dfrac{p \equiv q \quad q \xrightarrow{\alpha} q' \quad q' \equiv p'}{p \xrightarrow{\alpha} p'} & &
\end{array}
$$

Table 4: Direct labelled transition system (with early instantiation)

Coordination and Names. Sequential processes g are autonomous entities and are at the source of any activity. Hence, sequential processes can be seen as software components possibly distributed over a network. In this perspective, process combinators define the *coordination language*. However, in π-calculus (both synchronous and asynchronous), syntactic process combinators (i.e. parallel composition and restriction) are not the only language features which are relevant for defining configurations. In fact *free names* are used for this purpose, since they specify the *interfaces* of processes. Therefore the result of the operation of parallel composition heavily depends on which are the homonymous ports of the components (since $\mathtt{fn}(p \mid q) = \mathtt{fn}(p) \cup \mathtt{fn}(q)$).

This idea of viewing the free names as the specification of the communication interface leads to a more abstract representation of π-processes, and in particular of sequential processes.

Given any sequential process g, we assume the existence of a function *std* which returns an *abstract* sequential process $std(g)$ such that

$$g \equiv std(g)(a_1, \ldots, a_n)$$

for a suitable ordering of the free names $a_1, \ldots, a_n$ of g. We want *std* to be invariant under substitution (i.e. $std(g) = std(g\sigma)$ for any injective free name substitution σ) and under structural equivalence (i.e. if $g \equiv g'$ then $std(g) = std(g')$). For instance *std* could be defined by first selecting a standard representative g' of the equivalence class $[g]_\equiv$ and then by writing $std(g) = \lambda a_1. \ldots \lambda a_n.g'$, where $a_1, \ldots, a_n$ are the free names of g' (and of g) listed according to the order of their first occurrence in the *preorder tree walk* of the abstract syntax tree of the sequential process g'.

It is now easy to see that for any process p we have an equivalent process of the form

$$p \equiv \nu a_1 \ldots \nu a_k (g_1(\vec{a_1}) \mid \ldots \mid g_n(\vec{a_n}) \mid \overline{b_1} c_1 \mid \ldots \mid \overline{b_m} c_m)$$

where $g_1, \ldots, g_n$ are abstract sequential processes and $\vec{a_1}, \ldots, \vec{a_n}$ are suitable tuples of names.

5.2 A Tile Rewrite System for Asynchronous π-calculus

Hereafter, we introduce the components of the tile rewrite system, i.e. sorts, horizontal signature, vertical signature and rewrite rules. While the total number of rules will be infinite, finitely many rules will suffice to describe the evolution of any process.

Sorts. The first sort, e, labels a single name in every cogs-graph representing a configuration, and this name is shared by all sequential processes and messages. Its role is to sequentialize all the activities, since here we are modeling the interleaving version of the calculus. The symbol we choose, e, is reminiscent of the word *event*, since the only name of sort e can be considered as a name generated by the last transition and corresponding to the last event. The second sort, a, is reminiscent of *action*, and the names of this sort will correspond to names of the calculus.

Horizontal Signature. The symbols of Σ_H are

$$A[g] : \epsilon \to e\, a^{|\mathtt{fn}(g)|} \qquad (Agent)$$

$$M :: \epsilon \to e\, a\, a \qquad (Message)$$

$$\nu : a \to \epsilon \qquad (Restriction)$$

where we have symbols $A[g]$ for all abstract sequential processes g.

Vertical Signature. The symbols of Σ_V are

$$input : e\, a \to e\, a \qquad (Input)$$

$$output : e\, a\, a \to e \qquad (Output)$$

$$Output : e\, a \to e\, a \qquad (Bound\, Output)$$

$$tau : e \to e \qquad (Tau)$$

Rewrite Rules. There are two kinds of rewrite rules: *activity* rules and *coordination* rules. Activity rules describe the evolution of a sequential process and they depend on the specific structure of the process. Instead, coordination rules coordinate the evolution of sequential processes inside the operational environment.

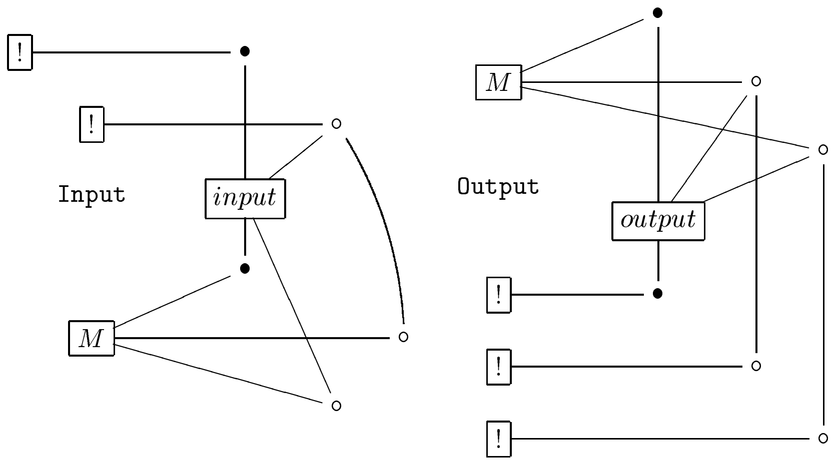
Activity Rules. We first present the `Input` and `Output` rules for messages. They model messages entering and leaving the system.

$$(\texttt{Input}) \quad e!, a! \xrightarrow[e'b:=input(e,a),a:=a]{} M(e', a, b)$$

$$(\texttt{Output}) \; M(e, a, b) \xrightarrow[e':=output(e,a,b),ab:=ab]{} e'!, a!, b!$$

Notice that the `Input` rule restricts messages entering the system to those messages whose subject is an already known name. Notice that, since we are considering the ground case, the formal parameter of input prefix is treated as a fresh name.

The wire-and-box diagrams for the `Input` and `Output` activity rules are displayed below. In the diagrams the symbol $\bullet$ is used to represent names of sort e, while symbol $\circ$ represents names of sort a.



We now introduce the activity rules for sequential processes. We employ an inference rule, called `Synch`, which creates a rewrite rule (without trigger) for each message reception.

$$g(\vec{a}) \mid \overline{b}c \xrightarrow{\tau} \nu a_1 \ldots \nu a_k (g_1(\vec{a_1}) \mid \ldots \mid g_n(\vec{a_n}) \mid \overline{b_1}c_1 \mid \ldots \mid \overline{b_m}c_m)$$

(`Synch`) implies

$$A[g](e, \vec{a}), M(e, b, c) \xrightarrow[e':=tau(e),\vec{a}bc:=\vec{a}bc]{} \begin{cases} A[g_1](e', \vec{a_1}), \ldots, A[g_n](e', \vec{a_n}) \\ M(e', b_1, c_1) \ldots, M(e', b_m, c_m) \\ a_1 := \nu, \ldots, a_k := \nu \end{cases}$$

As an example, let us consider the sequential process

$$g(a) = a(b).\nu c(c(d).0 \mid \overline{b}c)$$

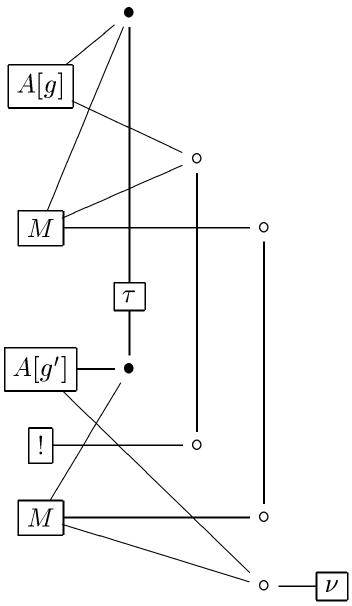
and the transition

$$g(a) \mid \overline{a}a' \xrightarrow{\tau} \nu c(c(d).0 \mid \overline{a'}c).$$

The corresponding activity tile is thus:

$$A[g](e, a), M(e, a, a') \xrightarrow[e':=tau(e),aa':=aa']{} A[g'](e', c), M(e', a', c), c := \nu, a!$$

with $g'(c) = c(d).0$. The wire-and-box representation of the tile is displayed below.



Notice that while the transition and the activity tile above look similar, they are actually very different, in that the names occurring in the transition are global, while those in the rule are only used in the linear notation for it. Thus if we rename the free names a and a' in the transition, we obtain a different transition, while if we rename them in the rule, provided we keep the same ordering between them, we obtain the same rule. Moreover, if we change the order, we obtain a tile which differs from the previous one only with regard to composition with perm tiles.

Coordination Rules. We have the following rewrite rules.

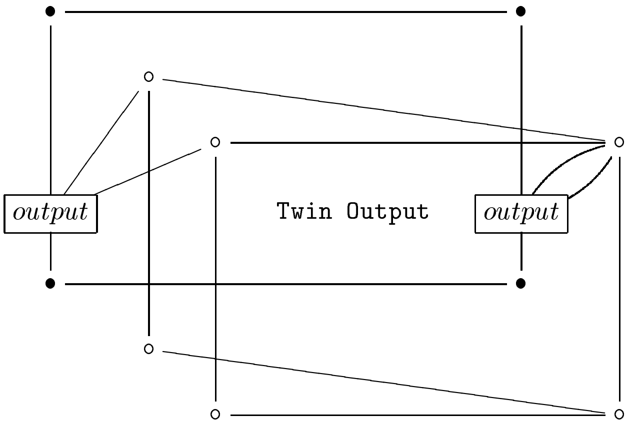
(`Twin Output`) $$eab := eaa \xrightarrow[e':=output(e,a,a),a:=a]{e':=output(e,a,b),ab:=ab} e'ab := e'aa$$

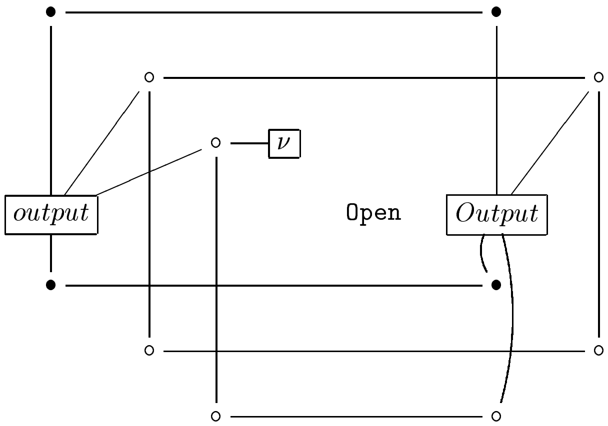
(`Open`) $$ea := ea, b := \nu \xrightarrow[e'b:=Output(e,a),a:=a]{e':=output(e,a,b),ab:=ab} e'ab := e'ab$$

(`Event Output Sharing`) $$eabe'' := eabe \xrightarrow[e':=output(e,a,b),ab:=ab]{e':=output(e,a,b),e''ab:=e''ab} e'abe'' := e'abe'$$

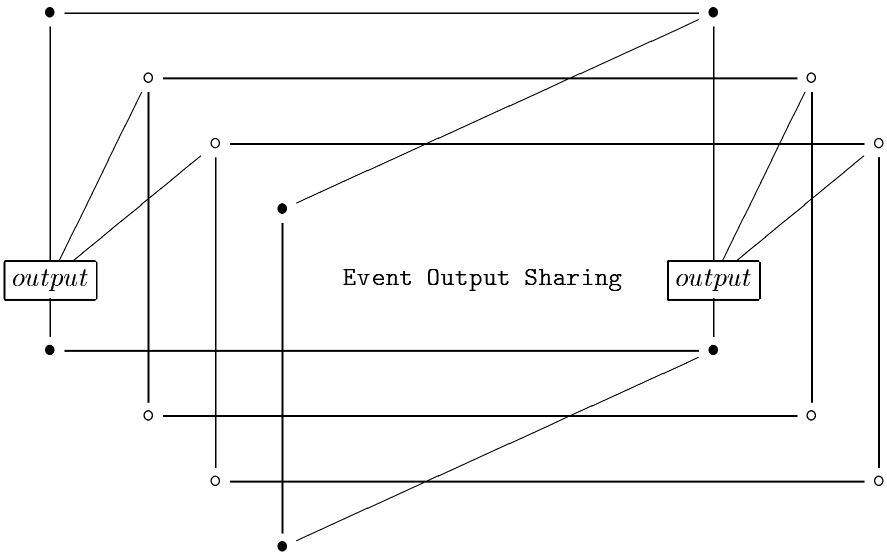
The first rule, (`Twin Output`), identifies subject and object of output. The `Open` rule models the extrusion of a name while a message is leaving the system. The trigger is the graph $\{e' := \mathit{output}(e, a, b), ab := ab\}$, and the application of the rule causes the transformation of the initial configuration $\{ea := ea, b := \nu\}$ into the final configuration $\{e'ab := e'ab\}$ (where the restriction is removed) and the creation of effect $\{e'b := \mathit{Output}(e, a), a := a\}$. Notice that the effect creates a new name b, which is clearly "the same" name which occurred restricted in the initial configuration.

The wire-and-box representation of rules `Twin Output` and `Open` is shown below.





The last rule, `Event Output Sharing`, has an extra event name e'' in the trigger, which is identified with e in the initial configuration and with e' in the final configuration. This rule corresponds directly to the *comp* rule in Table 4. The wire-and-box representation of rule `Event Output Sharing` is shown below.



Here we have several other sharing tiles and all of them allow for some trigger to be transmitted unmodified as effect, but sharing some name. More precisely, we have `x Output Sharing`, where `x` besides `Event` can be `Subject`, `Object`, `Event Twin` or `Event Extruding`, `Action Twin` or `Subject Extruding`. Likewise, we have `Event Input Sharing`, `Subject Input Sharing`, and `Tau Sharing`. We do not show all these rules.

While the above enumeration is clearly dictated by the connections of the vertical constructors, it would be better if all the sharing rules could be obtained in a conceptually more unified way. This is actually what happens, at least for sharing rules operating on the action sort a. The rule we need is

$$a_1 a_2 := a \xrightarrow[aa_1' a_2' := aaa]{a_1, a_1', a_2, a_2' := a_1, a_1, a_2, a_2} a_1 a_2 a_1' a_2' := aaa_1' a_2'.$$

It is easy to see that by composing this tile vertically with identity tiles it is possible to derive not only all the action sharing rules, but also the `Twin Output Rule`. A similar factorization is not possible for event sharing rules. This shows how interleaving semantics handles events quite differently than (other) names. On the contrary, an analogous tile for events works in the case of the concurrent semantics shown in Section 4.2. It yields rules `Twin`, $\texttt{Comp}_\lambda$ and `TwinComp`.

While there is an infinite number of activity tiles, it is possible to show the following finiteness result.

Proposition 5.3 (finiteness) *For any horizontal graph s, the rewrite rules needed to entail all the tile sequents having s as initial configuration are finitely many.*

Proof. Graph s will contain a finite number of symbols $A[g_i]$, $i = 1, \ldots, n$. Thus it suffices to note that in every reachable configuration only symbols $A[g_i']$ can appear, such that g_i' is a syntactic subagent of g_i. Since these are finite, only a finite number of activity tiles can apply. □

5.3 Comparing SOS and Tile Semantics

In this section we present the coincidence between tile bisimilarity on the tile rewrite system and direct ground bisimulation, which coincides with asynchronous bisimulation [40, 4], on π agents.

The proof is similar to the one given in Section 4.3 for location semantics. Here the translation function $[\![\_]\!]$ maps processes p into configurations $[\![p]\!]$ of type $\epsilon \to u$, where u contains only an occurrence of sort e. We have $[\![p]\!] = s$ iff

$$\begin{aligned} p &\equiv \nu a_1 \ldots \nu a_k (g_1(\vec{a_1}) \mid \ldots \mid g_n(\vec{a_n}) \mid \overline{b_1} c_1 \mid \ldots \mid \overline{b_m} c_m) \\ s &= \{A[g_1](e, \vec{a_1}), \ldots, A[g_n](e, \vec{a_n}), M(e, b_1, c_1), \ldots, M(e, b_m, c_m), \\ &\qquad a_1 := \nu, \ldots, a_k := \nu\} \end{aligned}$$

The equivalence result is stated by the following proposition.

Proposition 5.4 *Assume $S_1 = \texttt{fn}(p_1)$, $S_2 = \texttt{fn}(p_2)$, $S = S_1 \cup S_2$. Let $\{a_1, \ldots, a_n\} = S_1 \cap S_2$; $\{b_1, \ldots, b_h\} = S \setminus S_2$; $\{c_1, \ldots, c_k\} = S \setminus S_1$; $\{d_1, \ldots, d_{h+k}\} \cap S = \emptyset$; $a_i \leq d_j$ for $i = 1, \ldots, n$, $j = 1, \ldots, h + k$. Then*

$$p_1 \sim_D p_2 \textit{ iff } [\![p_1]\!]; id_e \otimes G_1 \sim_t [\![p_2]\!]; id_e \otimes G_2$$

where

$$G_1 = \{a_1 \dots a_n b_1 \dots b_h := a_1 \dots a_n d_1 \dots d_h, d_{h+1}!, \dots, d_{h+k}!\},$$

$$G_2 = \{a_1 \dots a_n c_1 \dots c_k := a_1 \dots a_n d_{h+1} \dots d_{h+k}, d_1!, \dots, d_h!\}.$$

Proof. The gs-graphs which are composed with $[\![p_1]\!]$ and $[\![p_2]\!]$ ensure that the names in $S_1 \cap S_2$ are listed first for both processes, and thus they are in correct correspondence, while the names in $S \setminus S_2$ and $S \setminus S_1$ correspond to new names. The proof is similar to the one of Proposition 4.35. □

6 Concluding Remarks

We have introduced a novel approach for the operational rules of process calculi. An innovative aspect is related to the use of gs-graphs, a simple extension of term graphs, as underlying algebraic structure for configurations and observations. By the use of gs-graphs, no global names exist in the system, but just operators for specifying name sharing among contiguous sequential processes, and rules for controlling the behavior accordingly. All the name handling issues, including alpha conversion and new name generation, are thus delegated to the underlying logic.

We have shown that tile systems are particularly suited for defining, in a compositional framework, operational models of calculi with localities and with bound names. For this purpose, we presented tile systems for located CCS and for asynchronous π-calculus, and proved them equivalent to the ordinary (bisimulation) semantics. An advantage of the tile approach is the full compositionality of the underlying logic, which is able to handle computations of (nonclosed) system components as they were new rewrite rules specifying complex coordinators.

As suggested by the title of the paper, we believe that our approach is quite systematic. We now outline the various steps needed to model a generic model of computation with tiles.

The natural targets of the tile approach are models of reactive systems which are compositional in both space and time. Thus the first step is to choose the compositional structures we want to use in order to represent the states of the system and the interactions with the external world. In the SOS approach, the states are represented by terms and the interactions by action sequences. In this paper we have shown that gs-graphs can be more natural and convenient when sharing of locations or of bound names must be modeled. In [50] states are ordinary hypergraphs, and compositionality is achieved by gluing them on the nodes: again, gs-graphs turn out to be the natural choice. In most cases, however, more general structures are needed. In [16] the normal form of gs-graphs shown in Section 2.2 is extended to structures that can represent partial orders and partitions. They are gs-graphs and cogs-graphs at the same time, and have additional axioms between duplicators and dischargers and their reversed versions. Partial orders are needed to represent causality, for instance in distributed versions of the π-calculus and of the actor model [51]. Partitions allow one to build feedback connections and to specify that certain resources are the same after being used, i.e. they are read but not consumed [34]. It would also be interesting to study the use of process contexts as observation labels as in [8, 58].

The next step is to choose auxiliary tiles. In this paper we have introduced permutations, duplicators and dischargers in states and observations, but we only used auxiliary tiles for permutations. Perm tiles allowed us to choose freely the ordering of the location created by a transition among the existing locations, a feature which is essential in location semantics. Had we also introduced auxiliary tiles for duplicators in π-calculus, arbitrary non-injective substitutions of names would have been possible at each step. This feature is incompatible with π-calculus semantics, but it is needed for instance in fusion calculus [53]

Then we have to choose how to model recursion or replication. In this paper, located CCS is presented without recursion, while π-calculus is equipped with activity tiles, which are essentially recursive definitions. Recursive definitions could also be directly included in the configurations using iteration theories [11]. We think that replication could be introduced simply by exploiting duplicators in observations and configurations.

In this paper we have employed flat tiles, i.e. here tiles have no proof terms associated with them and are characterized simply by their configurations and observations. The full version is more complicated, but it may be useful for a very precise modeling of the amount of concurrency present in a computational model. For instance in [22] it is shown that rewriting logic (which in the non-conditional case can be considered as tile logic with only identity observations) captures exactly Boudol's notion of concurrent term rewriting. In fact, we think that by employing non-flat tiles we could make Proposition 4.32 hold in both directions.

When choosing horizontal and vertical signatures, the first option concerns sorts. In this paper we had one sort for located CCS and two sorts for the π-calculus, representing processes and names. More sorts, or in general more elaborated type systems, may capture important restrictions in accessing resources and in communicating names. For instance in [51], a simple distinction between actors and actor references enforces the uniqueness of actor names. As shown in the paper, signature symbols have in our view the meaning of language constructs or of basic observations. Notice that symbols of an arity different from one in the vertical signature are needed for synchronization and for the creation of new names.

The format of the rewrite rules may be essential in proving important properties of the system. For instance, as highlighted by Proposition 4.8, 4.15 and 4.33 in the paper, a main proof method is given by the ability to vertically or horizontally decompose a derivable tile into more basic derivable tiles. This decomposition is driven by a decomposition of the effect or of the initial configuration, which in turn is possible if every rule (in a certain class) has an atomic initial configuration or an atomic effect. Among the properties provable in this way there are also important results such as "bisimulation is a congruence", as shown in [32] for the strong semantics of ordinary CCS. It would be interesting to develop similar results for more general tile formats, thus exploiting the powerful rule induction intrinsic in the tile model.

While tile logic can be considered as an extension of rewriting logic, a term rewriting engine is probably the best way to make tiles executable. In fact, mapping tile logic into rewriting logic is feasible both theoretically and practically [48, 17]. For instance, the rewrite system for located CCS shown in the paper runs on the Maude system at SRI International

[17], taking advantage of the reflection capabilities of that system for enforcing synchronization and for controlling nondeterminism.

On the application side, we would like to mention the graphical orientation of tiles. While no graphical support for tiles is currently available, we think that the wire-and-box representation of tiles is (after some training) expressive and suggestive. It could be useful for users with a limited theoretical background, or for applications where simplicity and flexibility is essential, such as software architectures. In the latter area, some work is in progress [38, 39] which relies on the integration of various specification methods (graph rewriting, process description languages, reconfiguration via coordination and constraint solving). Tile logic provides a natural setting where these formalisms can be reconciled.

7 Acknowledgments

We would like to thank Roberto Bruni, Fabio Gadducci and Marco Pistore for helpful comments. We are also grateful to the anonymous referees for their constructive comments which have helped us to improve the first version of the paper.

References

[1] Aceto, L. (1994) Deriving Complete Inference Systems for A Class of GSOS Languages Generating Regular Behaviours, in “Proeedings CONCUR 94”, LNCS 836, .449-464.

[2] Aceto, L., Bloom, B., and Vaandrager, F. (1992) Turning SOS Rules into Equations, *Information and Computation* **111**, 1-52.

[3] Aceto, L. and Ingolfsdottir, A. (1995) CPO models for GSOS Languages *Information and Computation* **129** (2), 107-141.

[4] Amadio, R., Castellani, I., and Sangiorgi, D. (1998) On Bisimulations for the Asynchronous π-calculus, *Theoretical Computer Science* **195**, 291-324.

[5] Badouel, E., and Darondeau, P. (1995) Trace Nets and Process Automata *Acta Informatica* **32** (7), 647-679.

[6] Barendregt, H., van Eekelrn, M.C.J.D., Glauert., J.R.W., Kennaway, J., Plasmeijer, M.J., and Sleep, M.R. (1987) Term Graph Reduction, in Proceedings PARLE, LNCS 259, 141–158..

[7] Baeten, J., and Verhoef, C. (1993) A Congruence Theorem for Structured Operational Semantics with Predicates, in: Proceedings CONCUR 93, LNCS 715.

[8] Bernstein, K., (1998) A Congruence Theorem for Structured Operational Semantics of Higher-Order Languages, in Proceedings LICS 98.

[9] Bloom, B.(1995) Structured Operational Semantics for Weak Bisimulations *Theoretical Computer Science* **146**, 25-68..

[10] Bloom, B., Istrail, S. and Meyer, A.R. (1995) Bisimulation Can't Be Traced, *Journal of the ACM* **42** 232–268.

[11] Bloom, S. and Ésik, Z., (1993) Iteration Theories, Springer Verlag, EATCS Monographs on Theoretical Computer Science.

[12] Boudol, G. (1992) Asynchrony and the π-calculus (note), *Rapport de Recherche 1702*, INRIA Sophia-Antipolis.

[13] Boudol, G, Castellani, I., Hennessy, M., and Kiehn, A. (1993) Observing Localities, *Theoretical Computer Science*, **114**: 31–61.

[14] Boudol, G, Castellani, I. (1994) Flow Models of Distributed Computations: Three Equivalent Semantics for CCS, *Information and Computation* **114**(2):247-314.

[15] Bruni, R. and Montanari, U. (1997) Zero-Safe Nets, or Transition Synchronization Made Simple, in Proceedings EXPRESS 97, ENTCS 7.

[16] Bruni, R., Gadducci, F. and Montanari, U. (1998) Normal Forms for Partitions and Relations, Abstract, in Proceedings Workshop on Algebraic Development Techniques.

[17] Bruni, R., Meseguer,J., and Montanari, U. (1998) Process and Term Tile Logic, Technical Report, SRI-CSL-98-06 SRI International; and TR-98-09, Dipartimento di Informatica, University of Pisa.

[18] Corradini, F., and De Nicola, R. (1994) Distribution and Locality of Concurrent Systems, in Proceedings ICALP 94, LNCS 920.

[19] Corradini, A., Ehrig, H., Löwe, M., Montanari, U., and Rossi, F. (1997) Abstract Approach to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach, in The Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations, (G Rozenberg, Ed.) World Scientific.

[20] Corradini, A., and Gadducci, F. (1998) An Algebraic Presentation of Term Graphs via Gs-Monoidal Categories, *Applied Categorical Structures*, to appear.

[21] Corradini, A., and Gadducci, F. (1997) A 2-Categorical Presentation of Term Graph Rewriting, in Proceedings CTCS 97, LNCS 1290.

[22] Corradini, A., Gadducci, F., and Montanari, U. (1995) Relating Two Categorical Models of Term Rewriting, in Rewriting Techniques and Applications, LNCS 914, 225-240.

[23] Darondeau, P., and Degano, P. (1989) Causal Trees, in Proceecings ICALP 89, LNCS, Vol. 372.

[24] Degano, P. De Nicola, R., and Montanari, U. (1989) Partial Ordering Descriptions and Observations of Nondeterministic Concurrent Processes, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354.

[25] Degano, P., Meseguer, J., and Montanari, U. (1996) Axiomatizing the Algebra of Net Computations and Processes, *Acta Informatica* **33**(7), 641-667.

[26] De Simone, R. (1995) Higher Level Synchronising Devices in MEIJE-SCCS, *Theoretical Computer Science* **37** (3), 245-267.

[27] Ehresmann, C. (1963) Catégories Structurées: I and II, Ann. Éc. Norm. Sup. 80, Paris (1963), 349-426; III, Topo. et Géo. diff. V, Paris.

[28] Ferrari, G. and Montanari, U. (1990) Towards the Unification of Models for Concurrency, in Proceedings 15th Colloquium on Trees and Algebra of Programming (CAAP 90), LNCS 431, 162-176.

[29] Ferrari, G. and Montanari, M. (1997) A Tile-Based Coordination View of the Asynchronous π-calculus, in Proceedings MFCS 97, LNCS 1295, 52-70.

[30] Ferrari, G. and Montanari, M. (1997) Tiles for Concurrent and Located Calculi, in Proceedings EXPRESS 97, ENTCS 7.

[31] Gadducci, F. (1996) On the Algebraic Approach to Concurrent Term Rewriting, PhD Thesis, Università di Pisa, Pisa, Technical Report TD-96-02, Department of Computer Science, University of Pisa.

[32] Gadducci, F and Montanari, U. (1998) The Tile Model, in Proofs, Languages and Interaction: Essays in Honour of Robin Milner, (G. Plotkin, C. Stirling and M. Tofte, Eds.) MIT Press.

[33] Gadducci, F and Montanari, U. (1996) Tiles, Rewriting Rules and CCS, in in Proceedings international workshop on Rewriting Logic and Applications, ENTCS 4, 1-19.

[34] Gadducci, F. and Montanari, U. (1998) Axioms for Contextual Net Processes, in Proceedings ICALP'98, LNCS 1443, 296-308.

[35] Gorrieri, R., and Montanari, U. (1995) On the Implementation of Concurrent Calculi into Net Calculi: Two Case Studies, *Theoretical Computer Science* **141**(1-2), 195-252.

[36] Groote, J.F. (1990) Transition System Specifications with Negative Premises, in Proceedings CONCUR 90, LNCS 458.

[37] Groote, J.F., and and Vandraager, F. (1992) Structured Operational Semantics and Bisimulation as a Congruence, *Information and Computation* **100**, 202-260.

[38] Hirsch, D., Inverardi, P., and Montanari, U., (1999) Graph Grammars and Constraint Solving for Software Architecture Styles, to appear in Proceedings ISAW 99.

[39] Hirsch, D., Inverardi, P., and Montanari, U., (1998) Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving, to appear in Proceedings WICSA 98.

[40] Honda, K., and Tokoro, M., (1992) An Object Calculus for Asynchronous Communication, Object-Based Concurrent Computing (Tokoro, Nierstrass and Wegner Eds.), LNCS 612, 21-51.

[41] Kiehn, A. (1994) Local and Global Causes, *Acta Informatica* **31**, 697-718.

[42] Laneve, C., and Montanari, U. (1996) Axiomatizing Permutation Equivalence, *Math. Struct. in Comp. Sc.* **6**, 219-249.

[43] Larsen, K., and Xinxin, L. (1991) Compositionality Through an Operational Semantics of Contexts, *Journal of Logic and Computation*, **1**(6).

[44] Lawvere, F.W., (1963) Functorial Semantics of Algebraic Theories, in Proceedings National Academy of Science **50**.

[45] Meseguer, J. (1992) Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science* **96**, 73-155.

[46] Milner, R. (1989) *Communication and Concurrency*, Prentice Hall.

[47] Milner, R., Parrow, J., and Walker, D. (1992) A Calculus of Mobile Processes (parts I and II), *Information and Computation*, **100** 1–77.

[48] Montanari, U., and Meseguer, J. (1998) Mapping Tile Logic into Rewriting Logic, in Recent Trends in Algebraic Development Techniques (Francesco Parisi-Presicce, Ed.), LNCS 1376, 62-91.

[49] Montanari, U., Pistore, M., and Yankelevich, D. Efficient Minimization up to Location Equivalence, in Proceedings ESOP 96, LNCS 1058.

[50] Montanari, U., and Rossi, F. (1998) Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems, to appear in *Applied Category Theory.*

[51] Montanari, U., and Talcott, C. (1998) Can Actors and π-Agents Live Together?, in Second Workshop on Higher-Order Operational Techniques in Semantics, (Andrew Gordon, Andrew Pitts and Carolyn Talcott, Eds.), ENTCS 10.

[52] Montanari, U., and Yankelevich, D. (1995) Location Equivalence in a Parametric Setting. *Theoretical Computer Science* **149**.

[53] Parrow, J., and Victor, B., (1998) The τ-Laws of Fusion, in Proceedings CONCUR'98, LNCS 1466, 99-114.

[54] Plotkin, G. (1981) A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

[55] Radestock, M., and Eisenbach, S. (1996) Semantics of Higher Order Coordination Languages, in Proceedings COORDINATION'96, LNCS 1061, 339-356.

[56] Rensink, A. (1997) Bisimilarity of Open Terms, in Proceedings EXPRESS 97, ENTCS 7.

[57] Sangiorgi, S. (1992) Expressing mobility in process algebras: first-order and higher-order paradigms, PhD Thesis CST-99-93, University of Edinburgh.

[58] Sewell, P. (1998) From Rewrite to Bisimulation Congruences, in Proceedings CONCUR 98, LNCS 1466, 269-284.

[59] Thomsen, B. (1993) Plain Chocs, *Acta Informatica*, **30**(1), 1-59.

[60] Ulidowski, I. (1995) Axiomatisations of Weak Equivalences for De Simone Languages, in Proccedings CONCUR'95, LNCS 962.

[61] Vaandrager, F. (1993) Expressiveness Results for Process Algebras, in *Semantics: Foundations and Applications*, LNCS 666.

[62] Walker., D. (1995) Ojects in the π-calculus, *Information and Computation* **116**(2), 252-271.

Appendix: Gs-Monoidal Theories

Here the interested reader may find the axiomatic definition (taken from [31, 32, 20, 21]) of gs-monoidal theories. These theories are similar to ordinary algebraic (Lawvere) theories [44], the differences being the extended signature and the missing naturality axioms for duplicators and dischargers. Here we introduce gs-monoidal theories as an extension of monoidal theories.

Definition 7.1 *(graphs)*
A *graph* G is a 4-tuple $\langle O_G, A_G, \delta_0, \delta_1 \rangle$: O_G, A_G are sets whose elements are called *objects* and *arrows* respectively (ranged over by $u, v, \ldots$ and $f, g, \ldots$), and $\delta_0, \delta_1 : A_G \to O_G$ are functions, called *source* and *target* respectively. A graph G is *reflexive* if there exists an *identity* function $id : O_G \to A_G$ such that $\delta_0(id(u)) = \delta_1(id(u)) = u$ for all $u \in O_G$; it is *with pairing* if its class O_G of objects forms a monoid; it is *monoidal* if it is reflexive with pairing and also its class of arrows forms a monoid, such that id, δ_0 and δ_1 respect the neutral element and the monoidal operation. $\square$

Definition 7.2 *(symmetric monoidal theories)*
Given a set S of sorts and a (hyper) signature Σ, the associated *symmetric monoidal theory* $\mathbf{SM}(\Sigma)$ is the monoidal graph whose objects are the elements of the free commutative monoid $(S^*, \otimes, \epsilon)$, and whose arrows are those generated by the following inference rules

$$(generators)\ \frac{f \in \Sigma_{u,v}}{f : u \to v \in \mathbf{SM}(\Sigma)} \qquad (sum)\ \frac{t : u \to v, t' : u' \to v'}{t \otimes t' : u \otimes u' \to v \otimes v'}$$

$$(identities)\ \frac{u \in S^*}{id_u : u \to u} \qquad (composition)\ \frac{t : u \to v, t' : v \to w}{t; t' : u \to w}$$

$$(permutations)\ \frac{u, v \in S^*}{\rho_{u,v} : u \otimes v \to v \otimes u}.$$

Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the *functoriality* axiom

$$(t_1 \otimes t_2); (t_1' \otimes t_2') = (t_1; t_1') \otimes (t_2; t_2')$$

whenever both sides are defined; the *identity* axiom $u; t = t = t; v$ for all $t : u \to v$; the *coherence* axioms

$$id_{u \otimes v} = id_u \otimes id_v \qquad \rho_{u \otimes v, w} = (id_u \otimes \rho_{v,w}); (\rho_{u,w} \otimes id_v)$$

$$\rho_{\epsilon,u} = \rho_{u,\epsilon} = id_u \qquad \rho_{u,v}; \rho_{v,u} = id_{u \otimes v}$$

for all $u, v, w \in S^*$; and the *naturality* axiom

$$(f \otimes f'); \rho_{v,v'} = \rho_{u,u'}; (f' \otimes f)$$

for all $f \in \Sigma_{u,v}$ and $f' :\in \Sigma_{u',v'}$. $\square$

Definition 7.3 *(gs-monoidal theories)*

Given a set S of sorts and a signature Σ, the associated *gs-monoidal theory* $\mathbf{GS}(\Sigma)$ is the monoidal graph whose objects are the elements of the free commutative monoid $(S^*, \otimes, \epsilon)$ and whose arrows are those generated by the inference rules for symmetric monoidal theories plus the following two additional rules

$$(duplicators)\ \frac{s \in S}{\nabla_s : s \to s \otimes s} \qquad (dischargers)\ \frac{s \in S}{!_s : s \to \epsilon}$$

The monoid of arrows satisfies all the axioms of symmetric monoidal theories and the following additional coherence axioms

$$\nabla_s; (id_s \otimes \nabla_s) = \nabla_s; (\nabla_s \otimes id_s) \qquad \nabla_s; \rho_{s,s} = \nabla_s \qquad \nabla_s; (id_s \otimes !_s) = id_s$$

for all $s \in S$. □