Process algebras with localities

Ilaria Castellani

INRIA, 2004 Route des Lucioles, 06902 Sophia-Antipolis, France.
Ilaria.Castellani@sophia.inria.fr

Abstract

Process algebras can be enriched with *localities* that explicitly describe the distribution of processes. Localities may represent physical machines, or more generally *distribution units* where processes are grouped according to some criterion like the sharing of resources. In a concurrent process, localities are naturally associated with (groups of) parallel components. These localities then intervene in the semantics of processes and become part, to some extent, of their observable behaviour.

In a first line of research, initiated in the early nineties, localities have been used to give *noninterleaving* semantics for process algebras, and particularly for Milner's calculus CCS. Here localities are used to differentiate parallel components. The resulting semantics, taking into account distribution, is more discriminating than the standard interleaving semantics of the calculus. It is also incomparable with other noninterleaving semantics proposed for CCS, based on the notion of causality.

More recently, localities have appeared in a number of new calculi for describing mobile processes. The idea here is that some "network awareness" is required to model wide-area distributed mobile computation. In these calculi localities are more than simple units of distribution. According to the case, they become units of failure, of communication, of migration or of security.

This chapter reviews in some detail the first body of work, and tries to delineate the main ideas of the more recent studies, which are still, for the most part, at an early stage of development.

Keywords: Process algebras and calculi, localities, concurrency, bisimulation, distribution, causality, mobility, migration, failure.

Contents

1 Introduction

The aim of this chapter is to review some established work on process calculi with explicit localities or locations[1], as has been developed in the last decade by several authors [27, 28], [5], [115], [37], [113, 167, 114], [116], [48], as well as to delineate new directions of research in this area that have been spurred by the growing interest in calculi for mobile processes.

The presentation will accordingly be structured in two parts: in the first (Sections 2–3) we shall present the more consolidated work on traditional process calculi, while in the second (Section 4) we will report more informally on recent proposals for combining locations and mobility. The emphasis, as well as most of the technical material, will be on the first part, where locations are used as a means to provide *noninterleaving semantics* for process algebras.

The starting idea of the former work is that *distribution* is an important aspect of parallel systems and should somehow be reflected in their semantics. If systems are modelled in

[1]We shall use the two words interchangeably throughout the chapter.

a process calculus equipped with a parallel operator, like Milner's calculus CCS [106, 108], this can be achieved by assigning *locations* to parallel components and then observing actions together with the location at which they occur. Using these enriched observations, one obtains so-called *distributed semantics* for the language, which are more discriminating than the standard interleaving semantics. The aim of these distributed semantics is twofold: (1) give a measure of the *degree of parallelism* present in a system, and (2) keep track of the *local behaviour* of components within a system. While (1) can be useful for implementing a system on a given physical architecture, (2) allows for the detection of local properties, like a *local deadlock* or the dependence from a local resource.

The above-mentioned papers are all based on the language CCS, and adopt semantic notions which are variants of the notion of *bisimulation* introduced by Park [123] and Milner [106]. However they differ for the various subsets of CCS they consider. They also vary according to the way locations are assigned to processes: whether statically, before processes are run ([5], [115], [37], [116], [113]), or dynamically as the execution proceeds ([27, 28]). We shall refer to the former as "static approaches" and to the latter as "dynamic approaches".

The *static approaches* are further distinguished by the way locations are observed. In [116], [48] the identities of locations are significant and the resulting semantics is very *concrete*: processes are equated if they reside on the same set of locations and present the same behaviour at each location. In [5], [37], on the other hand, locations are observed up to a concurrency-preserving renaming which is built incrementally along executions. Here the resulting semantics is more *abstract*, identifying processes which are bisimilar in the classical sense and whose computations exhibit the same degree of distribution and the same local behaviours. There is a little subtlety to note here, as processes presenting the same degree of parallelism *in each run* need not consist of the same number of parallel components.

In the *dynamic approach*, which was the first to be developed [27, 28], locations are associated with actions rather than with parallel components. They are built incrementally when actions are performed, and then recorded in the residual processes. Here the location of an action can be viewed as the *history* of that action in the component where it occurs.

Clearly locations do not have the same intuitive meaning in the two approaches. In the static approach they represent *sites*, much as one would expect. In the dynamic approach, on the other hand, the location of an action is a record of its *local causes*, the actions that locally precede it. In spite of this difference in intuition, the dynamic and (abstract) static approach turn out to yield the same semantic notions of *location equivalence* and *location preorder* [5, 37]. This means that observing distribution - in conjunction with the usual interleaving behaviour - is essentially the same as observing local causality.

As mentioned previously, these location semantics emerged as particular noninterleaving semantics for process algebras. To provide some historical background, Section 2 will start with a brief survey of such noninterleaving semantics. A comprehensive presentation of the abstract location semantics, both static and dynamic, will then be given in Sections 2.2–2.5. The two approaches will be compared in Section 2.6. Instead, the description of the concrete model of [116, 48, 45] will be deferred to Section 4.1, as it fits more naturally with the more recent issues examined in that section.

In Section 3 the abstract location semantics is compared with other distributed semantics (like *distributed bisimulation* [36, 38, 95], the *local mixed-ordering equivalence* of [113], and variants of them [46, 47]), as well as with noninterleaving semantics based on causality (like *causal bisimulation* [49, 50] and *history preserving bisimulation* [73, 76, 75]). This comparison will be mainly based on the papers [3], [28], [97]. In particular, we will recall in some detail the *local/global cause semantics* of [97], which incorporates distributed and causal semantics into a single framework. As it turns out, distributed semantics and causality-based semantics are orthogonal ways of accounting for the parallelism in systems, which only agree on the communication-free fragment of CCS, a class of processes also known as BPP (Basic Parallel Processes).

At the end of Section 3 we present the extension of the location semantics of [28] to the π-calculus, as proposed by Sangiorgi in [141]. An important result of this work is that the location machinery of [28] can be encoded into the standard π-calculus, in such a way that the location semantics reduces to ordinary interleaving semantics.

In Section 4.1, we describe the concrete approach of [116, 48, 45], where the *identities* of locations are fully observable. Observability of location names is also the standpoint adopted in most of the recent studies on languages for mobility and migration surveyed in Section 4.3.

In Section 4.2, we report on subsequent work dealing with *location failure* and its detection ([11], [8], [136]). This has been developed for extensions of CCS and the π-calculus, with features for creating processes at remote locations, and for testing and killing locations. Contrarily to the previous line of work, here the emphasis is on location failure rather than distribution, and locations are only observable in case of failure. While the previous approaches assumed a global observer for the whole distributed system, here one adopts the perspective of an individual user, wishing to have transparent access to resources irrespective of where they are located. This is the so-called *network transparency* principle. Awareness of locations is only requested in case of failure: a process that has spawned activities at another location wants to be notified in case of failure of this location, in order to stop interacting with it and transfer activities to other locations. This kind of semantics is particularly appropriate for describing mobile applications in distributed systems, where it may be important to distinguish between a process executed locally and a process spawned at a remote location which is liable to failure.

Finally, in Section 4.3, we discuss the use of locations in connection with *mobility* and *migration* in some more recent languages with explicit distribution like the distributed JOIN-calculus [69, 67], the distributed π-calculi of [137, 89] and [145], the AMBIENT calculus [32], the SEAL calculus [157] and the language Nomadic Pict [146]. These languages address some aspects of large-scale distributed computation in a very explicit way. In particular, most of them abandon the network transparency assumption. For instance, in the AMBIENT calculus the details of routing for migrating processes must be exposed. Similarly, messages are explicitly routed in the distributed π-calculus of [137, 89] and in the SEAL calculus. In general, in these models, processes have to know the locations of other processes and resources in order to be able to interact with them.

To conclude this introduction we could say that, while in the first approaches locations were treated simply as *units of distribution*, in these new languages they acquire additional meanings. For instance, in languages dealing with failure detection they become *units of failure*, in that the failure of a site entails the termination of all processes at that site. They constitute *units of communication* in languages with purely local communication, where processes have to convene in a common location in order to interact. In some languages supporting process migration locations are also *units of mobility*, in that the migration of a process from a location provokes the simultaneous migration of all processes placed at the same location. Finally, locations can be used as *units of security*, in the sense that a resource access request emanating from the location of the resource is not submitted to the same security checks as an external request. As their roles progressively accumulate, locations become more and more an integral part of the semantics of the language.

2 An abstract view of locations

This section reviews the approach to abstract locations as described in [27, 28] and [5, 37]. Generally speaking, this work is concerned with *distributed semantics* for CCS, which account for the spatial distribution of processes among different *sites* or *locations*. Such semantics emerged from a line of research on noninterleaving semantics for process algebras, which started in the early eighties and developed at a steady pace in the subsequent decade. By "noninterleaving semantics" we mean a semantics that does not simulate parallelism by a nondeterministic choice of sequential interleavings. Typically, such reduction of concurrency to interleaving is expressed by Milner's *expansion law* [106], a simple instance of which is:

$$a \mid b \;=\; a.b \;+\; b.a \tag{1}$$

where "|" and "." stand respectively for parallel and sequential composition, "+" for nondeterministic choice and a, b are any (non synchronisable) actions. This law is indeed valid for *bisimulation equivalence*, the standard behavioural equality for the calculus CCS. By contrast, semantic models that reject this equation and insist on treating concurrency as primitive are sometimes qualified as "true-concurrency" or "nonsequential" models. Such models mostly evolved from the early ideas of Petri [125]. Paradigmatic models of this kind are *Petri nets* [126, 124, 134], *event structures* [161, 120, 164] and *Mazurkiewicz traces* [103, 104]. Similar nonsequential models were studied by Shields [149] and Winkowski [160]. To provide some background for the coming material on location semantics, we shall start with a short review of this area of research, focussing on *operational* semantics for CCS. For a more comprehensive account, covering other process algebras like ACP [13] and the Petri Box Calculus, the reader is referred to the other chapters of Part 5 in this volume [16].

2.1 Noninterleaving semantics for process algebras

We give here a brief introduction to noninterleaving operational semantics for process algebras. We concentrate on branching-time semantics based on variations of the notion of *bisimulation*, and will mostly restrict our attention to the calculus CCS.

The search for *operational semantics* for process algebras accounting for the concurrent and causal structure of processes, started roughly around the mid-eighties. Previously, there had been a few proposals for interpreting languages like CCS, CSP and TCSP into nonsequential models for concurrent systems such as *Petri nets* [125, 126, 124] and *event structures* [161, 120, 164]. Among these early proposals, we will only mention those by Winskel [162], De Cindio et al. [44], and Goltz and Mycroft [81]. Many other interpretations of a similar kind were to be given in subsequent years, on variants of these models or on other semantic domains like *asynchronous transition systems* [14], *causal trees* [49], *trace automata* [12] and *transition systems with independence* [166]. For instance, Petri nets have been further used to give semantics for CCS-like languages by Goltz [81, 83, 80], Winskel [163, 165], Van Glabbeek and Vaandrager [78], Nielsen [119], Degano, De Nicola and Montanari [59], Degano, Gorrieri and Marchetti [55], Olderog [121, 122], Taubner [150, 151], Boudol and Castellani [26], Montanari and Gorrieri [84], Montanari and Ferrari [65], and others. Similarly, event structure interpretations have been proposed and studied, among others, by Winskel [163], Goltz and Loogen [102], Degano, De Nicola and Montanari [52], Vaandrager [155], Boudol and Castellani [22, 23, 26], van Glabbeek and Goltz [76, 75], and Langerak [99].

Event-based models like Petri nets and event structures are built from the outset on the primitive notions of *concurrency* and *causality* between events, and therefore naturally provide a noninterleaving semantics for process calculi. The question was then to devise a purely operational semantics for processes, by means of labelled transition systems specified via structural rules (in the style advocated by Plotkin [129], see also Chapter 1.3 in this issue), which could express the same information about concurrency and causality.

A natural idea was to generalise the labels of transitions, from simple actions to whole *partial orders* of actions representing nonsequential computations. The representation of concurrent computations as partial orders, where the ordering stands for causality (and the absence of ordering for concurrency), was already standard in nonsequential models like Petri nets, event structures and Mazurkiewicz traces, where these partial orders were called respectively *abstract processes* [82, 133], *configurations* and *dependency graphs*.

When dealing with process algebras, a simple way to achieve this generalisation of actions is to extract the information about concurrency and causality from the *syntax* of the terms themselves. More precisely, one looks for structural semantic rules by which the constructs of sequential and parallel composition are directly transferred from the terms to the actions labelling their transitions. This idea was successfully applied in [22] to a simple CCS-like language with sequential composition and disjoint parallel composition (not allowing communication): using the notion of *partially ordered multiset* [2], studied by Grabowski [85], Pratt [130, 131] (who coined for it the term "pomset") and Gischer [71], a definition of *pomset transition* was introduced in [22] for this language, with the associated notion of *pomset bisimulation*. This new notion of bisimulation distinguishes the two processes of equation (1) because the first has a pomset transition

$$a \mid b \xrightarrow{a|b} nil \mid nil$$

which the second cannot simulate. The second process has two possibilities for executing a and b in one step, but in each case the two actions are ordered. One of these pomset transitions is

$$a.b \ + \ b.a \xrightarrow{a.b} nil$$

To generalise the pomset semantics to the full language CCS [23], on the other hand, it appeared necessary to use an indirect construction of a pomset transition starting from

[2] Or *pomset*, a generalisation of the notion of partial order, which is necessary to deal with homonymous actions in computations. Formally, pomsets are defined as isomorphism classes of labelled partial orders.

a class of permutation-equivalent transition sequences[3]. This construction "a posteriori" of a partial order from an equivalence class of sequences is similar to the synthesis of a partial order from a Mazurkiewicz trace [104]. The main novelty of [23] was the use of enriched transitions for CCS called *proved transitions* (because they were labelled with a representation of their *proof* in the inference system of CCS), from which the concurrency relation between transitions could be derived by syntactic means. For instance, a sequence of proved transitions for the process $a \mid b$ is

$$a \mid b \xrightarrow{|_0\, a} nil \mid b \xrightarrow{|_1 b} nil \mid nil$$

These two transitions are concurrent because they are extracted from two different sides of the parallel operator (i.e. they are inferred using different rules for $|$), and this information is recorded in their labels. The transitions can therefore be commuted and the pomset corresponding to the equivalence class of this sequence is $a \mid b$. On the other hand the process $a.b \; + \; b.a$ has the sequence of proved transitions

$$a.b \; + \; b.a \xrightarrow{+_0 a} b \xrightarrow{b} nil$$

where the two transitions cannot be commuted, hence the corresponding pomset is $a.b$.

Indeed, the permutation semantics of [23] extends the pomset semantics of [22] and can be applied to more interesting examples. For instance the process $(a.\gamma \mid \overline{\gamma}.b)$, where γ and $\overline{\gamma}$ are two synchronisable actions, can do the sequence of proved transitions

$$a.\gamma \mid \overline{\gamma}.b \xrightarrow{|_0 a} \gamma \mid \overline{\gamma}.b \xrightarrow{(\gamma,\overline{\gamma})} nil \mid b \xrightarrow{|_1 b} nil \mid nil$$

Here the two transitions a and b, although marked as concurrent by their labels, cannot be commuted because they are not adjacent. The pomset generated by this transition sequence is $a.\tau.b$ (where τ represents a generic communication action). Indeed, the communication between γ and $\overline{\gamma}$ creates a *cross-causality* between the actions a and b.

Note that while in the direct pomset semantics of [22] atomic actions are replaced with composite actions representing whole nonsequential computations, in the proved transition semantics actions remain atomic but are decorated with additional information that identifies them uniquely in a term. Pomset transitions are retrieved only in a second step. Indeed, these two proposals by Boudol and Castellani exemplify two different ways of structuring actions: either relax atomicity of actions so as to obtain *compound actions*, or retain portions of their proof in order to pinpoint their identity – we shall speak in this case of *decorated (atomic) actions*.

Another track for defining noninterleaving semantics for process algebras was to structure the *states* – rather than the actions – of the transition system, so as to mark the components responsible for each action. This led first to Degano and Montanari's model of *concurrent histories* [56], concrete computations with structured initial and final states, which could be concatenated while keeping the causality information. From these computations a partial ordering of actions could then be abstracted away. This model was used by Degano, De Nicola and Montanari to define a partial ordering semantics for CCS

[3]This is because the class of pomsets is not closed with respect to parallel composition when communication is allowed.

in [57, 54]. Here the causality relation on actions is recovered by decomposing each CCS process into a set of sequential components called *grapes*, and by causally connecting only those successive actions that originate from the same component. For example the process $a \mid b$ is decomposed into two components $a \mid id$ and $id \mid b$ (here the markers $\mid id$ and $id \mid$ are used to specify the access paths of components inside a term, and therefore to identify them throughout a computation[4]). We have here a computation

$$\{a \mid id, id \mid b\} \xrightarrow{a} \{nil \mid id, id \mid b\} \xrightarrow{b} \{nil \mid id, id \mid nil\}$$

where actions a and b can be seen as emanating from different components and hence are not causally related[5]. On the other hand the process $a.b \ + \ b.a$ has only one component, itself. The computation in this case is

$$\{a.b \ + \ b.a\} \xrightarrow{a} \{b\} \xrightarrow{b} \{nil\}$$

Here actions a and b originate from the same component and thus are causally related. The resulting partial order semantics is the same as that derived from the previously mentioned proved transition systems [23]. In both cases a partial order computation is obtained from a sequential computation together with a relation on transitions (a causality relation here, and a concurrency relation in [23]).

On the other side, the same idea of structuring states led Castellani and Hennessy to the definition of distributed transition systems [36, 38], where transitions have distinct *local* and *concurrent* residuals[6]: that is to say, transitions $p \xrightarrow{a} p'$ are replaced by transitions of the form $p \xrightarrow{a} < p'_\ell, p'_c >$, where p'_ℓ is what locally follows the action a while p'_c is what is concurrent with it. A notion of *distributed bisimulation*, acting separately on the two residuals, was then proposed to compare these systems. For the two processes of our running example, the distributed transitions would be[7]

$$a \mid b \ \xrightarrow{a} \ < nil, b >$$
$$a.b \ + \ b.a \ \xrightarrow{a} \ < b, nil >$$

and the distributed bisimulation would tell them apart since the residuals are not pairwise equivalent. It should be pointed out that no partial order computation was synthesized here. Indeed this particular semantics, which will be reviewed in some detail in Section 3.2, departed somehow from the main line of research at the time, which aimed at defining partial ordering transitions. It was the first attempt at a *distributed semantics*, emphasizing local behaviours rather than the global causal behaviour. Unfortunately this distributed semantics was defined only for a subset of CCS without the restriction operator, and its extension to the whole language, investigated by Kiehn in [95], appeared to be difficult.

[4] Just like the markers $|_0$ and $|_1$ are used to identify occurrences of actions in proved transitions.

[5] This is a slightly simplified rendering of the semantics of [57]. As a matter of fact, also actions were enriched in [57, 54] - with a portion of their proof - but this information was somehow redundant, in particular it was not used to derive the partial order computations.

[6] This was the original formulation reported in [36]. An alternative formulation in terms of *local* and *global* residuals was considered in [38], as it seemed better suited to deal with the full language CCS.

[7] To be precise the first transition should be $a \mid b \ \xrightarrow{a} \ < nil, nil \mid b >$, but we are keeping deliberately informal here.

Indeed, the very idea of splitting terms into parts and observing these parts separately did not seem fully compatible with global operators such as restriction.

The partial ordering approach of [57] was pursued in subsequent works by Degano, De Nicola and Montanari [52, 59], where the grape semantics was adapted in various ways to match more closely event structure and Petri net interpretations for CCS. The same authors presented a different formulation of a partial ordering semantics for CCS in [58], based on a tree-model called Nondeterministic Measurement Systems (NMS). By applying so-called observation functions to NMS's, different equivalence notions could be obtained, among which a partial ordering equivalence called *NMS-equivalence*, which turned out to be unrelated to pomset bisimulation[8].

Another rather original tree-model that was put forward in the late eighties, with the express purpose of reconciling the interleaving and partial order approaches, was Darondeau and Degano's model of *causal trees* [49, 50]. These are a variant of Milner's synchronisation trees, where labels of observable transitions carry an additional element representing the *set of causes* of the corresponding action. More precisely, these labels are of the form (a, K), where K is an encoding - in the form of backward pointers in the tree - of the actions that causally precede a. The bisimulation associated with these new transitions, called *causal bisimulation*, was of course stronger than the usual bisimulation, but also stronger than pomset bisimulation and NMS-equivalence. It may be noted that in causal trees, the causal information is recorded in both the states (nodes) and the actions. In fact, this model illustrates a further way of decorating atomic actions, with parts of their *computational history* rather than parts of their proof.

The initial criterion for these noninterleaving operational semantics for CCS was their agreement with "denotational semantics" given by interpretations into event based semantic domains like event structures and Petri nets. Besides the above-mentioned papers [52, 59], which show the consistency of the grape operational semantics with interpretations of terms into prime event structures and condition/event systems, we can mention also [23, 24], where the pomset semantics is shown to agree with an interpretation of processes into more general event structures, and the work by Olderog [121], where a Petri net semantics for the language CCSP (a mixture of CCS and TCSP) is derived operationally. In [26] the results of [23, 24] are extended to Petri nets and asynchronous transition systems. A correspondence between (a variant of) the proved transition semantics and an interpretation of processes as *trace automata* is given by Badouel and Darondeau in [12]. As it were, the search for denotational interpretations of process algebras gave a new impulse to the study of semantic domains, and led to the definition of specific classes of event structures and nets, such as *bundle event structures* [99] and *flow event structures* [24, 39][9], *flow nets* [25], and Δ-*free event structures* [51]. Following the approach of Winskel [163], many of these denotational models had been meanwhile uniformly formalised in a categorical framework, which facilitated their analysis and comparison: several correspondence results between the various models, including Mazurkiewicz traces and transition systems with independence, could be established in this setting [166][10]. We refer to this work and

[8]Examples showing the incomparability of the two equivalences are given by Goltz and van Glabbeek in [73] and [77].

[9]Which are both weaker variants of Wisnkel's *stable event structures*.

[10]The correspondence in [166], as in most other cases, is given by the isomorphism of the computation domains of the models, while in [26] the correspondence between the three considered models was tighter,

to [143] for a comprehensive survey of such categorical models and their relationships. The unification of models of concurrency into a common algebraic framework had also been a central concern of Ferrari and Montanari's work [63, 65].

In the late eighties, a new criterion for assessing equivalences emerged in connection with the issue of *action refinement*. Here, in accordance with the methodology of stepwise refinement, one wishes to allow for a gradual specification of systems by successive refinements of atomic actions into more complicated behaviours. Then, to be applicable along such top-down design of systems, equivalence notions are required to be preserved under action refinement. This means that two equivalent processes should remain equivalent after replacing all occurrences of an action a by a process $r(a)$. At first, this new criterion appeared to be a further argument in favour of "true concurrency" semantics, since all the existing interleaving semantics failed to satisfy it. For instance, as mentioned earlier, the standard bisimulation equivalence $\sim$ satisfies the expansion law

$$a \mid b \;\sim\; a.b \;+\; b.a$$

On the other hand, if action a is refined into the process $a_1.a_2$, the equality no longer holds

$$a_1.a_2 \mid b \;\not\sim\; a_1.a_2.b \;+\; b.a_1.a_2$$

since after performing a_1 the first process can choose between a_2 and b , while the second cannot. On the basis of this example, the use of partial ordering semantics was advocated by Castellano, de Michelis and Pomello as more appropriate for dealing with action refinement [40]. However, in reaction to this position Goltz and van Glabbeek showed that none of the existing branching-time partial order equivalences was preserved by refinement [73, 74]. In particular pomset bisimulation and NMS equivalence[11] were not invariant under refinement. In conclusion it was proved that a finer equivalence called *history preserving bisimulation* – first proposed by Rabinovitch and Trakhtenbrot [132] under the name *BS-equivalence* (Behaviour Structure equivalence) and then reformulated for event structures by Goltz and van Glabbeek – was indeed preserved by refinement. This result was first proven in the setting of prime labelled event structures for a restricted form of refinement [73], and subsequently extended, for more general forms of refinement, to flow and stable event structures [76, 75, 77], and finally to *configuration structures* [77].

Well before the start of this debate on refinement, Hennessy had proposed in a somewhat confidential publication [86] a semantics for CCS taking into account beginnings and endings of actions, and studied the corresponding bisimulation. This equivalence, later called *split bisimulation*, was reproposed in [87] in a slightly updated form and subsequently studied by Aceto and Hennessy [6, 7] aw well as other authors. It appeared to be closely related to another equivalence, not conceived for true concurrency but for real-time, introduced by van Glabbeek and Vaandrager in [78] and named *ST-bisimulation*.[12] While ST-bisimulation is preserved by refinement [72], split bisimulation is only preserved

amounting to the equality of the families of computations.

[11] Which also coincides with an equivalence proposed by Devillers [61], called weak history preserving bisimulation by Goltz and van Glabbeek [73].

[12] This equivalence is finer than split bisimulation in that it matches action endings with specific beginnings, but it coincides with it on systems without autoconcurrency (i.e. concurrency between actions of the same name).

by refinement on a subset of CCS [6, 79]. Thus ST-bisimulation appeared to be the weakest strengthening of ordinary bisimulation which would be a congruence with respect to refinement. However, since this equivalence is not generally agreed to properly reflect causality (see [77]), it could not be adopted as a representative for partial order semantics.

Meanwhile, refinement had been studied in the setting of Petri nets [158] and causal trees [51][13], and history preserving bisimulation had been shown to coincide with *concurrent bisimulation* [18] on nets, and with causal bisimulation [3, 51] on causal trees. In [3] Aceto also showed the coincidence of causal bisimulation with the *mixed-ordering equivalence* of Degano et al. [53]. These results further consolidated the status of history preserving bisimulation as the best candidate equivalence when both causality and refinement are of concern [77]. It should be pointed out, however, that only the *strong* version of this equivalence was studied, taking into account internal actions as well as visible ones. This is because the weak version of history preserving bisimulation would not be preserved by refinement – as is the case, in fact, for most weak equivalences[14]. For more about history preserving bisimulation and the question of action refinement, we refer the reader to [2, 77] and to Chapter 5.4 in this issue.

Most of the noninterleaving semantics reviewed so far are based on the notion of causality, and can therefore be qualified as *causal semantics*. By contrast, *distributed semantics* focus on the local behaviour of parallel components and aim at capturing local dependencies. As mentioned earlier, the distributed bisimulation semantics of Castellani and Hennessy [36, 38], initially defined for a subset of CCS without the restriction operator, appeared to be hard to extend to the whole language. The problem can be illustrated by a simple example, where the restriction operator $\backslash\gamma$ has the effect of forcing actions γ and $\overline{\gamma}$ to occur simultaneously (and unobservably). Consider the process:

$$p \;=\; (a.\,b.\,\gamma \mid \overline{\gamma}.\,c)\backslash\gamma$$

Suppose action a is performed. Then, what should be its local and concurrent residuals? Clearly, distributing the restriction operator over the two residuals would unduly limit their behaviour: if we let $p = (a.\,b.\,\gamma \mid \overline{\gamma}.\,c)\backslash\gamma \xrightarrow{a} <(b.\,\gamma)\backslash\gamma,\ (\overline{\gamma}.\,c)\backslash\gamma>$, then the second residual would be equivalent to *nil*, and the whole process p would be equivalent to $a.\,b$, which is not what we expect. A variant of this example shows that we could also make unwanted distinctions between processes. Let $q \;=\; (a.\,\gamma.\,b \mid \overline{\gamma})\backslash\gamma$. Then q should be equivalent to $a.\,b$, since in both cases actions a and b occur in sequence at the same location. However a transition $q \;=\; (a.\,\gamma.\,b \mid \overline{\gamma})\backslash\gamma \xrightarrow{a} <(\gamma.\,b)\backslash\gamma,\ \overline{\gamma}\backslash\gamma>$ would yield a local residual equivalent to *nil*, and therefore distinguish q from $a.\,b$.

The other naïve solution would be to "resolve" the restriction by transforming restricted actions into unobservable τ-actions in the residuals. Again, this would lead to unwanted identifications and distinctions. For instance, we would have in this case a transition $p = (a.\,b.\,\gamma \mid \overline{\gamma}.\,c)\backslash\gamma \xrightarrow{a} < b.\,\tau,\ \tau.\,c >$ and p would behave like the process $(a.\,\gamma.\,b \mid \overline{\gamma}.\,c)\backslash\gamma$, where action c can occur before action b. On the other hand, the process $(a \mid \overline{\gamma}.\,b)\backslash\gamma$, which should be equivalent to just a, would be equated with $(a.\,\gamma \mid \overline{\gamma}.\,b)\backslash\gamma$.

[13] In fact, refinement was prevalently studied as a substitution operation on the semantic domains used to interpret terms (*semantic refinement*) rather than on the languages themselves (*syntactic refinement*). Among the above-mentioned papers, only [6] and [7] give a syntactic treatment of refinement. For other references see Chapter 5.4 in this book.

[14] In [7], Aceto and Hennessy studied a weak equivalence which is preserved by refinement. However this is essentially a weak ST-bisimulation equivalence.

In fact, it is not clear how to distribute an encapsulation operator such as restriction whithout losing information about its scope, which is crucial to determine both the local and the overall behaviours. For this reason, subsequent studies on distributed bisimulation tended to keep the residuals in their context (while marking them in some way) rather than observe them in isolation. The alternative formulation in terms of *local* and *global* residuals studied in [38] was a first step in this direction. It did not seem sufficient, however, to overcome the above-mentioned difficulties. Indeed, the later work by Kiehn [95] showed that in the presence of the restriction operator even rather ingenious reformulations of the semantics failed to capture some desired operational distinctions (the later variant considered by De Nicola and Corradini in [47], while getting around Kiehn's counterexample, still suffered from the same drawback).

A few years later, a fresh attack on the problem was resumed with the location semantics of Boudol, Castellani, Hennessy and Kiehn [27, 28]. The idea here was to insert locations into processes and observe actions together with the location at which they occur. In the initial work [27, 28] locations are created *dynamically* for actions when they are executed, and then recorded in the derivative process. The assignment of locations is such that concurrent actions have disjoint locations, while actions from the same component have related locations (one is a sublocation of the other). For instance the process $a \mid b$ has a sequence of transitions

$$a \mid b \quad \xrightarrow[\ell]{a} \quad \ell :: nil \mid b \quad \xrightarrow[k]{b} \quad \ell :: nil \mid k :: nil$$

where actions a and b occur at different locations. On the other hand the process $a.b + b.a$ has transitions

$$a.b \,+\, b.a \quad \xrightarrow[\ell]{a} \quad \ell :: b \quad \xrightarrow[\ell k]{b} \quad l :: k :: nil$$

where the location of b is a *sublocation* of that of a. Note that the idea underlying this semantics is a natural generalisation of that of splitting residuals: every component which becomes active is marked with a location which will distinguish it from any other component running in parallel.

Based on this new transition system, notions of *location equivalence* $\approx_\ell$ and *location preorder* $\sqsubseteq\!\!\!\!\!\sim_\ell$ were introduced. The equivalence $\approx_\ell$ formalises the idea that two processes are bisimilar, in the classical sense, and moreover are equally distributed and have the same local behaviours in each computationn. Formally, $\approx_\ell$ is just the ordinary bisimulation associated with the new transition system. It can be easily seen, looking at the transitions above, that

$$a \mid b \quad \not\approx_\ell \quad a.b + b.a$$

This is expected since the first process is distributed among two different locations while the second is confined within one location. The location preorder $\sqsubseteq\!\!\!\!\!\sim_\ell$, on the other hand, relates a process with another which is bisimilar but possibly more distributed. Technically, the locations of the second process are required to be *subwords* of the locations of the first. For example

$$a.b \,+\, b.a \quad \sqsubseteq\!\!\!\!\!\sim_\ell \quad a \mid b$$

Again, this is expected because the first process is less distributed than the second.

In two subsequent papers, Aceto [5] and Castellani [37] proposed an alternative approach where locations are assigned *statically* to parallel components, and observed modulo a concurrency-preserving renaming. This approach, although closer to the intuition, had been initially abandoned in favour of the dynamic one because of some technical difficulties it presented. In the static approach, processes are given an explicit distribution before they are executed, by inserting locations in front of parallel components. For instance, two *distributions* for the processes $a \mid b$ and $a.b \,+\, b.a$ would be

$$\ell :: a \mid k :: b \quad \text{and} \quad a.\, b \,+\, b.\, a$$

where ℓ and k are distinct locations, and processes with no explicit location are supposed to be placed at the location ε of the overall system. The semantics then simply exhibits the locations along transitions. For instance

$$\begin{array}{l} \ell :: a \mid k :: b \;\xrightarrow[\ell]{a}\; \ell :: nil \mid k :: b \;\xrightarrow[k]{b}\; \ell :: nil \mid k :: nil \\ a.b \,+\, b.a \;\xrightarrow[\varepsilon]{a}\; b \;\xrightarrow[\varepsilon]{b}\; nil \end{array}$$

Notions of location equivalence and preorder are now redefined in this setting. An immediate observation is that one cannot require here equality of locations. Otherwise one would distinguish processes like $a \mid (b \mid c)$ and $(a \mid b) \mid c$, whose distributions are of the form

$$\ell_1 :: a \mid k_1 :: (\ell'_1 :: b \mid k'_1 :: c) \quad \text{and} \quad \ell_2 :: (\ell'_2 :: a \mid k'_2 :: b) \mid k_2 :: c$$

The idea is then to compare transitions modulo particular *associations* of their locations. For this example the required association would be: $\{(\ell_1, \ell_2\ell'_2), (k_1\, \ell'_1, \ell_2\, k'_2), (k_1\, k'_1, k_2)\}$.

Interestingly the two semantics, dynamic and static, yield the same notions of location equivalence and preorder, in spite of the fact that locations reflect different intuitions in the two cases. In the static case locations represent *sites*, while in the dynamic case they represent *local causes* for an action. Indeed, in the static case locations may be seen as part of the proof of a transition, in the sense explained earlier, while in the dynamic case they are part of its computational history.

We shall not enter in further details here, since the location semantics will be presented in detail in the next section. Let us just mention that this semantics consistently extends the distributed bisimulation semantics of [38], while being in general incomparable with the causality-based semantics discussed earlier[15]. It therefore offers an interesting complementary point of view about observing parallelism in processes.

Semantics based on static locations were investigated also by other authors. In [115] an equivalence analogous to that of [37] was studied by Mukund and Nielsen for a class of asynchronous transition systems modelling a subset of CCS with guarded sums. It was conjectured to coincide with the dynamic location equivalence of [28], a result that could later be subsumed from those of [37]. A transition system for CCS labelled with static locations, called *spatial transition system*, was considered by Montanari and Yankelevich in [167, 114]. In this latter work, however, location transitions are only used as a first step to build a second transition system, labelled by *local mixed partial orders* (where the ordering is a mixture of temporal ordering and local causality), which is then used

[15]Except on the finite fragment of CCS without communication, where the location semantics coincides with the causal semantics of [49] and [73].

to define the behavioural equivalence. This equivalence, a variant of the mixed-ordering equivalence of [53] called *local mixed-ordering equivalence*, was shown to coincide with location equivalence [167]. The view of location equivalence as capturing local causality was further substantiated by the work of Kiehn [97], who characterised it as a variant of causal bisimulation called *local cause bisimulation*. In the same spirit, in [37] location equivalence was presented as a *local history preserving bisimulation*. The first characterisation was carried out within Kiehn's local/global cause transition system, an enriched transition system which unifies distributed and causal semantics into a single framework, thus shedding light on their relationship. In afterthought, the convergence of these three new formulations of location equivalence was somehow expected since their "global" counterparts (mixed-ordering equivalence, causal bisimulation and history preserving bisimulation) were known to coincide from Aceto's work [3].

A more concrete view of localities was proposed by D. Murphy in [116] (and later extended to unobservable transitions by Corradini and Murphy in [48]), for a subset of CCS with only top-level parallelism. Here again localities are assigned statically to parallel components. The main difference with respect to the other static approaches is that the names of localities are significant here, and processes are considered equivalent only if they reside on the same set of localities and present the same behaviour at each locality. The interest here is centered on implementing a system on a *fixed network of processors*, while in the previous cases the notions of local behaviour and degree of distribution where expressed somewhat abstractly. For instance the two distributed processes

$$(\ell :: a \mid k :: b) \quad \text{and} \quad (\ell :: b \mid k :: a)$$

are distinguished in Murphy's semantics, while they are equated in the other static location semantics.

More recently, Corradini and De Nicola have reformulated in [47] both the dynamic location equivalence and a variant of Kiehn's extended distributed bisimulation [95] within the grape semantics of [54]. Kiehn's extended distributed bisimulation was known to be weaker than location equivalence from [27] and [28]. While exploiting some of Kiehn's ideas, Corradini and De Nicola propose here a slightly stronger equivalence, which however still fails to recover the full power of location equivalence.

We conclude our review of noninterleaving semantics with a remark about structured actions and states. The idea of decorating atomic actions with (portions of) their proof to obtain more expressive semantics for CCS culminated on one side in the above-mentioned *proved transition systems* [23, 26], and on the other side in Montanari and Ferrari's *structured transition systems* [63, 65, 66]. In the first model only actions are structured while in the second both states and transitions are structured[16]. Both these transition systems retain in some sense a maximal information about the structure of the underlying terms, which can be weakened in various ways to obtain specialised semantics like pomset or distributed semantics. For instance the structured transitions of [65] are interpreted through a mapping into an algebra of observations, and the associated notion of bisimulation is

[16]In [26] a generalisation of the proved transition system called *event transition system* is considered, where also states are structured in order to represent partially executed terms: like the causal trees of Darondeau and Degano and Kiehn's local/global cause transition system, these systems incorporate information about the *computational history* of processes, and this information needs to be recorded in the states in order to be correctly propagated along the transitions.

parameterized with respect to the choice of the algebra. Similarly, proved transitions were exploited by Degano and Priami in [60] to obtain different semantics for CCS. Here, in order to capture the causal bisimulation semantics of [49], the authors use a simpler model based on derivation trees – called *proved trees* – rather than the more general proved transition systems discussed earlier.

2.2 Motivation for location semantics

In this section we introduce the abstract location semantics proposed in [27, 28] and [5, 37], by means of simple examples. These will be described using CCS notation (formally introduced only in the next section), which should be accessible to the unfamiliar reader with the help of comments and pictures.

The distributed structure of CCS processes can be made explicit by assigning different *locations* to their parallel components. This can be done using a *location prefixing* construct $l :: p$, which represents process p residing at a location l. The actions of such a process are observed together with their location. We have for instance

$$(l :: a \mid k :: b) \xrightarrow[l]{a} (l :: nil \mid k :: b) \xrightarrow[k]{b} (l :: nil \mid k :: nil)$$

In CCS, parallelism is not restricted to top-level. It may also appear after the execution of some actions, in which case it is best viewed as a *fork* operation, a future activation of two processes in parallel. Then the locations of actions will not be simple letters $l, k, \ldots$ but rather words $u = l_1 \cdots l_n$, and a "distributed process" will perform transitions of the form $p \xrightarrow[u]{a} p'$. For instance, if the process $(l :: a \mid k :: b)$ is a component of a larger process, we will have

$$l' :: c.\,(l :: a \mid k :: b) \mid k' :: d \;\xrightarrow[l']{c}\; l' :: (l :: a \mid k :: b) \mid k' :: d \;\xrightarrow[l'\,l]{a}\; l' :: (l :: nil \mid k :: b) \mid k' :: d$$

Let us consider a more concrete example, taken from [27]. We may describe in CCS a simple protocol, transferring data one at a time from one port to another, as follows:

$$\begin{array}{rcl} Protocol & \Leftarrow & (Sender \mid Receiver)\backslash \alpha, \beta \\ Sender & \Leftarrow & in.\,\overline{\alpha}.\,\beta.\,Sender \\ Receiver & \Leftarrow & \alpha.\,out.\,\overline{\beta}.\,Receiver \end{array}$$

where $\overline{\alpha}$ represents transmission of a message from the sender to the receiver, and $\overline{\beta}$ is an acknowledgement from the receiver to the sender, signalling that the last message has been processed. In the standard theory of *weak bisimulation equivalence*, usually noted $\approx$, one may prove that this system is equivalent to the following specification

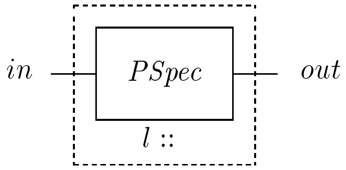
$$PSpec \;\Leftarrow\; in.\,out.\,PSpec$$

That is to say, $PSpec \approx Protocol$. The reader familiar with the (weak version of) *causal bisimulation* $\approx_c$ of [49][17], which is a strengthening of $\approx$, should also be easily convinced that $PSpec \approx_c Protocol$: intuitively, this is because the synchronizations on α, β in $Protocol$

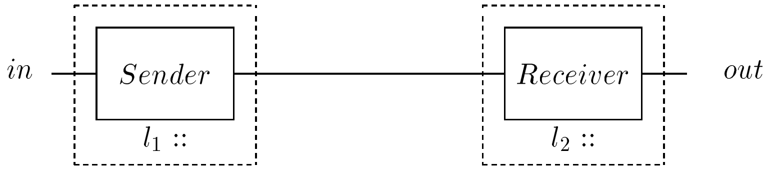
[17] This equivalence will be reviewed in Section 3.3.

create “cross-causalities” between the visible actions *in* and *out*, constraining them to happen alternately in sequence.

In a distributed view, on the other hand, it would be reasonable to distinguish these two systems, because *PSpec* is completely sequential and thus performs the actions *in* and *out* at the same location l, what can be represented graphically as follows



while *Protocol* is a system distributed among two different localities l_1 and l_2, with actions *in* and *out* occurring at l_1 and l_2 respectively. Thus *Protocol* may be represented as

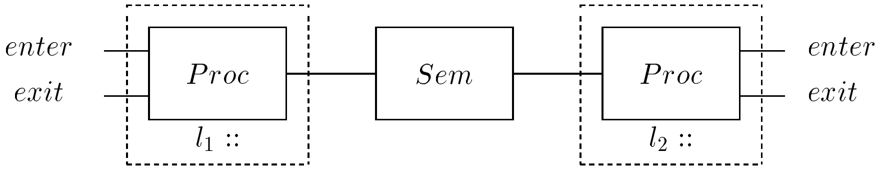


Here the unnamed link represents the communication lines α, β, which are private to the system. We shall see that *PSpec* and *Protocol* will not be equated in the location semantics. On the other hand they will be related by a weaker relation, a *preorder* that orders processes according to their degree of distribution.

Let us consider another example, taken from [28], which describes the solution to a simple mutual exclusion problem. In the system *Mutex*, two processes compete for a device, and a semaphore is used to serialize their accesses to this device

$$\begin{array}{rcl} Mutex & \Leftarrow & (Proc \mid Sem \mid Proc)\backslash\{p, v\} \\ Proc & \Leftarrow & \overline{p}.\, enter.\, exit.\, \overline{v}.\, Proc \\ Sem & \Leftarrow & p.\, v.\, Sem \end{array}$$

As a distributed system, *Mutex* can be pictured as follows, where l_1 and l_2 are again two distinct locations (the location of the semaphore is left implicit, since all its actions are unobservable)



Take now a variant of the system *Mutex*, where one of the processes is faulty and may deadlock after exiting the critical region (the deadlocked behaviour being modelled here as *nil*). This system, *FMutex*, may be defined by:

$$\begin{array}{rcl} \mathit{FMutex} & \Leftarrow & (\mathit{Proc} \mid \mathit{Sem} \mid \mathit{FProc})\backslash\{p, v\} \\ \mathit{FProc} & \Leftarrow & \overline{p}.\, \mathit{enter}.\, \mathit{exit}.\, (\overline{v}.\, \mathit{FProc} + \overline{v}.\, \mathit{nil}) \end{array}$$

where *Proc* and *Sem* are defined as above. In the standard theory of weak bisimulation, the two systems *Mutex* and *FMutex* are equivalent. Indeed, they are both observationally equivalent to the sequential specification:

$$\mathit{MSpec} \Leftarrow \mathit{enter}.\, \mathit{exit}.\, \mathit{MSpec}$$

Again, we could argue that these systems should be distinguished in a distributed semantics. Note that *FMutex* has a distributed representation similar to that of *Mutex*, with *FProc* replacing the second occurrence of *Proc*. Then *Mutex* and *FMutex* should have different behaviours, because *FMutex* may reach a state in which no more actions can occur at location l_2, while this is not possible for *Mutex*. In other words, there is a *local deadlock* at location l_2 in *FMutex*, which has no counterpart in *Mutex*. Moreover, both systems should be distinguished from the specification *MSpec*, because both have activities at distinct locations, while *MSpec* is confined to a single location. Note that a causal semantics would not help in this case either: we have $\mathit{Mutex} \approx_c \mathit{MSpec} \approx_c \mathit{FMutex}$ because the synchronizations on p, v in *Mutex* and *FMutex* create cross-causalities between the critical regions of the two competing processes.

We shall now proceed to a formalisation of these intuitions. In the next two sections we present the static approach to location semantics, as has been proposed in [5, 37].

2.3 A Language for Processes with Locations

We introduce here a language for specifying processes with locations, called LCCS. This language is a simple extension of CCS, including a new construct to deal with locations.

We start by recalling some conventions of CCS [106, 108]. One assumes a set of names Δ, ranged over by $\alpha, \beta, \ldots$, and a corresponding set of co-names $\overline{\Delta} = \{\overline{\alpha} \mid \alpha \in \Delta\}$, where $\bar{\ }$ is a bijection such that $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Delta$. The set of visible actions is given by $Act = \Delta \cup \overline{\Delta}$. Invisible actions – representing internal communications – are denoted by the symbol $\tau \notin Act$. The set of all actions is then $Act_\tau =_{\mathrm{def}} Act \cup \{\tau\}$. We use $a, b, c, \ldots$ to range over Act and $\mu, \nu, \ldots$ to range over Act_τ. We also assume a set V of process variables, ranged over by $x, y \ldots$.

In addition to the operators of CCS, the language LCCS includes a construct for building processes with explicit locations. Let *Loc*, ranged over by $l, k, \ldots$, be an infinite set of atomic locations. The new construct of *location prefixing*, noted $l :: p$, is used to represent process p residing at location l. Intuitively, the actions of such a process will be observed to occur "within location l". The syntax of LCCS is as follows:

$$p ::= \mathit{nil} \mid \mu.\, p \mid (p \mid q) \mid (p + q) \mid p\backslash\alpha \mid p\,\langle f\rangle \mid x \mid \mathit{rec}\, x.\, p \quad \mid \quad l :: p$$

Apart from $l :: p$, all the other constructs are borrowed from CCS. They are: the empty process *nil*, action prefixing $\mu.\, p$ (a simplified form of sequential composition), parallel composition $p \mid q$, nondeterministic choice $p + q$, restriction $p \backslash \alpha$, relabelling $p\,\langle f \rangle$[18] and recursion $rec\,x.\ p$. The intuitive meaning of these operators is as follows: nil represents the terminated process; $\mu.\,p$ is a process that performs action μ and then behaves like p; $(p \mid q)$ represents the concurrent execution of p and q, with possible communication; $(p+q)$ behaves like any of p and q, discarding the other; $p \backslash \alpha$ behaves like p but with the use of actions $\alpha, \overline{\alpha}$ restricted to internal communications; $p\,\langle f \rangle$ behaves like p with all its actions relabelled by f, where $f : Act_\tau \rightarrow Act_\tau$ is a function preserving complementation and τ-actions. Finally, $rec\,x.\ p$ denotes a distinguished solution of the recursive equation $x = p$. As usual, we shall omit trailing occurrences of nil in terms, writing for instance $a.\,b$ instead of $a.\,b.\,nil$.

The basic idea of the semantics is that the actions of processes are observed together with the locations at which they occur. In general, because of the nesting of parallelism and prefixing in terms, the locations of actions will not be atomic locations of Loc, but rather *words* over these locations. Thus general locations will be elements $u, v \ldots$ of Loc^*, and processes will be interpreted as performing transitions $p \xrightarrow[u]{\mu} p'$, where μ is an action and u is the location at which it occurs.

2.4 Static Approach

We start by presenting an operational semantics for LCCS based on the static notion of location, as described in [3, 37]. The idea of this semantics is very simple. Processes of LCCS have some components of the form $l :: p$, and the actions arising from these components are observed together with their location. The distribution of locations in a term remains fixed through execution. Location prefixing is a static construct and the operational rules do not create new locations; they simply exhibit the locations which are already present in terms. Formally, this is expressed by the operational rules for action prefixing and location prefixing. Recall that locations are words $u, v, \ldots \in Loc^*$. The empty word ε represents the location of the overall system. The rules for $\mu.\,p$ and $l :: p$ are respectively:

$$\text{(S1)} \qquad \mu.\,p \xrightarrow[\varepsilon]{\mu}_s p$$

$$\text{(S2)} \qquad p \xrightarrow[u]{\mu}_s p' \qquad \Rightarrow \qquad l :: p \xrightarrow[l\,u]{\mu}_s l :: p'$$

Rule (S1) says that an action which is not in the scope of any location l is observed as a global action of the system. Rule (S2) shows how locations are transferred from processes to actions. The rules for the remaining operators, apart from the communication rule, are similar to the standard interleaving rules for CCS, with transitions $\xrightarrow[u]{\mu}_s$ replacing the usual transitions $\xrightarrow{\mu}$.

The set of rules specifying the operational semantics of LCCS is given in Figure 1[19]. The rule for communication (S4) requires some explanation. In the strong location transition system defined here (where no abstraction is made from τ-transitions), we take the

[18]Here we adopt the slightly nonstandard notation $p\,\langle f \rangle$ for the renaming operator of CCS, as the notation $p[f]$ will be used for renaming of locations later on.

[19]For convenience we include in the same page the rules for the dynamic transition system, which will only come into play in Section 2.5. These rules (given in Figure 2) can be ignored for the time being.

location of a communication to be that of the smallest component which includes the two communicating subprocesses: the notation $u \sqcap v$ in rule (S4) stands for the longest common prefix of u and v. For instance we have:

Example 2.1 $$\begin{array}{rl} l :: \alpha \mid k :: \bar{\alpha}.\, (l' :: \beta \mid k' :: \bar{\beta}) & \xrightarrow[\varepsilon]{\tau}_s \; l :: nil \mid k :: (l' :: \beta \mid k' :: \bar{\beta}) \\ & \xrightarrow[k]{\tau}_s \; l :: nil \mid k :: (l' :: nil \mid k' :: nil) \end{array}$$

There are other possibilities for defining locations of τ-actions representing communication. In [113] the location of such an action is the pair of locations of the contributing actions. In the above example, the resulting location would be (l, k) for the first communication, and (kl', kk') for the second. In [27, 28], τ-actions have no location at all. In fact, these differences are not important since in all these studies the interest is centered on the *weak location transition system*, where the locations of τ-actions are simply ignored (since the actions themselves are not observable).

The weak location transitions $\overset{a}{\underset{u}{\Longrightarrow}}_s$ and $\overset{\tau}{\Longrightarrow}_s$ are defined by:

$$\begin{array}{lcl} p \overset{\tau}{\Longrightarrow}_s q & \Leftrightarrow & \exists\, u_1, \ldots, u_n,\, p_0, \ldots, p_n,\; n \geq 0 \quad s.t. \quad p = p_0 \xrightarrow[u_1]{\tau}_s p_1 \cdots \xrightarrow[u_n]{\tau}_s p_n = q \\ p \overset{a}{\underset{u}{\Longrightarrow}}_s q & \Leftrightarrow & \exists\, p_1, p_2 \quad s.t. \quad p \overset{\tau}{\Longrightarrow}_s p_1 \xrightarrow[u]{a}_s p_2 \overset{\tau}{\Longrightarrow}_s q \end{array}$$

The weak location transition system will be the basis for defining notions of *location equivalence* and *location preorder* accounting for the distributed behaviour of CCS processes.

The reader may have noticed, however, that applying the rules of Figure 1 to CCS terms just yields a transition $p \xrightarrow[\varepsilon]{\mu}_s p'$ whenever the standard semantics yields a transition $p \xrightarrow{\mu} p'$. In fact, these rules will not be applied directly to CCS terms. Instead, the idea is to first bring out the parallel structure of CCS terms by assigning locations to their parallel components, thus transforming them into particular LCCS terms called "distributed processes", and then execute these according to the given operational rules.

The set DIS $\subseteq$ LCCS of *distributed processes* is given by the grammar:

$$p \;::=\; nil \;\mid\; \mu.p \;\mid\; \underbrace{(l :: p \mid k :: q)}_{l \neq k} \;\mid\; (p + q) \;\mid\; p\backslash\alpha \;\mid\; p\,\langle f \rangle \;\mid\; x \;\mid\; rec\,x.\, p$$

Essentially, a distributed process is obtained by inserting a pair of distinct locations in a CCS term wherever there occurs a parallel operator. This is formalised by the following notion of *distribution*.

Definition 2.2 The distribution relation is the least relation $\mathcal{D} \subseteq$ (CCS$\times$DIS) satisfying:

- $nil \;\mathcal{D}\; nil$ and $x \,\mathcal{D}\, x$
- $p \,\mathcal{D}\, r \;\Rightarrow\; \begin{array}{l} \mu.p \;\mathcal{D}\; \mu.r \\ p\backslash\alpha \;\mathcal{D}\; r\backslash\alpha \\ p\,\langle f \rangle \;\mathcal{D}\; r\,\langle f \rangle \\ (rec\,x.\; p) \;\mathcal{D}\; (rec\,x.\; r) \end{array}$

For each $\mu \in Act_\tau$, $u \in Loc^*$, let $\xrightarrow[u]{\mu}_s$ be the least relation $\xrightarrow[u]{\mu}$ on LCCS processes satisfying the following axiom and rules.

(S1)	$\mu . p \xrightarrow[\varepsilon]{\mu} p$		
(S2)	$p \xrightarrow[u]{\mu} p'$	$\Rightarrow$	$l :: p \xrightarrow[lu]{\mu} l :: p'$
(S3)	$p \xrightarrow[u]{\mu} p'$	$\Rightarrow$	$p \mid q \xrightarrow[u]{\mu} p' \mid q$ $q \mid p \xrightarrow[u]{\mu} q \mid p'$
(S4)	$p \xrightarrow[u]{\alpha} p', \quad q \xrightarrow[v]{\bar{\alpha}} q'$	$\Rightarrow$	$p \mid q \xrightarrow[u \sqcap v]{\tau} p' \mid q'$
(S5)	$p \xrightarrow[u]{\mu} p'$	$\Rightarrow$	$p + q \xrightarrow[u]{\mu} p'$ $q + p \xrightarrow[u]{\mu} p'$
(S6)	$p \xrightarrow[u]{\mu} p', \quad \mu \notin \{\alpha, \bar{\alpha}\}$	$\Rightarrow$	$p \backslash \alpha \xrightarrow[u]{\mu} p' \backslash \alpha$
(S7)	$p \xrightarrow[u]{\mu} p'$	$\Rightarrow$	$p \langle f \rangle \xrightarrow[u]{f(\mu)} p' \langle f \rangle$
(S8)	$p[rec\, x.\ p/x] \xrightarrow[u]{\mu} p'$	$\Rightarrow$	$rec\, x.\ p \xrightarrow[u]{\mu} p'$

Figure 1: Static location transitions

Let $p \xrightarrow[u]{\tau}_d q \ \Leftrightarrow_{\mathrm{def}}\ p \xrightarrow[u]{\tau}_s q$, and for each $a \in Act$, $u \in Loc^*$, let $\xrightarrow[u]{a}_d$ be the least relation $\xrightarrow[u]{a}$ on LCCS processes satisfying rules (S2), (S3), (S5), (S6), (S7), (S8) and the axiom:

(D1) $\quad a.p \xrightarrow[l]{a} l :: p \quad$ for any $l \in Loc$

Figure 2: Dynamic location transitions

- $p\,\mathcal{D}\,r \;\&\; q\,\mathcal{D}\,s \;\Rightarrow\; \begin{array}{l}(p \mid q)\;\mathcal{D}\;(l :: r \mid k :: s)\,, \quad \forall\, l,k \;\text{ s.t. }\; l \neq k \\ (p+q)\;\mathcal{D}\;(r+s)\end{array}$

If $p\,\mathcal{D}\,r$ we say that r is a *distribution* of p.

Note that the same pair of locations may be used more than once in a distribution. We shall see in fact, at the end of this section, that distributions involving just *two atomic locations* are sufficient for describing the distributed behaviour of CCS processes in the static approach. This will not be the case for the dynamic approach of Section 2.5.

2.4.1 Static location equivalence

We shall now introduce an equivalence relation $\approx_\ell^s$ on CCS processes, which is based on a bisimulation-like relation on their distributions. The intuition for two CCS processes p, q to be equivalent is that there exist two distributions of them, say $\bar{p}$ and $\bar{q}$, which perform "the same" location transitions at each step.

However, we shall not go as far as to observe the concrete names of locations: the intention here is that locations should serve to reveal the degree of distribution of processes rather than to specify a placing of processes into a given network. Their precise identities should therefore not be significant. In fact, even if we want to observe distribution, we still aim, to some extent, at an extensional semantics. For instance, we do not want to observe the order in which parallel components have been assembled in a system, nor indeed the number of these components. We are only interested in the number of *active* components in each computation. Then we cannot require the identity of locations in corresponding transitions. For instance, if we want to identify the following processes:

(2) $$a \mid (b \mid c) \qquad \text{and} \qquad (a \mid b) \mid c$$

(3) $$a \qquad \text{and} \qquad a \mid nil$$

it is clear that, whatever distributions we choose, we must allow corresponding transitions to have different – although somehow related – static locations.

The idea is to compare transitions modulo an *association* between their locations. This was first proposed by Aceto for a subset of CCS called *nets of automata* [5], where parallelism is restricted to top-level, and then generalised by Castellani to full CCS in [37].

In the case of nets of automata the structure of locations is flat and the association is simply a partial bijection between the sets of locations of the two processes. For general CCS processes the association will not be a bijection, nor even a function. For instance, in order to equate the two processes (which should be both equivalent to $a.\,b.\,c$):

(4) $$a.\,(b.\,c \mid nil) \quad \text{and} \quad a.\,b.\,(c \mid nil)$$

we need an association containing the three pairs $(\varepsilon,\varepsilon), (l,\varepsilon), (l,l')$, for some $l, l' \in Loc$.

In fact, the only property we will require of location associations is that they respect independence of locations. To make this precise, let $\ll$ denote the prefix ordering on Loc^*. If $u \ll v$ we say that v is an extension or a *sublocation* of u. If $u \not\ll v$ and $v \not\ll u$, what we indicate by $u \diamond v$, we say that u and v are *independent*. Intuitively, transitions originating from different parallel components will have independent locations. On the other hand, transitions performed in sequence by the same component (like a and b in the two processes of example (4)) will have dependent locations: the location of the second transition will be a sublocation of the location of the first. We shall sometimes by extension talk of independent or dependent transitions.

We introduce now the notion of *consistent location association*, as defined in [37]. This is meant to express the requirement that independent transitions from one process be matched by independent transitions from the other process.

Definition 2.3 A relation $\varphi \subseteq (Loc^* \times Loc^*)$ is a *consistent location association* (*cla*) if:

$$(u, v) \in \varphi \quad \& \quad (u', v') \in \varphi \qquad \Rightarrow \qquad (u \diamond u' \;\Leftrightarrow\; v \diamond v')$$

The following properties of *cla's* are immediate to check:

Property 2.4 (Properties of cla's)

1. *If φ is a cla, then φ^{-1} is a cla.*
2. *If φ is a cla and $\psi \subseteq \varphi$, then ψ is a cla.*
3. *If φ and ψ are cla's, then $\varphi \circ \psi$ is a cla.*

We will see now that, for a given pair of distributed processes we want to equate, the required *cla* cannot in general be fixed statically, but has to be built incrementally. We will show that this is necessary to ensure such basic identifications as (2) and (3) above.

For consider the two processes, which should certainly be equated if we want parallel composition to absorb the *nil* process, and + to be idempotent, which is the very essence of a bisimulation semantics:

(5) $$(a \mid b) \qquad \text{and} \qquad (a \mid b) \;+\; (\, nil \mid (a \mid b)\,)$$

Any two distributions of these processes will have the following form, where $l \neq k$ and $l_i \neq k_i$ for $i = 1, 2, 3$:

$$(l :: a \mid k :: b) \qquad \text{and} \qquad (l_1 :: a \mid k_1 :: b) \;+\; (\, l_2 :: nil \mid k_2 :: (l_3 :: a \mid k_3 :: b)\,)$$

Suppose we want to equate these two distributions using a fixed location association φ. Clearly we will require φ to contain the pairs (l, l_1), $(k, k_1), (l, k_2 l_3), (k, k_2 k_3)$. But then, if φ is to be consistent, k_2 must be such that $l_1 \ll k_2 l_3$ and $k_1 \ll k_2 k_3$: however this would imply $l_1 = k_2$ and $k_1 = k_2$, contradicting the hypothesis $l_1 \neq k_1$.

Similarly, if we want parallel composition to be associative, we should identify the processes:

(6) $$a \mid (b \mid c) \qquad \text{and} \qquad a \mid (b \mid c) \;+\; (a \mid b) \mid c$$

which have the following generic distributions, where $l \neq k$ and $l_i \neq k_i$ for $i = 1, 2, 3, 4, 5$:

$$(\, l :: a \mid k :: (l_1 :: b \mid k_1 :: c)\,)$$

$$\text{and} \quad (\, l_2 :: a \mid k_2 :: (l_3 :: b \mid k_3 :: c)\,) \;+\; (\, l_4 :: (l_5 :: a \mid k_5 :: b) \mid k_4 :: c\,)$$

Here a global location association would have to contain the pairs (l, l_2), (kl_1, k_2l_3), (kk_1, k_2k_3), (l, l_4l_5), (kl_1, l_4k_5), (kk_1, k_4). But this would imply $l_2 = l_4$ and $k_2 = l_4$, whence the contradiction $l_2 = k_2$.

These examples show that, at least if we want to retain a simple notion of location[20], the location association cannot be fixed once and for all, but has to be built dynamically along computations. In example (5), depending on which summand is chosen in the second process, one will use the association $\varphi_1 = \{(l, l_1), (k, k_1)\}$ or the association $\varphi_2 = \{(l, k_2l_3), (k, k_2k_3)\}$. In example (6), one can choose $\varphi_1 = \{(l, l_2), (kl_1, k_2l_3), (kk_1, k_2k_3)\}$ for one computation, and $\varphi_2 = \{(l, l_4l_5), (kl_1, l_4k_5), (kk_1, k_4)\}$ for the other. Note incidentally that these situations are not due to the phenomenon known as "symmetric confusion" in Petri nets, and exemplified in CCS by the term $(p \mid q) + r$. Similar examples could be obtained in CCS with guarded sums (the language considered e.g. in [115]), by just prefixing all summands in the examples above by an action d.

Let us see now how these location associations can be built up dynamically. Let Φ be the set of consistent location associations. We define particular Φ-indexed families of relations S_φ over distributed processes, called *progressive bisimulation families* (although the relations that constitute a family are not themselves bisimulations). The idea is to start with the *empty* association of locations and extend it consistently as the execution proceeds.

Definition 2.5 A *progressive bisimulation family (pbf)* is a Φ-indexed family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ of relations over DIS such that, if $pS_\varphi q$ then for all $a \in Act, u \in Loc^*$:

(1) $p \overset{a}{\underset{u}{\Longrightarrow}}_s p' \;\Rightarrow\; \exists\, q', v$ such that $q \overset{a}{\underset{v}{\Longrightarrow}}_s q'$, $\varphi \cup \{(u, v)\} \in \Phi$ and $p'\, S_{\varphi \cup \{(u,v)\}}\, q'$

(2) $q \overset{a}{\underset{v}{\Longrightarrow}}_s q' \;\Rightarrow\; \exists\, p', u$ such that $p \overset{a}{\underset{u}{\Longrightarrow}}_s p'$, $\varphi \cup \{(u, v)\} \in \Phi$ and $p'\, S_{\varphi \cup \{(u,v)\}}\, q'$

(3) $p \overset{\tau}{\Longrightarrow}_s p' \;\Rightarrow\; \exists\, q'$ such that $q \overset{\tau}{\Longrightarrow}_s q'$ and $p' S_\varphi q'$

(4) $q \overset{\tau}{\Longrightarrow}_s q' \;\Rightarrow\; \exists\, p'$ such that $p \overset{\tau}{\Longrightarrow}_s p'$ and $p' S_\varphi q'$

We may now define the *location equivalence* $\approx_\ell^s$ on CCS terms as follows

Definition 2.6 (Static location equivalence) For $p, q \in$ CCS, let $p \approx_\ell^s q$ if and only if for some $\bar{p}, \bar{q} \in$ DIS such that $p\, \mathcal{D}\, \bar{p}$ and $q\, \mathcal{D}\, \bar{q}$, there exists a progressive bisimulation family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ such that $\bar{p}\, S_\emptyset\, \bar{q}$.

Let us prove that $\approx_\ell^s$ is indeed an equivalence relation. The reader may have noticed that the inverse $\mathcal{D}^{-1}$ of the distribution relation is a function. If we let $\pi =_{\mathrm{def}} \mathcal{D}^{-1}$, then $\pi(p)$ is the pure CCS process underlying the distributed process p. We start by showing that all distributions of the same process are in the relation $S_\emptyset$ for some progressive bisimulation family $\mathbf{S}$.

[20] One could envisage a more structured notion of location, incorporating information about the + operator; this would mark as *conflicting* (another form of dependency) the locations of the first and second summand in the right-hand process of examples (5) and (6). Then, identical locations would be allowed to be matched by conflicting locations and the counterexamples (5) and (6) would not hold anymore. In fact, this would bring us close to the notion of *proved transition* considered in [26], and would certainly complicate the theory as a new relation of conflict would have to be considered on locations.

Proposition 2.7 *Let $p_1, p_2 \in$ DIS. Then* $\pi(p_1) = \pi(p_2) \;\Rightarrow\; \exists\, pbf$ **S** *s.t.* $p_1 S_{\emptyset} p_2$.

The proof of this proposition relies on the following definition and lemma.

Definition 2.8 For any $p_1, p_2 \in$ DIS such that $\pi(p_1) = \pi(p_2)$, let $\varphi(p_1, p_2)$ be the least relation on locations satisfying:

$$\varphi(nil, nil) \;=\; \varphi(x, x) \;=\; \varphi(\mu.\, r_1, \mu.\, r_2) \;=\; \varphi(\,(r_1 + s_1)\,,\; (r_2 + s_2)\,) \;=\; \{(\varepsilon, \varepsilon)\}$$

$$\varphi(\, r_1\backslash\alpha,\, r_2\backslash\alpha\,) \;=\; \varphi(r_1\,\langle f\rangle,\, r_2\,\langle f\rangle) \;=\; \varphi(r_1, r_2)$$

$$\varphi(\, r_1[rec\, x.\; r_1/x],\, r_2[rec\, x.\; r_2/x]\,) \;\subseteq\; \varphi(\, rec\, x.\; r_1,\, rec\, x.\; r_2\,)$$

$$\begin{aligned}\varphi(\,(l_1 :: r_1 \mid k_1 :: s_1)\,,\; (l_2 :: r_2 \mid k_2 :: s_2)\,) \;=\;& \{(\varepsilon, \varepsilon)\} \\ & \cup\, (l_1, l_2) \cdot \varphi(r_1, r_2) \\ & \cup\, (k_1, k_2) \cdot \varphi(s_1, s_2)\end{aligned}$$

where for any relation φ, we let $(l, l') \cdot \varphi \;=_{\mathrm{def}}\; \{(lu, l'v) \mid (u, v) \in \varphi\}$.

It may be easily checked that the relation $\varphi(p_1, p_2)$ is a consistent location association. Note that $\varphi(p_1, p_2)$ only relates those locations of p_1 and p_2 which are "statically exhibited", i.e. which do not occur under a dynamic operator. The following lemma establishes the relation between the association $\varphi(p_1, p_2)$ and the transitions of p_1, p_2.

Lemma 2.9 *Let $p_1, p_2 \in$ DIS be such that $\pi(p_1) = \pi(p_2)$. Then:*

1. $p_1 \overset{a}{\underset{u}{\Longrightarrow}}_s p_1' \;\Rightarrow\; \exists\, p_2', v$ *s.t.* $p_2 \overset{a}{\underset{v}{\Longrightarrow}}_s p_2'\,,\; \varphi(p_1, p_2) \,\cup\, (u, v) \,\subseteq\, \varphi(p_1', p_2')$ *and* $\pi(p_1') = \pi(p_2')$
2. $p_1 \overset{\tau}{\Longrightarrow}_s p_1' \;\Rightarrow\; \exists\, p_2'$ *s.t.* $p_2 \overset{\tau}{\Longrightarrow}_s p_2'\,,\; \varphi(p_1, p_2) \subseteq \varphi(p_1', p_2')$ *and* $\pi(p_1') = \pi(p_2')$

Proof: By induction on the proofs of transitions of distributed processes. Note that in general $\varphi(p_1, p_2) \subseteq \varphi(p_1', p_2')$ because new parallel structure may appear as the computations proceed.

We may now prove the above proposition.

Proof of Proposition 2.7: Define the family $\mathbf{T} = \{T_\varphi \mid \varphi \in \Phi\}$ by letting:

$$T_\varphi = \{\, (r_1, r_2) \mid \pi(r_1) = \pi(r_2) \text{ and } \varphi \subseteq \varphi(r_1, r_2)\,\}$$

It is clear that $r_1 T_{\emptyset} r_2$ for any r_1, r_2 such that $\pi(r_1) = \pi(r_2)$. Let us show that $\mathbf{T}$ is a progressive bisimulation family. Suppose that $r_1 T_\varphi r_2$. If $r_1 \overset{a}{\underset{u}{\Longrightarrow}}_s r_1'$ then by Lemma 2.9 $r_2 \overset{a}{\underset{v}{\Longrightarrow}}_s r_2'$, with $\varphi(r_1, r_2) \cup (u, v) \subseteq \varphi(r_1', r_2')$ and $\pi(r_1') = \pi(r_2')$. We want to show that $\varphi' = \varphi \cup (u, v)$ is a *cla* and that $r_1' T_{\varphi'} r_2'$. But this follows from $\varphi \subseteq \varphi(r_1, r_2)$ and Lemma 2.9, since $\varphi' = \varphi \cup (u, v) \subseteq \varphi(r_1, r_2) \cup (u, v) \subseteq \varphi(r_1', r_2')$. □

Using this proposition, we can now show that:

Proposition 2.10 *The relation $\approx_\ell^s$ is an equivalence on CCS processes.*

PROOF: *Reflexivity:* Consider the family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ defined by:

$$S_\varphi = \begin{cases} \{ (q,q) \mid q \in \mathrm{DIS} \} & \text{if } \varphi \subseteq Id \\ \emptyset & \text{otherwise} \end{cases}$$

It is clear that $\mathbf{S}$ is a progressive bisimulation family such that $(q,q) \in S_\emptyset$ for any $q \in$ DIS. Hence $p \approx_\ell^s p$ for any $p \in$ CCS.

Symmetry: Let $p \approx_\ell^s q$. This means that for some $\bar{p}, \bar{q} \in$ DIS such that $\pi(\bar{p}) = p$ and $\pi(\bar{q}) = q$, there exists a progressive bisimulation family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ such that $\bar{p}\, S_\emptyset\, \bar{q}$. Define now a family $\mathbf{R} = \{R_\psi \mid \psi \in \Phi\}$ by:

$$R_\psi = \{ (r,s) \mid s\, S_{\psi^{-1}}\, r \}$$

Clearly $\bar{q}\, R_\emptyset\, \bar{p}$. We show now that $\mathbf{R}$ is a progressive bisimulation family. Suppose $(r,s) \in R_\psi$: this is because $s\, S_\varphi\, r$, with $\varphi = \psi^{-1}$. Then $r \overset{a}{\underset{u}{\Longrightarrow}}_s r'$ implies $s \overset{a}{\underset{v}{\Longrightarrow}}_s s'$, with $s' S_{\varphi \cup \{(v,u)\}} r'$. Now $[\varphi \cup \{(v,u)\}^{-1} = \psi^{-1} \cup \{(u,v)\}$, thus $(r', s') \in R_{\psi^{-1} \cup \{(u,v)\}}$. The case of unobservable transitions is similar. We can then conclude that $q \approx_\ell^s p$.

Transitivity: Let $p \approx_\ell^s r$ and $r \approx_\ell^s q$. This means that for some $\bar{p}, \bar{q}, r_1, r_2 \in$ DIS such that $\pi(\bar{p}) = p$, $\pi(\bar{q}) = q$, $\pi(r_1) = \pi(r_2) = r$, there exist *pbf's* $\mathbf{R}^1$ and $\mathbf{R}^2$ s.t. $\bar{p} R^1_\emptyset r_1$, $r_2 R^2_\emptyset \bar{q}$. Moreover, if $\mathbf{T}$ is the *pbf* introduced in the proof of Proposition 2.7, we know (from the same proposition) that $r_1 T_\emptyset r_2$. Hence if we define a family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ as follows

$$\begin{array}{rcl} (s,s') \in S_\varphi & \Leftrightarrow & \exists\, t_1, t_2 \in \mathrm{DIS} \text{ such that } \pi(t_1) = \pi(t_2); \\ & & \exists\, pbf's\ \mathbf{S}^1, \mathbf{S}^2,\ \exists\, cla's\ \varphi_1, \psi, \varphi_2, \text{ such that} \\ & & \varphi \subseteq \varphi_2 \circ \psi \circ \varphi_1 \text{ and } s\, S^1_{\varphi_1}\, t_1\, T_\psi\, t_2\, S^2_{\varphi_2}\, s' \end{array}$$

it is clear that $\bar{p}\, S_\emptyset\, \bar{q}$. Moreover $\mathbf{S}$ is a progressive bisimulation family, since for any u, v, v', w we have: $\varphi \cup \{(u,w)\} \subseteq \varphi_2 \cup \{(v',w)\} \circ \psi \cup \{(v,v')\} \circ \varphi_1 \cup \{(u,v)\}$. $\square$

A pleasant consequence of Proposition 2.7 is that $\approx_\ell^s$ is independent from the particular distributions we choose. If two CCS terms p and q are equivalent, then any two distributions of them are related by $S_\emptyset$, for some progressive bisimulation family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$.

Corollary 2.11 *For any $p, q \in$ CCS : $p \approx_\ell^s q \Leftrightarrow$ for all $\bar{p}, \bar{q} \in$ DIS such that $p\, \mathcal{D}\, \bar{p}$ and $q\, \mathcal{D}\, \bar{q}$ there exists a progressive bisimulation family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ such that $\bar{p}\, S_\emptyset\, \bar{q}$.*

By virtue of this result, we can restrict attention to particular "binary" distributions, systematically associating location 0 to the left operand and location 1 to the right operand of a parallel composition. A distribution of this kind will be called *canonical*. Similarly, elements of $\{0,1\}^*$ will be called *canonical locations*. These are in fact the locations used in work by Mukund and Nielsen [115] and, with a slightly different notation, by Montanari and Yankelevich [113, 167]. In fact, when applied to canonical distributions of CCS terms, the static transition rules give exactly the same transitions as the *spatial transition system* of [113, 167] (except for τ-transitions, for which these papers use pairs of locations). Note that since canonical locations can be extracted directly from the structure of terms (in fact they are just a part of the *proof* of a transition, in the sense of the proved transition systems mentioned in section 2.1), in the static approach the extension of the language with location prefixing is not really necessary. Using canonical locations only, the whole static location theory could be developed within the standard syntax of CCS. However, it would then not be possible to relax the theory to allow for the placing of more components into the same location, a point which will be discussed at page 31.

We show now that location equivalence is finer than the ordinary (weak) bisimulation, and that introducing locations only adds discriminations between processes in so far as their distributed aspects are concerned. Let $\mathrm{CCS_{seq}}$ be the set of sequential processes of CCS, built without the parallel operator. On this language location equivalence $\approx_\ell^s$ reduces to the standard weak bisimulation $\approx$:

Proposition 2.12 *For any processes $p, q \in \mathrm{CCS_{seq}}$:* $\quad p \approx_\ell^s q \;\Leftrightarrow\; p \approx q.$

Proof: The proof that $p \approx_\ell^s q \;\Rightarrow\; p \approx q$ is straightforward and left to the reader. We show here that $p \approx_\ell^s q \;\Leftarrow\; p \approx q$. Let R be an ordinary weak bisimulation between $p, q \in \mathrm{CCS_{seq}}$. Note that the distributions of processes in $\mathrm{CCS_{seq}}$ coincide with the processes themselves, and therefore all their transitions have the form $p \xrightarrow[\varepsilon]{\mu}_s p'$. Let $S_\emptyset = \{(p,q)\}$. Since the location association $\varphi = \{(\varepsilon, \varepsilon)\}$ is consistent it is clear that the family of relations $\mathbf{S} = \{S_\emptyset\} \;\cup\; \{S_\varphi \mid \varphi = \{(\varepsilon,\varepsilon)\} \text{ and } S_\varphi = R\} \;\cup\; \{S_\psi \mid \psi \in \Phi \backslash \{\emptyset, \varphi\},\ S_\psi = \emptyset\}$ is a progressive bisimulation family. $\square$

We give next some simple examples. The reader acquainted with causality-based equivalences such as the causal bisimulation of [50], will notice here the difference between location equivalence and causality-based equivalences (the exact relation between these equivalences will be discussed in Section 3.3).

Example 2.13 $\quad a.\, b + b.\, a \quad \not\approx_\ell^s \quad (a.\,\gamma \mid \bar\gamma.\, b)\backslash\gamma \;+\; (b.\,\gamma \mid \bar\gamma.\, a)\backslash\gamma \quad \approx_\ell^s \quad a \mid b$

By Corollary 2.11, to show $\not\approx_\ell^s$ it is enough to find a pair of distributions for which every association fails to be consistent. Using canonical distributions, it is easy to see that the computation a followed by b yields the association $\varphi = \{\, (\varepsilon, 0)\, , (\varepsilon, 1)\,\}$ between the locations of the first two processes, which is *not* consistent. On the other hand, for the second and third process we can build the consistent association $\varphi_1 = \{\, (0,0)\, , (1,1)\,\}$ for the computation a followed by b, and the consistent association $\varphi_2 = \{\, (0,1)\, , (1,0)\,\}$ for the computation b followed by a. A related example is:

Example 2.14 $\quad [\, (a.\,\gamma + b.\,\bar\gamma) \mid (\bar\gamma.\, b + \gamma.\, a)\,]\backslash\gamma \quad \approx_\ell^s \quad a \mid b$

Again, these two processes are intuitively equivalent since they both perform actions a and b in either order at different locations. The construction of the appropriate associations is left to the reader.

Example 2.15 $\quad [(\alpha + b) \mid \bar\alpha.\, b]\backslash\alpha \quad \approx_\ell^s \quad b$

Here the observable behaviour consists for both processes in just one action b occurring at some location. Although the first process may choose its action b from two different components, this cannot be detected by $\approx^s_\ell$. The reader may want to check this fact using canonical distributions.

These examples illustrate the extensional character of $\approx^s_\ell$: from Example 2.14 we see that equivalent processes do not need to be componentwise bisimilar, while Example 2.15 shows that equivalent processes need not even have the same number of parallel components. The following is an example of infinite processes which are not equivalent:

Example 2.16 $\quad rec\,x.\ a.\,x \quad \not\approx^s_\ell \quad (rec\,x.\ a.\,x \mid rec\,x.\ a.\,x)$

Note that in the standard CCS semantics these two processes give rise to isomorphic transition systems. We conclude this section by reconsidering the protocol and mutual exclusion examples discussed in Section 2.2. We let the reader verify the following.

Example 2.17 $\quad PSpec \not\approx^s_\ell Protocol$ and $Mutex \not\approx^s_\ell MSpec \not\approx^s_\ell FMutex \not\approx^s_\ell Mutex$

2.4.2 Static location preorder

We define now a preorder $\sqsubseteq^s_\ell$ on CCS processes, which formalises the idea that one process is more sequential or *less distributed* than another. When dealing with noninterleaving semantics, which are more discriminating than the usual bisimulation semantics, it is interesting to have such preorder notions to be able to relate, for instance, a sequential specification with a distributed implementation.

The location preorder $\sqsubseteq^s_\ell$ is obtained by slightly relaxing the notion of consistent association. The intuition for $p \sqsubseteq^s_\ell q$ is that there exist two distributions $\bar{p}$ and $\bar{q}$ of p and q such that whenever $\bar{p}$ can perform two transitions at independent locations, then $\bar{q}$ performs corresponding transitions at locations which are also independent, while the reverse need not be true. This is expressed by the following notion of left-consistency:

Definition 2.18 A relation $\varphi \subseteq (Loc^* \times Loc^*)$ is a left-consistent location association if:

$$(u, v) \in \varphi \quad \& \quad (u', v') \in \varphi \quad \Rightarrow \quad (u \diamond u' \ \Rightarrow \ v \diamond v')$$

Now, if Ψ is the set of left-consistent location associations, we may obtain a notion of *progressive pre-bisimulation family (ppbf)* on distributed processes of DIS by simply replacing Φ by Ψ in Definition 2.5. Again, this gives rise to a relation on CCS processes:

Definition 2.19 (Static location preorder) If $p, q \in$ CCS, let $p \sqsubseteq^s_\ell q$ if and only if for some $\bar{p}, \bar{q} \in$ DIS such that $p\,\mathcal{D}\,\bar{p}$ and $q\,\mathcal{D}\,\bar{q}$, there exists a progressive pre-bisimulation family $\mathbf{S} = \{S_\psi \mid \psi \in \Psi\}$ such that $\bar{p}\,S_\emptyset\,\bar{q}$.

It is easy to see that $p \approx^s_\ell q \ \Rightarrow \ p \sqsubseteq^s_\ell q$. As may be expected the reverse is not true. We have for instance, resuming the examples from the previous section:

Example 2.20 $\quad a.\,b + b.\,a \quad \sqsubseteq^s_\ell \quad (a.\,\gamma \mid \bar{\gamma}.\,b)\backslash\gamma \ + \ (b.\,\gamma \mid \bar{\gamma}.\,a)\backslash\gamma$

Example 2.21 $\quad rec\,x.\ a.x \quad \sqsubseteq\!\!\!\!\sim^{s}_{\ell} \quad (rec\,x.\ a.x \mid rec\,x.\ a.x)$

Example 2.22 $\quad PSpec \quad \sqsubseteq\!\!\!\!\sim^{s}_{\ell} \quad Protocol$

The reader may want to check, on the other hand, that the two systems *Mutex* and *FMutex* in the mutual exclusion example of p.16 are not related by $\sqsubseteq\!\!\!\!\sim_{\ell}$.

Having introduced both an equivalence $\approx^{s}_{\ell}$ and a preorder $\sqsubseteq\!\!\!\!\sim^{s}_{\ell}$ based on the same intuition, we may wonder whether $\approx^{s}_{\ell}$ coincides with the equivalence $\simeq^{s}_{\ell} =_{def} \sqsubseteq\!\!\!\!\sim^{s}_{\ell} \cap \sqsupseteq\!\!\!\!\sim^{s}_{\ell}$ induced by the preorder. It is clear that $\approx^{s}_{\ell} \subseteq \simeq^{s}_{\ell}$, since we have both $\approx^{s}_{\ell} \subseteq \sqsubseteq\!\!\!\!\sim^{s}_{\ell}$ and $\approx^{s}_{\ell} \subseteq \sqsupseteq\!\!\!\!\sim^{s}_{\ell}$. On the other hand, the kernel of the preorder is weaker than location equivalence, as shown by the following example. Consider the two processes:

$$a.\,a.\,a \;+\; (a \mid a \mid a) \qquad \text{and} \qquad a.\,a.\,a \;+\; a.\,a \mid a \;+\; (a \mid a \mid a)$$

These two processes are not equivalent w.r.t. $\approx^{s}_{\ell}$, but they are equivalent w.r.t. $\simeq^{s}_{\ell}$ because $a.\,a.\,a \;\sqsubseteq\!\!\!\!\sim^{s}_{\ell}\; a.\,a \mid a \;\sqsubseteq\!\!\!\!\sim^{s}_{\ell}\; a \mid a \mid a$.

2.5 Dynamic Approach

We present now the dynamic approach introduced in [28][21], and in particular the dynamic versions of the location equivalence and preorder, noted $\approx^{d}_{\ell}$ and $\sqsubseteq\!\!\!\!\sim^{d}_{\ell}$. In the dynamic approach, locations are associated with actions rather than with parallel components. This association is built dynamically, according to the rule:

$$\text{(D1)} \qquad a.\,p \;\xrightarrow[l]{a}_{d}\; l :: p \qquad \text{for any } \; l \in Loc$$

In some sense locations are transmitted from transitions to processes, whereas in the static approach we had the inverse situation. Rule (D1) is the essence of the dynamic location semantics. The remaining rules for observable transitions are just as in the static semantics, see Figure 2 at p. 20. Note that locations increase here at each step, even if the execution goes on within the same parallel component. In fact the location l created by rule (D1) may be seen as an identifier for action a, or more precisely, for that particular occurrence of a. By inspecting the rules, one can easily see that a generic transition has the form $p \;\xrightarrow[ul]{a}_{d}\; p'$, where $u \in Loc^{*}$: here the string u is a record of all action occurrences which causally precede action a (through the prefixing operator), what we shall call the *causal path* (or *access path*) to a. The whole string ul will be called the *dynamic location* of a.

Let us see some examples of dynamic transitions. The first and third process of Example 2.13 may both execute an a action followed by a b action. The corresponding sequences of dynamic transitions are (for any choice of locations $l_1, l_2 \in Loc$):

$$(7) \qquad a.\,b + b.\,a \;\xrightarrow[l_1]{a}_{d}\; l_1 :: b \;\xrightarrow[l_1\,l_2]{b}_{d}\; l_1 :: l_2 :: nil$$

$$(8) \qquad a \mid b \;\xrightarrow[l_1]{a}_{d}\; l_1 :: nil \mid b \;\xrightarrow[l_2]{b}_{d}\; l_1 :: nil \mid l_2 :: nil$$

[21] An earlier variant presented in [27] will be discussed in Section 3.2.

The observable dynamic transitions $p \xrightarrow[u]{a}_d p'$ are related to the static transitions $p \xrightarrow[u]{a}_s p'$ in a simple way. To see this, let us introduce a few notations. Let $\Delta_k(p)$ be the function that erases the atomic location k in p, wherever it occurs. Formally, $\Delta : (Loc \times \text{LCCS}) \rightarrow$ LCCS is defined by the clauses:

$$\begin{array}{lcl} \Delta_k(p) & = & p, \quad \text{if } p \in \text{CCS} \\ \Delta_k(l :: p) & = & \left\{ \begin{array}{ll} \Delta_k(p) & \text{if } k = l \\ l :: \Delta_k(p) & \text{otherwise} \end{array} \right. \end{array}$$

together with clauses stating the compatibility of Δ_k with the remaining operators (for instance $\Delta_k(p \mid q) = \Delta_k(p) \mid \Delta_k(q)$, etc.). Also, for any $p \in$ LCCS, let $Loc(p)$ be the set of atomic locations occurring in p, defined in the obvious way. We have then the following correspondence between the two kinds of location transitions:

Fact 2.23 *Let $p \in LCCS$, $l \notin Loc(p)$. Then:*

$$\begin{array}{llcl} (i) & p \xrightarrow[ul]{a}_d p' & \Rightarrow & p \xrightarrow[u]{a}_s \Delta_l(p') \\ (ii) & p \xrightarrow[u]{a}_s p' & \Rightarrow & \exists\, p'' \text{ s.t. } p \xrightarrow[ul]{a}_d p'' \text{ and } \Delta_l(p'') = p' \end{array}$$

The proof, by induction on the inference of the transition, is left to the reader.

Because of rule (D1), the dynamic location transition system is both infinitely branching and acyclic: it thus gives infinite representations for all regular processes. In fact, while the infinite branching may be overcome easily (through a *canonical* choice of dynamic locations [167, 37], as will be explained later), the infinite progression is intrinsic to the dynamic semantics.

For τ-transitions, for which we do not want to introduce additional locations, we simply use the static transition rules. Although this last point differentiates our strong dynamic location transition system from that originally introduced in [28], where no locations were associated with τ-transitions, the resulting *weak (dynamic) location transition system* is the same. The definition of the weak dynamic transitions $\overset{a}{\underset{u}{\Longrightarrow}}_d$ and $\overset{\tau}{\Longrightarrow}_d$ is similar to that of the static transitions $\overset{a}{\underset{u}{\Longrightarrow}}_s$ and $\overset{\tau}{\Longrightarrow}_s$.

For instance, the second process of Example 2.13 has the following sequence of strong dynamic transitions:

$$\begin{array}{lll} (a.\gamma \mid \bar{\gamma}.b)\backslash\gamma \;+\; (b.\gamma \mid \bar{\gamma}.a)\backslash\gamma \;\xrightarrow[l_1]{a}_d\; (l_1 :: \gamma \mid \bar{\gamma}.b)\backslash\gamma & \xrightarrow[\varepsilon]{\tau}_d & (l_1 :: nil \mid b)\backslash\gamma \\ & \xrightarrow[l_2]{b}_d & (l_1 :: nil \mid l_2 :: nil)\backslash\gamma \end{array}$$

From this one can derive (for instance) the following sequence of weak transitions:

$$(9) \qquad (a.\gamma \mid \bar{\gamma}.b)\backslash\gamma \;+\; (b.\gamma \mid \bar{\gamma}.a)\backslash\gamma \;\overset{a}{\underset{l_1}{\Longrightarrow}}_d\; (nil \mid b)\backslash\gamma \;\overset{b}{\underset{l_2}{\Longrightarrow}}_d\; (nil \mid nil)\backslash\gamma$$

We define now the *dynamic location equivalence* $\approx_\ell^d$ and the *dynamic location preorder* $\sqsubseteq_\ell^d$. Because of the flexibility in the choice of locations offered by rule (D1), these definitions are much simpler than in the static case. In [28] the relations $\approx_\ell^d$ and $\sqsubseteq_\ell^d$ were obtained as instances of a general notion of *parameterized location bisimulation*. We shall give here directly the instantiated definitions.

2.5.1 Dynamic location equivalence

The dynamic location equivalence $\approx_\ell^d$ is just the ordinary bisimulation equivalence associated with the weak dynamic location transition system.

Definition 2.24 (Dynamic location equivalence) A relation $R \subseteq$ LCCS $\times$ LCCS is called a *dynamic location bisimulation (dlb)* iff for all $(p, q) \in R$ and for all $a \in Act, u \in Loc^+$:

$$\begin{array}{ll}
\textbf{(1)} & p \overset{a}{\underset{u}{\Longrightarrow}}_d p' \;\Rightarrow\; \exists\, q' \text{ such that } q \overset{a}{\underset{u}{\Longrightarrow}}_d q' \text{ and } (p', q') \in R \\
\textbf{(2)} & q \overset{a}{\underset{u}{\Longrightarrow}}_d q' \;\Rightarrow\; \exists\, p' \text{ such that } p \overset{a}{\underset{u}{\Longrightarrow}}_d p' \text{ and } (p', q') \in R \\
\textbf{(3)} & p \overset{\tau}{\Longrightarrow}_d p' \;\Rightarrow\; \exists\, q' \text{ such that } q \overset{\tau}{\Longrightarrow}_d q' \text{ and } (p', q') \in R \\
\textbf{(4)} & q \overset{\tau}{\Longrightarrow}_d q' \;\Rightarrow\; \exists\, p' \text{ such that } p \overset{\tau}{\Longrightarrow}_d p' \text{ and } (p', q') \in R
\end{array}$$

The largest *dlb* is called *dynamic location equivalence* and denoted $\approx_\ell^d$.

For instance, the three processes of Example 2.13 are related as follows (as it is easy to check by looking at their transition sequences given in (7), (9) and (8) respectively):

Example 2.25 $\quad a.\,b + b.\,a \quad \not\approx_\ell^d \quad (a.\,\gamma \mid \bar{\gamma}.\,b)\backslash\gamma \;+\; (b.\,\gamma \mid \bar{\gamma}.\,a)\backslash\gamma \quad \approx_\ell^d \quad a \mid b$

In fact, all the examples given in Section 2.4 for the static location equivalence $\approx_\ell^s$ equally apply to the dynamic location equivalence $\approx_\ell^d$. We refer the reader to [28] for more detailed examples of equivalent and unequivalent processes.

2.5.2 Dynamic location preorder

We consider now the location preorder $\sqsubseteq\!\!\!\!\!\!\approx_\ell^d$. Here, instead of requiring the identity of locations in corresponding transitions, we demand that the locations in the second (more distributed) process be subwords of the locations in the first (more sequential) process. Formally, the *subword* relation $\leq_{\text{sub}}$ on Loc^* is defined by:

$$v \leq_{\text{sub}} u \;\Leftrightarrow\; \exists\, v_1, \ldots, v_k\,,\; \exists\, w_1, \ldots, w_{k+1} \;\text{ s.t. }\; v = v_1 \cdots v_k \;\text{ and }\; u = w_1 v_1 \cdots w_k v_k w_{k+1}$$

Definition 2.26 (Dynamic location preorder) A relation $R \subseteq$ LCCS$\times$LCCS is called a *dynamic location pre-bisimulation (dlpb)* iff for all $(p, q) \in R$ and for all $a \in Act, u \in Loc^+$:

$$\begin{array}{ll}
\textbf{(1)} & p \overset{a}{\underset{u}{\Longrightarrow}}_d p' \;\Rightarrow\; \exists\, v.\; v \leq_{\text{sub}} u, \exists\, q' \text{ such that } q \overset{a}{\underset{v}{\Longrightarrow}}_d q' \text{ and } (p', q') \in R \\
\textbf{(2)} & q \overset{a}{\underset{v}{\Longrightarrow}}_d q' \;\Rightarrow\; \exists\, u.\; v \leq_{\text{sub}} u, \exists\, p' \text{ such that } p \overset{a}{\underset{u}{\Longrightarrow}}_d p' \text{ and } (p', q') \in R \\
\textbf{(3)} & p \overset{\tau}{\Longrightarrow}_d p' \;\Rightarrow\; \exists\, q' \text{ such that } q \overset{\tau}{\Longrightarrow}_d q' \text{ and } (p', q') \in R \\
\textbf{(4)} & q \overset{\tau}{\Longrightarrow}_d q' \;\Rightarrow\; \exists\, p' \text{ such that } p \overset{\tau}{\Longrightarrow}_d p' \text{ and } (p', q') \in R
\end{array}$$

The largest *dlpb* is called *dynamic location preorder* and denoted $\sqsubseteq\!\!\!\!\!\!\approx_\ell^d$.

The intuition is as follows. If p is a sequentialized version of q, then each component of p corresponds to a group of parallel components in q. Thus the local causes of any action of q will correspond to a subset of local causes of the corresponding action of p. This may be easily verified for the following examples:

Example 2.27 $\quad a.\,a.\,a \;\; \sqsubseteq\!\!\!\!\sim{}^{d}_{\ell} \;\; a.\,a \mid a \quad$ and $\quad a.\,b \,+\, b.\,a \;\; \sqsubseteq\!\!\!\!\sim{}^{d}_{\ell} \;\; a \mid b$

Let us briefly return to the Protocol example of Section 2.2. This example was already proposed for study under the static semantics (Examples (2.17) and (2.22)), but it is somewhat easier to handle in the dynamic semantics. We want to argue that:

Example 2.28 $\quad PSpec \not\approx^{d}_{\ell} Protocol$ but $PSpec \sqsubseteq\!\!\!\!\sim{}^{d}_{\ell} Protocol$

Recall that the sequential specification *PSpec* was given by $PSpec \Leftarrow in.\,out.\,PSpec$, while the system *Protocol* was defined as follows:

$$\begin{aligned} Protocol &\Leftarrow (Sender \mid Receiver)\backslash\alpha,\beta \\ Sender &\Leftarrow in.\,\overline{\alpha}.\,\beta.\,Sender \\ Receiver &\Leftarrow \alpha.\,out.\,\overline{\beta}.\,Receiver \end{aligned}$$

To see that $PSpec \not\approx^{d}_{\ell} Protocol$, consider the sequence of transitions of *Protocol*:

$$Protocol \overset{in}{\underset{l}{\Longrightarrow}}_d \overset{out}{\underset{k}{\Longrightarrow}}_d (l :: Sender \mid k :: Receiver)\backslash\alpha,\beta$$

The specification *PSpec* can only respond by performing the sequence of transitions:

$$PSpec \overset{in}{\underset{l}{\Longrightarrow}}_d \overset{out}{\underset{lk}{\Longrightarrow}}_d l :: k :: PSpec$$

Clearly, this transition sequence does not match that of *Protocol*. Note however that the locations indexing the transitions of *Protocol* are subwords of those indexing the transitions of *PSpec*. This suggests how to build a dynamic location pre-bisimulation to prove that $PSpec \sqsubseteq\!\!\!\!\sim{}^{d}_{\ell} Protocol$.

All the other examples given in Section 2.4 for the static location preorder $\sqsubseteq\!\!\!\!\sim{}^{s}_{\ell}$ also hold for $\sqsubseteq\!\!\!\!\sim{}^{d}_{\ell}$. Indeed, as we shall see in the next section, the dynamic relations $\approx^{d}_{\ell}$ and $\sqsubseteq\!\!\!\!\sim{}^{d}_{\ell}$ coincide with the static relations $\approx^{s}_{\ell}$ and $\sqsubseteq\!\!\!\!\sim{}^{s}_{\ell}$ introduced in the previous section.

Discussion

We now briefly recount how the static and dynamic approaches to locations presented in this section emerged. The static approach may appear more natural than the dynamic one as it matches more closely the intuition behind locations as *units of distribution* for processes. However this approach was initially abandoned in favour of the dynamic one, as it was not clear how to define an equivalence based on static locations. In particular the idea of *dynamic location association* did not immediately come to mind. This idea was first proposed by Aceto in [5] for a class of CCS process with only top level parallelism, called *nets of automata*[22], and then extended to the whole language CCS by Castellani in [37]. Similarly, the coincidence of the static notions of equivalence and preorder with the dynamic ones (which will be exposed in the next section) was first established for nets of automata in [5] and then for CCS in [37]. Because of the arbitrary nesting of parallelism and prefixing in CCS terms, and of the interplay between sum and parallelism, this extension was not completely straightforward. An intermediate step was taken by Mukund and Nielsen in [115], where a notion of bisimulation equivalence based on static locations was introduced for a class of asynchronous transition systems modelling CCS with guarded sums. This equivalence, similar to that of [37], was already conjectured to coincide with the dynamic location equivalence of [28].

The first definition of the dynamic location semantics appeared in [27]. In that formulation, rule (D1) generated a location *word* u for each action, rather than an atomic location l. As it turned out, the resulting equivalence was slightly too weak[23], hence this first definition was replaced in [28] by the one presented here.

In [27] a logical characterisation is given for the early variant of dynamic location equivalence, which can be easily adapted to the later variant of [28]. An equational characterisation for both the location equivalence $\approx_\ell^d$ and the location preorder $\lesssim_\ell^d$ is presented in [28]. This axiomatisation makes use of a new prefixing construct $<a@ux>.\,p$, whose semantics is given by:

$$<a@ux>.\,p \xrightarrow[ul]{a}_d p[l/x] \qquad \text{for any } l \in Loc$$

The idea is to reduce terms to normal forms by replacing the ordinary prefixing $a.p$ with the new construct, according to the basic transformation law:

$$a.\,p \;=\; <a@x>.\; x :: p$$

In fact, the logical characterisation is based on a very similar idea. Here modalities of the form $<a>_u$ are added to a classical Hennessy-Milner logic, with the following semantics:

$$p \models \; <a>_u \Phi \quad \text{if, for some } p'\,,\; p \overset{a}{\underset{u}{\Longrightarrow}}_d p' \;\text{ and }\; p' \models \Phi$$

Location equivalence also inherits axiomatic characterisations given on various subsets of CCS for other distributed equivalences[24]. For instance an equational characterisation was given for distributed bisimulation in [36, 38], for a subset of CCS with communication but without restriction. This made use of auxiliary operators of left-merge and communication merge, taken from [13] and previous work on ACP (interestingly, however, the laws for these operators were somewhat different from those holding in ACP). A proof system for local cause equivalence, which also uses these operators (together with a new operator of cause prefixing, see Section 3.2) and covers all finite CCS processes, is presented by Kiehn in [96]. An axiomatisation for a related equivalence may also be found in [64].

We conclude this section with a discussion on the choice of the language LCCS as a basis for the location semantics. The reader may have noticed that, while the relations $\approx_\ell^d$

[22]For this class of processes a consistent location association is just a partial bijection between two sets of locations.

[23]An example of unwanted identification under this equivalence is given in Section 2.6, where this earlier variant is described.

[24]These equivalences will be discussed in Section 2.6.

and $\sqsubseteq\!\!\!\approx^{d}_{\ell}$ have been defined on the language LCCS, the static relations $\approx^s_\ell$ and $\sqsubseteq\!\!\!\approx^{s}_{\ell}$, on the other hand, have been defined only on CCS. In fact, these relations are defined on CCS by lifting them up from relations on DIS, which is itself a proper sublanguage of LCCS. We shall now briefly comment on this choice, and show that some results, like the coincidence of the static and dynamic notions of equivalence and preorder, would fail to hold on the whole language LCCS.

At first sight, we could envisage to extend the static equivalence $\approx^s_\ell$ to the rest of LCCS by letting, for any terms p and q in LCCS\CCS: $p \approx^s_\ell q$ if and only if $pS_\emptyset q$ for some progressive bisimulation family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ (the notion of progressive bisimulation family being itself generalised). Note that in LCCS the distribution of locations may be quite liberal: we have for instance terms like $(l :: a \mid l :: b)$ and $(l :: a + k :: b)$. With our generalised definition of $\approx^s_\ell$ we would have the following identification:

$$(10) \qquad (l :: a \mid l :: b) \qquad \approx^s_\ell \qquad l :: (a.\, b + b.\, a)$$

Now, if this could be argued to be a reasonable equation in a locality based semantics, it would not be compatible with our intended interpretation of $\approx^s_\ell$, which is supposed to preserve the degree of parallelism of processes. In fact, these two processes would *not* be equated in the dynamic semantics:

$$(11) \qquad (l :: a \mid l :: b) \qquad \not\approx^d_\ell \qquad l :: (a.\, b + b.\, a)$$

because the first has a computation:

$$(l :: a \mid l :: b) \xrightarrow[l\, l_1]{a}_d l :: (l_1 :: nil \mid b) \xrightarrow[l\, l_2]{b}_d l :: (l_1 :: nil \mid l_2 :: nil)$$

which the second is not able to mimic. As it were, the dynamic semantics will *always* distinguish between parallel components, no matter what locations have been inserted in front of them. On the other hand the static semantics can only distinguish between parallel components that have been previously tagged with disjoint locations. This example also suggests that $\approx^s_\ell$ would not be preserved by parallel composition on LCCS: we would have for instance $l :: b \ \approx^s_\ell \ k :: b$ but $(l :: a \mid l :: b) \ \not\approx^s_\ell \ (l :: a \mid k :: b)$.

Therefore, it seems fair to say that the appropriate language for defining the static relations $\approx^s_\ell$ and $\sqsubseteq\!\!\!\approx^{s}_{\ell}$ is DIS, rather than LCCS. On the other hand, a language larger than DIS is needed for the dynamic semantics: for example the term $(a \mid l :: b)$ can arise in the dynamic semantics as a derivative of some CCS term, for instance of the term $(a \mid c.\, b)$. This is why LCCS was chosen as the basic language in [28]. It was kept as the reference language in [37], because it is a natural extension of CCS where the dynamic and the static location transitions can be defined in a simple and uniform way.

Another point which is worth discussing is the possibility of relaxing the location semantics to allow for the placing of several parallel components on the same location. Clearly, this would slightly change the focus of the semantics: as it stands, the location semantics prescribes a maximal distribution of processes, identifying their *degree of distribution* with their *degree of parallelism*. In some cases however, it could be interesting to dissociate these two measures, allowing the degree of distribution to be lower than the degree of parallelism. For instance, one may describe in CCS a two-place buffer as a system made of two parallel components. As for the Protocol example discussed earlier, the location semantics would distinguish this system from its sequential specification (see [27] for a detailed description of the buffer example). On the other hand, it may seem excessive to view such an elementary system – representing a simple data structure – as being distributed over different sites. In this and similar cases we may want to allow different parallel components to be sitting together at the same location.

To accomodate this possibility, the static location semantics would require very little modification. It would be enough to omit the requirement that parallel components have different locations in the language DIS. The dynamic semantics, on the other hand, would be slightly more difficult to adapt: since this semantics is sensitive to the degree of parallelism of processes, as shown by Example (11), it would be necessary to introduce a kind of *sequential encapsulation* operator $[\![p]\!]$, whose effect would be to conceal all locations in p. The semantics of this operator could be given by:

$$p \xrightarrow[u]{a}_d p' \quad \Rightarrow \quad [\![p]\!] \xrightarrow[l]{a}_d l :: [\![p']\!] \qquad \text{for any } \; l \in Loc$$

Then, for instance, the process $[\![a \mid b]\!]$ would have a sequence of transitions:

$$[\![a \mid b]\!] \quad \xrightarrow[l]{a}_d \quad l :: [\![l_1 :: nil \mid b]\!] \quad \xrightarrow[lk]{b}_d \quad l :: k :: [\![l_1 :: nil \mid l_2 :: nil]\!]$$

Introducing this operator, however, would not have exactly the same effect as allowing processes of the form $(l :: p \mid l :: q)$ in the static semantics, since in the latter case the sublocations of l could still be distinguished, while the operator $[\![p]\!]$ conceals locations in all the subterms of p. We leave this point open for speculation.

2.6 Equivalence of the Two Approaches

This section is devoted to showing the equivalence of the static and dynamic approach. More precisely, we shall establish the following theorem.

Theorem 2.29 *Let* $p, q \in$ *CCS. Then:* $(\mathbf{1})\; p \approx_\ell^s q \;\Leftrightarrow\; p \approx_\ell^d q$; $(\mathbf{2})\; p \sqsubseteq\!\!\!\approx_\ell^s q \;\Leftrightarrow\; p \sqsubseteq\!\!\!\approx_\ell^d q$.

To this end, we introduce a new transition system on CCS, called *occurrence transition system*, which in some sense incorporates the information of both location transition systems. This system will serve as an intermediate between the static and the dynamic semantics. The main point will be to prove that starting from a static or a dynamic location computation, one may always reconstruct a corresponding occurrence computation. This means, essentially, that all the information about distribution and local causality is already present in both location transition systems.

The two location transition systems could also be compared directly, without recourse to an auxiliary transition system. However the occurrence transition system is interesting in its own right, since it provides a concrete level of description where the notions of *occurrence* of an action, computational *history* of an occurrence and *computation state* of a process have a precise definition. Moreover, as we shall see, it allows for the definition of a notion of *local history preserving bisimulation*, which turns out to be a third equivalent formulation of location equivalence.

2.6.1 The occurrence transition system

We define here a new transition system on CCS, called *occurrence (transition) system*, whose states represent CCS computation states with a "past", and whose labels are occurrences of actions within a computation. This system, which is based on a syntactic notion of *occurrence* of action, is essentially a simplification of the *event (transition) system* introduced in [26] to compare different models of CCS: it is simpler because we do not try to identify uniquely all occurrences of action in a term, as in [26], but only those which can coexist in a computation. Moreover, since we are interested here in weak semantics, we shall not distinguish between different occurrences of τ-actions and we concentrate on *abstract* occurrences, in which τ-actions and communications are absorbed. Formally, the set $\mathcal{O}_\tau$ of *occurrences* is defined as $\mathcal{O}_\tau = \mathcal{O} \cup \{\tau\}$, where the elements of $\mathcal{O}$, the visible occurrences, are given by:

$$e \;::=\; a \;\mid\; a\,e \;\mid\; 0\,e \;\mid\; 1\,e$$

The meaning of the occurrence constructors is as follows: a denotes an initial occurrence of action a (possibly preceded or followed by some τ actions in the computation), $a\,e$ denotes the occurrence e after an action a, while $0\,e, 1\,e$ represent the occurrence e at the left, resp. at the right of a parallel operator. Finally the symbol τ is used - with abuse of notation - to represent any occurrence of a τ-action in a computation. We use $e, e', \ldots$ to range over the whole set $\mathcal{O}_\tau$.

We shall see that a visible occurrence $e \in \mathcal{O}$ incorporates both its static and its dynamic location. Note that $\mathcal{O}$ could also be defined as:

$$\mathcal{O} = (Act \cup \{0,1\})^*\, Act$$

Then an occurrence $e \in \mathcal{O}_\tau$ is either τ or a word σa, for some $\sigma \in (Act \cup \{0,1\})^*$ and $a \in Act$. The *label* of $e \in \mathcal{O}_\tau$ is the action of which e is an occurrence:

Definition 2.30 (Label) The function $\lambda : \mathcal{O}_\tau \rightarrow Act_\tau$ is defined by:

$$\lambda(\tau) = \tau, \quad \lambda(\sigma a) = a$$

This alternative presentation of $\mathcal{O}$ makes it also particularly easy to define the location and the causal path (also called access path in [37]) of a visible occurrence e. The *location* $loc(e)$ of an occurrence $e \in \mathcal{O}$ is the canonical location obtained by projecting e onto $\{0,1\}$ (here we use $proj_\Sigma(w)$ to denote the projection of a word w on an alphabet Σ):

Definition 2.31 (Location) The function $loc : \mathcal{O} \rightarrow \{0,1\}^*$ is defined by:

$$loc(e) = proj_{\{0,1\}}(e)$$

The prefix ordering on $\mathcal{O}$ defines a relation of causality $\preceq$ on visible occurrences:

Definition 2.32 (Local causality) The relation $\preceq \subseteq (\mathcal{O} \times \mathcal{O})$ is given by:

$$e \preceq e' \;\Leftrightarrow\; e = e' \;\text{ or }\; \exists\, e'' \;\text{ s.t. }\; ee'' = e'$$

We know that $\preceq$ is a partial ordering. We call $\preceq$ *local causality* because it connects occurrences within the same parallel component: $e \preceq e' \;\Rightarrow\; loc(e) \ll loc(e')$. Let $e \prec e'$ stand for $(e \preceq e' \;\&\; e \neq e')$. For $e \in \mathcal{O}$, we define $\downarrow\! e = \{e' \mid e' \prec e\}$ to be the set of local causes of e. Then the *causal path* of e is the sequence of such causes:

Definition 2.33 (Causal path) For $e \in \mathcal{O}$, $path(e) =_{\text{def}} e_1 \cdot \cdots \cdot e_n$, where $\{e_1, \ldots, e_n\} = \downarrow\! e$ and $e_i \prec e_{i+1}$, $1 \leq i < n$.

For instance, if $e = 0a10b11c$, then $\downarrow\! e = \{0a, 0a10b\}$ and $path(e) = (0a) \cdot (0a10b)$. We call $e \in \mathcal{O}$ an *initial occurrence* if $\downarrow\! e = \emptyset$ (equivalently $path(e) = \varepsilon$). An initial occurrence has always the form $e = loc(e) \cdot \lambda(e)$. More generally, if $\eta' \ll \eta$ and η/η' is the residual of η after η', defined by $\eta/\eta' = \eta''$ if $\eta = \eta'\eta''$, we have the following characterisation for visible occurrences:

Fact 2.34 *An occurrence $e \in \mathcal{O}$ is completely determined by its label, location and causal path. Namely, if $\lambda(e) = a$, $loc(e) = \eta$ and $path(e) = e_1 \cdot \cdots \cdot e_n$, $n \geq 1$, the occurrence e is given by $e = (loc(e_1) \cdot \lambda(e_1)) \cdot (loc(e_2)/loc(e_1) \cdot \lambda(e_2)) \cdot \cdots \cdot (\eta/loc(e_n) \cdot a)$. If $path(e) = \varepsilon$ then $e = \eta \cdot a$.*

Note that $loc(e)$ may be seen as the *static location* of an occurrence e, while $path(e) \cdot \lambda(e)$ represents its *dynamic location*. We define now the relation of *concurrency* on visible occurrences:

Definition 2.35 (Concurrency) The relation $\smile \subseteq (\mathcal{O} \times \mathcal{O})$ is defined by:

$$e \smile e' \;\Leftrightarrow\; \begin{cases} \textit{either} & e = \sigma\, 0\, e_0 \;\;\&\;\; e' = \sigma\, 1\, e'_0 \\ \quad \textit{or} & e = \sigma\, 1\, e_0 \;\;\&\;\; e' = \sigma\, 0\, e'_0 \end{cases}$$

where $\sigma \in (Act \cup \{0,1\})^*$, $e_0, e'_0 \in \mathcal{O}$.

Clearly, the relation $\smile$ is symmetric and irreflexive, and $e \smile e' \;\Leftrightarrow\; loc(e) \diamond loc(e')$.

The occurrences e will be the labels of the occurrence transition system. Let us now shift attention to the states of this transition system. As we said, these states are meant to represent processes with a *past*. The past records the observable guards which have been passed along a computation. Formally, the set $\mathcal{S}$ of *computation states* is given by:

$$\xi \;::=\; nil \;\mid\; \mu.p \;\mid\; p+q \;\mid\; x \;\mid\; rec\,x.\,p \;\mid\; \widehat{a}.\xi \;\mid\; (\xi \mid \xi') \;\mid\; \xi\backslash\alpha \;\mid\; \xi\,\langle f\rangle$$

where $p, q \in$ CCS. The construct $\widehat{a}.\xi$ is used to represent the state ξ with "past" a, that is, after a guard a has been passed. The idea is that any transition labelled by a visible occurrence will introduce a "hat" in the resulting state. The basic operational rule is:

(O1) $\qquad a.p \xrightarrow{a} \widehat{a}.p$

On the other hand an invisible occurrence τ does not leave any trace in the past. This is expressed by the rule:

(O1$'$) $\qquad \tau.p \xrightarrow{\tau} p$

The hats recorded in states are used to build up "deep" occurrences along computations, according to the rule:

(O2) $\quad \xi \xrightarrow{e} \xi',\;\; e \neq \tau \quad\Rightarrow\quad \widehat{a}.\xi \xrightarrow{a\,e} \widehat{a}.\xi'$

Occurrences of the form $i\,e$, for $i = 0, 1$, originate from parallel terms $\xi \mid \xi'$:

(O3) $\quad \xi \xrightarrow{e} \xi',\;\; e \neq \tau \quad\Rightarrow\quad \xi \mid \xi'' \xrightarrow{0\,e} \xi' \mid \xi''\,, \quad \xi'' \mid \xi \xrightarrow{1\,e} \xi'' \mid \xi'$

For defining the whole occurrence system we need a few more notations. First, we extend action relabellings f to occurrences by letting: $f(a\,e) = f(a)\,f(e)$ and $f(i\,e) = i\,f(e)$ for $i = 0, 1$. Moreover, we shall use an auxiliary function for defining the communication rule. In the occurrence system communication arises from concurrent occurrences with complementary labels. However the resulting τ-occurrence should not contribute to the past, since this only keeps track of observable actions. Thus we need to take back the hats introduced by the synchronising occurrences. To this end we introduce a function $\delta_e(\xi)$, which erases the hat corresponding to occurrence e in ξ (somewhat similar to the function $\Delta_k(p)$ used in Section 2.4). The partial function $\delta : (\mathcal{O} \times \mathcal{S}) \rightarrow \mathcal{S}$ is given by:

$$\begin{array}{lcl}
\delta_a(\widehat{a}.p) & = & p \\
\delta_{ae}(\widehat{a}.\xi) & = & \widehat{a}.\,\delta_e(\xi) \\
\delta_{0e}(\xi \mid \xi') & = & \delta_e(\xi) \mid \xi' \\
\delta_{1e}(\xi \mid \xi') & = & \xi \mid \delta_e(\xi') \\
\delta_e(\xi\backslash\alpha) & = & \delta_e(\xi)\backslash\alpha \\
\delta_{f(e)}(\xi\,\langle f\rangle) & = & \delta_e(\xi)\,\langle f\rangle
\end{array}$$

We have now all the elements to define the occurrence system for CCS. The rules specifying this system are listed in Figure 3. Note that the condition $\lambda(e) \neq \{\alpha, \bar{\alpha}\}$ in rule (O6) could be strenghtened to $proj_{\{\alpha,\bar{\alpha}\}}(e) = \varepsilon$, to prevent transitions like $\widehat{a}\,.\,b.\,p \xrightarrow{a\,b} \widehat{a}\,.\,\widehat{b}.\,p$. However this would make no difference for states ξ obtained via an occurrence computation from a CCS term (more will be said on this point below). The *weak occurrence system* is now given by:

$$\xi \stackrel{\tau}{\Longrightarrow} \xi' \quad \Leftrightarrow \quad \exists\, \xi_0, \ldots, \xi_n\,,\ n \geq 0 \quad s.t. \quad \xi = \xi_0 \xrightarrow{\tau} \xi_1 \cdots \xrightarrow{\tau} \xi_n = \xi'$$
$$\xi \stackrel{e}{\Longrightarrow} \xi' \quad \Leftrightarrow \quad \exists\, \xi_1, \xi_2 \quad s.t. \quad \xi \stackrel{\tau}{\Longrightarrow} \xi_1 \xrightarrow{e} \xi_2 \stackrel{\tau}{\Longrightarrow} \xi'$$

Let us examine some properties of this weak occurrence system. It is clear that any term ξ gives an intensional representation of a CCS computation state. In fact from each state ξ one may extract the set of visible occurrences that have led to it. Obviously, this set should be empty for a CCS term. Formally, the set of *past occurrences* of a term ξ is defined by:

$$\begin{array}{lcl} occ\,(p) & = & \emptyset, \quad \text{if } p \in \text{CCS} \\ occ\,(\widehat{a}.\,\xi) & = & \{a\} \cup a \cdot occ\,(\xi) \\ occ\,(\xi \mid \xi') & = & 0 \cdot occ\,(\xi) \cup 1 \cdot occ\,(\xi') \\ occ\,(\xi \backslash \alpha) & = & \{\, e \in occ\,(\xi) \mid \lambda(e) \neq \alpha, \bar{\alpha}\,\} \\ occ\,(\xi\,\langle f \rangle) & = & f(occ\,(\xi)) \end{array}$$

For states ξ reachable from a CCS term, the clause for restriction reduces to $occ\,(\xi\backslash\alpha) = occ\,(\xi)$. In fact, one may easily verify that such states, what we shall call CCS *computation states*, are exactly those ξ whose subterms $\xi'\backslash\alpha$ satisfy $\alpha, \bar{\alpha} \notin \lambda(occ\,(\xi'))$. Moreover for a CCS computation state ξ each relabelling f is injective on $occ\,(\xi)$ (this justifies, for instance, the last clause in the definition of the function δ above).

Remark 2.36 *If $\delta_e(\xi)$ is defined, then* $occ\,(\delta_e(\xi)) = occ\,(\xi) - \{e\}$.

The next statements explain how visible occurrences are generated along computations, and how they are related.

Lemma 2.37 *Let ξ be a* CCS *computation state. Then:*

1. $$\xi \xrightarrow{e} \xi',\ e \neq \tau \quad \Rightarrow \quad \begin{cases} (i) \ \ occ\,(\xi') = occ\,(\xi) \cup \{e\} \\ (ii) \ \ \forall e' \in occ\,(\xi):\ e' \prec e \ \text{ or } \ e' \smile e\,) \\ (iii) \ \ {\downarrow}e \subseteq occ\,(\xi) \end{cases}$$

2. $$\xi \xrightarrow{\tau} \xi' \quad \Rightarrow \quad occ\,(\xi') = occ\,(\xi)$$

Corollary 2.38 *Let* $p \in$ CCS. *If* $p \stackrel{e_1}{\Longrightarrow} \xi_1 \ \cdots \stackrel{e_n}{\Longrightarrow} \xi_n$ *, where* $\forall i:\ e_i \neq \tau$*, then:*

1. $\forall i:\ occ\,(\xi_i) = \{\, e_1, \ldots, e_i \,\}$

2. $i < j \ \Rightarrow\ e_i \prec e_j$ or $e_i \smile e_j$

The following proposition shows how local causality may be recovered from static locations along a computation:

Proposition 2.39 *Let $p \in CCS$. If* $p \stackrel{e_1}{\Longrightarrow} \xi_1 \ \cdots \stackrel{e_n}{\Longrightarrow} \xi_n$ *, where* $\forall i:\ e_i \neq \tau$*, then:*

$$e_i \prec e_j \quad \Longleftrightarrow \quad i < j \ \text{ and } \ loc(e_i) \ll loc(e_j)$$

Proof: $\Leftarrow$: Since $i < j$, by Corollary 2.38 either $e_i \prec e_j$ or $e_i \smile e_j$. But it cannot be $e_i \smile e_j$, since this would imply $loc(e_i) \diamond loc(e_j)$. Thus $e_i \prec e_j$. $\Rightarrow$: If $e_i \prec e_j$, then it cannot be $j < i$, because of Corollary 2.38 again. Moreover, since e_i is a prefix of e_j, also $loc(e_i)$ is a prefix of $loc(e_j)$. $\square$

We proceed now to define a notion of bisimulation on the weak occurrence system. Once again we use a notion of consistency and progressive bisimulation family.

Definition 2.40 A *consistent occurrence aliasing* is a partial injective function $g : \mathcal{O} \to \mathcal{O}$ which satisfies, for any e, e' on which it is defined:

(i) $\lambda(e) = \lambda(g(e))$

(ii) $e' \prec e \;\Leftrightarrow\; g(e') \prec g(e)$

Let $\mathcal{G}$ be the set of consistent occurrence aliasings on $\mathcal{O}$.

Definition 2.41 A *progressive o-bisimulation family* is a $\mathcal{G}$-indexed family of relations over $\mathcal{S}$, $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$, such that if $\xi_0 \, R_g \, \xi_0'$ then for all $e \in \mathcal{O}$:

(1) $\xi_0 \stackrel{e}{\Longrightarrow} \xi_1 \;\Rightarrow\; \exists\, e', \xi_1' \;\; s.t. \;\; \xi_0' \stackrel{e'}{\Longrightarrow} \xi_1', \;\; g \cup \{(e,e')\} \in \mathcal{G} \;\; and \;\; \xi_1 \, R_{g \cup \{(e,e')\}} \, \xi_1'$

(2) $\xi_0' \stackrel{e'}{\Longrightarrow} \xi_1' \;\Rightarrow\; \exists\, e, \xi_1 \;\; s.t. \;\; \xi_0 \stackrel{e}{\Longrightarrow} \xi_1, \;\; g \cup \{(e,e')\} \in \mathcal{G} \;\; and \;\; \xi_1 \, R_{g \cup \{(e,e')\}} \, \xi_1'$

(3) $\xi_0 \stackrel{\tau}{\Longrightarrow} \xi_1 \;\Rightarrow\; \exists \xi_1' \in \mathcal{S} \;\; s.t. \;\; \xi_0' \stackrel{\tau}{\Longrightarrow} \xi_1' \;\; and \;\; \xi_1 R_g \xi_1'$

(4) $\xi_0' \stackrel{\tau}{\Longrightarrow} \xi_1' \;\Rightarrow\; \exists \xi_1 \in \mathcal{S} \;\; s.t. \;\; \xi_0 \stackrel{\tau}{\Longrightarrow} \xi_1 \;\; and \;\; \xi_1 R_g \xi_1'$

These relations induce an equivalence $\approx^{occ}$ on CCS processes as follows:

Definition 2.42 (Equivalence on the occurrence system) For any $p, q \in$ CCS, let $p \approx^{occ} q$ iff $p R_\emptyset q$ for some progressive o-bisimulation family $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$.

The reader familiar with the notion of *history preserving bisimulation* (*cf* e.g. [73]) may have noticed the similarity with our definition of $\approx^{occ}$. In fact history preserving bisimulation is itself a "progressive" notion, and it is clear that a consistent occurrence aliasing g is nothing else than an isomorphism between two partially ordered sets of occurrences. In Section 3.3 we give a formal definition of *local history preserving bisimulation* on the occurrence system (so-called because the ordering is that of local causality), and show that it is a direct reformulation of the equivalence $\approx^{occ}$.

The preorder $\sqsubseteq\!\!\!\!\sim^{occ}$ is obtained using the same definition, after weakening the notion of *consistency* as follows:

Definition 2.43 A *right-consistent occurrence aliasing* is a partial injective function $g : \mathcal{O} \to \mathcal{O}$ which satisfies, for any e, e' on which it is defined:

(i) $\lambda(e) = \lambda(g(e))$

(ii) $g(e') \prec g(e) \;\Rightarrow\; e' \prec e$

The main result, presented in the next sections, is that the equivalence $\approx^{occ}$ coincides with both $\approx_\ell^s$ and $\approx_\ell^d$, and similarly that $\sqsubseteq\!\!\!\!\sim^{occ}$ coincides with both $\sqsubseteq\!\!\!\!\sim_\ell^s$ and $\sqsubseteq\!\!\!\!\sim_\ell^d$. The proofs rely on the above properties of the occurrence system, and on proving conversion lemmas between the different kinds of transitions. Most of these proofs will be omitted or only outlined. The reader is referred to [37] for a full exposition.

2.6.2 Occurrence semantics = static location semantics

We establish here the relationship between $\approx^{occ}$ and $\approx^s_\ell$. We saw in Section 2.4 that $\approx^s_\ell$ can be defined in terms of canonical distributions. We recall that these are distributions always associating location 0 to the left operand and 1 to the right operand of a parallel composition. Let CDIS denote the set of these canonical distributions, and $\eta, \zeta \in \{0,1\}^*$ range over canonical locations. With each state ξ, we associate a distributed process $dis(\xi) \in$ CDIS as follows:

$$\begin{array}{lcl}
dis(\xi) & = & \xi, \text{ if } \xi = nil \text{ or } \xi = x \\
dis(a.\, p) & = & a.\, dis(p) \\
dis(p+q) & = & dis(p) + dis(q) \\
dis(rec\, x.\; p) & = & rec\, x.\; dis(p) \\
dis(\widehat{a}.\, \xi) & = & dis(\xi) \\
dis(\xi \mid \xi') & = & 0 :: dis(\xi) \mid 1 :: dis(\xi') \\
dis(\xi \backslash \alpha) & = & dis(\xi) \backslash \alpha \\
dis(\xi \langle f \rangle) & = & dis(\xi) \langle f \rangle
\end{array}$$

Thus $dis(\xi)$ is the canonical distribution of the CCS term underlying ξ. We can now give the conversion lemma between occurrence transitions and static location transitions:

Lemma 2.44 (**Conversion : static $\leftrightarrow$ occurrence**) *Let $p, p' \in$ CDIS and $\xi, \xi' \in \mathcal{S}$. Then:*

$$\begin{array}{llcl}
(i) & \xi \overset{\tau}{\Longrightarrow} \xi' & \Rightarrow & dis(\xi) \overset{\tau}{\Longrightarrow}_s dis(\xi') \\
(ii) & p \overset{\tau}{\Longrightarrow}_s p' & \Rightarrow & \forall \xi \;\; s.t. \;\; dis(\xi) = p \;\; \exists \xi' \;\; s.t. \;\; \xi \overset{\tau}{\Longrightarrow} \xi' \;\; and \;\; dis(\xi') = p' \\
(iii) & \xi \overset{e}{\Longrightarrow} \xi' & \Rightarrow & dis(\xi) \underset{\eta}{\overset{a}{\Longrightarrow}}_s dis(\xi'), \;\; where \;\; a = \lambda(e) \;\; and \;\; \eta = loc(e) \\
(iv) & p \underset{\eta}{\overset{a}{\Longrightarrow}}_s p' & \Rightarrow & \forall \xi \;\; s.t. \;\; dis(\xi) = p \; \exists\, e, \xi' \;\; s.t. \;\; \lambda(e) = a, \;\; loc(e) = \eta, \\
& & & \qquad dis(\xi') = p' \;\; and \;\; \xi \overset{e}{\Longrightarrow} \xi'
\end{array}$$

Using Lemma 2.44 and Proposition 2.39 we may now prove our equivalence result.

Theorem 2.45 *For any $p, q \in$ CCS : $\quad p \approx^s_\ell q \;\Leftrightarrow\; p \approx^{occ} q$.*

Proof outline. $\Rightarrow$: Suppose $p \approx^s_\ell q$, and let $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ be a progressive bisimulation family such that $dis(p)\, S_\emptyset\, dis(q)$. Define a $\mathcal{G}$-indexed family of relations $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ as follows:

$(\xi, \xi') \in R_g \iff$ there exist occurrence transition sequences

$$p \overset{\tau}{\Longrightarrow} \xi_0 \overset{e_1}{\Longrightarrow} \cdots \overset{e_n}{\Longrightarrow} \xi_n = \xi$$
$$q \overset{\tau}{\Longrightarrow} \xi'_0 \overset{e'_1}{\Longrightarrow} \cdots \overset{e'_n}{\Longrightarrow} \xi'_n = \xi'$$

such that

(1) $g \;=\; \{(e_1, e'_1), \ldots, (e_n, e'_n)\}$

(2) $dis(\xi)\; S_\varphi\, dis(\xi')$, where $\varphi = \{\, (loc(e), loc(e')) \mid (e, e') \in g \,\}$

Clearly $(p,q) \in R_\emptyset$. It may be shown that $\mathbf{R}$ is a progressive o-bisimulation family.

$\Leftarrow$: Suppose $p \approx^{occ} q$, and let $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ be a progressive o-bisimulation family such that $p\, R_\emptyset\, q$. Consider the family $\mathbf{S} = \{S_\varphi \mid \varphi \in \Phi\}$ of relations over CDIS given by:

$$
\begin{array}{lll}
(r,s) \in S_\varphi & \Longleftrightarrow & \text{there exist occurrence transition sequences} \\
& & \qquad p \stackrel{\tau}{\Longrightarrow} \xi_0 \stackrel{e_1}{\Longrightarrow} \cdots \stackrel{e_n}{\Longrightarrow} \xi_n = \xi \\
& & \qquad q \stackrel{\tau}{\Longrightarrow} \xi'_0 \stackrel{e'_1}{\Longrightarrow} \cdots \stackrel{e'_n}{\Longrightarrow} \xi'_n = \xi' \\
& & \text{such that} \\
& & \qquad (1)\;\; dis(\xi) = r\,, \quad dis(\xi') = s \\
& & \qquad (2)\;\; \varphi = \{\, (\, loc(e_i), loc(e'_i)\,) \mid 1 \le i \le n \,\} \\
& & \qquad (3)\;\; \xi\, R_g\, \xi' \;\text{ for }\; g = \{\, (e_i, e'_i) \mid 1 \le i \le n \,\}
\end{array}
$$

It may be shown that $\mathbf{S}$ is a progressive bisimulation family. □

Using a variation of this proof, a similar result can be established for the preorders:

Theorem 2.46 *For any* $p, q \in$ CCS*:* $\quad p \sqsubseteq\!\!\!\!\sim^{s}_{\ell}\, q \;\Leftrightarrow\; p \sqsubseteq\!\!\!\!\sim^{occ}\, q$.

This concludes the comparison between the static location semantics and the occurrence transition semantics.

2.6.3 Occurrence semantics = dynamic location semantics

We turn now to the relation between $\approx^{occ}$ and $\approx^d_\ell$. To establish the coincidence of the two equivalences, we will use a property of $\approx^d_\ell$ which was first pointed out by Kiehn in [97], namely that $\approx^d_\ell$ only depends on computations where *distinct* atomic locations are chosen at each step. We start by recalling some definitions and results from [28]:

Definition 2.47 A *location renaming* is a mapping $\rho : Loc \to Loc^*$. For any $p \in$ LCCS, let $p[\rho]$ denote the process obtained by replacing all occurrences of l in p with $\rho(l)$, for any $l \in Loc$.

We use the notation $\rho\{u/l\}$ for the renaming which maps l to u and acts like ρ on $Loc \setminus \{l\}$. Also, we shall abbreviate $p[id\{u/l\}]$ to $p\{u/l\}$. In what follows, we shall mainly consider alphabetic renamings $\rho : Loc \to Loc$. Note that any partial function $f : Loc \to Loc$ may be seen as a location renaming $\rho : Loc \to Loc$, by letting:

$$
\rho(l) \;=\; \begin{cases} l & \text{if } f(l) \text{ is not defined} \\ f(l) & \text{otherwise.} \end{cases}
$$

For instance the empty function $\emptyset$ corresponds to the identity renaming id. In the following we shall freely use the renaming notation $p[f]$ whenever f is a partial function $f : Loc \to Loc$.

The following lemma relates the transitions of $p[\rho]$ with those of p.

Lemma 2.48 *Let $p \in LCCS$. Then for any alphabetic location renaming $\rho : Loc \rightarrow Loc$:*

1. a) $p \xrightarrow[u]{\tau}_d p' \;\Rightarrow\; \exists v$ *s.t.* $\rho(u) \ll v$ *and* $p[\rho] \xrightarrow[v]{\tau}_d p'[\rho]$.

   b) $p \overset{\tau}{\Longrightarrow}_d p' \;\Rightarrow\; p[\rho] \overset{\tau}{\Longrightarrow}_d p'[\rho]$.

2. a) $p[\rho] \xrightarrow[v]{\tau}_d p' \;\Rightarrow\; \exists u, p''$ *such that* $\rho(u) \ll v$, $p''[\rho] = p'$ *and* $p \xrightarrow[v]{\tau}_d p''$.

   b) $p[\rho] \overset{\tau}{\Longrightarrow}_d p' \;\Rightarrow\; \exists\, p''$ *such that* $p''[\rho] = p'$ *and* $p \overset{\tau}{\Longrightarrow}_d p''$.

3. a) $p \xrightarrow[ul]{a}_d p'$, $l \notin Loc(p) \;\Rightarrow\; \forall\, k \in Loc$, $p[\rho] \xrightarrow[vk]{a}_d p'[\rho\{k/l\}]$, *where* $v = \rho(u)$.

   b) *Same as a), with weak transitions.*

4. a) $p[\rho] \xrightarrow[vl]{a}_d p' \;\Rightarrow\; \exists\, u$ *such that* $\rho(u) = v$ *and* $\forall\, k \notin Loc(p)\; \exists\, p''$ *such that* $p''[\rho\{l/k\}] = p'$ *and* $p \xrightarrow[uk]{a}_d p''$.

   b) *Same as a), with weak transitions.*

We recall now Kiehn's definition for $\approx_\ell^d$, and show that it is equivalent to the original one.

Notation 2.49 *Let $LCCS^\nu$ be the set of LCCS processes whose atomic locations are all distinct.*

Definition 2.50 (ν–dynamic location equivalence)

A relation $R \subseteq (\mathrm{LCCS}^\nu \times \mathrm{LCCS}^\nu)$ is a *ν–dynamic location bisimulation (ν–dlb)* iff for all $(p, q) \in R$ and for all $a \in Act, u \in Loc^*$:

(1) $p \overset{a}{\underset{ul}{\Longrightarrow}}_d p'$, $l \notin Loc(p) \cup Loc(q) \;\Rightarrow\; \exists\, q'$ *s.t.* $q \overset{a}{\underset{ul}{\Longrightarrow}}_d q'$ *and* $(p', q') \in R$

(2) $q \overset{a}{\underset{ul}{\Longrightarrow}}_d q'$, $l \notin Loc(p) \cup Loc(q) \;\Rightarrow\; \exists\, p'$ *s.t.* $p \overset{a}{\underset{ul}{\Longrightarrow}}_d p'$ *and* $(p', q') \in R$

(3) $p \overset{\tau}{\Longrightarrow}_d p' \;\Rightarrow\; \exists\, q'$ *s.t.* $q \overset{\tau}{\Longrightarrow}_d q'$ *and* $(p', q') \in R$

(4) $q \overset{\tau}{\Longrightarrow}_d q' \;\Rightarrow\; \exists\, p'$ *s.t.* $p \overset{\tau}{\Longrightarrow}_d p'$ *and* $(p', q') \in R$

The largest ν–dlb is called ν–dynamic location equivalence and denoted $\approx_\ell^\nu$.

Fact 2.51 *For any processes $p, q \in CCS$: $p \approx_\ell^d q \;\Leftrightarrow\; p \approx_\ell^\nu q$.*

Proof outline: The $\Rightarrow$ direction is trivial. For the $\Leftarrow$ direction, Lemma 2.48 is used in a straightforward way. □

We proceed now to show that $\approx^{occ} = \approx_\ell^\nu$. To do this, we need to establish a conversion between occurrence transitions and dynamic location transitions. We start by converting terms ξ into LCCS terms which represent the same state of computation. The idea is to replace every "hat" in ξ by a canonical atomic location representing uniquely the corresponding occurrence. The simplest way to do this is to take the occurrences themselves as *canonical (dynamic) locations.* We shall then assume, from now onwards, that $\mathcal{O} \subseteq Loc$. We also introduce, for any $\gamma \in Act \cup \{0, 1\}$, a renaming ρ_γ which prefixes by γ all the occurrences appearing as locations in p, namely $\rho_\gamma(e) = \gamma e$. Then the canonical LCCS process $proc(\xi)$ corresponding to a computation state $\xi \in \mathcal{S}$ is defined by:

$$\begin{array}{lcl} proc(\xi) & = & \xi, \text{ if } \xi \in \text{CCS} \\ proc(\widehat{a}.\,\xi) & = & a :: proc(\xi)[\rho_a] \\ proc(\xi \mid \xi') & = & proc(\xi)[\rho_0] \mid proc(\xi')[\rho_1] \\ proc(\xi \backslash \alpha) & = & proc(\xi)\backslash\alpha \\ proc(\xi\,\langle f \rangle) & = & proc(\xi)[f]\,\langle f \rangle \end{array}$$

Note that in the last clause the first f is a location renaming, while the second is an action relabelling. We have for instance: $proc(\widehat{a}.\,\widehat{b}.\,nil \mid c.\,nil) = 0a :: 0ab :: nil \mid c.\,nil$. Similarly, if f is the relabelling given by $f(a) = a_1, f(b) = b_1, f(c) = c_1$, then $proc((\widehat{a}.\,\widehat{b}.\,nil \mid c.\,nil)\,\langle f \rangle) = (0a :: 0ab :: nil \mid c.\,nil)[f]\,\langle f \rangle = 0a_1 :: 0a_1b_1 :: nil \mid c_1.\,nil$. It can be easily checked that $Loc(proc(\xi)) = occ(\xi)$ and $proc(\xi) \in \text{LCCS}^{\nu}$.

We give next the conversion lemma between occurrence transitions and dynamic location transitions, followed by the coincidence result:

Lemma 2.52 (**Conversion : dynamic $\leftrightarrow$ occurrence**) *If $\xi, \xi' \in \mathcal{S},\ f : occ(\xi) \to Loc$, then:*

$$\begin{array}{lll} (i) & \xi \stackrel{\tau}{\Longrightarrow} \xi' & \Rightarrow\ proc(\xi)[f] \stackrel{\tau}{\Longrightarrow}_d proc(\xi')[f] \\ (ii) & proc(\xi)[f] \stackrel{\tau}{\Longrightarrow}_d p' & \Rightarrow\ \exists\, \xi' \text{ s.t. } proc(\xi')[f] = p' \text{ and } \xi \stackrel{\tau}{\Longrightarrow} \xi' \\ (iii) & \xi \stackrel{e}{\Longrightarrow} \xi' & \Rightarrow\ \forall l \in Loc :\ proc(\xi)[f] \underset{ul}{\stackrel{a}{\Longrightarrow}}_d proc(\xi')\,[f\{l/e\}], \text{ where} \\ & & \ a = \lambda(e) \text{ and } u = f(path(e)) \\ (iv) & proc(\xi)[f] \underset{ul}{\stackrel{a}{\Longrightarrow}}_d p' & \Rightarrow\ \exists\, e, \xi' \text{ s.t. } \lambda(e) = a,\ f(path(e)) = u,\ \xi \stackrel{e}{\Longrightarrow} \xi' \text{ and} \\ & & \ proc(\xi')\,[f\{l/e\}] = p' \end{array}$$

Theorem 2.53 *For any $p, q \in \text{CCS}$:* $\quad p \approx^{occ} q \ \Leftrightarrow\ p \approx^{d}_{\ell} q.$

PROOF: $\Rightarrow$: Let $p \approx^{occ} q$. Then there exists a $\mathcal{G}$-indexed family of relations $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ such that $pR_\emptyset q$. Define a relation S on processes by:

$(r, s) \in S \iff$ there exist occurrence transition sequences

$$p \stackrel{\tau}{\Longrightarrow} \xi_0 \stackrel{e_1}{\Longrightarrow} \cdots \stackrel{e_n}{\Longrightarrow} \xi_n = \xi$$
$$q \stackrel{\tau}{\Longrightarrow} \xi'_0 \stackrel{e'_1}{\Longrightarrow} \cdots \stackrel{e'_n}{\Longrightarrow} \xi'_n = \xi'$$

and there exist two functions:

$$f_1 : \{e_1, \ldots, e_n\} \to Loc\,, \quad f_2 : \{e'_1, \ldots, e'_n\} \to Loc$$

such that

(1) $\forall i \in \{1, \ldots, n\} :\ f_1(e_i) = f_2(e'_i)$

(2) $proc(\xi)\,[f_1] = r\,,\ \ proc(\xi')[f_2] = s$

(3) $\xi\, R_g\, \xi'$ for $g = \{\, (e_1, e'_1), \ldots, (e_n, e'_n)\,\}$

It may be shown that S is a dynamic location bisimulation.

$\Leftarrow$: to prove this direction, we use the alternative definition $\approx_\ell^\nu$ of dynamic location equivalence. Suppose $p \approx_\ell^\nu q$. Define a $\mathcal{G}$-indexed family of relations $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ as follows:

$$
\begin{aligned}
(\xi, \xi') \in R_g \quad \Longleftrightarrow \quad & \text{there exist occurrence transition sequences} \\
& \qquad p \stackrel{\tau}{\Longrightarrow} \xi_0 \stackrel{e_1}{\Longrightarrow} \cdots \stackrel{e_n}{\Longrightarrow} \xi_n = \xi \\
& \qquad q \stackrel{\tau}{\Longrightarrow} \xi'_0 \stackrel{e'_1}{\Longrightarrow} \cdots \stackrel{e'_n}{\Longrightarrow} \xi'_n = \xi' \\
& \text{such that} \\
& \qquad (1)\ \ g \ = \ \{ (e_1, e'_1), \ldots, (e_n, e'_n) \} \\
& \qquad (2)\ \ \exists \text{ injection } f : \{e'_1, \ldots e'_n\} \rightarrow Loc \text{ such that :} \\
& \qquad\qquad proc(\xi)\,[f \circ g] \ \approx_\ell^\nu \ proc(\xi')[f]
\end{aligned}
$$

Note that $proc(\xi)\,[f \circ g]$, $proc(\xi')[f] \in \mathrm{LCCS}^\nu$, given that f and g are injections. Also, it is easy to see that $(p, q) \in R_\emptyset$, since $proc(p)[\emptyset] = p$ and $proc(q)[\emptyset] = q$ (recall that the partial function $\emptyset$ corresponds to the identity renaming id). It may be shown that $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ is a progressive o-bisimulation family. □

We state now the analogous result for the preorders. To prove $p \sqsubseteq\!\!\!\sim_\ell^d q \ \Rightarrow \ p \sqsubseteq\!\!\!\sim^{occ} q$ one uses a preorder $\sqsubseteq\!\!\!\sim_\ell^\nu$ (the obvious variant of $\approx_\ell^\nu$) in place of $\sqsubseteq\!\!\!\sim_\ell^d$. The proof is an easy adaptation of that for the equivalences.

Theorem 2.54 *For any* $p, q \in \mathrm{CCS}$*:* $\quad p \sqsubseteq\!\!\!\sim_\ell^d q \ \Leftrightarrow \ p \sqsubseteq\!\!\!\sim^{occ} q$.

This concludes our comparison between the dynamic location semantics and the occurrence transition semantics.

Putting together Theorems 2.45 and 2.53, we obtain Theorem 2.29 **(1)**, stating the coincidence of $\approx_\ell^s$ and $\approx_\ell^d$. Similarly the coincidence of $\sqsubseteq\!\!\!\sim_\ell^s$ and $\sqsubseteq\!\!\!\sim_\ell^d$, stated in Theorem 2.29 **(2)**, follows from Theorems 2.46 and 2.54.

In the light of these results, the location equivalence and preorder will be simply denoted $\approx_\ell$ and $\sqsubseteq\!\!\!\sim_\ell$ in the following, unless an explicit reference to their specific definition is required.

For each $e \in \mathcal{O}_\tau$ let $\xrightarrow{e} \subseteq (\mathcal{S} \times \mathcal{S})$ be the least binary relation satisfying the following axioms and rules.

$$
\begin{array}{lllll}
\text{(O1)} & a.p \xrightarrow{a} \widehat{a}.\,p & & & \\
\text{(O1}') & \tau.p \xrightarrow{\tau} p & & & \\
\text{(O2)} & \xi \xrightarrow{e} \xi'\,, \quad e \neq \tau & \Rightarrow & \widehat{a}.\,\xi \xrightarrow{a\,e} \widehat{a}.\,\xi' \\
\text{(O3)} & \xi \xrightarrow{e} \xi'\,, \quad e \neq \tau & \Rightarrow & \xi \mid \xi'' \xrightarrow{0\,e} \xi' \mid \xi'' \\
 & & & \xi'' \mid \xi \xrightarrow{1\,e} \xi'' \mid \xi' \\
\text{(O2}') & \xi \xrightarrow{\tau} \xi' & \Rightarrow & \widehat{a}.\,\xi \xrightarrow{\tau} \widehat{a}.\,\xi' \\
\text{(O3}') & \xi \xrightarrow{\tau} \xi' & \Rightarrow & \xi \mid \xi'' \xrightarrow{\tau} \xi' \mid \xi'' \\
 & & & \xi'' \mid \xi \xrightarrow{\tau} \xi'' \mid \xi' \\
\text{(O4)} & \left.\begin{array}{l} \xi_0 \xrightarrow{e_0} \xi_0'\,, \quad \xi_1 \xrightarrow{e_1} \xi_1' \\ \text{and} \;\; \lambda(e_0) = \overline{\lambda(e_1)} \end{array}\right\} & \Rightarrow & \xi_0 \mid \xi_1 \xrightarrow{\tau} \delta_{e_0}(\xi_0') \mid \delta_{e_1}(\xi_1') \\
\text{(O5)} & p \xrightarrow{e} \xi & \Rightarrow & p + q \xrightarrow{e} \xi \\
 & & & q + p \xrightarrow{e} \xi \\
\text{(O6)} & \xi \xrightarrow{e} \xi'\,, \quad \lambda(e) \notin \{\alpha, \bar{\alpha}\} & \Rightarrow & \xi \backslash \alpha \xrightarrow{e} \xi' \backslash \alpha \\
\text{(O7)} & \xi \xrightarrow{e} \xi' & \Rightarrow & \xi\,\langle f \rangle \xrightarrow{f(e)} \xi'\,\langle f \rangle \\
\text{(O8)} & p\,[rec\,x.\;p/x] \xrightarrow{e} \xi & \Rightarrow & rec\,x.\;p \xrightarrow{e} \xi
\end{array}
$$

Figure 3: Occurrence transition system

3 Extensions and comparison with other approaches

We compare here the location semantics presented in the previous Section with other noninterleaving semantics that have been proposed for CCS in the literature, all based on variations of the notion of bisimulation. An extension of the location semantics to the π-calculus is also presented in Section 3.4. This section is mainly addressed to readers with a keen interest in understanding the various noninterleaving semantics for CCS. The reader preferring to keep to the main exposition of location-based semantics may safely skip to Section 4.3, which surveys more recent approaches to process algebras with localities.

To carry out our comparison we will rely on the *local/global cause transition system* of Kiehn [97], which records both local and global causes of actions along computations. This is a convenient formalism for our purpose, as it brings together the ideas of distributed and causal semantics. Intuitively, distributed semantics deal with *local causality*, which is induced by the structure of terms, while causal semantics are concerned with *global causality*, which is determined by the flow of control of computations. We shall see in fact that both the location semantics of [28] and the causal semantics of [49] can be characterised as particular instantiations of the local/global cause-semantics.

As announced already in Section 2.2, distributed and causal semantics are in general incomparable. However they coincide over the communication and restriction-free fragment of CCS, a class of processes often referred to as BPP (Basic Parallel Processes). This convergence result, first proved for finite processes in [97, 3], has been recently generalised to recursive BPP processes by Kiehn [94]. For easy reference we recall here the syntax of BPP:

$$p \ ::= \ nil \mid \mu.p \mid (p \mid q) \mid (p+q) \mid x \mid rec\, x.\, p$$

In BPP, communication is not allowed in parallel composition. The same language, where communication is allowed, is referred to as CPP (Communicating Parallel Processes). We will use $\mathrm{BPP_f}$ (resp. $\mathrm{CPP_f}$) to denote the finite fragment of BPP (resp. CPP).

We start by introducing the local/global cause transition system, which will serve as our reference model in this section.

3.1 The local/global cause semantics

We present here the *local/global cause (lgc) transition system*, which was proposed by Kiehn in [97] (and further studied in [94]) as a unifying framework for studying distributed and causal semantics. This transition system also provides a new, stronger semantics arising from the joint observation of local and global causality.

The *lgc*–semantics is defined on an extended syntax, which we call here LGCCS. This includes, besides the operators of CCS, a construct of *cause prefixing* $<\Lambda, \Gamma> :: p$, where Λ and Γ represent the sets of *local causes*, respectively *global causes*, of the process p. Intuitively, these causes have been generated along the computation leading to state p. Local causes correspond to actions wich precede p via the standard prefixing operator, while global causes may also be *inherited* from another component by means of a communication. Thus $\Lambda \subseteq \Gamma$. A CCS process p is identified with the extended process $<\emptyset, \emptyset> :: p$. Formally, Λ and Γ are sets of pairs of the form (a, i), where a is a visible action in Act and $i \in I\!N$. For simplicity, a pair (a, i) is rendered as a_i (this will imply $a_i \neq a_j$ for all $i \neq j$). We let $\mathcal{K} = \{a_i \mid a \in Act,\ i \in I\!N\}$ be the set of all causes. By construction $\mathcal{K} \cap Act = \emptyset$.

We extend the complementation function to causes by setting $\overline{(a,i)} = (\bar{a}, i)$. We also use $\mathcal{K}_{\mathrm{L}}(p)$, resp. $\mathcal{K}_{\mathrm{G}}(p)$, to denote the set of causes occurring in some Λ, resp. Γ, inside p. Finally, we let $\mathcal{K}(p) = \mathcal{K}_{\mathrm{L}}(p) \cup \mathcal{K}_{\mathrm{G}}(p)$ denote the set of all causes occurring in p.

Note the formal similarity between cause prefixing $<\Lambda, \Gamma> :: p$ and location prefixing $l :: p$. In fact the semantics will operate much in the same way as the location semantics, exhibiting the causes of actions in transitions. These transitions have the general form $p \xrightarrow{a_i, <\Lambda,\Gamma>} p'$, representing an occurrence of action a with set of local causes Λ and set of global causes Γ. The most notable difference with the (dynamic) location semantics is in the rule for communication, which propagates the global causes of each communicating process to the other partner. Again, causes are only assigned to visible actions. Thus τ-transitions have the form $p \xrightarrow{\tau} p'$ and are inferred using the standard rules of CCS.

Formally the semantics is given by the rules in Figure 4. We write $p \xrightarrow{obs} p'$ when the actual label of the transition is not important. In the communication rule (LG4) the notation $\{\Gamma_q / a_i\}$ is used to indicate the *cause replacement* of a_i by Γ_q in p': this is the process obtained by removing a_i from all local cause sets occurring in p' and replacing a_i by the elements of Γ_q in all the global cause sets in p'.

Based on these transitions, one has the following notion of *lgc*-bisimulation:

Definition 3.1 (Local/global cause bisimulation) A relation $R \subseteq$ LGCCS $\times$ LGCCS is a local/global cause bisimulation (*lgc*-bisimulation) iff for all $(p, q) \in R$ and for each label $obs \in \{\tau\} \cup (\mathcal{K} \times 2^{\mathcal{K}} \times 2^{\mathcal{K}})$:

$$\textbf{(1)}\ \ p \xrightarrow{obs} p' \ \ \Rightarrow\ \ \exists\, q' \text{ such that } q \xrightarrow{obs} q' \text{ and } (p', q') \in R$$

$$\textbf{(2)}\ \ q \xrightarrow{obs} q' \ \ \Rightarrow\ \ \exists\, p' \text{ such that } p \xrightarrow{obs} p' \text{ and } (p', q') \in R$$

The largest *lgc*-bisimulation is called local/global cause (bisimulation) equivalence and denoted $\sim_{\ell gc}$.

Local/global cause equivalence requires the equality of both local and global causes in matching transitions. If this condition is relaxed by allowing $obs = a_i, <\Lambda, \Gamma>$ to be matched by another label $obs' = a_i, <\Lambda, \Gamma'>$, one obtains a weaker equivalence notion called *local cause equivalence* and denoted by $\sim_{\ell c}$. Similarly, if $obs = a_i, <\Lambda, \Gamma>$ is allowed to be matched by $obs' = a_i, <\Lambda', \Gamma>$, one obtains a weaker equivalence called *global cause equivalence* and written $\sim_{gc}$. In [97] a parametric definition is given in place of Definition 3.1, where labels *obs* are filtered through an observation function. Then the three equivalences $\sim_{\ell gc}$, $\sim_{\ell c}$ and $\sim_{gc}$, as well as the ordinary bisimulation equivalence $\sim$, can be obtained as special instances of this parametric definition. In fact, the easiest way to look upon *lc*-equivalence and *gc*-equivalence is to regard them as the bisimulations associated with the *reduced lgc*-transition system where cause prefixing $<\Lambda, \Gamma> :: p$ is replaced by $\Lambda :: p$ and $\Gamma :: p$ respectively (and the operational rules are simplified accordingly). We shall call these reduced transition systems the *lc*-transition system and *gc*-transition system.

The weak *lgc*-transitions are derived from the strong transitions in the standard way, and induce notions of weak *lgc*-equivalence, weak *lc*-equivalence, and weak *gc*-equivalence, denoted respectively $\approx_{\ell gc}$, $\approx_{\ell c}$ and $\approx_{gc}$.

We recall from [97] the main results concerning these equivalences. The first is a simple consequence of the definitions.

$$\text{(LG1)} \qquad a.\,p \xrightarrow{a_i,\,<\emptyset,\emptyset>} \,<\{a_i\},\{a_i\}>::p \qquad\qquad a_i \notin \mathcal{K}(p)$$

$$\text{(LG2)} \qquad p \xrightarrow{a_i,\,<\Lambda',\Gamma'>} p' \quad \Rightarrow \quad <\Lambda,\Gamma>::p \xrightarrow{a_i,\,<\Lambda\cup\Lambda',\Gamma\cup\Gamma'>} \,<\Lambda,\Gamma>::p'$$

$$\text{(LG3)} \qquad p \xrightarrow{a_i,\,<\Lambda,\Gamma>} p', \quad a_i \notin \mathcal{K}(q) \quad \Rightarrow \quad \begin{array}{l} p \mid q \xrightarrow{a_i,\,<\Lambda,\Gamma>} p' \mid q \\ q \mid p \xrightarrow{a_i,\,<\Lambda,\Gamma>} q \mid p' \end{array}$$

$$\text{(LG4)} \qquad \left.\begin{array}{l} p \xrightarrow{a_i,\,<\Lambda_p,\Gamma_p>} p', \quad q \xrightarrow{\bar{a}_j,\,<\Lambda_q,\Gamma_q>} q' \\ a_i \notin \mathcal{K}(q),\ \bar{a}_j \notin \mathcal{K}(p), \\ \mathcal{K}_{\mathrm{L}}(p) \cap \mathcal{K}_{\mathrm{L}}(q) = \emptyset \end{array}\right\} \quad \Rightarrow \quad p \mid q \xrightarrow{\tau} p'\{\Gamma_q/a_i\} \mid q'\{\Gamma_p/\bar{a}_j\}$$

$$\text{(LG5)} \qquad p \xrightarrow{obs} p' \quad \Rightarrow \quad \begin{array}{l} p + q \xrightarrow{obs} p' \\ q + p \xrightarrow{obs} p' \end{array}$$

$$\text{(LG6)} \qquad \left.\begin{array}{l} p \xrightarrow{a_i,\,<\Lambda_q,\Gamma_q>} p' \\ a \notin \{\alpha,\bar{\alpha}\},\ \Gamma_q \cap \{\alpha,\bar{\alpha}\} = \emptyset \end{array}\right\} \quad \Rightarrow \quad p\backslash\alpha \xrightarrow{a_i,\,<\Lambda_q,\Gamma_q>} p'\backslash\alpha$$

$$\text{(LG7)} \qquad \left.\begin{array}{l} p \xrightarrow{a_i,\,<\Lambda_q,\Gamma_q>} p' \\ f(a)_j \notin \mathcal{K}(p\,\langle f\rangle) \end{array}\right\} \quad \Rightarrow \quad p\,\langle f\rangle \xrightarrow{f(a)_j,\,<f(\Lambda_q),f(\Gamma_q)>} p'\,\langle f\rangle$$

$$\text{(LG8)} \qquad p[rec\,x.\ p/x] \xrightarrow{obs} p' \quad \Rightarrow \quad rec\,x.\ p \xrightarrow{obs} p'$$

Figure 4: Local/global cause transitions

Transitions $p \xrightarrow{\tau} p'$ are inferred from (LG4) for communication, from the standard rules of CCS for the remaining operators, and from the following rule for cause prefixing:

$$\text{(LG9)} \qquad p \xrightarrow{\tau} p' \quad \Rightarrow \quad <\Lambda,\Gamma>::p \xrightarrow{\tau} \,<\Lambda,\Gamma>::p'$$

Figure 5: Rules for τ-transitions

Proposition 3.2 *Let* $p, q \in$ LGCCS. *If* $p \approx_{\ell gc} q$ *then* $p \approx_{\ell c} q$ *and* $p \approx_{gc} q$.

The reverse of this proposition is not true. The following example, taken from [97], shows that $\approx_{\ell gc}$ is strictly finer than the intersection of $\approx_{\ell c}$ and $\approx_{gc}$.

Example 3.3 For the following processes p and q, we have $p \approx_{\ell c} q$ and $p \approx_{gc} q$ but not $p \approx_{\ell gc} q$:

$$\begin{aligned} p &= a \mid b.\,c \quad + \quad (a.\,\gamma \mid \bar{\gamma}.\,b.\,\delta \mid \bar{\delta}.\,c)\backslash\gamma, \delta \\ q &= p \qquad\quad + \quad (a.\,\gamma \mid \bar{\gamma}.\,b.\,c)\backslash\gamma, \delta \end{aligned}$$

In [97] it is also shown that local cause equivalence and global cause equivalence are incomparable, even on the simple sublanguage without restriction:

Proposition 3.4 *Let* $p, q \in$ LGCCS. *Then* $p \approx_{\ell c} q \not\Rightarrow p \approx_{gc} q$, *and* $p \approx_{gc} q \not\Rightarrow p \approx_{\ell c} q$.

The proof is given by the example:

Example 3.5

$$(a.\,\gamma \mid \bar{\gamma}.\,b) \;+\; a.\,b \quad \begin{matrix} \not\approx_{\ell c} \\ \approx_{gc} \end{matrix} \quad (a.\,\gamma \mid \bar{\gamma}.\,b)$$

$$(a.\,\gamma \mid \bar{\gamma}.\,b) \;+\; (b.\,\gamma \mid \bar{\gamma}.\,a) \;+\; a \mid b \quad \begin{matrix} \approx_{\ell c} \\ \not\approx_{gc} \end{matrix} \quad (a.\,\gamma \mid \bar{\gamma}.\,b) \;+\; (b.\,\gamma \mid \bar{\gamma}.\,a)$$

On the other hand, the two equivalences coincide when no communication is allowed. This is expected since the only rule that differentiates the *lc*-transition system and the *gc*-transition system is the rule for communication.

Proposition 3.6 *Let* $p, q \in$ BPP. *Then* $p \approx_{\ell c} q \;\Leftrightarrow\; p \approx_{gc} q$.

In fact this coincidence already holds for the strong equivalences $\sim_{\ell c}$ and $\sim_{gc}$. The main results of [97] are the characterisations of location equivalence $\approx_{\ell}$ and causal bisimulation $\approx_c$ [49] for CCS processes, respectively as the local cause equivalence $\approx_{\ell c}$ and the global cause equivalence $\approx_{gc}$. We simply state here these results, referring to Kiehn's paper for the proofs.

Proposition 3.7 (Characterisation of location equivalence) *Let* $p, q \in$ CCS. *Then* $p \approx_{\ell} q \;\Leftrightarrow\; p \approx_{\ell c} q$.

Proposition 3.8 (Characterisation of causal bisimulation) *Let* $p, q \in$ CCS. *Then* $p \approx_c q \;\Leftrightarrow\; p \approx_{gc} q$.

A couple of comments may be helpful here. As regards Proposition 3.7, we noted already the similarity between the *lc*-semantics and the location semantics. This becomes even more evident if we consider the *lc*-transition system to be given by the rules of Figure 4 where $< \Lambda, \Gamma > :: p$ is replaced by just $\Lambda :: p$. There remain some differences though:

in the location transition system locations are not required to be new at each step[25]; moreover, local causes (locations) are recorded together with their occurrence ordering, while in the *lc*-transition system they are just collected into a set. In fact for general LGCCS processes location equivalence is finer than *lc*-equivalence. For instance, using $l :: p$ as an abbreviation for $< \{l\}, \emptyset > ::\ p$, the two processes $l :: k :: a$ and $k :: l :: a$ are local cause equivalent but not location equivalent. Indeed, the fact that $\approx_\ell = \approx_{\ell c}$ for CCS processes does not mean that the occurrence ordering of causes is immaterial here, but rather that for cause-free processes this ordering is already accounted for by the bisimulation relations.

The relationships between the *gc*-transition system and the causal transition system of [49, 50] will be discussed in Section 3.3, where causal bisimulation is reviewed.

3.2 Comparison with other distributed semantics

We examine here how location equivalence relates to an earlier version introduced in [27], called *loose location equivalence.* We also compare it with *distributed bisimulation* ([38], [36], [95, 47]), the first distribution based equivalence proposed in the literature, and with the *local mixed-ordering equivalence* of [167, 114].

We shall arrive at the following picture for distribution based equivalences: for finite restriction–free processes of $\mathrm{CPP_f}$ they all coincide; on the whole language CCS, distributed bisimulation equivalence and loose location equivalence are incomparable, and both are weaker than location equivalence and local mixed-ordering equivalence, which coincide.

3.2.1 Loose location equivalence

We start by comparing the dynamic location equivalence $\approx_\ell^d$ with its earlier formulation given in [27]. While the definition of the two equivalences is formally the same, the underlying location transition systems are slightly different. Transitions in [27] are more general in that the location allocated at each step is a word $u \in Loc^*$ instead of an atomic location $l \in Loc$. To avoid confusion the transitions of [27] will be called *loose location transitions*, and denoted $\overset{a}{\underset{u}{\rightsquigarrow}}$. For the loose transition system the rule (D1) of Figure 2 is replaced by

$$\text{(LD1)} \qquad a.p \overset{a}{\underset{u}{\rightsquigarrow}} u :: p \qquad u \in Loc^*$$

The rules concerning the other process constructors, the τ-transitions and weak transitions are the same as for the dynamic location transition system of Section 2.5. We denote the weak loose transitions by $p \overset{a}{\underset{u}{\Rrightarrow}} p'$. The location equivalence based on these transitions is called *loose location equivalence* and denoted $\simeq_{\ell\ell}$.

Although the difference between $(D1)$ and $(LD1)$ may seem quite inoffensive (and indeed it was thought to be so when the formulation in [27] was chosen), it turns out that the loose transition system gives more freedom for relating process behaviours. While for

[25] We have seen already in Section 2.6 that the location equivalence and preorder for CCS processes do not change if new locations are created at each step. Indeed, this was established by Kiehn in [97] precisely for the purpose of comparing location equivalence with *lc*-equivalence.

the location equivalence $\approx_\ell$, based on atomic allocation, we implicitly require the equality of the last allocated locations, this is not true for loose location equivalence. The latter can introduce more than one atomic location in one step and thus is able to fill up "missing locations". The following example (due to Rob Van Glabbeek) shows that loose location equivalence $\simeq_{\ell\ell}$ can equate processes which are distinguished by location equivalence $\approx_\ell$. Let p and q represent respectively the processes $(l :: \alpha \mid \bar{\alpha}.\, b)\backslash\alpha$ and $(l :: (\alpha + b) \mid \bar{\alpha}.\, b)\backslash\alpha$. Then the move of q

$$(l :: (\alpha + b) \mid \bar{\alpha}.\, b)\backslash\alpha \ \xrightarrow[lk]{b}\ (l :: k :: nil \mid \bar{\alpha}.\, b)\backslash\alpha$$

which is also a loose move, can only be matched by a loose move of p, introducing the location lk in one step:

$$(l :: \alpha \mid \bar{\alpha}.\, b)\backslash\alpha \ \overset{b}{\underset{lk}{\Rrightarrow}}\ (l :: nil \mid lk :: nil)\backslash\alpha$$

Indeed we have $p \approx_{\ell\ell} q$ but $p \not\approx_\ell q$. This also implies that the CCS terms $(a.\, \alpha \mid \bar{\alpha}.\, b)\backslash\alpha$ and $(a.\, (\alpha + b) \mid \bar{\alpha}.\, b)\backslash\alpha$ are loosely location equivalent but not location equivalent. Now, intuitively it is not desirable to equate these processes, since in the first process action b is not locally dependent upon action a .

From this example we see that the two location equivalences are different for CCS terms. However the property of filling up "missing locations" only comes into play when processes containing the restriction operator are considered. It can be shown, in fact, that for finite restriction-free processes the two equivalences coincide [28]:

Proposition 3.9 *Let* $p, q \in \mathrm{CPP_f}$ *. Then* $p \simeq_{\ell\ell} q \ \Leftrightarrow\ p \approx_\ell q$.

On the full language, on the other hand, location equivalence is finer than loose location equivalence.

Proposition 3.10 *Let* $p, q \in \mathrm{CCS}$*. Then* $p \approx_\ell q \ \Rightarrow\ p \simeq_{\ell\ell} q$.

In conclusion, $\approx_\ell$ is stronger than $\simeq_{\ell\ell}$ and captures more precisely the intuition about local dependencies. This explains why in [28] the first formulation of location equivalence $\simeq_{\ell\ell}$ was abandoned in favour of $\approx_\ell$. The interested reader is referred to this paper for more details on these results.

3.2.2 Distributed bisimulation

We now turn to the relationship between $\approx_\ell$ and distributed bisimulation. Distributed bisimulation was the first attempt to define a semantics for CCS taking the distributed nature of processes into account. It was introduced by Castellani and Hennessy in [36, 38] and further studied by Kiehn in [95, 94] and by Corradini and De Nicola in [47].

The basic idea is similar to that of location equivalence. The capabilities of observers are increased so that they can observe actions together with the location where they are performed. However, locations are not explicitly introduced here; instead, there are multiple observers which can move from one location to another as the computation proceeds. When observing an action, the current observer moves to its location and appoints a new observer for the remainder of the process. Thus the number of observers increases by one each time a visible action is performed. In this way locations are observed much in the same way as in the dynamic approach, but without assigning names to them.

We use here the original definition of distributed bisimulation, as given in [36] for the subset $\mathrm{CPP_f}$ of CCS. This is based on a *distributed transition system* where transitions give rise to a pair of residuals, a *local residual* and a *concurrent residual* (or *remote residual*). Intuitively, the first represents what locally follows the action, while the second is the part of the process which is concurrent with the action[26]. We shall give here a slightly different presentation of the operational semantics of [36], which is taken from [95].

We will distinguish the local residual by including it in brackets $[\ ]$, as for instance in $(a.nil \mid [b.nil]) \mid c.\bar{b}.nil$. The syntax for "processes with a local component" is given by the following grammar, where p stands for any $\mathrm{CPP_f}$ -term,

$$P \ ::= \ [p] \mid \ (P \mid p) \ \mid \ (p \mid P)$$

Let $\mathcal{P}_{[\ ]}$ denote this language. We use $\mathcal{C}[p], \mathcal{D}[q], \ldots$ to range over $\mathcal{P}_{[\ ]}$. In $\mathcal{C}[p]$, p is the *local process* and $\mathcal{C}$ is its *context*. For example in $(a.nil \mid [b.nil]) \mid c.\bar{b}.nil$ the local process is $p = b.nil$ and the context is $\mathcal{C} = (a.nil \mid .) \mid c.\bar{b}.nil$. We also use the notation $\mathcal{C}(p)$ to represent the CCS process obtained by simply embedding p in $\mathcal{C}$ without the $[\ ]$–brackets. Processes in $\mathcal{P}_{[\ ]}$ are used to exhibit the location associated with the last visible action. Visible transitions have the form $p \xrightarrow{a}_d \mathcal{C}[p']$. The observer who sees action a moves to its location, and from then onwards only observes the local component p'. The newly appointed observer takes care of the remote component $\mathcal{C}(nil)$.

The rules for (strong) distributed transitions are given in Figure 6. These rules apply only to restriction–free terms.

For each $a \in Act$ let $\xrightarrow{a}_d \subseteq (\mathrm{CPP_f} \times \mathcal{P}_{[\ ]})$ be the least binary relation satisfying the following axiom and rules.

(D1)	$a.p \xrightarrow{a}_d [p]$		
(D2)	$p \xrightarrow{a}_d \mathcal{C}[p']$	implies	$p + q \xrightarrow{a}_d \mathcal{C}[p']$ $q + p \xrightarrow{a}_d \mathcal{C}[p']$
(D3)	$p \xrightarrow{a}_d \mathcal{C}[p']$	implies	$p \mid q \xrightarrow{a}_d \mathcal{C}[p'] \mid q$ $q \mid p \xrightarrow{a}_d q \mid \mathcal{C}[p']$

Figure 6: Distributed Transitions

Weak transitions are derived by the rule

$$\text{(WD)} \qquad p \overset{\varepsilon}{\Rightarrow} p_1 \xrightarrow{a}_d \mathcal{C}[p'] \overset{\varepsilon}{\Rightarrow} \mathcal{D}[p''] \quad \text{implies} \quad p \overset{a}{\Rightarrow}_d \mathcal{D}[p'']$$

[26] An alternative formulation in terms of *local* and *global* residuals was proposed in [38].

where transitions $\mathcal{C}[p'] \overset{\varepsilon}{\Rightarrow} \mathcal{D}[p'']$ are defined in the usual way from the transitions $\xrightarrow{\tau}$, which in turn are based on the relations $\xrightarrow{\mu}$ given by the standard operational rules of CCS together with the rule:

$$\text{(D4))} \qquad p \xrightarrow{\mu} p' \quad \text{implies} \quad [p] \xrightarrow{\mu} [p']$$

For example we can derive

$$\begin{array}{lll} (a \mid \bar{a}.b) \mid c.\bar{b} & \xrightarrow{\bar{a}}_d & (a \mid [b]) \mid c.\bar{b} \\ (a \mid \bar{a}.b) \mid c.\bar{b} & \overset{c}{\Rightarrow}_d & (nil \mid nil) \mid [nil] \end{array}$$

The main difference between the location and the distributed semantics does not lie in the underlying transition systems but in the way the equivalences work; in the first case observing a location results in assigning a name to it, while in the latter it results in splitting the process into a local part and a remote part. Note however that in the distributed case we have at each step only two "locations", while in the location semantics the number of locations appearing in a computation state is unbounded.

Definition 3.11 (Distributed Bisimulation Equivalence)
A symmetric relation $R \subseteq \mathrm{CPP_f} \times \mathrm{CPP_f}$ is called a *distributed bisimulation* iff $R \subseteq D(R)$ where $(p, q) \in D(R)$ iff

(**1**) $p \overset{\varepsilon}{\Rightarrow} p'$ implies $q \overset{\varepsilon}{\Rightarrow} q'$ for some $q' \in \mathrm{CPP_f}$ such that $(p', q') \in R$

(**2**) $p \overset{a}{\Rightarrow}_d \mathcal{C}[p']$ implies $q \overset{a}{\Rightarrow}_d \mathcal{D}[q']$ for some $\mathcal{D}[q'] \in \mathcal{P}_{[\ ]}$ such that $(p', q') \in R$ and $(\mathcal{C}(nil), \mathcal{D}(nil)) \in R$

Two distributed processes p and q are said to be *distributed bisimulation equivalent*, $p \approx_d q$, if there is a distributed bisimulation R such that $(p, q) \in R$.

As an example of non–equivalent processes consider $a \mid b$ and $a.b + b.a$. If $a.b + b.a \overset{a}{\Rightarrow}_d [b]$ then $a \mid b$ can match this move only with $a \mid b \overset{a}{\Rightarrow}_d [nil] \mid b$. So we have to compare the local subprocesses, b and nil, and the remaining ones, i.e. $\mathcal{C}(nil) = nil$ and $\mathcal{D}(nil) = nil \mid b$. Obviously in both cases the equivalence does not hold. On the other hand we have

$$p \;=\; (a \mid \bar{a}.b) \mid c.\bar{b} + c.\bar{b} \mid b \quad \approx_d \quad (a \mid \bar{a}.b) \mid c.\bar{b} \;=\; q$$

For example the transition $p \overset{c}{\Rightarrow}_d [\bar{b}] \mid b$ can be matched $q \overset{c}{\Rightarrow}_d nil \mid b \mid [\bar{b}]$; it is obvious that the processes at the current locations are equivalent and so are the remaining processes $nil \mid b$ and $nil \mid b \mid nil$.

We next state the coincidence of $\approx_\ell$ and $\approx_d$ on $\mathrm{CPP_f}$. The main difference between these two equivalences resides in the fact that distributed bisimulation acts separately on the two residuals of a transition. The crux of the proof is a decomposition property for location equivalence, which allows located terms themselves to be split. This property says that if $u :: p \mid q \approx_\ell u :: r \mid s$ then $p \approx_\ell r$ and $q \approx_\ell s$. A proof of this property, as well as of the following coincidence result, may be found in [27].

Proposition 3.12 *Let $p, q \in \mathrm{CPP_f}$. Then $p \approx_d q \;\Leftrightarrow\; p \approx_\ell q$.*

A consequence of this proposition is that results available for distributed bisimulation can be reused for location equivalence. For instance, from [36, 38] one gains a complete axiomatization for location equivalence on $\mathrm{CPP_f}$, which is significantly different from that given in [28].

In [95] the definition of distributed bisimulation was generalised by Kiehn to all finite CCS processes. We will not give the definition of this *generalised distributed bisimulation* here, but simply mention that this equivalence is weaker than $\approx_\ell$. This is shown by a rather involved example in [95], which we therefore do not report here.

Intuitively the difference between the two equivalences is that distributed bisimulation may "forget" locations which have been observed in previous (not immediately preceding) states. Hence, a transition of some component of one process can be matched by a transition of a component of the other process which originally was "at a different location". This shows that location equivalence is a better choice than generalized distributed bisimulation when looking for an extension of distributed bisimulation to general CCS processes.

A further proposal for generalising distributed bisimulation to full CCS was presented by Corradini and De Nicola in [47]. Here distributed bisimulation is reformulated within the grape semantics of Degano, De Nicola and Montanari, allowing for a more symmetric treatment of the local and global residuals with respect to [95]. The resulting equivalence is finer than Kiehn's one, but still fails to recover the full power of location equivalence. An example is provided to show that the grape distributed bisimulation is coarser than location equivalence. Indeed, as observed by Kiehn in [94], this example illustrates quite clearly that using a pair of locations (which is essentially what distributed bisimulation does) is not enough to capture location equivalence. It is also argued by Kiehn that by suitably generalising this example, one could show that *no finite number of locations* would be enough to capture location equivalence.

Since Kiehn's example from [95] may also be used to show that generalised distributed bisimulation is weaker than loose location equivalence, which in turn is weaker than location equivalence, the relationships among these three equivalences can be sumarised as follows: for finite and restriction–free processes of $\mathrm{CPP_f}$ they all coincide; on the whole language CCS, distributed bisimulation equivalence and loose location equivalence are incomparable, while location equivalence is finer than both of them.

3.2.3 Local mixed ordering equivalence

A notion of bisimulation based on canonical static locations[27], for a class of asynchronous transition systems modelling a subset of CCS with guarded sums, was proposed by Mukund and Nielsen in [115]. The resulting equivalence notion is essentially the same as the static location equivalence reviewed in Section 2.4. Indeed, this notion was conjectured in [115] to coincide with the dynamic location equivalence $\approx_\ell^d$, a result which was to be subsumed by those of [37].

Another transition system labelled with canonical static locations, called *spatial transition system*, was defined by Montanari and Yankelevich for the whole language CCS

[27]That is, words over $\{0, 1\}$, see the definition at page 25.

in [167, 114]. Here location transitions are used as a first step to build a second transition system, labelled by *mixed partial orders* (where the ordering is a combination of temporal ordering and local causality), which is then used to define the behavioural equivalence. This equivalence, a variant of the mixed-ordering equivalence of [53] called *local mixed-ordering equivalence*, was shown to coincide with location equivalence, applying techniques similar to those used in Section 2.6. In [167, 114], the authors also consider the equivalence obtained by ignoring the temporal ordering and retaining only the local causality ordering: this equivalence, which they call *abstract* location equivalence and provide with some intuitive justification, does not seem to coincide with any other known notion. The *local mixed-ordering equivalence* is obtained in [114] as an instance of a parametric approach; it thus inherits some general results of this approach, such as a complete axiomatisation [64].

We shall not elaborate further on the notions of [115] and [114] here, and refer the reader to these papers for further details. Let us only note that in both cases, since canonical locations can be extracted from the syntax of terms, there is no need to extend the language with a location prefixing construct. A disadvantage of such implicit locations, on the other hand, could be the difficulty of adapting the semantics to allow the placing of several parallel components in the same location (see discussion at page 31).

Another reformulation of the location semantics of [28] was given by Corradini and De Nicola [47], within the grape semantics of [54]. The resulting equivalence is called *maximal distribution equivalence* and shown to coincide with location equivalence. In fact, their formulation is conceptually very close to the original one, once it is observed that sequential components themselves (*grapes* in that approach) can be used to represent localities.

3.3 Comparison with causal semantics

We compare here the location semantics, which is designed to reflect distribution in space, with a number of semantics aimed at reflecting *global causality*. This notion of causality covers both the local causality determined by the structure of terms, and the *inherited causality* induced by the flow of control among different components, which may vary according to the computations. Semantics which are based on such notion of global causality will be collectively referred to as *causal semantics*.

We shall see that, in general, equivalence notions based on causal semantics, such as pomset bisimulation and causal bisimulation, are incomparable with equivalences based on distribution. This is already apparent from the characterisations given by Kiehn [97], reviewed in the previous section. On the other hand, on the sublanguage BPP all these equivalences reduce to the same one, apart from pomset bisimulation which is less discriminating than the others.

3.3.1 Pomset bisimulation

We relate here the location equivalence $\approx_\ell$ and preorder $\sqsubseteq\!\!\!\!\!\!\sim_\ell$ with similar notions associated with *pomset semantics*. This semantics, where transitions are labelled with partially ordered multisets (pomsets) of actions, was studied in [22, 23] for a subset of CCS without communication and restriction, essentially the class BPP (in fact, a variation of BPP with full sequential composition in place of action prefixing), and extended to the whole language in [24]. The partial ordering semantics proposed by Degano, De Nicola and Montanari in [57, 58, 52] were also designed to generate pomset transitions for CCS.

On the language BPP, the pomset semantics can be defined directly by means of structural rules. We give here the rules for finite processes:

(P1)	$a.\,p \xrightarrow{a} p$		
(P2)	$p \xrightarrow{u} p'$	$\Rightarrow$	$a.\,p \xrightarrow{a.\,u} p'$
(P3)	$p \xrightarrow{u} p'$	$\Rightarrow$	$p + q \xrightarrow{u} p'$ $q + p \xrightarrow{u} p'$
(P4)	$p \xrightarrow{u} p'$	$\Rightarrow$	$p \mid q \xrightarrow{u} p' \mid q$ $q \mid p \xrightarrow{u} q \mid p'$
(P5)	$p \xrightarrow{u} p'\,,\ q \xrightarrow{v} q'$	$\Rightarrow$	$p \mid q \xrightarrow{u \mid v} p' \mid q'$

Figure 7: Pomset transitions

Note how rules (P2) and (P5) are used to build pomsets u, v of actions on top of transitions. As an operation on pomsets, prefixing $a.\,u$ is interpreted as generating causality between a and the actions of u, and parallel composition $u \mid v$ as generating concurrency between the actions of u and the actions of v.

The *pomset bisimulation* on BPP, noted $\sim_{pom}$, is just the ordinary bisimulation equivalence associated with this transition system [22]. Weak transitions can then be defined by restricting to visible actions the pomsets labelling the strong transitions. The corresponding weak pomset bisimulation is written $\approx_{pom}$.

We have seen in the previous section that on BPP location equivalence coincides with distributed bisimulation, which in turn was known from [36] to be finer than pomset bisimulation on this language. The following is an example of processes which are distinguished by location (and distributed) bisimulation but equated by pomset bisimulation.

Example 3.13

$$p \;=\; a.\,(b\,+\,c) \;+\; (a \mid b) \quad \begin{array}{c}\not\approx_{\ell}\\ \approx_{pom}\end{array} \quad a.\,(b\,+\,c) \;+\; (a \mid b) \;+\; (a.\,b) \;=\; q$$

These two processes are pomset bisimilar because the behaviour of the additional summand of q can be simulated by p: the transition $q \xrightarrow{a} b$ can be matched by the summand $(a \mid b)$ of p, while the transition $q \xrightarrow{a.\,b} nil$ is matched by the first summand of p.

Using the result of [36], the situation can be summarised as follows.

Proposition 3.14 *Let $p, q \in BPP$. Then $p \approx_\ell q \;\Rightarrow\; p \approx_{pom} q$ but $p \approx_{pom} q \not\Rightarrow\; p \approx_\ell q$.*

Intuitively, location and distributed bisimulation are stronger than pomset bisimulation on BPP because they also capture the property that matching computations should always be *extensions* of smaller matching computations.

As soon as communication and restriction are introduced in the language, location equivalence becomes incomparable with pomset bisimulation. We have for instance:

Example 3.15

$$p \;=\; (a.\,\alpha + b.\,\beta \mid \bar{\alpha}.\,b + \bar{\beta}.\,a) \quad \begin{array}{c}\approx_\ell \\ \not\approx_{pom}\end{array} \quad (a.\,\alpha + b.\,\beta \mid \bar{\alpha}.\,b + \bar{\beta}.\,a) \;+\; (a \mid b) \;=\; q$$

Similar examples were given in Section 3.2 to differentiate local cause equivalence from global cause equivalence (Example 3.5). These examples also hold for pomset bisimulation, replacing $\approx_{gc}$ with $\approx_{pom}$. For general CCS processes we thus have the following.

Proposition 3.16 *Let $p, q \in$ CCS. Then $p \approx_\ell q \not\Rightarrow\; p \approx_{pom} q$ and $p \approx_{pom} q \not\Rightarrow\; p \approx_\ell q$.*

Let us now turn to the preorder relations. To our knowledge the first definition of a preorder expressing the idea that one process is more sequential than another was proposed by Aceto in [1] for a subset of CCS. This preorder is based on the pomset transition semantics: one process is more sequential than another, in notation $p \sqsubseteq\!\!\!\!\sim_{pom} q$, when the pomsets labelling the transitions of p are more sequential than those labelling the transitions of q (this "more sequential than" ordering on pomsets had already been introduced by Grabowski [85] and Gischer [71]). Thus the intuition underlying Aceto's preorder is different from that of the location preorder $\sqsubseteq\!\!\!\!\sim_\ell$, in the same way as pomset bisimulation is different from location equivalence.

Indeed, for the preorders we have a similar situation as for the equivalences: the causality-based preorder $\sqsubseteq\!\!\!\!\sim_{pom}$ and the distribution-based preorder $\sqsubseteq\!\!\!\!\sim_\ell$ are already differentiated on BPP. The following example, suggested by Aceto and already reported in [28], shows that $\sqsubseteq\!\!\!\!\sim_{pom} \not\subseteq \sqsubseteq\!\!\!\!\sim_\ell$. Let:

$$\begin{aligned} p \;&=\; a.\,b.\,c \;+\; c.\,a.\,b \;+\; (a \mid b \mid c) \\ q \;&=\; p \;+\; a.\,b \mid c \end{aligned}$$

Then $p \sqsubseteq\!\!\!\!\sim_{pom} q$ but $p \not\sqsubseteq\!\!\!\!\sim_\ell q$ (while we have both $q \sqsubseteq\!\!\!\!\sim_{pom} p$ and $q \sqsubseteq\!\!\!\!\sim_\ell p$). To see why $p \not\sqsubseteq\!\!\!\!\sim_\ell q$ consider the move $q \xrightarrow[l]{a} l :: b \mid c$, due to the summand $a.\,b \mid c$ of q. Now p has two ways to match this transition, namely $p \xrightarrow[l]{a} l :: b.\,c$ and $p \xrightarrow[l]{a} l :: nil \mid b \mid c$, but neither of them is appropriate. Another example showing that $p \sqsubseteq\!\!\!\!\sim_{pom} q \not\Rightarrow p \sqsubseteq\!\!\!\!\sim_\ell q$ is Example 3.13.

On BPP, it would be plausible that $p \sqsubseteq\!\!\!\!\sim_\ell q$ implies $p \sqsubseteq\!\!\!\!\sim_{pom} q$ (although this has not been shown, to our knowledge). On full CCS, the two preorders $\sqsubseteq\!\!\!\!\sim_{pom}$ and $\sqsubseteq\!\!\!\!\sim_\ell$ are incomparable. Example 3.15 can be reused here. For the two processes p and q of this example we have $q \sqsubseteq\!\!\!\!\sim_\ell p$ but $q \not\sqsubseteq\!\!\!\!\sim_{pom} p$. In conclusion we have:

Proposition 3.17 *Let $p, q \in CCS$. Then $p \sqsubseteq_{\ell} q \not\Rightarrow p \sqsubseteq_{pom} q$ and $p \sqsubseteq_{pom} q \not\Rightarrow p \sqsubseteq_{\ell} q$.*

3.3.2 Causal bisimulation

The causal bisimulation semantics was introduced by Darondeau and Degano in [49], and further investigated in relation to action refinement in [51]. The causal transition system is again defined on an extended language. The new construct here has the form $K \Rightarrow p$, representing process p with set of (global) causes K. The set K is the exact analogue of the global cause set Γ in the *lgc*-transition system, which in fact was specifically designed to encompass both the causal and the location transition systems.

As stated already in section 3.2, causal bisimulation coincides with the global cause equivalence of Kiehn. Although the language is essentially the same as that considered by Kiehn, the transition systems present a number of differences.

The first difference is that causes are introduced here in the form of *backward pointers* in a computation. A term $K \Rightarrow p$ represents a CCS process in some computation state, where it has "accumulated" the causes K. A process in its initial state has the form $\emptyset \Rightarrow p$. In $K \Rightarrow p$ the set K consists of numbers $n, m, \ldots$ representing the positions of the causes of p (as backward displacements along the computation). This backward referencing requires an updating of pointers at each step, which is not needed in the *gc*-transition system.

Assuming, for the sake of comparison, that backward pointers are replaced by uniquely named causes, as in Kiehn's system, there remain a couple of differences between the two semantics. In the causal transition system the new cause introduced for an action does not appear in the label of the transition, but only in the resulting state. Instead of rules (LG1), (LG2), we have the following rule

$$\text{(C1)} \quad K \Rightarrow a.p \xrightarrow{a,K} K \cup \{a_i\} \Rightarrow p, \qquad\qquad a_i \notin K$$

Finally, the rule for communication resorts here to an auxiliary transition system, whose transitions have the form

$$p \xrightarrow{a,K,K'} p'$$

Intuitively, the set K' represents new causes which may be inherited from another component via a communication. These new causes are "guessed" by a kind of early instantiation scheme through the rule:

$$\text{(CC1)} \quad K \Rightarrow a.p \xrightarrow{a,K,K'} K \cup K' \Rightarrow p$$

Note that no new cause is created here to represent the action a in the rest of the computation. This is because the transitions $p \xrightarrow{a,K,K'} p'$ are only introduced in order to be used in the communication rule:

$$\text{(C2)} \quad p \xrightarrow{a,K,K'} p',\ q \xrightarrow{a,K',K} q' \qquad \text{imply} \qquad p \mid q \xrightarrow{\tau} p' \mid q'$$

The remaining rules are essentially the same as in the global cause transition system.

Because of these few differences in formulation, the coincidence of $\sim_{gc}$ and $\approx_c$ is not completely straightforward, although intuitively expected. This coincidence allows us to merge results obtained for the two equivalences. For instance, the examples given in Section 3.2 to differentiate local and global cause equivalence may be applied to location and causal bisimulation. A similar example is the following. Let $r = (a.\,\alpha{+}b.\,\beta \mid \bar{\alpha}.\,b{+}\bar{\beta}.\,a)$. Then

$$\begin{array}{ccccc} r + (a \mid b) & \approx_\ell & r & \not\approx_\ell & r + a.\,b \\ & \not\approx_c & & \approx_c & \end{array}$$

Note also that

$$(a \mid b) \quad \approx_\ell \quad (r\backslash\alpha,\beta) \quad \approx_c \quad a.\,b + b.\,a$$

These absorption phenomena, resp. of $(a \mid b)$ in r w.r.t. $\approx_\ell$, and of $a.\,b$ in r w.r.t. $\approx_c$, clearly show the difference between the two equivalences: the former equates processes with the same parallel structure, while the latter equates processes with the same causal structure. Another interesting example, involving the restriction operator, is the following

$$p = (a.\,\alpha.\,c \mid b.\,\bar{\alpha}.\,d)\backslash\alpha \quad \begin{array}{c}\approx_c \\ \\ \not\approx_\ell\end{array} \quad (a.\,\alpha.\,d \mid b.\,\bar{\alpha}.\,c)\backslash\alpha = q$$

Here $p \approx_c q$ since in both processes actions c and d causally depend on a and b. On the other hand $p \not\approx_\ell q$ since in p action d is not spatially dependent upon action a. One can see that in a distributed view, the "cross-causalities" induced by communication are observed as purely temporal dependencies, while in the causal approach they are assimilated to local causalities.

Let us now revisit the mutual exclusion and protocol examples from Section 2. It is easy to see that *Mutex* $\approx_c$ *MSpec* $\approx_c$ *FMutex*, because the synchronizations on p, v in *Mutex* and *FMutex* create cross-causalities between the critical regions of the two competing processes. The same holds for the protocol example, for which we have *PSpec* $\approx_c$ *Protocol*.

Causal bisimulation has been shown to coincide with *history preserving bisimulation* and with an instance of "NMS equivalence" in [76]. The coincidence of causal and distributed bisimulations in absence of communication (on $\mathrm{BPP_f}$) was also shown in [4]. Finally, causal bisimulation has been studied in relation to action refinement in [50, 51]. It was shown to be indeed robust with respect to action refinement.

The model obtained by unfolding the causal transition system, called *causal trees*, has been used to define a direct "denotational" interpretation of CCS terms, which agrees with a standard event structure semantics; it has also been taken as the basis for a complete axiomatisation for causal bisimulation [49]. Indeed, one of the initial motivations of Darondeau and Degano's work was the search for a *tree-model* on which standard results from the usual theory of bisimulation could be carried over.

3.3.3 History-preserving bisimulation

We noted already that the original definitions of history-preserving bisimulation were given on event structures or similar semantic domains. More precisely, history-preserving bisimulation was originally defined in [132] and [73] for prime event structures, and extended in [76] and [3] to flow and stable event structures respectively. More recently, this equivalence has also been studied on *configurations structures* [77]. Two variants of the notion of history preserving bisimulation have also been considered: *hereditary* history preserving bisimulation [15] and *strong* history preserving bisimulation [92]. Both are finer than the original notion.

Using the occurrence transition system of Section 2.6, we may define a notion of *local history preserving bisimulation* for CCS processes. Essentially, a history-preserving bisimulation is a bisimulation which preserves, at each state of computation, the partially ordered set of events that led to that state. Moreover, it also accounts for the way this partial ordering is *extended* along a computation.

The following definition, taken from [37], differs from that of [73] and [3] in two respects: it is "syntactic", in that it is defined directly on (an enrichment of) the CCS transition system, and it is based on the local rather than the global causality ordering.

The occurrence system provides a notion of *state* (or configuration) for CCS terms. For $p \in$ CCS, define:

$$States(p) \quad = \quad \{\, \xi \mid \exists\, e_i, \xi_i \ \text{ s.t. } p \stackrel{\tau}{\Longrightarrow} \xi_0 \stackrel{e_1}{\Longrightarrow} \cdots \stackrel{e_n}{\Longrightarrow} \xi_n = \xi \,\}$$

Recall that each state ξ has an associated set of events $occ\,(\xi)$, ordered by the local causality relation $\preceq$. Unlike the global causality ordering in flow and stable event structures, which is relative to a configuration, the local causality ordering $\preceq$, which is essentially a static notion, is the same for all computations.

Definition 3.18 (Local history preserving bisimulation)

Let $p, q \in$ CCS. A relation $R \subseteq\ States(p) \times States(q) \times \wp(\,occ\,(States(p)) \times occ\,(States(q))\,)$ is a *local history preserving bisimulation (lhp-bisimulation)* between p and q if $(p, q, \emptyset) \in R$ and whenever $(\xi_0, \xi'_0, g) \in R$ then:

(1) g is an isomorphism between $(\,occ\,(\xi_0),\ \preceq)$ and $(\,occ\,(\xi'_0),\ \preceq)$

(2) *a)* $\xi_0 \stackrel{e}{\Longrightarrow} \xi_1 \ \Rightarrow\ \exists\, e', \xi'_1$ s.t. $\xi'_0 \stackrel{e'}{\Longrightarrow} \xi'_1$ and $(\xi_1, \xi'_1, g \cup (e, e')) \in R$

b) $\xi'_0 \stackrel{e'}{\Longrightarrow} \xi'_1 \ \Rightarrow\ \exists\, e, \xi_1$ s.t. $\xi_0 \stackrel{e}{\Longrightarrow} \xi_1$ and $(\xi_1, \xi'_1, g \cup (e, e')) \in R$

(3) *a)* $\xi_0 \stackrel{\tau}{\Longrightarrow} \xi_1 \ \Rightarrow\ \exists\, \xi'_1$ s.t. $\xi'_0 \stackrel{\tau}{\Longrightarrow} \xi'_1$ and $(\xi_1, \xi'_1, g) \in R$

b) $\xi'_0 \stackrel{\tau}{\Longrightarrow} \xi'_1 \ \Rightarrow\ \exists\, \xi_1$ s.t. $\xi_0 \stackrel{\tau}{\Longrightarrow} \xi_1$ and $(\xi_1, \xi'_1, g) \in R$

We say that p and q are local history preserving equivalent, $p \approx^{lhp} q$, if there exists a local history preserving bisimulation between them.

We have then the following charcterisation:

Fact 3.19 *For any processes $p, q \in$ CCS:* $p \approx^{lhp} q \ \Leftrightarrow\ p \approx^{occ} q$.

Proof: It should be clear that if $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ is a progressive bisimulation family such that $pR_\emptyset q$, then the relation:

$$S = \{\, (\xi, \xi', g) \mid \xi \in States(p),\ \xi' \in States(q),\ g \in \mathcal{G} \ \text{ and } \ \xi R_g \xi' \,\}$$

is a lhp-bisimulation between p and q. In fact, if $\xi R_g \xi'$ for $\xi \in States(p)$ and $\xi' \in States(q)$, then g is an occurrence aliasing such that $occ(\xi) \subseteq dom(g)$ and $occ(\xi') \subseteq range(g)$, that is an isomorphism between $occ(\xi)$ and $occ(\xi')$.

Similarly, if S is a lhp-bisimulation between p and q then the family $\mathbf{R} = \{R_g \mid g \in \mathcal{G}\}$ defined by:

$$R_g = \{ (\xi, \xi') \mid (\xi, \xi', g) \in S \}$$

is a progressive bisimulation family such that $pR_\emptyset q$. □

In the light of the results of Section 2.6, we have then also:

Corollary 3.20 *For any processes $p, q \in$ CCS: $p \approx_\ell^d q \Leftrightarrow p \approx^{lhp} q \Leftrightarrow p \approx_\ell^s q$.*

A similar notion of *local history preserving preorder*, $\sqsubseteq\!\!\!\!\raisebox{-1ex}{$\sim^{lhp}, can be obtained by requiring g, in Def. 3.18, to be a bijection between $(occ(\xi_0), \preceq)$ and $(occ(\xi'_0), \preceq)$ whose inverse is a homomorphism. This preorder may be shown to coincide with $\sqsubseteq\!\!\!\!\raisebox{-1ex}{$\sim$}_\ell$.

Note that the local history-preserving bisimulation just defined is *weak*, in the sense that it ignores τ-actions. This distinguishes it from the other notions of history-preserving bisimulation, which are *strong*. Indeed, as noted already in Section 2.1, the notion of history-preserving bisimulation was designed to be robust under refinement, and most weak equivalences are not. Nevertheless, it could be worth investigating a similar syntactic definition for the usual notion of (global) history preserving bisimulation, by taking a slightly more concrete notion of occurrence where communications are pairs of visible occurrences, and adopting the corresponding global causality ordering (as defined in [26]).

3.4 Extension to the π-calculus

In this section we give an informal account of Sangiorgi's extension of the location equivalence $\approx_\ell$ to the π-calculus, as presented in [141]. An important result of this research is that the location machinery of [28] can be *implemented* in the π-calculus by means of a fully abstract encoding, in such a way that the study of location equivalence $\approx_\ell$ reduces to that of ordinary bisimulation equivalence.

The π-calculus is an extension of CCS proposed by Milner, Parrow and Walker in [111] to deal with mobile processes, building on earlier work by Engberg and Nielsen [62]. In the π-calculus, processes interact by exchanging *channel names* (also called simply *names*) over channels. In particular, a restricted name can be sent outside its original scope, which is then enlarged to include the receiver process, a phenomenon known as *scope extrusion*. The combination of channel transmission and scope extrusion is the most powerful feature of the π-calculus with respect to its predecessor CCS. Using these mechanisms, it is possible to model the creation of *unique* names and simulate the transmission of resources and processes. For a formal definition of the π-calculus, the reader is referred to Parrow's chapter in this issue. The syntax of (a core version of) the language may also be found at page 71.

The language used by Sangiorgi is in fact the *polyadic* variant of the π-calculus, proposed by Milner in [109], which allows names to be transmitted in tuples rather than one at a time. This variant is particularly useful for writing programming examples as well as encoding other languages.

The version of location equivalence used by Sangiorgi is a generalisation of the *dynamic location equivalence* $\approx_\ell^d$ examined in Section 2.5 (hereafter referred to simply as $\approx_\ell$). To accomodate the definition of location equivalence, the syntax and operational semantics of the π-calculus are enriched with location names, exactly in the same way as for the language LCCS examined in Section 2. This adaptation can be done quite smoothly, as name-passing does not introduce additional complications in this respect.

We shall now give an outline of Sangiorgi's encoding, which maps *located* π-processes into more complex π-processes that incorporate the location information. This encoding exploits in a crucial way the π-calculus ability to generate and transmit unique names. We shall use $\mathcal{E}[\![P]\!]$ to denote the encoding of the located process P. As can be expected, locations will be represented by names in $\mathcal{E}[\![P]\!]$. Let a, b, x, y, z range over names, and $\langle\tilde{a}\rangle$ represent a tuple of names $a_1, \ldots, a_k$. The idea is the following.

Suppose that $P \xrightarrow[ul]{\overline{a}\langle\tilde{b}\rangle}_d P'$. Then $\mathcal{E}[\![P]\!]$ will send an additional name x which will serve as a "handle" for the observer, enabling him to access both the location ul and the causal path u of action $\overline{a}$. Assume that P is located at y. Then we will have approximately:

$$\mathcal{E}[\![P]\!] \xrightarrow{\overline{a}\langle\tilde{b}x\rangle} (\nu z)(\, Loc\langle xzy\rangle \mid \mathcal{E}[\![P']\!]\,) \tag{12}$$

where the process $Loc\langle xzy\rangle$ implements the location mechanism and can be interrogated by the observer using name x. Here, y is the name representing the causal path u of action $\overline{a}$, while z represents the location ul of the derivative P' in $\mathcal{E}[\![P']\!]$. The process $Loc\langle xzy\rangle$ is defined by:

$$Loc\langle xzy\rangle \;=\; \overline{x}\langle z\rangle \mid {!}\, z.\, \overline{y}$$

In fact, to avoid interference with the environment, the name x needs to be unique in (12).

We give now the formal encoding for input and output processes, where the parameter y represents the location u of the encoded process (which is initially the empty string ε):

$$\mathcal{E}[\![\overline{a}\langle\tilde{b}\rangle.\, P]\!]\langle y\rangle \;=\; (\nu x)\, \overline{a}\langle\tilde{b}x\rangle.\, (\nu z)(\, Loc\langle xzy\rangle \mid \mathcal{E}[\![P']\!]\langle z\rangle\,) \tag{13}$$

$$\mathcal{E}[\![a(\tilde{b}).\, P]\!]\langle y\rangle \;=\; a(\tilde{b}x).\, (\nu z)(\, Loc\langle xzy\rangle \mid \mathcal{E}[\![P]\!]'\langle z\rangle\,) \tag{14}$$

These two clauses give the essence of the translation; further details are omitted. The main result is that this encoding is *fully abstract*, as expressed by the following theorem by Sangiorgi [141]. Here $\approx^c$ and $\approx_\ell^c$ are the congruences associated respectively with the early bisimulation equivalence of the π-calculus and with the location equivalence $\approx_\ell$.

Theorem 3.21 *For any located π-calculus processes P, Q, $P \approx_\ell^c Q \Leftrightarrow \mathcal{E}[\![P]\!] \approx^c \mathcal{E}[\![Q]\!]$.*

Note that this is essentially an expressiveness result about the π-calculus. In this sense it falls in the same class as the results by Milner and Sangiorgi [110, 139] on the encoding of the λ-calculus and higher-order calculi, and that by Walker [159] on the encoding of object-oriented languages.

4 A concrete view of locations

In this section we consider a *concrete* approach to the use of localities in process algebras, where *location identities* are significant and cannot be abstracted away as in the abstract approaches examined in Section 2. In the abstract approaches locations were used essentially to *differentiate* parallel components. Their exact names were immaterial, as long as they bore the same dependence or independence relation with each other. In the concrete approach we shall describe now, on the other hand, location names are important; indeed the topmost locations of a process may be viewed here as *physical addresses* in a network.

4.1 A concrete location semantics for CCS

We start with the basic and most concrete approach to processes with explicit locations, as was originally proposed by Murphy in [116] and further pursued by Corradini and Murphy in [48] and [45]. Although this concrete approach might be seen as the first step towards a theory of processes with locations, we postponed its presentation because it is easier to formalise once the general set-up of location equivalence has been understood. Indeed, this approach followed, chronologically, the more abstract approaches exposed in the previous sections. It also responds to a rather different concern: that of placing parallel processes onto a network of processors with a fixed geometry. In this context, the identities of locations are significant, as they may refer to physical processors. Accordingly, the semantics is very concrete: processes are equated if they reside on the same set of locations and present the same behaviour at each location.

The language considered in [116] and [48, 45] is a simple subset of CCS, essentially a finite combination of regular processes by means of the static operators of CCS, parallel composition, restriction and renaming[28]. This is the same class of processes considered by Aceto in [5] under the name *nets of automata*, and we shall retain this terminology here. In this language all the occurrences of the parallel operator appear at top level, and locations are assigned to parallel components in the same way as in the static approach examined in Section 2.4, using the location prefix construct $\ell :: p$. The only difference is that here the two operands of a parallel composition are allowed to be placed at the same location. This is required for solving the placing problem, in case the number of processes is larger than that of processors. It is then immediately clear that locations will not serve as a measure of parallelism in this model. Also, since there is no nesting of parallelism in nets of automata, the structure of locations will be *flat*. Formally, we have the following two-level syntax.

The set of *sequential processes* is given by the grammar:

$$p ::= nil \mid \mu.p \mid (p+p) \mid x \mid rec\, x.\, p$$

Then the set of *(distributed) nets of automata* is given by:

$$n ::= l :: p \mid (n \mid n) \mid n \backslash \alpha \mid n \langle f \rangle$$

[28]The language studied in [116] and [48] is in fact a slight variant of CCS, with an asymmetric notion of communication where reception is directed (tagged with the location from which the value is expected to come) while emission is not. This choice was motivated by a similar asymmetry present in several synchronisation protocols. Here, for the sake of simplicity and ease of comparison with other approaches, we take a more standard CCS-like language, as was done already by Corradini in [45].

In concentrating on this simple language, the authors were motivated by implementation concerns: they wanted to rule out distributed choices like $(\ell :: p + k :: q)$, and also terms of the form $(\ell :: p \parallel k :: q) + r$, since in both cases a synchronisation between possibly distant sites is required.

The original work [116] presents a strong transition semantics for this language, with the associated strong bisimulation, while [48] and [45] consider the corresponding weak semantics, abstracting from τ-transitions. We shall mainly consider the latter here.

Formally, the semantics of nets of automata is given by the same rules used for static location transitions in Figure 1: the transitions are simply indexed by the location at which they occur. Then the *weak located equivalence* of Corradini and Murphy [48, 45], which we denote here by $\approx_{CM}$, is defined as the ordinary weak bisimulation equivalence associated with these transitions. We have for instance:

$$
\begin{array}{lrcl}
(15) & (\ell :: a \parallel \ell :: b) & \approx_{CM} & \ell :: (a.b + b.a) \\
(16) & (\ell :: a \parallel k :: b) & \not\approx_{CM} & (\ell :: b \parallel k :: a) \\
(17) & (\ell :: (\alpha + b) \mid k :: \bar{\alpha}.\, b)\backslash\alpha & \not\approx_{CM} & \ell :: b
\end{array}
$$

Note that the nets of examples (16) and (17) are equated by location equivalence $\approx_\ell$ (we refer here to the static location equivalence $\approx_\ell^s$ defined in Section 2). As for the nets of example (15), these are *not* admissible distributed processes in the static approach of Section 2.4, and therefore $\approx_\ell$ is not defined on them[29].

In fact, it can be easily shown that the equivalence $\approx_{CM}$ is *stronger* than location equivalence $\approx_\ell$ on the intersection of the two languages, that is, on the language of nets of automata where parallel components have different locations). A proof of this fact may be found in [48]. In the light of this result it is fair to say that the model of Murphy and Corradini reflects a *concrete view of localities.*

It may be remarked that this concrete model could very nearly be simulated in ordinary CCS, by using the relabelling operator to distinguish actions of different parallel components. This is only approximately true, because one should then adopt a slightly more flexible communication discipline to allow relabelled complementary actions like α_i and $\bar{\alpha}_j$ to synchronise. For instance one could use a *communication function* as in [13] or a *synchronisation algebra* as in [162]. Modulo this little twist in defining communication, one would indeed be able to define a simple encoding $\mathcal{E}[\![n]\!]$ of distributed nets of automata into CCS, such that

$$n_1 \approx_{CM} n_2 \;\;\Leftrightarrow\;\; \mathcal{E}[\![n_1]\!] \approx \mathcal{E}[\![n_2]\!]$$

where $\approx$ is the ordinary weak bisimulation of CCS. Note on the other hand that this would not hold for location equivalence $\approx_\ell$, even in the simple setting of nets of automata, as shown by the example:

$$[\,(a.\,\gamma + b.\,\bar{\gamma}) \mid (\bar{\gamma}.\,b + \gamma.\,a)\,]\backslash\gamma \;\;\approx_\ell\;\; a \mid b$$

It should be clear that any relabelling here would distinguish the two actions a (as well as the two actions b) in the first process, and therefore the two relabelled processes would be not be identified by $\approx$.

[29]Note on the other hand that the dynamic location equivalence $\approx_\ell^d$ could be applied to these terms and would distinguish them.

Concerning located semantics, two additional results were presented in [116] and [48]: a calculus of *efficient placings* of processes into networks of processors, and an axiomatisation, along standard lines, of the weak located equivalence $\approx_{CM}$.

An intermediary location semantics, which could be called "super-static", was also considered by Corradini in [45]: the idea was to use a static assignment of locations, as in the calculus here, but compare transitions modulo a *static* injective renaming of their locations. This semantics would equate the two systems of example (16) above, while still distinguishing the systems of example (17). Indeed, it not difficult to show that the resulting equivalence is strictly included between the located equivalence $\approx_{CM}$ and the location equivalence $\approx_\ell$ of Section 2.

To conclude this section, let us mention another study on distributed CCS conducted by Khrisnan in [93], which is also based on a concrete view of locations. The author considers an extension of CCS with locations and explicit message passing primitives. The semantics he proposes for the language is rather different from those presented here, and we shall not attempt a detailed comparison; it is also more directly driven by practical concerns, like the limitations on the degree of parallelism imposed by a physical architecture. Applications of the calculus to the description of configuration and routing in distributed systems are also discussed.

As pointed out already, the names of localities are important in the concrete approach, since localities are meant to represent physical sites in a distributed architecture. This leads quite naturally to the treatment of locations in more recent and powerful languages for distributed *mobile* computation. These languages will be the subject of Section 4.3.

4.2 The location failure approach

In this section we briefly review the location failure approach, as has been developed by Amadio and Prasad in [11] and further pursued by Amadio in [8], and by Riely and Hennessy in [136]. Location failure has been considered also in other calculi, for instance in the distributed JOIN calculus [69]. In these latter calculi, however, locations have more complex meanings, hence their treatment is deferred to Section 4.3.

The standpoint here is rather different from the one taken so far, where a global observer for a whole distributed system was implicitly assumed, and the issue was to identify the degree of distribution of the system. In the "location failure" approach, one starts from the hypothesis that locations - nodes in a distributed system - are liable to failure. The perspective adopted here is that of an individual programmer or process, wishing to have transparent access to resources irrespective of where they are located, at least as long as these locations are running. On the other hand, some awareness of locations is requested in case of failure: a process that has spawned activities at another location wants to be notified in case of failure of this location, in order to stop interacting with it and transfer activities to other locations.

The language considered in [136] is based on pure CCS with no value-passing, while those of [11, 8] build on more powerful formalisms, respectively the programming language FACILE of [153] and the asynchronous π-calculus [91, 20], which allow for the transmission of processes, respectively names, in communications.

We will concentrate here on [136] and [8], which have been developed independently but appear to share several features. Common to both studies is the idea to enrich the basic language (CCS and the asynchronous π-calculus[30], respectively) with new operators for *spawning* a process at a remote location, *killing* a location and *testing* the status of a location. Locations $\ell \in Loc$ have two possible states, dead and alive. In the case of [8] locations are also transmissible values – in fact they are just a particular category of π-calculus names.

Formally, *basic processes* are built with the standard operators on the underlying calculus together with the following new operators (taking a blending of the constructs of [8] and [136]):

- $\mathsf{spawn}\,(\ell\,, p)$, which creates a new process p at location ℓ;
- $\mathsf{kill}\ \ell$, which kills location ℓ, entailing the termination of all processes running at ℓ;
- $\mathsf{if}\ \ell\ \mathsf{then}\ p\ \mathsf{else}\ q$, which evolves to either p or q, depending on whether ℓ is alive or dead.

Then *distributed processes* are built on top of basic processes using the static operators of the language as well as an operator for *locating* processes, $\ell :: p$, which represents process p running at location ℓ (this is analogous to the construct encountered in the previous sections, so we use the same notation). We therefore have a two-level syntax similar to that of [116, 48], although with a richer basic language.

The operational semantics, which we shall not report here, is then defined directly on distributed processes: as can be expected it makes use of the additional information about the state of locations (for instance the process $\ell :: p$ can only execute if ℓ is alive), but it is otherwise very close to the standard semantics of the underlying languages. In particular, unlike all the semantics considered so far in this paper, it does not record the location at which actions are performed. In fact, the common intention of the works [8] and [136] is that distribution should be completely *transparent* in the absence of failure.

We should point out, however, that the names of locations are significant in this model, just like in the concrete approach of [116, 48]: for instance the two systems $(\ell :: a \parallel k :: b)$ and $(\ell :: b \parallel k :: a)$, which would be equated by the location equivalence $\approx_\ell$ of Section 2, can be distinguished here by an observer which has the capability of killing ℓ or k.

The semantic notion which is adopted in [8] and [136] is a form of contextual equivalence called *barbed equivalence*, essentially a bisimulation based on silent transitions and basic observations called "barbs", which is further closed with respect to contexts. For more details on this kind of equivalence see Parrow's chapter in this issue. In [136] the barbed equivalence is given a more tractable formulation as a bisimulation relation called *located-failure equivalence*, noted $\approx_{LF}$. An example of distributed processes that are equated by $\approx_{LF}$ is:

$$\begin{array}{rcl} P_1 & = & (\ell :: \bar{\alpha} \parallel k :: (\alpha.b))\backslash\alpha \\ Q_1 & = & \ell :: (\mathsf{spawn}\,(k\,, b))\backslash\alpha \end{array}$$

Intuitively, these are equivalent because even an observer capable of killing ℓ is not able to tell them apart. In [136], located-failure equivalence is also given an alternative symbolic characterisation (along the lines of [88]), which further improves its tractability.

[30]The reader unfamiliar with the π-calculus is referred to Parrow's chapter in this issue. However, knowledge of the π-calculus is not strictly necessary for the discussion in this section.

Given the similarity of the underlying languages, the equivalence $\approx_{LF}$ can be easily compared with the concrete located equivalence $\approx_{CM}$ of [48] and the location equivalence $\approx_\ell$ of [28]. It turns out that $\approx_{LF}$ is neither stronger nor weaker than either of them. Here is an example, taken from [136], of processes which are both $\approx_{CM}$-equivalent and $\approx_\ell$-equivalent but are distinguished by $\approx_{LF}$:

$$\begin{array}{rcl} P_2 & = & (\ell :: \alpha \parallel k :: (\bar{\alpha} + \tau.b))\backslash\alpha \\ Q_2 & = & (\ell :: (\alpha + \tau) \parallel k :: (\bar{\alpha}.b))\backslash\alpha \end{array}$$

These are not $\approx_{LF}$-equivalent because if ℓ is killed by some context, then P_2 is capable of doing a b-transition which Q_2 cannot match. In [136] one may find a slightly larger example, due to F. Corradini, showing that two $\approx_{LF}$-equivalent processes need not be $\approx_{CM}$-equivalent nor $\approx_\ell$-equivalent. Therefore $\approx_{LF}$ reflects indeed a different perception of localities.

The basic language considered by Amadio in [8] is a "disciplined" asynchronous π-calculus where for each channel name there is at most one receiver, which can be viewed as a *server* for that channel. Calculi of this kind are often said to have the *unique receiver property*. To emphasize this property, Amadio calls his language the π_1-calculus, and its located version the $\pi_{1\ell}$-calculus. The $\pi_{1\ell}$-calculus will be further discussed in Section 4.3.

In conclusion of this section, we can remark that in the location-failure approach, locations are *units of failure* as well as units of distribution; and while distribution in itself is not observable, failure may be observed (as the absence of certain communication capabilities) and thus reveal some parts of the distribution.

4.3 Combining locations and mobility

We give now an overview of some recent work combining locations and mobility. Most of this work is still at an early stage and therefore our account will be essentially descriptive, trying to isolate those features that are more relevant to our subject of interest. We will only consider languages which are presented as process algebras or strongly based on them, namely distributed versions of the π-calculus [111] and the JOIN calculus [68], the Nomadic π-calculus of [147], and new calculi expressly designed with distribution and security concerns in mind, such as the AMBIENT calculus of Cardelli and Gordon [32] and the SEAL calculus of Castagna and Vitek [157]. We shall also briefly touch upon a distributed calculus based on the coordination language Linda, the calculus LLINDA of De Nicola, Ferrari and Pugliese [117].

It can be noted [144] that while most early work on process calculi was centered around modelling protocols (or other concurrent systems) and reasoning about them, these new process calculi have been more specifically designed to be the basis of programming languages. Indeed, a few of them have already given rise to prototypical programming languages such as the Join Calculus Language [70], NOMADIC PICT [148] and KLAIM [118].

With the advent of the *network* as a computing support, *mobility* is bound to play an important role in programming. However theoretical models for mobile computation, such as the π-calculus [111] (see Chapter 3.2 in this issue), were already proposed and thoroughly studied before the wide generalisation of network programming that we witness today. This partly explains why, although the very idea of mobility seems to presuppose the existence of different sites among which to move, the notion of *distribution* among sites was not originally present in calculi for mobile processes. Only very recently have "distributed versions" of these calculi been defined and started to be investigated. In fact, there are two aspects to the notion of mobility as it is currently used:

(1) the *transmission* of processes or links to processes as *arguments* of communication along channels. This form of mobility, which could perhaps be better qualified as "process-passing" and "name-passing" (or "link mobility"), is exemplified by the π-calculus and its higher-order variants, CHOCS [152] and HOPI [139, 140], as well as by the more practical languages derived from them such as FACILE [153] and PICT [128].

(2) the actual *movement* of processes from one place to another, what is commonly called *migration*. This is the kind of mobility emphasized in formalisms such as the AMBIENT calculus and the SEAL calculus, and also accounted for in the distributed JOIN calculus [69, 67] and in distributed versions of the π-calculus such as [8, 145, 89] and NOMADIC PICT [147, 148].

Strictly speaking, only the second form of mobility requires an explicit distribution of processes among localities, as a kind of "reference frame" for the movement to take place. For the first form, it may indeed be sufficient to assume an implicit distribution given by the parallel structure of processes, since mobility is here only a facet of communication.

To fix ideas, we will say that a model or language is "distribution-oriented" or simply "distributed" if it explicitly integrates some notion of *site*, *locality* or *domain*. In this sense, current developments of the π-calculus such as Amadio's $\pi_{1\ell}$-calculus [8], the distributed π-calculi *dpi* of Sewell [145] and $D\pi$ of Hennessy and Riely [137, 89], the distributed JOIN calculus [69, 67] and NOMADIC PICT [148] are distributed, and so are newly proposed calculi such as the AMBIENT calculus [32] and the SEAL calculus [157]. These are essentially all the models we shall examine in this section. Now, what exactly is a locality in these models? According to the model localities may have different attributes, but some common features can be identified:

- A locality is the host of a number of processes and resources. Sometimes, the locality itself is assimilated to a particular process (the "locality process" of the $\pi_{1\ell}$-calculus) which plays the role of a controller for the processes in its scope.
- Localities can be observed or controlled. The need for controlling localities is often motivated by failure detection or security concerns. The observation of localities is required to implement migration.
- A locality can be named, and as such designated as the target of a communication (if communication across localities is allowed) or the destination of a migration.

Moreover, localities have a bearing on the handling of communication. In a distributed calculus there are two forms of communication: *local communication*, between processes at the same locality, and *distant communication*, between processes at different localities.

Distant communication can be transparent - in the sense that distant destinations are reachable as if they were local (distributed JOIN calculus, Amadio's calculus, Sewell's *dpi*-calculus), restricted to the immediate neighbourhood (SEAL calculus), or even completely forbidden (AMBIENT calculus, Hennessy and Riely's $D\pi$-calculus). In the last case only local communication is possible, and processes have to convene in a common location in order to communicate. Typically, a process wishing to send a message will have to move to the location of the destination process. Clearly this choice only makes sense in calculi with a migration primitive, where distant communication can be simulated.

Depending on the form of distant communication allowed, one can distinguish between *global communication*, *proximity communication* and *purely local communication* models. In most of these models both local and distant communication are *asynchronous* (in the sense that the sending of a message is non-blocking[31], see also Chapter 3.2 in this issue). In general, distributed calculi have tended to adopt asynchronous rather than synchronous communication, as the former seems more primitive, easier to implement, and closer to the asynchronous message delivery mechanisms of actual networks. In the SEAL calculus, on the other hand, both local and distant communication are synchronous. This choice may be due to the fact that the primary concern of this calculus being security, a more rigid communication discipline is preferred (the implementation issue is less crucial here since only short-range communication is allowed). Similarly, communication is synchronous in the $D\pi$-calculus of Hennessy and Riely, where only local communication is allowed.

The structure of locations can be *flat* or *hierarchical*, in which case they can be organised in two levels or arbitrarily nested in a tree. In a hierarchy of locations, the first level may be seen as representing physical machines[32] while lower levels correspond to software agents. Locations are flat in Amadio's $\pi_{1\ell}$-calculus, in Hennessy and Riely's $D\pi$-calculus, and in NOMADIC PICT. They have a two-level structure in Sewell's *dpi*-calculus and a tree structure in the distributed JOIN calculus, the AMBIENT calculus and the SEAL calculus. Hierarchical locations have a relevance in the presence of failures, since the failure of a location entails the failure of all its sublocations. In a hierarchy of locations, moreover, locations may be themselves migrating entities: in this case a migrating location will bring together its sublocations. Structured locations also allow for a finer control of mobility and a more flexible handling of security issues when modelling networks partitioned into administrative or "trust" domains.

Locations are the reference frame for migration. Migration is intended as the movement of processes or some other entity having executable content (such as locations themselves), as opposed to simple data transfer. One can distinguish between *subjective* and *objective* migration. In the first case the migration instruction comes from the migrating entity itself (possibly from a process enclosed in it, if this entity is for instance a location), while in the second it is outside its control.

A further classification can be made into *strong* and *weak* migration. In the first case a process can be interrupted during its execution and moved together with its current environment to another location where execution is pursued. In this case, where "active" code can be moved, one often speaks of *migrating agent*. In the case of weak mobility, instead, only "passive" code can be moved. Finally, strong migration can be *blocking* for the migrating entity, in the sense that this has to suspend execution until the migration is

[31] This is not to be confused with the use of "asynchronous" and "synchronous" in reference to Milner's calculi CCS and SCCS (Synchronous CCS) [107]. In SCCS synchrony means that parallel components proceed together in lock-step.

[32] Although, strictly speaking, this would imply a different treatment of topmost locations, to forbid them to move or be dynamically generated. This issue is explicitly addressed only in NOMADIC PICT, where the flat space of locations is fixed from the start, and the *dpi*-calculus, where new locations may only be created as sublocations of existing ones.

completed, or *asynchronous*, as is the case for the migration of localities, which can still receive messages while being subject to a migration instruction.

In the next sections, we shall examine in some detail a selection of these models for distributed mobile computation.

4.3.1 Distributed π-calculi

The π-calculus does not have an explicit notion of location, although its constructs are powerful enough, as shown in Section 2.6, to simulate the location machinery of [28]. Indeed the π-calculus, while being one of the most influential foundational calculi for mobility, does not provide for direct process mobility. The movement of processes is represented indirectly via the movement of channels that refer to them: roughly speaking, two processes move close to each other when they acquire a common channel. It could then be natural to view the notion of *name scoping* as an abstraction of that of location. However this analogy cannot be brought very far, since processes may share several names with other processes. For instance one cannot define two processes to be co-located whenever they share a private name, since this notion of "co-localization" would not be transitive. A more convincing interpretation of name scoping would then be as "trust domains" (that is, groups of processes that trust each other), since these are allowed to overlap and membership to a common trust domain is not expected to be transitive.

To directly model distributed aspects such as migration of agents and failure of machines, one must add to the π-calculus primitives for grouping processes into units of migration or units of failure. An early proposal was presented by Amadio and Prasad in [11], to model the distributed module of the FACILE programming language [153]. More recently, enrichments of the π-calculus with explicit locations have been studied by Amadio [8], Hennessy and Riely [137, 89, 90, 138], Sewell [145], and Sewell et al. [147, 148]. The emphasis in this work is in developing type systems for mobile distributed computation, building on existing type systems for the π-calculus. These type systems may serve a variety of purposes. For instance, Amadio's type system is used to ensure the uniqueness of receptors on communication channels. The type systems of Hennessy and Riely's [137, 89] aim at regulating the use of resources according to permissions expressed by types; these type systems are further refined in [90, 138], to deal with the case where some components of the network, representing malicious agents, may not be assumed to be well-typed. The goal of Sewell's type system [145] is to distinguish local and remote channels in order to allow efficient implementation of communication. The language Nomadic Pict [147, 148] stands a little aside from this track: it is conceived as a distributed layer over the language Pict, offering high-level global communication while providing at the same time its implementation in terms of low-level located communication.

Most of these distributed π-calculi use a flat space of locations, and a primitive for the migration of processes. In the $D\pi$-calculus, this primitive is coupled with a purely local communication mechanism, while in the other calculi communication is global. Other models like the distributed JOIN calculus, the SEAL calculus and the AMBIENT calculus, although strongly inspired by the π-calculus, have rather distinct characteristics. Here the space of locations is tree–structured, and the hierarchy is relevant for migration as the units of migration are locations. In these calculi the emphasis is put respectively on implementation, security and administration[33] issues. Regulation on the use of channels and migration is imposed mostly by syntactic means. Type systems have not been a main concern in these models, although they are starting to be actively investigated for the AMBIENT calculus.

We proceed now to a more detailed description of the various models. Most of them are based on the mini π-calculus studied by Milner in [110], or on its asynchronous version [20, 91]. For convenience we recall here the syntax of these calculi, referring to Parrow's chapter in this issue for more details. The syntax of the mini π-calculus is given by:

$$P, Q \ ::= \ 0 \ \mid \ x(u).\,P \ \mid \ \overline{x}\langle u\rangle.\,P \ \mid \ P \mid Q \ \mid \ (\nu x)P \ \mid !\,P$$

The asynchronous π-calculus is obtained by replacing the output prefix construct $\overline{x}\langle u\rangle.\,P$ by simple messages $\overline{x}\langle u\rangle$, thus preventing output actions from blocking a process. Syntactic variants may be used: for instance replication may be replaced by general recursion or by *replicated inputs*, that is processes of the form $!\,x(u).\,P$. The *polyadic* version may be preferred, where input and output prefixes are generalised to allow transmission of tuples of names, noted $x(\overrightarrow{y})$ and $\overline{x}\langle\overrightarrow{y}\rangle$.

The semantics of these calculi is given by first defining a structural congruence $\equiv$, which is used to predispose terms for communication, and then a set of *reduction rules* describing process execution. This kind of semantics, which factors out structural rearrangements of terms, was first proposed by Milner in [110], taking inspiration from the *Chemical Abstract Machine (CHAM)* style of semantics advocated by Berry and Boudol [17].

The $\pi_{1\ell}$-calculus

The first extension of the π-calculus with locations was proposed by Amadio and Prasad in [11], to model an abstraction of the failure semantics of the language FACILE. The language considered in that paper, called the π_ℓ-calculus, is based on a variant of the synchronous polyadic π-calculus with guarded sums; moreover, a *flat* space of locations is introduced, and failure primitives similar to those described in Section 4.2 are provided. We shall not describe the calculus in detail, but only mention its main features, some of which reappear in later calculi.

In the π_ℓ-calculus *locations* are added as a new category of *names*, which can be dynamically created and transmitted in communication; all channels and processes are *located*. The reduction semantics of the π_ℓ-calculus is defined relative to an environment $\mathcal{L}$, describing the assignment of locations to channels. Communication is *global*: it may occur between any two complementary channels irrespective of their location. In general, locations intervene only in the failure semantics of the language. Although all locations are "failure-independent" from each other (since they are unstructured), channels are dependent on the location where they are placed: the failure of a location prevents any further communication on its channels.

The abstract semantics of the calculus is given by defining a *barbed equivalence* (see Parrow's chapter for a definition of this notion) on configurations $(\mathcal{L}, p)$. Although the observation with failures is rather different from the usual observation of the π-calculus, an encoding of the π_ℓ-calculus into the π-calculus is given, which is proved adequate with respect to the barbed equivalences of the two languages.

[33] In the sense of modelling navigation through administrative domains.

The $\pi_{1\ell}$-calculus studied subsequently by Amadio in [8] (this is the version of the calculus we shall refer to in the following) is based on the *asynchronous* rather than the synchronous π-calculus. More precisely, the $\pi_{1\ell}$-calculus builds on the π_1-calculus, a typed asynchronous π-calculus satisfying the *unique receiver* property. This property, which requires each channel to have at most one receiver, was already mentioned in Section 4.2. As we shall see, it is a common requirement in mobile distributed calculi, as it simplifies the implementation of distant communication. It also brings the π-calculus closer to object-oriented programming, where interaction arises when an object calls a method of another uniquely determined object. In the π_1-calculus, the unique receiver property is enforced by means of a simple type system. A barbed congruence is also defined on the π_1-calculus, which is characterised as a variant of the asynchronous bisimulation of [10]. The located $\pi_{1\ell}$-calculus adds to π_1 the model of locations and failures of the π_ℓ-calculus, together with a primitive $\mathsf{spawn}\,(\ell\,,p)$ for process migration. The failure primitives of the $\pi_{1\ell}$-calculus have already been discussed in Section 4.2. We shall only add here a few remarks about semantic and typing issues.

The $\mathsf{spawn}\,(\ell\,,p)$ construct provides a form of *objective* migration: the process containing the instruction causes process p to move to location ℓ. A distinctive feature of the $\pi_{1\ell}$-calculus is that locations have an associated controller process, called *locality-process*, which records the status of its controlled location and handles messages addressed to the location, in particular all spawn, test and halt requests[34]. In the $\pi_{1\ell}$-calculus communication is global, just like in the π_ℓ-calculus. However the asynchronous setting allows for a more elegant communication rule. In fact, the movement of a message towards its destination is described in two steps: first, the message is exported in the ether (that is, out of its current location[35]), and then it enters the location of the destination process (which is uniquely determined, because of the unique receiver property). Although divided in two steps, the routing of messages is completely transparent: indeed, local communication is implemented in exactly the same way. Note that the movement of messages towards their destination provides another implicit form of migration, besides that given by the $\mathsf{spawn}\,(\ell\,,p)$ primitive.

To conclude, let us mention a few results established for the $\pi_{1\ell}$-calculus. The type system proposed in [8] ensures both the unique receiver property and the invariability of the location of channel receivers. Similar properties, as we shall see, will be imposed by syntactic restrictions in the distributed JOIN calculus. Indeed the $\pi_{1\ell}$-calculus shares several ideas with the distributed JOIN calculus, like the localization of receptors and the transparency of distant communication and migration in the absence of failures. Another notable result of [8], which extends the analogous result for the π_ℓ-calculus recalled earlier, is that the $\pi_{1\ell}$-calculus can be encoded into the π_1-calculus, which in turn is sufficiently expressive to encode the full asynchronous π-calculus. Although applied to different models of locations, these results are also similar in spirit to the encoding of the location semantics of [28] into the π-calculus, given by Sangiorgi in [141] (see Section 3.4).

[34] In Section 4.2 we omitted the controller process, for the sake of simplicity. In fact, this controller may be seen as a way of implementing side-conditions on the status of locations in the semantic rules.

[35] What requires this location to be alive, a condition ensured by the controller process. In general, liveness is required for the *source* location of a communication or a migration, but not for the *target* location.

The $D\pi$-calculus of Hennessy-Riely

The $D\pi$-calculus of Hennessy and Riely is a typed synchronous π-calculus where types are used for resource access control. The calculus offers constructs for process migration and local communication. In its simplest form [89], a *flat* space of locations is introduced. In the richer variant presented in an earlier paper [137], which we may call the $D\pi^+$-calculus, structured locations and a primitive for moving locations were also considered; in addition the issue of site failure was addressed there. Since the treatement of structured locations and failures in [137] is essentially the same as in the distributed JOIN-calculus described in Section 4.3.2, we shall concentrate here on the simpler version of [89], which is also the one retained by the authors in subsequent papers [90, 138]. At the end of this section we will briefly discuss an asynchronous "receptive" version of the $D\pi$-calculus, recently studied by Amadio, Boudol and Lhoussaine [9].

The basic entities of the $D\pi$-calculus are threads (or processes), systems and resources. *Threads* p, q are processes of the polyadic π-calculus, with an additional operator $\mathtt{goto}\ \ell.\, p$ to allow movement of code p towards location ℓ. This operator is similar to the construct $\mathsf{spawn}\,(\ell\,, p)$ considered by the authors in [136] and by Amadio in [11, 8] (see Section 4.2). *Locations* ℓ, k have a flat structure. They are just a particular category of π-calculus names, and can be created via the operator $(\nu\ell)\, p$ and transmitted in communication.

Systems P, Q are collections of *located threads* of the form $\ell[p]$, also called *agents*. More precisely, systems are obtained by combining located threads via the static operators of parallel composition and name creation $(\mathtt{new}\ u : T)\ P$, an operator similar to restriction which allows sharing of information among different locations.

The *resources* of the calculus are channels, supporting synchronous communication between agents. Channels are also *located* and communication is restricted to being *purely local*: it is only allowed between agents placed at the same location, using channels that have been allocated at that location. Thus processes wishing to interact first have to move to a common location. This means that in the $D\pi$-calculus, as in other languages adopting the "go and communicate" paradigm, locations are used as *units of communication*.

The reduction semantics is given directly on systems P. It is defined modulo a structural congruence $\equiv$ that allows co-located threads to be split and reassembled, that is $\ell[p \mid q] \equiv \ell[p] \mid \ell[q]$, and empty agents to be garbage collected, $\ell[0] \equiv 0$. The rules for code movement and communication[36] are the following:

$$\ell\,[\,\mathtt{goto}\ k.\, p\,] \quad\rightarrow\quad k[p]$$

$$\ell\,[\,\overline{a}\langle u\rangle.\, p\,] \;\mid\; \ell\,[\,a(x).\, q\,] \quad\rightarrow\quad \ell\,[p] \;\mid\; \ell\,[\,q\{^{u}/_{x}\}\,]$$

Then, assuming a is a channel declared in location k, remote communication can be "implemented" using migration and local communication as follows:

$$\ell\,[\,\mathtt{goto}\ k.\ \overline{a}\langle u\rangle.\, p \;\mid\; r\,] \;\mid\; k\,[\,a(x).\, q\,] \quad\rightarrow^{*}\quad \ell\,[r] \;\mid\; k\,[\,p \;\mid\; q\,\{^{u}/_{x}\}\,]$$

Note that in $D\pi$, the units of migration are simple threads of the form $\ell\,[\,\mathtt{goto}\ k.\, p\,]$. If such a thread runs in parallel with other threads located at ℓ, these will stay put while p moves to k. Moreover process p will only be activated when reaching its destination.

[36]For simplicity we consider here the monadic version of the calculus.

Hence migration is *weak* in $D\pi$, that is, only passive code can be moved. This makes a difference with other calculi such as NOMADIC PICT, where migrating agents may contain several threads in parallel, which continue to run until the migration takes effect.

Local communication is the main parting point of the $D\pi$-calculus with respect to Amadio's $\pi_{1\ell}$-calculus (examined in the previous section), at least as far as the untyped languages are concerned. The *type systems* of the two languages, on the other hand, are rather different. While Amadio's type system is used to ensure uniqueness of receptor channels, types in the $D\pi$-calculus are designed to control the use of resources. The idea is that agents should not be allowed to use a non-local resource before being granted the appropriate *capability*. The type system of $D\pi$ is based on a new notion of *location type*, which describes the set of resources available to an agent at a particular location. Resources, that is channels, are themselves equipped with capabilities described by types. The types of channels describe the values they can transmit. They may be further refined to prescribe the way in which channels should be used, thus extending the input/output types studied in previous work by Pierce and Sangiorgi [127].

More formally, a location type has the form $\mathtt{loc}\{\overrightarrow{a}:\overrightarrow{A}\}$, where $\overrightarrow{a}$ is a tuple of distinct channel names and $\overrightarrow{A}$ is the associated tuple of capabilities. For instance the location $\ell : \mathtt{loc}\{a : A,\, b : B\}$ offers two channels a and b with capabilities A and B respectively. A simple *channel type* has the form $\mathtt{chan}\langle T\rangle$, where T is the type of the transmitted value. This value may be a location name, a *local* channel name, or a *remote* channel name marked with its location of creation, written $\ell[a]$. Compound names $\ell[a]$ may only be used as arguments in communication (not as transmission channels), and their types are of the form $\mathtt{loc}[A]$.

The syntax is then enriched with type information for received values and created names. The structural congruence $\equiv$ allows manipulations on these terms, like for instance $\ell[\,(\mathtt{new}\ a : A)\ p\,] \equiv (\,\mathtt{new}\ \ell[a] : \mathtt{loc}[A]\,)\ \ell[p]$. A new semantic rule for communication of remote channels is added:

$$\ell\,[\,\overline{a}\langle k[b]\rangle.\,p\,]\ \mid\ \ell\,[\,a(z[x]:\mathtt{loc}[A]).\,q\,]\quad\rightarrow\quad \ell\,[p]\ \mid\ \ell\,[\,q\{^{k,b}/_{z,x}\}\,]$$

For process $q\{^{k,b}/_{z,x}\}$ to be well-typed, it is necessary that the types of names k and b match those of z and x. Thus the typing ensures that names are received with the same capabilities they are sent with. In fact, thanks to subtyping, names can be more generally transmitted with a *subset* of the sender's capabilities. This means that a process sending a name can control the use that will be made of it, which is indeed the property the calculus was trying to ensure.

A remote procedure call may be described in the calculus as follows:

$$\ell\,[\,(\nu r)\ \mathtt{goto}\ k.\ \overline{a}\langle \ell[r]\rangle\ \mid\ r(X).\,p\,]\quad\mid\quad k\,[\,a(\,z[x]:\mathtt{loc}[A]\,).\ \mathtt{goto}\ z.\overline{x}\langle V\rangle\,]$$

We shall not elaborate further on the type system of [89] here. Let us just recall that this type system, which is rather sophisticated, satisfies the standard soundness properties of subject reduction (type preservation) and type safety. In later work [90, 138] the authors have adapted this type system to guarantee that even a partially typed system (such as can be an open network with untrusted components) will continue to satisfy the same property of safe access to resources.

To conclude, we will mention a result established in [9] by Amadio, Boudol and Lhoussaine, for an *asynchronous* version of the $D\pi$-calculus. Combining ideas from previous studies by Amadio [8], Boudol [21] and Sangiorgi [142], the authors propose a simple type system for this calculus, ensuring localization of receptors and *receptiveness* of channel names (a kind of input-enabledness). They show that well-typed systems enjoy a local deadlock-freedom property called "message deliverability". This property, which states that every migrating message will find a receiver at its target locality, is particularly desirable in an asynchronous setting, where there is no control on message consumption.

The *dpi* calculus of Sewell

The *dpi*-calculus studied by Sewell in [145] is an asynchronous π-calculus extended with location and migration primitives similar to those of the $D\pi$-calculus. Its hierarchical model of locations, on the other hand, as well as its global communication discipline, are inspired from those of the distributed JOIN calculus. A few syntactic variations with respect to the $D\pi$-calculus and the distributed JOIN calculus are worth mentioning: (1) the *dpi*-calculus adopts a two-level rather than a tree-structured hierarchy of locations; (2) locations are associated with each elementary subterm of a system, for instance every prefix is located; (3) when used for a location u, the construct for name creation ($\mathtt{new}\ u : @_\ell\ T$) only allows the creation of u as a *sublocation* of an existing location ℓ. This means that new locations cannot be created at the topmost level, which is interpreted as a set of physical addresses.

We shall not dwell longer on the description of the language here, but rather sketch the main contribution of the paper [145], which is the study of a type system distinguishing *local* and *global* channels with the aim of localizing communication whenever possible. The idea is that restricting the use of some channels to be local may be useful both for efficient implementation (these channels do not need to be globally unique nor registered with a central name server) and for robustness against external attacks.

The proposed type system generalises that of Pierce and Sangiorgi [127], where channel types are annotated with input and output *capabilities*. Here the input/output types of channels are refined to indicate whether their use is allowed to be global or restricted to be local. Formally, a channel type has the form $\updownarrow_{(i,o)} T$, where the input and output capabilities i and o are taken from the set $\{\mathtt{G}, \mathtt{L}, -\}$. In such a channel's type the input (resp. output) capability can be *global* ($\mathtt{G}$), in which case it can be exercised at any location, *local* ($\mathtt{L}$), in which case it may only be exercised at the channel's location, or absent ($\mathtt{-}$), in which case the channel cannot be used at all for input (resp. output). For instance, if x is a channel located at ℓ of type $\updownarrow_{(\mathtt{L},\mathtt{G})}\ T$, then x can be used for output anywhere, but for input only at location ℓ. Such a channel may be used, for instance, to send requests to a server located at ℓ. Note that in calculi enjoying the *unique receiver property*, such as the $\pi_{1\ell}$-calculus and the JOIN calculus (see next section), all channels would have such a type.

The type system ensures that channel names are used with the appropriate capabilities and that local capabilities are not sent outside the location they refer to. Subtyping allows for the transmission of subcapabilities. A type preservation result (subject reduction) is proven, and a method for inferring the *most local capabilities* of a channel is proposed. In the light of these results, it can be said that the *dpi*-calculus allows for compile-time optimization of communication, while retaining the expressiveness of global channel communication. The same idea of offering both local and global communication while clearly distinguishing between them, may be found also in the language NOMADIC PICT [148].

4.3.2 The distributed Join Calculus

The JOIN calculus was introduced by Fournet and Gonthier in [68], followed shortly later by its distributed version [69]. As it was presented by its authors in [68], the JOIN calculus is "an offspring of the π-calculus, in the asynchronous branch of the family"[37]. This calculus was specifically designed to facilitate distributed implementations of channel mechanisms. Compared to the asynchronous π-calculus, it offers a more explicit notion of places of interaction. The distributed JOIN calculus [69, 67] goes farther still, adding to the original calculus notions of *named location* and *distributed failure*.

We start with a description of the basic JOIN calculus, and then move to its distributed version. We will mainly follow the presentation of [101], which gives a good overview of the two calculi and the initial results that have been established for them.

The main feature of the JOIN calculus is that every input channel is persistent and enclosed in a *definition*, a construct that combines scope restriction and replicated reception.
In the π-calculus, there is no constraint on the use of receptor channels, and it is possible to write:

$$x(y).\, P \;\mid\; x(z).\, Q \;\mid\; \overline{x}a$$

In practice, if the two receptors $x(y).\, P$ and $x(z).\, Q$ are far from each other, one faces a *distributed consensus problem*, since the two processes have to agree about who takes the value. The JOIN calculus tries to avoid this problem by forcing all the receptors on a given channel to be grouped together in a *definition* of the form:

$$\texttt{def}\,(x\langle y\rangle \triangleright P\,) \wedge (x\langle z\rangle \triangleright Q\,)\;\texttt{in}\; x\langle a\rangle \tag{18}$$

In a definition $\texttt{def}\ D\ \texttt{in}\ R$ the notation $x\langle y\rangle$ is overloaded to mean reception of y on x in the D-part (declaration[38] part) and emission of y on x in the R-part (the running part, or body of the definition). The `def` construct delimits the scope of its defined names, that is the receptor channels declared in the D-part. Moreover, the receptors $x\langle y\rangle$ are *permanently defined* and correspond to π-calculus replicated inputs $!\,x(y).\, P$. Thus the analogue of definition (18) in the π-calculus is:

$$(\nu x)\,(\,!\,x(y).\, P \;\mid !\,x(z).\, Q \;\mid\; \overline{x}a\,)$$

An important point to note is that in the JOIN calculus every receptor channel is *statically defined*. Since their names are *bound* by their definition, receptors cannot be renamed: two different receptors will never be equated[39]. Moreover, receptors are replicated in the

[37]This branch was to grow quite loaded in subsequent years.

[38]The D-part of a definition construct was also called *definition* in [68, 69]. For clarity, we prefer to use "declaration" here.

[39]Indeed the JOIN calculus satisfies a property similar to the *ouput only capability* introduced explicitly in later calculi [105]. This property says that a transmitted channel name can never be used by the recipient as a receptor channel.

definition where they occur, and thus are always available. This gives a precise meaning to the statement that receptors are *localized* in the JOIN calculus (even in its basic version).

As in the asynchronous π-calculus [20, 91], communication is *asynchronous* in the JOIN calculus: the syntax only allows simple messages $\overline{x}\langle\overrightarrow{y}\rangle$, without continuation. To provide some synchronization mechanism, the calculus allows *join-patterns* in definitions. These are groups of guards expecting to receive a corresponding group of messages atomically. A *join pattern* J is either the reception of a tuple of names, $x\langle\overrightarrow{v}\rangle$, or a composition $J' \mid J''$ of such receptions. In a join pattern, the names of the receptors are supposed to be all distinct. A *declaration* D can then be an elementary declaration $J \triangleright P$, or a conjunction $D' \wedge D''$ of such declarations. It can also be empty. In each conjunct $J \triangleright P$, the names of the join pattern J are binding for the continuation process P. Join patterns in different conjuncts need not have disjoint receptor names. For instance, in example (18) above, the same receptor x was used in two different conjuncts.

Besides messages and the construct $\texttt{def}\ D\ \texttt{in}\ R$, the calculus has constructs for termination and parallel composition. An example of a JOIN process is the following:

$$\texttt{def}\,(\,x\langle y\rangle \mid t\langle u\rangle\ \triangleright P\,)\ \wedge\ (\,x\langle z\rangle \mid t\langle v\rangle\ \triangleright Q\,)\ \texttt{in}\ x\langle a\rangle \mid t\langle c\rangle \mid x\langle b\rangle$$

According to our informal description of the def construct, this process may reduce to:

$$\texttt{def}\,(\,x\langle y\rangle \mid t\langle u\rangle\ \triangleright P\,)\ \wedge\ (\,x\langle z\rangle \mid t\langle v\rangle\ \triangleright Q\,)\ \texttt{in}\ P\{{}^{a,c}/_{y,u}\} \mid x\langle b\rangle$$

The notation $P\{{}^{a,c}/_{y,u}\}$ indicates the simultaneous substitution of a for y and c for u. In this reduction the two messages $x\langle a\rangle$ and $t\langle c\rangle$ have been consumed by the first join-pattern, triggering the process P. Alternatively, the two messages $x\langle a\rangle$ and $t\langle c\rangle$ (or also $x\langle b\rangle$ and $t\langle c\rangle$) could have been used to trigger process Q. Thus join-patterns may be a source of nondeterminism in the language. Note on the other hand that if another message on t had been available, both P and Q could have been triggered in two successive reduction steps.

The general form of the reduction above is given by the following rule:

$$(19)\qquad \texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ J\sigma \mid Q \quad\rightarrow\quad \texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ P\sigma \mid Q$$

where the substitution σ is an instantiation of the received values of the pattern J. The semantics of the JOIN calculus is completely determined by the reduction rule (19), together with standard contextual rules and the use of a structural congruence $\equiv$ which allows, for instance, all occurrences of def to be pushed in front of a term[40].

The distributed JOIN calculus (DJOIN for short) extends the basic calculus with explicit *locations* and primitives for *migration*. Locations $a, b, \ldots$ are introduced in the syntax by adding a new clause for declarations, the *located declaration* $a[D : P]$, which represents a declaration of input channels *located* at a. The process P is used to "initialise" location a when it becomes active. Just like channels, locations are defined together with their code: in $a[D : P]$, the name a is a declared name, whose scope ranges over the whole def expression which contains it. Unlike a receptor, however, a location can only be

[40]In fact, the semantics of the JOIN calculus was originally defined using a *Chemical Abstract Machine* (CHAM) in the style proposed by Berry and Boudol [17]. The CHAM for the JOIN calculus is said to be *reflexive* in that the definition mechanism of the calculus allows new sets of rules to be added to the reduction rules of the machine.

declared *once* in a definition. For instance, the syntax does not allow the expression $\texttt{def}\, a[D : P] \wedge a[D' : Q] \triangleright R \;\texttt{in}\; S$.

Receptors are required to be declared in the join-patterns of a *unique location*. For instance, the expression $\texttt{def}\, a[x\langle y\rangle \triangleright P : Q] \wedge b[x\langle z\rangle \triangleright Q : R] \;\texttt{in}\; T$ is forbidden. On the other hand $\texttt{def}\, a[x\langle y\rangle \triangleright P \wedge x\langle z\rangle \triangleright Q : R] \;\texttt{in}\; S$ is a correct definition.

Like in the distributed π-calculi examined previously (see Section 4.3.1), in the DJOIN calculus locations are first class objects and can be transmitted in communication. It is worth stressing that locations are *globally declared* in the DJOIN calculus: just like a receptor channel, a location a declared in some definition cannot be declared in any other definition. It can only be used by a process in another definition if it has become known to it (by effect of a communication). Distant communication is transparent in the calculus: a message located at b will automatically move to the unique location a of its receptor.

As the syntax allows nested located declarations, locations are *hierarchically structured* in a tree. At the top of the tree, there is an unlocated root. Since location names are required to be *globally unique*, every proper node in the tree corresponds to a different location a[41]. The location path φa from the root to a node a represents the position of a in the location tree. If location b has position $\varphi a \psi b$, then b is a sublocation of a. The path φa is not necessary to identify a location a (the name a is sufficient, by the uniqueness assumption); it is useful however to express consistency conditions in the semantic rules, like the fact that a location does not migrate to a sublocation.

Migration is expressed by a new process construct $\texttt{go}\,\langle a, k\rangle$. A process executing the instruction $\texttt{go}\,\langle a, k\rangle$ causes its *current location*, say b, to become an immediate sublocation of location a; a null message $k\langle\,\rangle$ is emitted on channel k upon termination of the migration. This message notifies that the migration is completed: it can be used, for instance, to render a migration blocking. If the position of a in the location tree is given by φa, the new position of location b will be φab. For the migration to take effect, it is necessary that b does not occur in φa, that is b is not a superlocation of a. This is because in the DJOIN calculus, like in other calculi with structured locations, a migrating location b moves together with all its sublocations (the subtree rooted at b). This means that the location tree may be substantially reconfigurated at each migration. Note that migration is *subjective* (since the go instruction applies to its enclosing location), and that the units of migration are the locations themselves[42].

Let us consider an example of reduction in the DJOIN calculus. Without entering into technical details, and borrowing ideas from [137] and the *open* JOIN calculus [19], we may assume a runtime environment $\mathcal{L}$ which records the position of locations and declarations at any given time. Then the reduction relation would be defined on configurations $(\mathcal{L}, P)$. For instance, supposing the environment $\mathcal{L}$ defines the position of b to be ψb, and the position of a to be a string φa not containing b, we would have the following reduction:

$$(20)\;(\,\mathcal{L},\; \texttt{def}\;\, b\,[D : P \mid \texttt{go}\,\langle a,k\rangle] \wedge D' \;\texttt{in}\; R\,) \;\;\rightarrow\;\; (\,\mathcal{L}',\; \texttt{def}\;\, b\,[D : P \mid k\langle\,\rangle] \wedge D' \;\texttt{in}\; R\,)$$

where $\mathcal{L}'$ is obtained from $\mathcal{L}$ by replacing all positions of the form $\psi b \psi'$ by $\varphi a b \psi'$. In other words, the effect of this reduction is to move the tree rooted at b to become an immediate

[41] This contrasts with other models with structured locations, where a general location is represented by the whole path from the root to a node and an atomic location may appear in different paths.

[42] In fact, the movement of messages towards their receptor's location could also be seen as a simple form of *process* migration. This point will be discussed further in Section 4.3.5.

subtree of the tree rooted at a.

To summarize, the DJOIN calculus offers *global access* to locations as well as to channels. Once a process knows the name of a channel (resp. a location), it can make direct use of it independently of its location (resp. its position in the location tree). The distributed infrastructure required to implement such global mechanisms is simplified by the assumption of global uniqueness of locations and localization of receptors.

In other calculi, such as Amadio's $\pi_{1\ell}$-calculus considered in the previous section, the localization of receptors was ensured by means of a static type system. The DJOIN calculus makes the choice of handling locations directly, without recourse to a type system.

In calculi with global communication and migration one expects a *network transparency* result in the absence of node failures. As for the location failure calculi examined in Section 4.2, this result also holds for the DJOIN calculus, on the condition that systems are free of *circular migration* (such migration would appear, for instance, in the process of example (20) if location b were replaced by a). More precisely, under this condition two processes of the DJOIN calculus are equivalent if and only if they are equivalent in the basic JOIN calculus once stripped of all location information.

Clearly, one cannot hope for transparency results in the presence of location failures. Like the calculi for location failure discussed in Section 4.2, the DJOIN calculus has been equipped with a way of marking locations (as dead or alive), and constructs for *halting* a location and *testing* its status. As in Amadio's $\pi_{1\ell}$-calculus, it is still possible to send a message or migrate to a dead location. On the other hand emission or migration from a dead location is ruled out. Rules for communication and migration are then conditioned on the *liveness* of the source location. As in other models where locations are structured, a location fails with all its sublocations. Thus the dependence between locations is meaningful for failure as well as for migration.

While the location failure calculi in [11] and [136] are synchronous, Amadio's $\pi_{1\ell}$-calculus and the DJOIN calculus are asynchronous. As observed in [101], in an asynchronous world the assumption of a global knowledge of the liveness of every location may be unrealistic. For a detailed discussion about the treatment of failures in the DJOIN calculus, and more generally about results concerning the calculus, the reader is invited to consult Fournet's thesis [67].

Compared to other calculi described in this section, the JOIN and DJOIN calculi have a relatively developed theory. Standard equivalence relations (barbed equivalences) have been defined, although reasoning techniques still have to be worked out. Encodings of the JOIN calculus into and from the asynchronous π-calculus have been given in [68, 69, 67]. The paper [19] presents a characterisation of the barbed congruence of the JOIN calculus as an asynchronous bisimulation, inspired from [10]. A language based on the JOIN calculus has been implemented, the Join Calculus Language presented in [70].

4.3.3 The Ambient Calculus

The ambient calculus of Cardelli and Gordon [32] is a process calculus which emphasizes process mobility rather than communication. In fact, it appears that the mobility primitives are sufficient to give the calculus its full expressiveness, and the communication primitives are mostly added for convenience.

An *ambient* is an area delimited by a boundary, where multi-threaded computation takes place. Each ambient has a *name*, a collection of local *processes* (or threads) and a collection of *subambients*. Thus ambients can be arbitrarily nested in a tree structure. Processes may cause the enclosing ambient to move inside and outside other ambients. The form of migration provided by the calculus is therefore *subjective*.[43] Processes may also "open" an ambient at the same level, that is dissolve its boundary causing its contents to spread into the parent ambient. Finally, processes within the same ambient may communicate with each other asynchronously, by releasing and consuming messages in the local ether. The arguments of communication may be names, *capabilities* (that is, permissions to operate on ambients with a given name), or sequences of such arguments. We shall not consider further the communication primitives here.

Ambients move as a whole, bringing together all their threads and subambients. They are therefore *units of migration*. The tree structure of ambients allows for a close control of mobility: an ambient can only enter a sibling ambient and exit a parent ambient. Similarly, a process can only open an ambient which is enclosed in the same parent ambient. This form of mobility, which could be called *proximity mobility*, is well-suited to model navigation through a hierarchy of *administrative domains*, which was the original inspiration for the calculus. The ability to move and open ambients is regulated by capabilities, possibly acquired through communication. These capabilities model the notion of *authorisation*, necessary to regulate access to resources and, before that, the crossing of administrative domains.

Formally, an ambient is written $n[P]$, where n is the name of the ambient and P its contents, described by the following syntax, where M represents one of the capabilities $\mathtt{in}\ n$, $\mathtt{out}\ n$, $\mathtt{open}\ n$ (and the communication primitives are omitted):

$$P, Q \ ::= \ 0 \ \mid \ M.P \ \mid \ P \mid Q \ \mid \ (\nu n)P \ \mid !P \ \mid \ n[P]$$

In $n[P]$, it is understood that P is actively running, even when the surrounding ambient is moving. This fact is expressed by a semantic rule saying that any reduction of P can also take place within n:

$$P \rightarrow Q \quad \Rightarrow \quad n[P] \rightarrow n[Q]$$

The *mobility constructs* are $\mathtt{in}\ n.P$ (to enter an ambient named n), $\mathtt{out}\ n.P$ (to exit an ambient named n), and $\mathtt{open}\ n.P$ (to open an ambient named n). Their semantics is given by the following reduction rules:

$$\begin{array}{rcl} n[\mathtt{in}\ m.P \mid Q] \mid m[R] & \rightarrow & m[n[P \mid Q] \mid R] \\ m[n[\mathtt{out}\ m.P \mid Q] \mid R] & \rightarrow & n[P \mid Q] \mid m[R] \\ \mathtt{open}\ m.P \mid m[Q] & \rightarrow & P \mid Q \end{array}$$

Clearly, the `in` and `out` operations are duals of each other. The `open` operation, on the other hand, has a more definitive and disruptive effect. However, its use is regulated by a capability which must have been given out by the ambient to which it is applied.[44]

[43] Indeed the terminology "subjective/objective" for migration was introduced by Cardelli and Gordon.

[44] Note that the `open` operation provides a form of *objective* migration, of the contents of an ambient into its father ambient.

Ambient *names* can be created, and used to name multiple ambients. Ambients with the same name may coexist both at the same level and at different levels in the hierarchy. In case a mobility instruction matches several ambients, one of them is chosen nondeterministically. Ambients with the same name have separate identities, and an empty ambient is still observable through its name. This is expressed by the following inequalities (where $\equiv$ stands for semantic equality) :

$$n[P] \mid n[Q] \;\not\equiv\; n[P \mid Q]$$

$$n[\mathbf{0}] \;\not\equiv\; \mathbf{0}$$

An interesting equality which holds for ambients is the so-called *perfect firewall equation*:

$$(\nu n)\, n[P] \;\equiv\; \mathbf{0} \qquad \text{if } \; n \notin fn(P)$$

This law says that a process enclosed in an ambient whose name is not known by the environment (and cannot be used by the process itself) is equivalent to the terminated process.

In many ways, an ambient resembles a *named location*. Like a location, it supports internal communication and can receive ambients and messages (represented themselves by ambients) from other places. However, ambients have a more general meaning. For instance, they can move across places while carrying their running contents with them. In this respect they behave rather like *mobile agents*. Moreover, as we have seen, the handling of names is rather liberal: the same name can be borne by several ambients within the same scope. In this way, ambients may be used to model *service providers*, of which there may be more than one, as long as the same service is granted.

As mentioned earlier, one of the initial motivations for the AMBIENT calculus was to model firewall crossing in large networks. In fact, the calculus is rather ambitious, and intends to model a variety of distributed programming concepts such as channels, network nodes, packets, services and software agents. Some examples may be found in [32]. In the same work, it is also shown how the calculus can be used to encode languages such as the asynchronous π-calculus and some λ-calculi. Current work on the AMBIENT calculus is focussed on type systems for controlling mobility and the type of values transmitted within ambients [31]. In [100], a modified calculus called *Safe Ambients* is proposed, with a type system for controlling mobility and preventing so-called "dangerous interferences". Modal logics accounting for the spatial structure of ambients via the use of "spatial" modalities are also under investigation [33].

4.3.4 The Seal Calculus

Like the AMBIENT calculus, the SEAL calculus of Castagna and Vitek [157] is designed to model distributed computing over large scale open networks. As some of the previous formalisms examined in this section, the SEAL calculus takes the stand of exposing the network topology and handing over control of localities to the system programmer. However, compared to other calculi, the SEAL calculus focusses more particularly on the *security issues* raised by the new paradigm of network programming: it also aims to provide programmers with protection mechanisms for their resources and against malevolent hosts. Among the calculi we have seen, only the $D\pi$-calculus of Hennessy and Riely and the AMBIENT calculus had a strong concern in security.

The SEAL calculus may be concisely described as a polyadic *synchronous* π-calculus with hierarchical mobile locations and resource access control. The main entities of the calculus are *processes*, *locations* and *resources*.

A *process* P is a term of the mini synchronous π-calculus, where channels are tagged with the relative location to which they belong. This will be explained more precisely below, when discussing the use of channels for supporting communication and mobility. Moreover a process can also be a named location containing a process, that is a *seal* $n[P]$.

Locations are called *seals* in the calculus. A seal is written $n[P]$, where n is a name (not necessarily unique) and P is the process running inside the seal. Like channel names, seal names can be transmitted in communication and restricted upon. Since a seal encapsulates a process, and a seal is itself a process, it is clear that seals are hierarchically structured. Processes which are not contained in a seal are assumed to be placed at the root's location, thus all processes are – explicitly or implicitly – located at some seal. In $n[P]$, the seals contained in P are called the *children* of n, while n is called their *parent*. Seals are also the migration units of the calculus. The migration of a seal is ordered by the parent seal, and a seal can only be moved one level up or one level down the hierarchy. This is achieved, as we will see, by transmitting the name of the seal to be moved.

As in the $D\pi$-calculus, the only *resources* of the SEAL calculus are channels. Channels are used, as usual, for the communication of names. The SEAL calculus provides both *local communication*, between processes running in the same seal, and *linear proximity communication*, between processes placed in the parent-child relationship. No direct communication is possible, for instance, between sibling seals. For distant communication, the routing of messages in the hierarchy of seals must be explicitly programmed.

Channels are located in seals (inherited from the enclosing process)[45], and processes have access to local channels as well as to their parent's and children's channels. To enable processes to refer to proximity channels, channel names are annotated with tags referring to their relative place in the hierarchy. Formally, *tags* are defined by $\eta ::= \star \mid \uparrow \mid n$, denoting respectively the current seal, the parent seal, and a child seal bearing name n. Then input and output actions for transmitting channel names are given by:

$$a \ ::= \ \overline{x}^{\eta}(\overrightarrow{y}) \ \mid \ x^{\eta}(\overrightarrow{y})$$

Two co-located processes communicate via a local channel. We have for instance:

$$\overline{x}^{\star}(\overrightarrow{z}).\,P \mid x^{\star}(\overrightarrow{y}).\,Q \quad \rightarrow \quad P \mid Q\{\overrightarrow{z}/\overrightarrow{y}\}$$

For immediate upward or downward communication, a channel of either of the two partners may be chosen. Therefore the communication will always involve a channel tagged $\star$ and a complementary channel tagged with $\uparrow$ or some name n. For instance we have:

$$n[\overline{x}^{\uparrow}(\overrightarrow{z}).\,P] \mid x^{\star}(\overrightarrow{y}).\,Q \quad \rightarrow \quad n[P] \mid Q\{\overrightarrow{z}/\overrightarrow{y}\} \tag{21}$$

In fact, this is only approximately true. Because of the security orientation of the calculus, a further ingredient will be required for non-local communication to take place. The idea

[45] Thus channels may be located in different seals.

is that allowing non-local processes (like $\overline{x}^{\uparrow}(\overrightarrow{z}).P$ in this example) to use local channels (like channel x of process $x^{\star}(\overrightarrow{y}).Q$) constitutes a threat to security. Indeed the first process may have migrated to location n from some untrusted location. Then it should not be allowed to access the local channel x, unless it is given explicit permission to do so.

To control inter-seal communication, the calculus proposes a protection mechanism called *portal*. The idea is the following: for seal A to be able to use a channel x of seal B, seal B must first *open* a portal for A at x. A portal is a *linear access permission* for a channel: it can only be used once. Formally, a new construct $\mathtt{open}_\eta\ x.P$ (respectively $\mathtt{open}_\eta\ \overline{x}.P$) is introduced: it denotes a process which opens a portal for seal η allowing it to perform an input (resp. output) on local channel x, and then continues like P. The correct reduction replacing (21) is then:

$$(22) \qquad n[\overline{x}^{\uparrow}(\overrightarrow{z}).P] \mid x^{\star}(\overrightarrow{y}).Q \mid \mathtt{open}_n\ \overline{x}.0 \quad \rightarrow \quad n[P] \mid Q\{\overrightarrow{z}/\overrightarrow{y}\} \mid 0$$

The last important point to examine is *mobility of seals*. As mentioned earlier, seals may be transmitted over channels, and that is the way mobility is modelled in the calculus. The prefixing actions for exchanging seal names will be written $\overline{x}^{\eta}\{y\}$ and $x^{\eta}\{\overrightarrow{y}\}$, to distinguish them from the actions used for transmission of channel names. Note that an output action can only send a single seal name, for a reason that will soon be made clear. The complete syntax for prefixing actions is then:

$$a \ ::= \ \overline{x}^{\eta}(\overrightarrow{y}) \ \mid \ x^{\eta}(\overrightarrow{y}) \ \mid \ \overline{x}^{\eta}\{y\} \ \mid \ x^{\eta}\{\overrightarrow{y}\}$$

A seal can only be moved by a process contained in the parent seal. A seal *move action*, represented by an output $\overline{x}^{\eta}\{y\}$, requires that a seal with name y be present in the current seal, otherwise it blocks. If several seals match the name, then one of them is chosen nondeterministically. Note the difference with the ambient calculus: migration is *objective* here (a process causes a seal at the same level to move), while it was subjective in the ambient calculus (a process was causing the whole enclosing ambient to move).

The synchronisation of a move action $\overline{x}^{\eta}\{y\}$ with a receive action $x^{\eta}\{\overrightarrow{z}\}$ where $\overrightarrow{z} = z_1, \ldots, z_k$ has the effect of creating k copies of seal y under the names $z_1, \ldots, z_k$. The seal y will disappear from the location of the sender and the copies $z_1, \ldots, z_k$ will appear at the location of the receptor. Thus seals may be both *renamed* and *duplicated* during a migration. This may be useful respectively for protection and fault tolerance.

Regarded as a tree rewriting operation on the hierarchy of seals, a move $\overline{x}^{\eta}\{y\}$ has the effect of disconnecting a subtree rooted at seal y and grafting it either onto the parent of y, or onto one of y's children, or back onto y itself. Each of these moves can be accompanied by renaming and introduce copies. For instance, suppose $P = \overline{x}^{\uparrow}\{y\}.P'$ and $R = x^{\star}\{z\}.R'$. Then we have the following reduction:

$$(23) \qquad R \mid n[\,P \mid m[Q] \mid y[S]\,] \quad \rightarrow \quad R' \mid z[S] \mid n[\,P' \mid m[Q]\,]$$

where P causes the seal named y to move into its parent seal (the root seal in this case) renaming it with z[46]. This is an example of upward migration. Note that the location of the transmission channel is not important: the same reduction would have taken place

[46] Again this is not completely exact, and a portal is needed to authorize the use of non-local channels, just as for channel name transmission.

if $P = \overline{x}^{\star}\{y\}\,.\,P'$ and $R = x^{n}\{z\}\,.\,R'$. The direction of the migration is always from the sender to the receptor seal. The following is an example of downward migration, assuming $P = \overline{x}^{m}\{y\}\,.\,P'$ and $Q = x^{\star}\{z\}\,.\,Q'$:

$$P \mid m[Q] \mid y[S] \quad \to \quad P' \mid m[\,Q' \mid z[S]\,] \tag{24}$$

The two reductions (23) and (24) show that the transmission of a seal name is really a `spawn` operation in disguise. Its semantics is rather different from that of channel transmission. The protection mechanism by means of portals, on the other hand, is the same as for channel names, thus the formulation of migration as a form of communication is convenient.

Note that as regards protection, the direction (upward or downward) in the hierarchy of seals makes no difference: portals are used in the same way to protect a seal from its parent seal or from a subseal. For migration instead this direction is important, since a (process in the) parent can instruct a child to move but not viceversa.

The SEAL calculus was designed to provide protection against attacks from visitor hosts as well as attacks from the environment. A detailed discussion about the protection mechanisms of the calculus against various kinds of typical attacks (like disclosure of information, denial of service and Trojan horses) may be found in [156]. Some programming examples are presented in [157]. An implementation of the calculus is also under development.

We conclude with a brief comparison with the AMBIENT calculus, which was one of the sources of inspiration for the SEAL calculus. We have already remarked that seal mobility is objective while ambient mobility is subjective. Moreover, mobility of ambients is controlled by capabilities which can be transmitted and used more than once. By contrast, the SEAL calculus gives full control of mobility to the environment; moreover, this control is fine-grained since permissions must be renewed for each non local access. Furthermore, the SEAL calculus does not allow boundaries to be dissolved: the `open` operation of the AMBIENT calculus is considered dangerous in that it allows a lower-level agent, which should be subject to some control, to step at the local level. Finally, while the AMBIENT calculus focusses on migration rather than communication, the SEAL calculus gives priority to communication, and indeed implements migration via communication. Like the AMBIENT calculus, the SEAL calculus also satisfies the *perfect firewall equation* $(\nu\, x)x[P] \;\equiv\; 0$, which in this case holds without requirements on P. This equation states that an arbitrary process can be completely isolated from the rest of the system, i.e. prevented from any communication with it. It is a useful security property as it guarantees that a seal trapped in a firewall cannot disclose any information.

4.3.5 Nomadic Pict

The language NOMADIC PICT [148] of Sewell, Wojciechowski and Pierce is an extension of PICT, a strongly-typed high-level concurrent language based on the asynchronous π-calculus [111, 91, 20], which was developed by Pierce and Turner [154, 128]. PICT supports a wide range of high-level constructs including data structures, higher-order functional programming, concurrent control structures and objects. The vocation of NOMADIC PICT is to "complete" PICT to make it suitable for wide-area distributed programming. In particular, Nomadic Pict accomodates notions of location, agent and migration. The language is layered in two *levels of abstraction*:

- A *low level* that makes distribution and network communication clear; at this level one has *location-dependent* constructs, like agent migration and asynchronous communication between located agents.
- A *high level* where the distributed infrastructure can be ignored; this level provides *location-independent* constructs to allow mobile agents to communicate without having to explicitly track each other's movements.

In this brief description we shall not make a sharp distinction between the language Nomadic Pict itself [148] and the underlying calculus, the Nomadic π-calculus studied in [147]. We shall essentially present the calculus, but using mostly the notation of [148].

The main entities of the language are sites, agents and channels. *Sites* may be thought of as physical machines: they are unstructured and each site has a unique name s. *Agents* are units of executing computation that can migrate between sites. At any moment, an agent is located at a particular site s. Every agent has a name a, which is unique in the whole network. *Channels* are the support for communication: they are used for asynchronous communication both within an agent and between agents (possibly located at different sites). Channel names c can be created dynamically, as in the π-calculus. New names of agents can also be created dynamically, while names of sites are fixed once and for all. All names can be transmitted in communication.

Low-level language

The syntax is rigidly structured into the three categories of sites, agents and processes. *Sites* are assumed to be given: they are the set of machines, or rather an instantiation of the runtime system on these machines. Sites contain a number of *agents* running in parallel: the initial assignment of agents to sites is also supposed to be known. Finally, *processes* always appear as the body of some agent. They are built with the usual constructs of the asynchronous π-calculus (with typed channels), together with two new primitives for agent creation and migration:

$$
\begin{array}{ll}
\texttt{agent}\ a = P\ \ \texttt{in}\ \ Q & \quad \text{agent creation} \\[6pt]
\texttt{migrate to}\ s.\,P & \quad \text{agent migration}
\end{array}
$$

The effect of the first instruction in the body of some agent b is to create a new agent $a[P]$ at the same site, with name a and body P. After the creation, both P and Q start executing in parallel with the rest of agent b. The new name a is unique and is binding in both P and Q.

The execution of the second instruction in an agent a makes the whole agent move to site s. After the migration, P starts executing at site s in parallel with the rest of agent a.

More formally, assuming an environment $\mathcal{L}$ recording the sites of agents, with $\mathcal{L}(a) = s_a$, the semantics of these constructs is given by:

$$
\begin{array}{rcl}
(\,\mathcal{L},\ a[\,(\texttt{agent}\ b = P\ \ \texttt{in}\ \ Q) \mid R\,]\,) & \rightarrow & (\,\mathcal{L},\ (\nu b@s_a)(\,b[P] \mid a[\,Q \mid R\,]\,)\,) \\[6pt]
(\,\mathcal{L},\ a[\,(\texttt{migrate to}\ s.\,P) \mid Q\,]\,) & \rightarrow & (\,\mathcal{L}\{a \mapsto s\},\ a[P \mid Q]\,)
\end{array}
$$

The notation $(\nu b@s_a)$ represents creation of a new name b at site s_a, while $\mathcal{L}\{a \mapsto s\}$ is environment update. The semantics of the construct $(\nu a@s)$ is given by the rule:

$$\frac{(\,\mathcal{L}\{a \mapsto s\},\ P\,) \;\rightarrow\; (\,\mathcal{L}'\{a \mapsto s'\},\ P'\,)}{(\,\mathcal{L},\ (\nu a@s)\ P\,) \;\rightarrow\; (\,\mathcal{L}', (\nu a@s')\ P'\,)}$$

Note that migration is *subjective*, as in the AMBIENT and DJOIN calculi. It is is also *asynchronous*, since processes inside the moving agent a - other than the prefixed process P - continue to run until the migration takes place. Note also that the `migrate` instruction is the only means provided by the syntax to explicitly specify the site of an agent.

Direct communication on channels is only supported between processes running in the same agent. For interaction between agents, the low-level language provides two distinct constructs for *local* and *distant* communication:

$$\left.\begin{array}{ll} \langle a\rangle c\,!\,v & \text{send to local agent} \\ \langle a@s\rangle c\,!\,v & \text{send to distant agent} \end{array}\right\}\ \text{send to located agent}$$

The construct $\langle a\rangle c\,!\,v$ in an agent b attempts to deliver the message $c\,!\,v$ ("value" v on channel c) to agent a on the same site as b. The construct $\langle a@s\rangle c\,!\,v$, instead, tries to deliver the message $c\,!\,v$ to agent a on site s. Both operations fail silently if agent a is not found on the expected site. The implicit assumption is that at the low level agents must know each other's location in order to communicate successfully. In fact, these two constructs can be derived from another instruction available in the low-level language:

$$\texttt{if local}\ \langle a\rangle c\,!\,v\ \texttt{then}\ P\ \texttt{else}\ Q \qquad \text{test-and-send to local agent}$$

This instruction in the body of an agent b has two possible effects: if agent a is present on the same site, the message $c\,!\,v$ is delivered to a and process P is spawned in parallel with the rest of b; otherwise the message is discarded and Q is spawned in parallel with the rest of b. This construct, which reintroduces some implicit synchrony in the language, is mainly used for implementing the global communication mechanism of the high-level language. Formally, its semantics is given by the rule:

$$(\,\mathcal{L},\ a[\,\texttt{if local}\ \langle b\rangle c\,!\,v\ \texttt{then}\ P\ \texttt{else}\ Q\,]\,) \;\rightarrow\; \begin{cases} (\,\mathcal{L},\ b[c!v] \mid a[P]\,) & \text{if } \mathcal{L}(a) = \mathcal{L}(b) \\ (\,\mathcal{L},\ a[Q]\,) & \text{otherwise} \end{cases}$$

where again a structural congruence is used for splitting and grouping parts of an agent: $b[R] \mid b[S] \equiv b[R \mid S]$. The test-and-send is the only *primitive* communication construct of the low-level language, as the previous two constructs can be encoded as follows:

$$\begin{array}{rcl} \langle a\rangle c\,!\,v & = & \texttt{if local}\ \langle a\rangle c\,!\,v\ \texttt{then}\ nil\ \texttt{else}\ nil \\ \langle a@s\rangle c\,!\,v & = & \texttt{if local}\ \langle a\rangle c\,!\,v\ \texttt{then}\ nil\ \texttt{else} \\ & & \texttt{agent}\ b = \texttt{migrate to}\ s.\,\langle a\rangle c\,!\,v\ \texttt{in}\ nil \end{array}$$

High-level language

The high-level language is obtained by adding a single location-independent construct to the low-level language, namely one for *global communication*:

$$\langle a@\,?\,\rangle c\,!\,v \qquad \text{send to agent anywhere}$$

The effect of such a global output is to deliver the message $c\,!\,v$ to agent a, no matter what its current location and what further migrations it can undergo.

The idea is that at the high level an agent must be able to access all other agents uniformly, without having to locate them explicitly. It is important to note, on the other hand, that the global communication construct is *not primitive* in the language, and has to be implemented using the low-level communication primitives.

The low-level primitives remain available in the high-level language, for interacting with agents whose locations are predictable. This possibility of choice is interesting since global communication is potentially expensive to implement.

As can be seen, the model of NOMADIC PICT is very simple. The units of migration are agents, while sites are simply units of distribution.

The calculus LLinda

Linda [34] is a coordination language where processes share a global environment called a *tuple space*, and interact by fetching and releasing tuples asynchronously in this space. Distributed versions of Linda have been proposed, where multiple tuple spaces are introduced and remote operations on them are provided. A variant considered in [35] also allows nested tuple spaces. The calculus LLinda of De Nicola, Ferrari and Pugliese [117] combines a core distributed Linda with a set of operators borrowed from Milner's calculus CCS. Both tuple spaces and operations over tuples are located. A primitive for spawning a process at a remote tuple space is added (generalising an existing operation on tuples). The located tuple spaces of LLinda have a flat structure, and can be accessed from other tuple spaces without restrictions. Some typical examples (like a remote procedure call and a remote server) are provided to illustrate the use of the language for remote programming. An evolution of LLinda, the calculus KLAIM studied subsequently by the same authors [118], is also equipped with a type system for checking access rights of mobile agents. A prototype language based on KLAIM has been implemented.

Discussion

To summarize, the models reviewed in this section can be broadly divided in two classes as regards their treatment of locations:

- Models based on a *flat space of locations*, where the migrating entities are processes or agents. These are essentially the distributed π-calculi, excluding the *dpi*-calculus. Migration is mostly weak in these calculi; it is only strong (and asynchronous) in NOMADIC PICT. Research has focussed on type systems guaranteeing a correct or safe use of names, be they locations or channels. In some cases, as in the $D\pi$-calculus and in its asynchronous version, locations are also *units of communication.* In the $\pi_{1\ell}$-calculus, the only calculus in this category to account for failure, locations are also *units of failure*.

- Models based on *hierarchical locations*, where the migrating entities are locations themselves. The representation of migration is more sophisticated here, and better suited for handling failure and security issues. In all these models locations are *units* *of migration* and migration is strong and asynchronous. In the DJOIN calculus[47], locations are also *units of failure*. In the AMBIENT calculus and the SEAL calculus the issue of failure is neglected (or judged inappropriate for large networks), and priority is given to security questions. In the SEAL calculus locations are clearly treated as *units of security*. In the AMBIENT calculus locations are *units of communication* and it is more debatable whether they can be seen as units of security. In some of these models, a form of process migration is also provided. This is the case for instance in the $D\pi^+$-calculus. In the AMBIENT calculus, process migration is introduced by the `open` construct, which allows processes at one location to diffuse into their father's location. In the DJOIN calculus and in the *dpi*-calculus, message movement towards a distant destination may be seen as an elementary form of process migration.

All these languages provide an explicit migration primitive (which in the SEAL calculus is rendered as communication of location names). In all models locations have names which can be transmitted in communication. In all cases except for NOMADIC PICT, where they are fixed, locations may also be dynamically created; the *dpi*-calculus forbids creation of new locations at the top level, since this is meant to represent a fixed architecture of physical machines.

The main sources for this survey on distributed calculi for mobility were the original papers cited in the various sections. The references [29] and [144] were also of help.

5 Conclusions

In this chapter we have reviewed two main approaches to process algebras with locations, which we called *abstract* and *concrete*. These approaches respond to different concerns: in the first case locations are introduced to distinguish parallel components, as the basis for particular noninterleaving semantics. Locations are treated here as "units of sequential computation", although parallel subcomputations may be spawned at sublocations as the execution proceeds. In the second approach locations are used to support a new style of *network-aware programming*. In general, locations contain multi-threaded computation here, and their space may vary as a consequence of location creation and migration.

In the first approach transitions are indexed by the location where they occur, while in the second approach there is a variety of possible observations of locations, ranging from indexed transitions (as in Murphy and Corradini's model) to more implicit observations. Locations may even become completely unobservable under certain conditions (for instance in the DJOIN and $\pi_{1\ell}$ calculi in the absence of failures).

Since locations are directly reflected in the semantics in the abstract approach and not in the concrete approach (if we except Murphy and Corradini's work), the reader might be puzzled by our terminology "abstract vs concrete": this refers to the fact that in the "abstract" case locations are observed modulo a relabelling of their names, while in the "concrete" case the names of locations, even though their observation is more restricted, have an absolute meaning and cannot be abstracted upon.

Although locations do not have the same role in the two approaches, the *models* used to represent them, be they flat or structured, are not substantially different. Within the

[47]As in other calculi inspired from it, like the *dpi*-calculus and the $D\pi^+$-calculus.

abstract approaches, techniques have been developed for reasoning about traditional (CCS-like) process calculi enriched with locations, and a body of results have been obtained. As regards the integration of locations and mobility, on the other hand, interesting calculi have been proposed, with various foci, but the theory has not yet stabilised. The question is then: to which extent can techniques and results developed for a simple process calculus (CCS) in the first approach be extended to the more sophisticated calculi for mobile computation addressed by the concrete approach?

In exposing the abstract approach, we have concentrated on two different presentations of location semantics, based respectively on static and dynamic notions of location. The relation between the two formulations has been examined in full detail. The connection with other work on noninterleaving semantics for CCS has also been discussed in some depth. Equational and logical characterisations for the location equivalence and preorder were also mentioned. Very little has been said, on the other hand, about questions of decidability and verification for location equivalence and other distributed equivalences. We shall briefly touch upon such issues here. As regards *decidability*, a first decision procedure for distributed bisimulation, based on a tableau method, was presented by Christensen in [41], for a recursive fragment of CCS with parallelism known as BPP (Basic Parallel Processes, see Section 3.2). The correctness of this procedure was based on a cancellation property of the form $p \mid q \sim_d p \mid r \quad \Rightarrow \; q \sim_d r$, which does not hold for other noninterleaving equivalences such as location equivalence or causal bisimulation. Building on work by Christensen et al. [42, 43], an alternative method for the decision of such equivalences over BPP was proposed by Kiehn and Hennessy in [98]. An extension of this decision procedure to the weak versions of the equivalences over representative subclasses of BPP is also offered in the same work.

Another concern was to find *compact representations* for the location transition system and efficient verification procedures for location equivalence. In its original dynamic formulation of [28], the location semantics associates an infinitely branching transition system with any non trivial CCS process. As soon as recursion is involved, this transition system also becomes infinite state. We thus lose the correspondence between regular behaviours and finite transition system representations that we have in the standard CCS semantics. Now, while infinite-branching can be easily removed by choosing canonical locations at each step (as in the numbered transition system of Yankelevich [167] or the occurrence transition system considered in Section 2.6), the infinite progression is really intrinsic to the dynamic location semantics (as to any other *history dependent* semantics).

In its dynamic formulation, location equivalence involves the creation of new locations at each step. It thus requires the use of specific algorithms. In particular these algorithms work only on pairs of processes – following the so-called *on the fly* verification technique – and do not allow for the definition of minimal representatives for equivalence classes of processes. To overcome this problem, Montanari and Pistore have proposed in [112] a model called *location automata*: these are enriched transition systems where the information on locations appears on the labels, and is used in such a way that location equivalence on terms can be reduced to the standard bisimulation on the corresponding location automata. Moreover, location automata provide *finite representations* for all finitary CCS processes (those where no parallelism appears under recursion). This has the pleasant consequence that standard algorithms can be applied for the verification of location equivalence, with a worst case complexity which is comparable to that of verifying ordinary bisimulation, and *minimal representatives* for processes can be defined.

Recently, Bravetti and Gorrieri have proposed a technique that improves on Montanari and Pistore's model in that it offers a *compositional* interpretation of processes into a variation of location automata [30]. Although this technique has been mainly applied to ST-bisimulation, it may reveal useful also for location bisimulation.

As regards the static definition of location equivalence, a simple algorithm was proposed by Aceto in [5] for checking this equivalence on a subset of CCS (nets of automata). This line of investigation was not further pursued, however, and no experimentation was conducted on the static formulation of the equivalence for full CCS.

Concerning the concrete approach to localities, and particularly the new formalisms for combining locations and mobility, our account in Section 4.3 was admittedly very focussed and far from complete. For the sake of uniformity, we have concentrated on a family of calculi derived from the π-calculus. However, mobility of agents and processes is a vast and rapidly evolving field, and several other languages for mobile agents have been proposed in the literature in the last decade. To mention but a few, proposals like Telescript, Obliq, Oz, Kali Scheme, Odyssey, Voyager and Agent Tcl all bear some relation with the calculi examined here.

Aspects of distribution have also been studied in other settings; most of the languages cited in the previous paragraph are based on the object-oriented paradigm; for a treatment of locations in the context of concurrent constraint programming see for instance [135].

Acknowledgements

I would like to thank the anonymous referee for helpful comments on a previous version of this chapter. Many thanks also to Luca Aceto, Roberto Amadio, Gérard Boudol, Rob van Glabbeek, Matthew Hennessy, Astrid Kiehn, Alban Ponse and Davide Sangiorgi for giving me useful feedbacks on various parts of this material. This work was partly supported by the european Working Group CONFER2 and by the french RNRT Project MARVEL.

References

[1] L. Aceto. On relating concurrency and nondeterminism. In *Proceedings MFPS 91*, number 598 in LNCS, 1991.

[2] L. Aceto. *Action-refinement in process algebras.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1992.

[3] L. Aceto. History preserving, causal and mixed-ordering equivalence over stable event structures (Note). *Fundamenta Informaticae*, 17(4):319–331, 1992.

[4] L. Aceto. Relating distributed, temporal and causal observations of simple processes. *Fundamenta Informaticae*, 17(4):369–397, 1992.

[5] L. Aceto. A static view of localities. *Formal Aspects of Computing*, 6:201–222, 1994.

[6] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103(2):204–269, 1993.

[7] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, December 1994.

[8] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings COORDINATION 97*, number 1282 in LNCS, 1997.

[9] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus. In *Proceedings FST-TCS'99*, volume 1738 of *Lecture Notes in Computer Science*, 1999.

[10] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195:291–324, 1998.

[11] R. Amadio and S. Prasad. Localities and failures. In *Proceedings FST-TCS 94*, number 880 in LNCS, pages 205–216, 1994.

[12] E. Badouel and P. Darondeau. Structural operational specifications and trace automata. In *Proceedings CONCUR 92*, number 630 in LNCS, 1992.

[13] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[14] M. Bednarczyk. *Categories of Asynchronous Systems.* PhD thesis, University of Sussex, 1988.

[15] M. Bednarczyk. Hereditary history-preserving bisimulation, or what is the power of the future perfect in program logics. Research report, Institute of Computer Science, Polish Academy of Sciences, Gdansk, 1991.

[16] J. Bergstra, A. Ponse, and S. Smolka. *Handbook of Process Algebra.* 2000.

[17] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[18] E. Best, R. Devillers, A. Kiehn, and L. Pomello. Concurrent bisimulations in Petri nets. *Acta Informatica*, 28(3):231–264, 1991.

[19] M. Boreale, C. Fournet, and C. Lanve. Bisimulations in the join-calculus. In *Proceedings PROCOMET 98*, 1998.

[20] G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, Sophia-Antipolis, 1992.

[21] G. Boudol. The pi-calculus in direct style. In *Proceedings POPL 97*, 1997.

[22] G. Boudol and I. Castellani. On the semantics of concurrency: partial orders and transition systems. In *Proceedings TAPSOFT 87*, number 249 in LNCS, pages 123–137, 1987.

[23] G. Boudol and I. Castellani. Concurrency and atomicity. *Theoretical Computer Science*, 59:25–84, 1988.

[24] G. Boudol and I. Castellani. Permutation of transitions: an event structure semantics for CCS and SCCS. In *Proceedings REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency,* Noordwijkerhout, number 354 in LNCS, 1988.

[25] G. Boudol and I. Castellani. Flow models of distributed computations: event structures and nets. Report 1482, INRIA, 1991. Previous version by G. Boudol in *Proceedings LITP Spring School*, La Roche-Posay, number 469 in LNCS, 1990.

[26] G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, 1994.

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

[28] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.

[29] G. Boudol, F. Germain, and M. Lacoste. Projet MARVEL: Document d'analyse des langages et modèles de la mobilité. Document of the french RNRT Project MARVEL, 1999.

[30] M. Bravetti and R. Gorrieri. Deciding and axiomatizing ST-bisimulation for a process algebra with recursion and action refinement. In *Proceedings EXPRESS 99*, number 27 in ENTCS, 1999.

[31] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *Proceedings ICALP 98*, number 1644 in LNCS, pages 230–239, 1998.

[32] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS 98*, number 1378 in LNCS, 1998.

[33] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings POPL 2000*. ACM Press, 2000.

[34] N. Carriero and D. Gelernter. Linda in context. *JACM*, 32(4):444–458, 1989.

[35] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus linda. In *Proceedings Object-Based Models and Languages for Concurrent Systems*, number 924 in LNCS, 1995.

[36] I. Castellani. *Bisimulations for Concurrency.* PhD thesis, University of Edinburgh, 1988.

[37] I. Castellani. Observing distribution in processes: static and dynamic localities. *Int. Journal of Foundations of Computer Science*, 4(6):353–393, 1995.

[38] I. Castellani and M. Hennessy. Distributed bisimulations. *JACM*, 36(4):887–911, 1989.

[39] I. Castellani and G. Q. Zhang. Parallel product of event structures. *Theoretical Computer Science*, 179(1-2):203–215, 1997.

[40] L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs interleaving: an instructive example. *Bulletin of the EATCS*, 31:12–15, 1987.

[41] S. Christensen. Distributed bisimilarity is decidable for a class of infinite state-space systems. In *Proceedings CONCUR 92*, number 630 in LNCS, 1992.

[42] S. Christensen. *Decidability and decomposition in process algebras.* PhD thesis, University of Edinburgh, 1993.

[43] S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for all parallel processes. In *Proceedings LICS 93*, pages 143–157. IEEE Computer Society Press, 1993.

[44] F. De Cindio, G. De Michelis, L. Pomello, and C. Simone. A Petri net model for CSP. In *Proceedings CIL 81*, 1981.

[45] F. Corradini. *Space, Time and Nondeterminism in Process Algebras.* PhD thesis, Università di Roma La Sapienza, 1996.

[46] F. Corradini and R. De Nicola. On the relationships between four partial ordering semantics. *Fundamenta Informaticae*, 34(4):349–384, 1996.

[47] F. Corradini and R. De Nicola. Locality based semantics for process algebras. *Acta Informatica*, 34:291–324, 1997.

[48] F. Corradini and D. Murphy. Located processes and located processors. Report SI/RR - 94/02, Università di Roma La Sapienza, 1994.

[49] Ph. Darondeau and P. Degano. Causal trees. In *Proceedings ICALP 89*, number 372 in LNCS, pages 234–248, 1989.

[50] Ph. Darondeau and P. Degano. Causal trees: interleaving + causality. In *Proceedings LITP Spring School,* La Roche-Posay, number 469 in LNCS, 1990.

[51] Ph. Darondeau and P. Degano. Refinement of actions in event structures and causal trees. *Theoretical Computer Science*, 118(1):21–48, 1993.

[52] P. Degano, R. De Nicola, and U. Montanari. On the consistency of truly concurrent operational and denotational semantics. In *Proceedings LICS 88.* IEEE Computer Society Press, 1988.

[53] P. Degano, R. De Nicola, and U. Montanari. Partial orderings descriptions and observations of nondeterministic concurrent processes. In *Proceedings REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency,* Noordwijkerhout, number 354 in LNCS, 1989.

[54] P. Degano, R. De Nicola, and U. Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, 75(3):223–262, 1990.

[55] P. Degano, R. Gorrieri, and S. Marchetti. An exercise in concurrency: a CSP process as a C/E system. In *Advances in Petri nets 1988*, number 154 in LNCS, 1988.

[56] P. Degano and U. Montanari. Concurrent histories: A basis for observing distributed systems. *Journal of Computer and System Sciences*, 34(2/3):422–461, 1987.

[57] P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings FCT 85*, number 199 in LNCS, 1985.

[58] P. Degano, R. De Nicola, and U. Montanari. Observational equivalences for concurrency models. In *Formal Description of Programming Languages III*, pages 105–132. North Holland, 1987.

[59] P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica*, 26(1/2):59–91, 1988.

[60] P. Degano and C. Priami. Proved trees. In *Proceedings ICALP 92*, number 623 in LNCS, 1992.

[61] R. Devillers. On the definition of a bisimulation notion based on partial words. *Petri Net Newsletter*, 29:16–19, 1988.

[62] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Report DAIMI PB-208, Computer Science Department, University of Aarhus, Denmark, 1986.

[63] G. Ferrari. *Unifying Models of Concurrency*. PhD thesis, University of Pisa, 1990.

[64] G. Ferrari, R. Gorrieri, and U. Montanari. An extended expansion theorem. In S. Abramsky and T. Maibaum, editors, *Proceedings TAPSOFT '91*, number 494 in LNCS, 1991.

[65] G. Ferrari and U. Montanari. Towards the unification of models of concurrency. In *Proceedings CAAP 90*, number 431 in LNCS, 1990.

[66] G. Ferrari and U. Montanari. Parametrized structured operational semantics. *Fundamenta Informaticae*, 34:1–31, 1998.

[67] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, 1998.

[68] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings POPL 96*, pages 372–385, 1996.

[69] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings CONCUR 96*, number 1119 in LNCS, 1996.

[70] C. Fournet and L. Maranget. The join-calculus language, release 1.04 (documentation and user's manual), 1999.

[71] J.L. Gischer. *Partial orders and the axiomatic theory of shuffle*. PhD thesis, Stanford University, 1984.

[72] R.J. van Glabbeek. The refinement theorem for ST-bisimulation semantics. In M. Broy and C.B. Jones, editors, Proceedings IFIP TC2 Working Conference on *Programming Concepts and Methods,* Israel, April 1990, pages 27–52. North-Holland, 1990.

[73] R.J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In *Proceedings MFCS 89*, number 379 in LNCS, 1989.

[74] R.J. van Glabbeek and U. Goltz. Partial order semantics for refinement of actions – neither necessary nor always sufficient but appropriate when used with care. *Bulletin of the EATCS*, 38, 1989.

[75] R.J. van Glabbeek and U. Goltz. Equivalences and refinement. In *Proceedings LITP Spring School,* La Roche-Posay, number 469 in LNCS, 1990.

[76] R.J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness, volume 430 of LNCS, 1990.

[77] R.J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. Hildesheimer Informatik-Berichte 6/98, Institut für Informatik, Universität Hildesheim, 1998. To appear in *Acta Informatica.*

[78] R.J. van Glabbeek and F.W. Vaandrager. Petri net models for algebraic theories of concurrency. In *Proceedings PARLE 87*, number 259 in LNCS, 1987.

[79] R.J. van Glabbeek and F.W. Vaandrager. The difference between splitting in n and $n+1$. *Information and Computation*, 136(2):109–142, 1997.

[80] U. Goltz. On representing CCS programs by finite Petri nets. In *Proc. MFCS 88*, volume 324 of *lncs*, pages 339–350. Springer, 1988.

[81] U. Goltz and A. Mycroft. On the relationship of CCS and Petri nets. In *Proceedings ICALP 84*, number 172 in LNCS, 1984.

[82] U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57(2-3):125–147, 1983.

[83] U. Goltz and W. Reisig. CSP programs as nets with individual tokens. In *Advances in Petri nets 1984*, number 188 in LNCS, 1985.

[84] R. Gorrieri and U. Montanari. SCONE: A simple calculus of nets. In *Proceedings CONCUR 90*, number 458 in LNCS, 1990.

[85] J. Grabowski. On partial languages. *Fundamenta Informaticae*, 4(1):427–498, 1981.

[86] M. Hennessy. On the relationship between time and interleaving. Unpublished draft, Sophia-Antipolis, 1980.

[87] M. Hennessy. Axiomatising finite concurrent processes. *SIAM Journal on Computing*, 17(5), 1988.

[88] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138, 1995.

[89] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings HLCL 98*, volume 16 of ENTCS, 1998.

[90] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Proceedings of the 1999 Workshop on Secure Internet Programming*, number 1603 in LNCS. 1999.

[91] K. Honda and M. Tokoro. On asynchronous communication semantics. In *Proceedings Workshop on Object-based concurrent computing*, number 612 in LNCS, 1992.

[92] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation and open maps. In *Proceedings LICS 93*, pages 418–427. IEEE Computer Society Press, 1993.

[93] P. Khrisnan. Distributed ccs. In *Proceedings CONCUR 91*, volume 527 of LNCS, 1991.

[94] A. Kiehn. *Concurrency in process algebras.* Habilitation Thesis, Technische Universität München, 1999.

[95] A. Kiehn. Distributed bisimulations for finite CCS. Report 7/89, University of Sussex, 1989.

[96] A. Kiehn. Proof systems for cause based equivalences. In *Proceedings MFCS 93*, volume 711 of LNCS, 1993.

[97] A. Kiehn. Comparing locality and causality based equivalences. *Acta Informatica*, 31:697–718, 1994.

[98] A. Kiehn and M. Hennessy. On the decidability of noninterleaving process equivalences. *Fundamenta Informaticae*, 30:23–43, 1997.

[99] R. Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. Technical report, University of Twente, 1991.

[100] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings POPL 2000.* ACM Press, 2000.

[101] J.-J. Lévy. Some results in the join-calculus. In *Proceedings TACS 97*, number 1281 in LNCS, 1997.

[102] R. Loogen and U. Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae*, XIV(1):39–74, 1991.

[103] A. Mazurkiewicz. Concurrent program schemes and their interpretation. Report DAIMI PB-78, Aarhus University, 1977.

[104] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, number 255 in LNCS*, 1987.

[105] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. 25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer–Verlag, 1998.

[106] R. Milner. *A Calculus of Communicating Systems*, volume 92 of LNCS. Springer–Verlag, 1980.

[107] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:269–310, 1983.

[108] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[109] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[110] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[111] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[112] U. Montanari and M. Pistore. Efficient minimization up to location equivalence. In *Proceedings ESOP 96*, number 1058 in LNCS, 1996.

[113] U. Montanari and D. Yankelevich. A parametric approach to localities. In *Proceedings ICALP 92*, number 623 in LNCS, 1992.

[114] U. Montanari and D. Yankelevich. Location equivalence in a parametric setting. *Theoretical Computer Science*, 149:299–332, 1995.

[115] M. Mukund and M. Nielsen. CCS, locations and asynchronous transition systems. In *Proceedings FST-TCS 92*, number 652 in LNCS, 1992.

[116] D. Murphy. Observing located concurrency. In *Proceedings MFCS 93*, number 711 in LNCS, 1993.

[117] R. De Nicola, G. Ferrari, and R. Pugliese. Locality based linda: programming with explicit localities. In *Proceedings TAPSOFT-FASE'97*, number 1214 in LNCS, 1997.

[118] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.

[119] M. Nielsen. CCS and its relationship to net theory. In *Advances in Petri Nets 1986, number 255 in LNCS*, 1987.

[120] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[121] E.-R. Olderog. Operational Petri net semantics for CCSP. In *Advances in Petri Nets 1987*, volume 266 of LNCS, 1987.

[122] E.-R. Olderog. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship.* Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, 1991.

[123] D. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI-Conf. on Theoretical Computer Science*, number 104 in LNCS, 1981.

[124] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[125] C.A. Petri. Kommunication mit automaten. Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[126] C.A. Petri. Nonsequential processes. Research Report 77-05, GMD, Sankt Augustin, 1977.

[127] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[128] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press.

[129] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[130] V.R. Pratt. The pomset model of parallel processes: unifying the temporal and the spatial. In *Proceedings of Seminar on Concurrency*, number 197 in LNCS, 1985.

[131] V.R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.

[132] A. Rabinovich and B.A. Trakhtenbrot. Behaviour structures and nets. *Fundamenta Informaticae*, XI(4):357–404, 1988.

[133] W. Reisig. On the semantics of Petri nets. In *Formal Models in Programming*, pages 347–372. North-Holland, 1985.

[134] W. Reisig. *Petri Nets.* EATCS Monographs on Theoretical Computer Science. 1985.

[135] J.-H. Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34:323–346, 1998.

[136] J. Riely and M. Hennessy. Distributed processes and location failures. In *Proceedings ICALP 97*, number 1256 in LNCS, 1997.

[137] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings POPL 98*, 1998.

[138] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings POPL 99*, 1999.

[139] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher order paradigms.* PhD thesis, University of Edinburgh, 1992.

[140] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93*, volume 668 of LNCS, pages 151–166. Springer–Verlag, 1993.

[141] D. Sangiorgi. Locality and interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.

[142] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

[143] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency,. *Theoretical Computer Science*, 170(1–2):297–348, 1996.

[144] P. Sewell. A brief introduction to applied π. Notes from lectures at the MATHFIT Instructional Meeting on Recent Advances in Semantics and Types for Concurrency, Imperial College, 1998.

[145] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings ICALP 98*, number 1443 in LNCS, pages 695–706, 1998.

[146] P. Sewell, P.T. Wojciechowski, and B.C. Pierce. Location independence for mobile agents. In *Proceedings of the 1998 Workshop on Internet Programming Languages*, 1998.

[147] P. Sewell, P.T. Wojciechowski, and B.C. Pierce. Location-independent communication for mobile agents: a two-level architecture, 1998. Technical Report 462, Computer Laboratory, University of Cambridge. Submitted for publication.

[148] P. Sewell, P.T. Wojciechowski, and B.C. Pierce. Nomadic pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA'99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents)*, 1999.

[149] M.W. Shields. Concurrent machines. *The Computer Journal*, 28:449–465, 1985.

[150] D.A. Taubner. *Finite representation of CCS and TCSP programs by automata and Petri nets*, volume 369 of LNCS. Springer-Verlag, 1989.

[151] D.A. Taubner. Representing CCS programs by finite predicate/transition nets. *Acta Informatica*, 27, 1990.

[152] B. Thomsen. Plain CHOCS, a second generation calculus for higher-order processes. *Acta Informatica*, 30:1–59, 1993.

[153] B. Thomsen, L. Leth, S. Prasad, T.M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile Antigua release programming guide. Technical Report ECRC-93-20, ECRC, Munich, 1993.

[154] N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[155] F.W. Vaandrager. A simple definition of parallel composition for prime event structures. Technical Report CS-R8903, CWI, Amsterdam, 1989.

[156] J. Vitek and G. Castagna. Mobile computations and hostile hosts. In *Proceedings of the 10th JFLA (Journées Francophones des Langages Applicatifs*, 1999.

[157] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In D. Tsichritzis, editor, *Workshop on Internet Programming Languages*, 1999.

[158] W. Vogler. Behaviour preserving refinements of Petri nets. In *Proceedings Workshop on Graph-Theoretic Concepts in Computer Science*, number 246 in LNCS, 1987.

[159] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

[160] J. Winkowski. Behaviours of concurrent systems. *Theoretical Computer Science*, 12:39–60, 1980.

[161] G. Winskel. *Events in Computation.* PhD thesis, University of Edinburgh, 1980.

[162] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings ICALP 82*, number 140 in LNCS, 1982.

[163] G. Winskel. Categories of models for concurrency. In *Proceedings Seminar on concurrency*, number 197 in LNCS, 1984.

[164] G. Winskel. Event structures. In *Advances in Petri Nets 1986, number 255 in LNCS*, 1987.

[165] G. Winskel. Petri nets, algebras, morphisms and compositionality. *Information and Control*, 72:197–238, 1987.

[166] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford, 1995.

[167] D. Yankelevich. *Parametric Views of Process Description Languages.* PhD thesis, University of Pisa, 1993.