Thesis Proposal

Synchronization strategies for global computing models

Ivan Lanese

Dipartimento di Informatica, Università di Pisa
e-mail:lanese@di.unipi.it

Abstract

This proposal belongs to the field of formal models for global computing systems. In particular, we are interested in models at an high level of abstraction, where complex synchronizations involving multiple components are available as primitives. Our initial effort is devoted to study and compare existing models. In particular, we analyze relations between synchronization, name-handling and concurrency by comparing a process-calculus, namely Fusion Calculus, a graph transformation formalism, namely Synchronized Hyperedge Replacement, and logic programming. As a result, we also present an extension of logic programming with a restriction operator, which allows to define an observational semantics. We have also analyzed compositionality issues, comparing the algebraic approach of the Tile Model with the categorical approach of CommUnity. Future works will be devoted to the analysis of more complex synchronization strategies, in particular using a declarative approach where equations are associated to interfaces, and possible behaviours of the system are derived by solving the obtained system of equations. We will not concentrate on the operational semantics of systems as done until now for Fusion Calculus and Synchronized Hyperedge Replacement, but we will focus on the observational one. In order to give a unified view to synchronization strategies, we will extend the approach of synchronization algebras to deal with name-handling and mobility. This unified view will be used to define general models, which can be instantiated by choosing a synchronization algebra to define their synchronization strategy. Finally, we will study the properties and the expressive power of hybrid models, that is models obtained by combining features taken from different existing models. An example of hybrid model is the already mentioned logic programming with restriction.

Contents

Chapter 1

Introduction

1.1 Global computing

Global computing (GC), that is computing using networks deployed on world-scale areas, is becoming more and more important. In fact Internet, the most famous and developed GC system, interconnects today hundreds of millions of users, ranging from personal users with their home PC to industrial LANs (local area networks). In addition to Internet, also other WANs (wide area networks) exist, such as wireless communication networks or TV cable networks.

GC systems have different characteristics with respect to previous kinds of systems such as LANs, and one must keep these differences into account not only when designing them, but also in the most frequent case of developing applications for this environment.

First of all, GC systems are *distributed*, but in addition to normal problems posed by distribution in LANs, such as routing problems or remote resource-access problems, new difficulties arise because of the great distances involved. Since the cost of communications (in terms of time, but maybe also in terms of money) is not uniform, applications can not abstract from physical locations. Thus applications must deal with them explicitly in order to reach the wanted levels of quality of service (maybe by connecting to a near server instead of a far away one). In some cases, physical locations can also be involved in a more semantic way in the behaviour of an application. A classical example of location-dependent application is one that provides touristic information, such as restaurant or hotel advertising: it should possibly suggest places near to the user.

This kind of problems becomes fundamental for mobile systems. The importance of *mobility* nowadays is enormous, since everyone has its cellular phone, and maybe also a mobile computer and a PDA. These examples refer to physical mobility. Since one wants that these devices, while moving, keep computing and remain connected to some networks, one wants GC systems to be able to support mobility. This means that locations must be treated as dynamic information, and connectivity can not be taken for granted. Another kind of mobility is software mobility, where hardware is fixed but code can change its location. A common example of software mobility is given by Java applets that can be downloaded from a remote server and then executed locally. A more interesting example is that of software agents, which can move from host to host in order to collect some data and then go back home to deliver the result of their computation. The important thing here is that, not only code, but the whole computation (with its state) is moved. Software mobility can be useful since moving code may be cheaper than moving the data needed by the computation.

Another important feature of GC systems is *eterogeneity*: usually they are composed by hosts which differ in computational power (from mainframes to PDAs), in connectivity (from fast LANs to telephone cables), in memory and resource availability, in operating system and language support and in applicative environment. Thus a main issue in GC is to make eterogeneous systems able to communicate, synchronize and coordinate among themselves, abstracting away from the differences and allowing interoperability. In other cases, differences must not be abstracted, but taken into account in order to exploit them, e.g. by connecting to some host where a particular kind of resource is available at the minimum cost. In order to do that, a negotiation between the client and some possible servers may be needed. Another kind of eterogeneity is of administrative nature: systems can be composed of domains which belong to different organizations, thus being subject to different accounting and security policies. In order to interact with or to move to other domains applications may need to exhibit some authorizations or passwords.

Security is also a direct concern for GC applications, since they can not always trust all the components of the system they communicate with or even their own execution environment (consider e.g. the case of an agent executing on a remote server). Thus issues such as secrecy of communications, integrity of data and code and availability of the provided services must be considered.

Another constraint on most GC systems is that they must be *open*. This means that components can be added to or removed from the system while its other parts continue their executions. A simple, striking example: you can not ask to shut down Internet in order to connect or disconnect your PC. Thus available resources, the parts of a system to be synchronized or the subsystems participating in a computation can change dynamically and without warning. This aspect is even more stressed in the case of mobile systems which can disconnect and then reconnect, maybe from another location. Sometimes not only components are added or removed, but the whole structure of the system can be changed, clearly while keeping the system running. This may happen for many different reasons, for instance in order to satisfy new requirements for the system, or simply in order to optimize the structure of the system w.r.t. the current service requests. Mobile systems are continuously reconfigurated in response to the movement of their components.

Among the features of GC models discussed above, we leave aside security issues and focus on distribution and mobility, since security is usually analyzed using dedicated techniques. As far as eterogeneity and openness are concerned, even when they do not appear explicitly, they must be kept in mind since they influence the choice of the best approach to use. For instance, we can not use models that impose specific requirements on the structure of components since they are not suitable for modeling eterogeneous systems.

1.2 Formal methods for global computing

In order to manage the complexity of the development of applications for such a complex environment, where different problems exist and interact in complex ways, *formal approaches* are needed. Using formal methods one can build models of the system to be developed, in order to be able to analyze some aspects of it, before starting the actual implementation process. This is useful, first of all, to make clear the requirements and then the structure of the system to be built, concentrating on the aspects of interest and abstracting away from details. The developed models can be used to guide and help the implementation of the system (tools exist for generating part of the code from models) and then as reference when testing the actual system. For instance, an executable specification of a system can be used to generate the correct behaviours of a system under test, and then the real system, which is more complex since it must handle many more aspects, can be tested against these behaviours.

The use of formal methods is consolidated in computer science, but traditional models, built for sequential or parallel programs, are not enough for dealing with GC systems. In fact, in addition to standard concerns such as efficiency and input-output behaviour, other aspects have to be considered. As an example, GC models usually handle aspects such as locations or probability (for security concerns for instance) explicitly. Furthermore GC systems are usually made of traditional pre-existing systems which interact in order to reach a common goal. Thus what is usually required is to model *coordination* between given entities, which interact using some set of primitives that depends on the underlying architecture. Thus an important aspect in these models is the separation between computation (which can be abstracted) and coordination (which is the central aspect). Also after this first abstraction, GC models can be large, thus further levels of abstraction can be necessary. For instance, at the low level of abstraction, communications are asynchronous (and maybe faulty) as they are in real systems, but, at the more abstract level, one can consider (part of) the system as synchronous, leaving to other levels the implementation of synchronization mechanisms. Furthermore, at the low level, the basic interacting mechanism is usually a synchronization between two entities, that is an abstraction of some kind of message passing, while at the higher levels synchronizations are wider in space, involving multiple entities, and in time, grouped in transactions that must be completed atomically, or parts of complex reconfigurations. For instance different components of the system must agree on some information, such as the bandwidth dedicated to a data transfer, and at the low level one is interested in the protocol that drives the negotiation. At the more abstract level, components provide their requests, as constraints on the behaviour of the system, and the result of the synchronization is a behaviour that satisfies all the constraints. It is useful to have these complex forms of synchronization as primitives in the model, without the need of building them from simpler ones. The relationships between models at different levels of abstraction can be explicitly defined as refinements between them.

Finally, because of the huge size of the systems involved, it is important to have *compositional* models, that is to be able to combine models of different parts of the system to have a model of the whole system. A common approach to this problem is to define systems that can be interconnected through some kind of interfaces, and then to consider as behaviour of the system the one that can be inferred by analyzing what happens on its interface. A system is compositional if its behaviour can be inferred from the behaviours of its components. Different methods for analyzing this kind of behaviours exist in literature (testing equivalence, bisimulation, ...), and one of their goals is to define when two systems are equivalent, that is when they exhibit the same set of behaviours. Equivalent subsystems are interchangeable without altering the behaviour of the whole system. Furthermore if a system is equivalent to a simpler specification (w.r.t. some set of properties) then it correctly satisfies the requirements imposed by it.

The importance of GC systems, and in particular the need for theoretical research in the field, has been recognized by the European community, which started the GC initiative. This initiative consists of a cluster of projects which are devoted to the analysis of different issues raised by GC systems, with particular attention to the foundational aspect.

Our work is carried on in this framework, and it is mainly related to the projects Agile on architectures for mobility and Profundis on proofs of functionality for mobile distributed systems.

Our particular aim is to analyze and compare existing models for GC systems, in order to understand their commonalities and their differences. This will be useful to understand which features are required in a good GC system, and which are the choices that define a dimension in a "space of GC models". This is important since in the GC area no model has been able to impose itself as a standard, thus our work can both organize the existing models helping the choice of the right model for each particular application and go towards the creation of a general model. In particular, we look for models able to deal with GC systems at the high level of abstraction, thus modeling complex synchronization strategies between multiple components in a declarative style.

In this wide field we concentrate on the aspect of synchronization and on its relations with concurrency, mobility and coordination.

1.3 Existing approaches

We present now some existing models useful in the GC area. We will in particular emphasize their merits and drawbacks.

An important approach is the *process-calculi* one: since process-calculi are situated between real programming languages and mere mathematical abstractions, they facilitate rigorous system analysis, offering the basis for prototypical implementations and for verification tools. Many running implementations exist of languages based on calculi originally proposed to experiment basic interaction primitives. In the process-calculi approach a system is modeled with a term in a suitable term algebra, which has operators such as parallel composition, non-deterministic choice and local resource declaration. The semantics of a system is defined by rule induction as a labelled transition system (SOS semantics, see [Plo81]) or via reduction rules. SOS semantics is quite important since it allows to easily define observational equivalences such as bisimulation or testing equivalence. An important drawback of process-calculi is that they use the same syntactical structures to define both the topological structure of the system and its allowed behaviours. For instance the parallel composition operator can be considered both as parallel composition of systems and as defining the concurrent execution of two processes. Similarly, a restriction operator can be interpreted as definition of an administrative domain, but also as name declaration.

The process-calculi approach borns in the area of concurrent systems with CCS [Mil89] and CSP [Hoa85], which allow to define systems interacting using respectively *Milner synchronization*, where two processes synchronize by doing complementary actions, simulating message passing systems, and *Hoare synchronization*, where any number of processes can synchronize by performing the same action. These calculi are not apt to model GC systems since they can not handle e.g. mobility. This problem is analyzed in π*-calculus* [MPW92], a successor of CCS, which introduces name mobility: instead of moving a process, a name, i.e. a reference to it, is transmitted. This kind of mobility is mathematically simpler but it has been shown powerful enough to model also process mobility [San01]. An important feature of π-calculus is its ability to generate new names which are guarenteed distinct from any other known name. This allows to give access to a resource to exactly one process, that is the one that receives the name, and to model security features such as nonces. The π-calculus is the first calculus of a whole family of calculi for mobility. Among these we are mainly interested in the *Fusion Calculus* [PV98, Vic98], which is a convenient simplification of π-calculus. In fact, it has just one binder for names whereas the π-calculus has two and has symmetric input and output operators while in π-calculus they are not. Nevertheless π-calculus can be syntactically mapped into a subset of Fusion Calculus preserving the one-step semantics [Vic98]. Thus Fusion Calculus is, from this point of view, at least as expressive as its predecessor, and furthermore it allows fusions of names, thus modelling a further mobility feature. An asynchronous version of Fusion Calculus is presented in [GW00].

The Join Calculus [FG96] is similar to the asynchronous π-calculus but enforces a unique receptor for each channel, making it more easily implementable in a distributed setting (since no conflicts among receivers of a message are possible).

Another important calculus is *Ambient calculus* [CG98], which is more suited to study hierarchical models where the mobility space is partitioned in different environments called ambients. Processes live inside ambients and they can drive their ambients with primitives for moving or destroying an ambient. A known problem of Ambient calculus is that it has an unnatural and involved SOS semantics [MZ03], while reduction semantics is simpler.

An interesting calculus is KLAIM [DFP98], a process-calculus based on LINDA [Gel85], which thus uses (distributed) tuple spaces as communication infrastructure. KLAIM is important since it contains primitives for modeling software mobility, thus allowing easy creation of systems composed by mobile agents. Furthermore KLAIM is not only a process-calculus, but also an implemented programming language, and thus can be used for developing real mobile applications.

Finally, we cite *synchronization algebras* [Win86], since even if they do not consider any kind of GC feature such as mobility or openness, still they try to unify different existing models and to give a classification for them. In particular, they present Hoare and Milner synchronizations as different instances of a general pattern.

A quite related approach is the coalgebraic one [Rut00]: a coalgebra in a suitable category[1] can be used e.g. to model a labelled transition system, and the notion of bisimulation naturally arises as the dual notion to that of algebraic congruence. This kind of models is well-suited for studying the observational properties of systems and in particular their relations with compositionality. With this aim, one is interested in combining the observational behaviour of the system, i.e. its coalgebraic structure, with its compositional structure, i.e. its algebraic structure. In particular, one wants that the coalgebraic notion of bisimulation has the algebraic property of being a congruence w.r.t. the operators in the algebra. The first condition proposed that guarentees, by imposing syntactic restrictions on the form of SOS rules, that bisimilarity (i.e. the largest bisimulation) is a congruence, is the so called De Simone format [de 85].

An extension of this approach is the so called *tile logic* [GM00], which is also inspired by rewriting logic [Mes92] and context systems [LX90]. In the Tile Model both the system and the observations are modeled as arrows in a category, and basic blocks called tiles represent the evolution of open system components. In particular, a tile specifies how a given state is rewritten into another one, which are the triggers required from subsystems that make the evolution possible and which is the observable behaviour of the step. Basic tiles can be composed in different ways in order to model interactions of different parts of the system or sequences of computational steps. The composition is allowed only when some conditions

[1] An introduction to category theory can be found in [AHS90].

are met, e.g. the observation of the tile that models a subsystem must coincide with the trigger required by the tile describing the rest of the system. Even if tile logic foundations are in category theory, tile logic can not be considered a truly categorical formalism as far as compositionality is concerned, since its composition primitives do not rely on universal constructions as required by [Gog73], but are essentially algebraic.

A categorical model for communicating systems is *CommUnity* [FM97], a formalism that allows composition of designs in the style of Unity [CM88] but that is based on action sharing and synchronization. Composition is performed using universal constructions from category theory. In particular designs form a category together with morphisms specifying a subsystem relation, and composition is given through colimit. This is an interesting approach to compositionality since a colimit is naturally defined up-to isomorphisms, thus directly abstracting from uninteresting details. A main aspect of CommUnity is the separation of concerns between computation (described inside designs) and coordination (described by morphisms and by special designs called cables). CommUnity is used as an architectural description language.

A classical approach to concurrent systems is the *Petri net* one [Pet62, Rei85]. In this approach, a system is represented by a network of places and transitions. Places hold tokens, which represent the state of the system, and transitions executions can consume tokens from some places producing tokens in other places. Many transitions can be executed simultaneously. Interestingly, both states and computations have a monoidal structure [MM90]. Some steps in the Petri net field towards GC have been accomplished with the introduction of zero-safe nets [BM00], which allow to synchronize different transitions in a transactional style. Furthermore this model can be seen as a refinement of the classical one. The (quite developed) theory of Petri Nets can be generalized to other approaches more apt to GC field, in particular to *graph transformations* [Roz97].

Graphs are a natural model for systems made up of interconnected components, since the topological structure can be directly mapped into the structure of the graph. We mainly consider the approach where edges, or in general hyperedges, namely edges connecting any number of nodes, represent processes or hosts which communicate through ports or links represented by shared nodes, but dual approaches also exist. Graph transformations can animate the graph, thus allowing to model a system whose configuration changes dynamically. In general a graph transformation system is made up of rules that specify how a particular graph (the LHS of the rule) can be rewritten into another one (the RHS of the rule). The approaches differ mainly in the constraints on the structure of the LHS (one node, one edge, a generic graph), in the method used to match the LHS with a subgraph of the graph to be rewritten and on the gluing strategy used to merge the RHS of the rule with the rest of the graph. Sometimes special notation is provided to express the preservation of a part of the LHS in the RHS.

The algebraic approach (so called because graphs are seen as particular algebras), which allows to rewrite generic graphs, started with the double pushout definition [EPS73], which uses two pushout constructions in the category of graphs and total graph morphisms to apply rules. More recently a single pushout in the category of graphs and partial graph morphisms has been used to the same purpose giving rise to the single pushout approach [Löw93]. The use of categorical constructions to define these graph transformations has the advantage of being largely independent on the exact structure of objects to be rewritten, thus it is easily generalized to more complex situations such as higher-level graphs [EHKPP91]. This approach to graph transformations is difficult to implement in a GC setting where graphs are distributed systems. In fact, when writing applications to implement reconfigurations modeled as graph transformations of this kind, we may discover that hosts have not enough information about the topology of the whole system to decide if a rule needs to be applied or not.

Thus a more compositional approach, the *Synchronized Hyperedge Replacement* (SHR), has been proposed. SHR uses context-free rules called productions to describe the evolution of single hyperedges (that is single processes or hosts). Process synchronizations are modeled by imposing conditions on nodes, which specify constraints on which productions can be applied in a single step. When the productions to be applied have been chosen (this is called the rule-matching problem) in such a way that conditions are satisfied, the actual rewriting can be carried on separately by single processes. This approach was first proposed in [DM87] with the name of "Grammars for Distributed Systems". The rule-matching problem can be modeled as a finite domain constraint problem [CDM85] and solved through a distributed algorithm. An interesting extension of this approach allows also communication of nodes during transitions [HIM00], what allows to model mobile systems whose connections change at runtime.

A well-known computational model that can also be used fruitfully in the GC area is *logic programming* [Llo93]. To this end however one must abandon the usual concept of logic programming, where the aim is to find an SLD-refutation for a goal, that is a computation that reaches the empty goal. We consider instead logical predicates as concurrent processes that interact through shared variables using the unification engine as communication mechanism. In this framework one is not interested in the empty goal, which stands for an empty system, but must consider partial or even infinite computations that describe how a system evolves. The first step in this direction is the work on compositionality in logic programming (see e.g. [GS89]). An interesting paper for the GC perspective is [BMR01] where observational semantics is analyzed and open systems are considered (i.e. the semantics can be composed to deal with instantiation of variables and insertion of the goal into a larger context).

1.4 Structure of the proposal

Chapter 2 presents in details the models used in this proposal. In particular, we introduce the algebraic presentation for graphs and the Synchronized Hyperedge

Replacement (SHR) (Section 2.1) and the Fusion Calculus (Section 2.2). Then a short reminder on logic programming is presented (Section 2.3). Finally, we introduce the Tile Model (Section 2.4) and CommUnity (Section 2.5).

Chapter 3 presents a comparative study of Fusion Calculus, SHR with both Hoare and Milner synchronizations and logic programming. In particular, Section 3.1 presents a mapping from Fusion Calculus to Milner SHR (with special rules to force an interleaving behaviour). This part of the work is treated in [LM03]. Section 3.2 presents a comparison between Hoare and Milner synchronization in the framework of SHR, that is with mobility and multiple synchronizations. This is ongoing work, but some of the ideas are presented in [LM04]. Section 3.3 extends logic programming by adding a restriction operator, analogous to the scope operator of Fusion Calculus and to the restriction of SHR, to make it more suitable to model GC systems. A first approach to this problem is shown in [Lan02]. Section 3.4 presents a comparison between Hoare SHR and logic programming (also in the version with restriction), defining a subset of logic programming which has strong analogies with SHR. This work has been presented first in [Lan02], and more refined versions (but without the restriction operator) are shown in [LM02, LM04]. Finally, Section 3.5 presents an example that shows how the different steps can be composed, going from Fusion Calculus to logic programming.

Chapter 4 concentrates on strategies for composing systems, comparing the categorical approach of CommUnity with the algebraic approach of the Tile Model. In particular, we present a decomposition for CommUnity designs that highlights their algebraic structure (Section 4.1), a mapping from CommUnity into the Tile Model that preserves the operational behaviour (Section 4.2) and an axiomatization of systems in the Tile Model whose equivalence classes correspond to colimits of CommUnity systems (Section 4.3). What we do is to mimic algebraically the categorical universal construction that CommUnity uses to build systems.

Chapter 5 presents the roadmap for future work, with some ideas on the approaches that can be used to solve the problem of modeling GC systems at the abstract level.

Chapter 2

Technical background

2.1 Synchronized Hyperedge Replacement

Synchronized Hyperedge Replacement (SHR) [DM87] is an approach to *(hyper)graph transformations* that defines *global transitions* using *local productions*. Productions define how a single hyperedge can be rewritten and the *conditions* that this rewriting imposes on adjacent nodes. Thus the global transition is obtained by applying in parallel different productions whose conditions are compatible. What exactly compatible means depends on which *synchronization model* we use. In this work we will use both the Hoare and the Milner synchronization model.
We use the extension of SHR with *mobility* [HIM00, FMT01, Lan02], that allows edges to expose node references together with actions, and nodes whose references are matched during synchronization are unified.

We give a formal description of SHR in terms of labelled transition system, but first of all we need an algebraic representation for hypergraphs.

A hyperedge, or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,\ldots}$) and with as many ordered tentacles as the rank of its label. A set of nodes, together with a set of such edges, forms a hypergraph (or simply a graph) if each edge is connected, by its tentacles, to its attachment nodes. A graph is equipped with a set of *external nodes* identified by distinct names. External nodes can be seen as the connecting points of a graph with its environment (i.e. the context). We consider graphs up-to isomorphisms that preserve external nodes, labels of edges, and connections between edges and nodes.
Now, we present a definition of graphs as syntactic judgements, where nodes correspond to names, external nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE_n$.

Definition 1 (Graphs as syntactic judgements) *Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A syntactic judgement (or simply a judgement) is of the form $\Gamma \vdash G$ where:*

1. *$\Gamma \subseteq \mathcal{N}$ is a finite set of names (the external nodes of the graph).*

2. *G is a term generated by the grammar*
   $G ::= L(\vec{x}) \mid G|G \mid \nu y\ G \mid nil$
   where $\vec{x}$ is a vector of names and L is an edge label with $\mathrm{rank}(L) = |\vec{x}|$ *and y is a name.*

We define ν as a binder, so in $\nu y\ G$ all the occurrences of y are bound occurrences. We denote with fn, bn and n the functions that given a graph G return the sets $\mathrm{fn}(G)$, $\mathrm{bn}(G)$ and $\mathrm{n}(G)$ of respectively free, bound and all the names in G. We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2.

Definition 2 (Structural congruence and well-formed judgements) *The structural congruence $\equiv$ on terms G obeys the following axioms:*

$$\textit{(AG1)}\ (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3)$$

$$\textit{(AG2)}\ G_1|G_2 \equiv G_2|G_1$$

$$\textit{(AG3)}\ G|nil \equiv G$$

$$\textit{(AG4)}\ \nu x\ \nu y\ G \equiv \nu y\ \nu x\ G$$

$$\textit{(AG5)}\ \nu x\ G \equiv G\ \textit{if}\ x \notin \mathrm{fn}(G)$$

$$\textit{(AG6)}\ \nu x\ G \equiv \nu y\ G[y/x]\ \textit{if}\ y \notin \mathrm{fn}(G)$$

$$\textit{(AG7)}\ \nu x\ (G_1|G_2) \equiv (\nu x\ G_1)|G_2\ \textit{if}\ x \notin \mathrm{fn}(G_2)$$

The well-formed judgements for constructing graphs over LE and $\mathcal{N}$ are those generated by applying the following syntactic rules.

$$\textit{(RG1)}\ \frac{x_i \in \mathcal{N}, i = 1, \ldots, n}{x_1, \ldots, x_n \vdash nil}$$

$$\textit{(RG2)}\ \frac{L \in LE_m \qquad x_i \in \mathcal{N}, i = 1, \ldots, n \qquad y_j \in \{x_1, \ldots, x_n\}, j = 1, \ldots, m}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}$$

$$\textit{(RG3)}\ \frac{\Gamma \vdash G_1 \qquad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2}$$

$$\textit{(RG4)}\ \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu x\ G}$$

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

Rule (RG1) creates a graph with no edges and n nodes and rule (RG2) creates a graph with n nodes and one edge labelled by L and with m tentacles (note that some tentacles can be attached to the same node). Rule (RG3) allows the composition of two graphs that share the same set of nodes. Rule (RG4) hides a node.

Note that function fn is well-defined also on equivalence classes. Thanks to axiom (AG4), we can write νX, with $X = \bigcup_{i=1\ldots n}\{x_i\}$, to abbreviate $\nu x_1\, \nu x_2 \ldots \nu x_n$. Note that by using the axioms, we can always put any judgement in the normal form $\Gamma \vdash \nu X\; G$, where G is a subterm containing only compositions of edges. We can also make Γ and X be disjoint sets of nodes. Note that $\mathrm{n}(G) \subseteq \Gamma \cup X$.

We can state the following correspondence theorem.

Theorem 1 (Correspondence of graphs and judgements)
Well-formed judgements up to structural axioms are isomorphic to graphs up to isomorphisms.

We present now the steps of an SHR computation.

Definition 3 (SHR transition) *Let Act be a ranked set of actions. The rank of $a \in Act$ is $\mathrm{ar}(a)$. A SHR transition is of the form:*

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

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

We define:

- $\mathrm{n}(\Lambda) = \{z | \exists x. z \in \mathrm{n}_\Lambda(x)\}$
  set of exposed names;
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$
  set of fresh names that are exposed;
- $\mathrm{n}(\pi) = \{x | \exists x' \neq x. \pi(x) = \pi(x')\}$
  set of fused names.

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

Note that the set of names Φ of the resulting graph is fully determined by Γ, Λ and π and that we can write a SHR transition as

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \pi(\Gamma), \Gamma_\Lambda \vdash G'.$$

We usually assume to have an action $\epsilon \in Act$ of arity 0 to denote "no synchronization". We may not write explicitly π if it is the identity, and some actions if they are $(\epsilon, \langle\rangle)$. Furthermore we use Λ_ϵ to denote the function such that $\Lambda_\epsilon(x) = (\epsilon, \langle\rangle)$ for each node x in Γ (note that the dependence on Γ is implicit). When writing Λ as set of pairs we write a triple $(x, a, \vec{y})$ for the pair $(x, (a, \vec{y}))$.

We derive SHR transitions from basic productions using a set of inference rules. Productions define the behaviours of single edges.

Definition 4 (Production) *A production is an SHR transition of the form:*

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$$

where all x_i, $i = 1, \ldots, n$ are distinct.

Productions are to be considered as schemas and so when we have a production we suppose to have also all the productions obtainable through α-conversion of names in $\{x_1, \ldots, x_n\} \cup \Phi$.

Given a set of productions $\mathcal{P}$ we write $\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ if $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ can be obtained from the productions in $\mathcal{P}$ using the inference rules.

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

Definition 5 (Rules for Hoare synchronization)

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

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

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

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

i. $\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$

ii. $\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$

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

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

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

where:
$(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
$Z = \Phi \setminus \Phi'$

$$\textit{(idle)} \quad \Gamma \vdash G \xrightarrow{\Lambda_\epsilon,id} \Gamma \vdash G$$

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

A transition is obtained by composing productions, which are first applied on disconnected edges, and then by connecting the edges by merging nodes. In particular, rule (par) deals with the composition of transitions which use disjoint sets of nodes and rule (merge) allows to merge nodes (note that σ is a projection into representatives of equivalence classes). When two nodes are merged, (i) we must have on them the same action, with possibly different tuples of names. Definition (ii) introduces the most general unifier ρ of the union of two sets of equations: the first set identifies (the representatives of) the tuples associated to nodes merged by σ, while the second set of equations propagates merges in π to their representatives. Thus ρ is the merge resulting from both π and σ. Note that (iii) Λ is updated with these merges and that (iv) π' is ρ restricted to the nodes of the graph which is the source of the transition. Rule (res) allows to restrict a node x in the starting graph. We require this node not to be a representative of an equivalence class according to π. The set Z contains new nodes which are exposed only on x and thus must be hidden. Rule (idle) guarantees that each edge can always make an explicit idle step. Rule (new) allows adding to the source graph an isolated node where arbitrary actions are exposed.

We write $\mathcal{P} \Vdash_H (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ to stress the fact that the transition is derived using Hoare synchronization model.

Another important synchronization model is the Milner one. In Milner synchronization we suppose actions to be grouped in pairs (action-coaction) (such as $(a, \overline{a})$). The overline operator is such that $\overline{\overline{a}} = a$. Actions are usually considered as inputs and coactions as outputs. Milner synchronization happens when an edge exposes an action and another one exposes the corresponding coaction. All other edges must perform the ϵ action on that node. We also have a τ action of arity 0 that stands for an internal synchronization (this is the result of synchronizing two complementary actions). Note that the overline operator is not defined on ϵ and τ.

We will now present the set of inference rules for Milner synchronization.

Definition 6 (Rules for Milner synchronization)

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

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

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

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

i. $\sigma(x) = \sigma(y) \wedge \Lambda(x) \neq (\epsilon, \langle\rangle) \wedge \Lambda(y) \neq (\epsilon, \langle\rangle) \wedge x \neq y \Rightarrow$
$(\forall z \in \mathcal{N} \setminus \{x, y\}.\sigma(z) = \sigma(x) \Rightarrow \Lambda(z) = (\epsilon, \langle\rangle))$
$\wedge\Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w})$

ii. $\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w})\}$
$\cup\{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$

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

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

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

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

where:

- $(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
- $\Lambda(x) = (\epsilon, \langle\rangle) \vee \Lambda(x) = (\tau, \langle\rangle)$
- $Z = \{x\}$ *if* $x \notin \mathrm{n}(\Lambda|_\Gamma), Z = \emptyset$ *otherwise*

$$\textit{(idle)}\quad \Gamma \vdash G \xrightarrow{\Lambda_\epsilon,id} \Gamma \vdash G$$

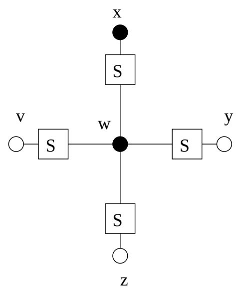


Figure 2.1: Star graph.

This set of rules is quite similar to the one for Hoare synchronization. The main differences are that in rule (merge) we require (i) that in a group of merged nodes at most two expose non trivial actions, furthermore if they are exactly two then they must send complementary actions. The result of such a synchronization is (iii) a τ action. Finally, since τ action has arity 0 then nodes exposed only by synchronized actions must be hidden (v). In rule (res) we have the additional condition that only nodes with ϵ or τ can be hidden, since other actions stand for partial synchronizations that must be completed. Note also that we have no rule (new) since on isolated nodes we can have just ϵ, what can be obtained using rules (idle) and (par).

We write $\mathcal{P} \Vdash_M (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ to stress the fact that the transition is derived using Milner synchronization model.

We will now present an example of Hoare SHR (HSHR) computation.

Example 1 ([HIM00]) *We will now show how to use HSHR to derive a 4 elements ring starting from a one element ring, and how we can then specify a reconfiguration that transforms the ring into the star graph in Figure 2.1.*

We use the following productions:

$$x, y \vdash C(x, y) \xrightarrow{(x,\epsilon,\langle\rangle),(y,\epsilon,\langle\rangle)} x, y \vdash \nu z\ C(x, z)|C(z, y)$$

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

that are graphically represented in Figure 2.2. Notice that Λ is represented by decorating every node x in the left hand with $\mathrm{act}_\Lambda(x)$ and $\mathrm{n}_\Lambda(x)$. Hidden nodes are drawn in white while external nodes are black. The first rule allows to create rings, in fact we can create all rings with computations like:

$$\begin{aligned} & x \vdash C(x, x) \to x \vdash \nu y\ C(x, y)|C(y, x) \to \\ & \qquad \to x \vdash \nu y, z\ C(x, y)|C(y, z)|C(z, x) \to \\ & \qquad\qquad \to x \vdash \nu y, z, v\ C(x, y)|C(y, z)|C(z, v)|C(v, x) \end{aligned}$$

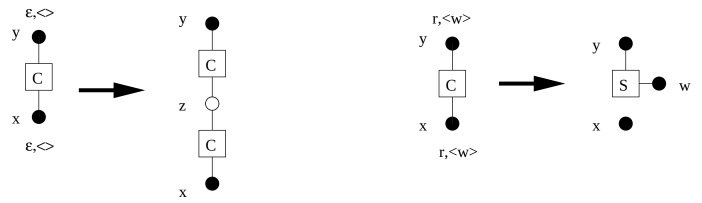


Figure 2.2: Productions.

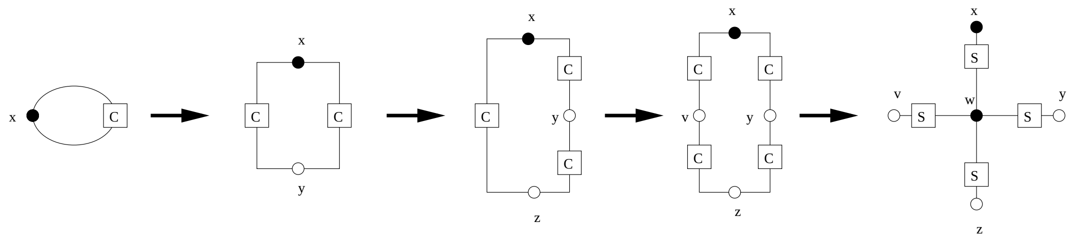


Figure 2.3: Ring creation and reconfiguration to star.

Note that new nodes are all internal nodes. In order to perform the reconfiguration into a star, we need rules with nontrivial actions, like the second one. This allows to do:

$$x \vdash \nu y, z, v\ C(x,y)|C(y,z)|C(z,v)|C(v,x) \xrightarrow{(x,r,\langle w\rangle)}$$
$$\to x, w \vdash \nu y, z, v\ S(x,w)|S(y,w)|S(z,w)|S(v,w)$$

Note that if an edge C is rewritten into an edge S, then all the edges in the ring must use the same production, since they must synchronize via action r. They must agree also on $\mathrm{n}_\Lambda(x)$ for every x, thus all the newly created nodes are merged. Note that node w is external since it has been exposed on the external node x. The whole transition is represented in Figure 2.3.

It is easy to show that if we can derive a transition T, then we can also derive every transition obtainable from T by applying an injective renaming.

Proposition 1 *Let $\mathcal{P}$ be a set of productions and σ an injective substitution.*
$\mathcal{P} \Vdash (\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2)$ *iff:*
$\mathcal{P} \Vdash (\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \sigma(\Phi) \vdash \sigma(G_2))$
where $\Lambda'(\sigma(x)) = \sigma(\Lambda(x))$ *and* $\pi'(\sigma(x)) = \sigma(\pi(x))$.

2.2 Fusion Calculus

The Fusion Calculus [PV98] is a calculus for modeling distributed and mobile systems which is based on the concepts of *fusion* and *scope*. It is an evolution of the

π-calculus [MPW92], and the interesting point is that it is obtained by simplifying the calculus. In fact, the two action prefixes for communication are symmetric whereas in the π-calculus they are not, and we have just one binding operator called scope whereas the π-calculus has two (restriction and input). As shown in [PV98] the π-calculus is syntactically a subcalculus of the Fusion Calculus (the key point is that the input of π-calculus is obtained using input and scope). In order to have these properties fusion actions have to be introduced.

We present here in detail the syntax and the SOS semantics of Fusion Calculus. In our discussion we distinguish between *sequential agents* (which have a summation as topmost operator) and general agents.

We suppose to have an infinite set $\mathcal{N}$ of names ranged over by $u, v, \ldots, z$. Names represent communication channels. We use $\vec{x}$ to denote a vector of names and ϕ to denote an equivalence relation on $\mathcal{N}$ which can be represented by a finite set of equalities (we denote with 1 the identity relation).

Definition 7 *The free actions are defined by:*

$$\begin{array}{rcl} \alpha ::= & u\vec{x} & \textit{(Input)} \\ & \overline{u}\vec{x} & \textit{(Output)} \\ & \phi & \textit{(Fusion)} \end{array}$$

Definition 8 *The agents are defined by:*

$$\begin{array}{rcl} S ::= & \sum_i \alpha_i . P_i & \textit{(Guarded sum)} \\ \\ P ::= & 0 & \textit{(Inaction)} \\ & S & \textit{(Sequential Agent)} \\ & P_1 | P_2 & \textit{(Composition)} \\ & (x)P & \textit{(Scope)} \\ & \mathrm{rec}\, X.P & \textit{(Recursion)} \\ & X & \textit{(Agent variable)} \\ & [x = y]P & \textit{(Match)} \\ & [x \neq y]P & \textit{(Mismatch)} \end{array}$$

The scope operator is a binder for names thus x is bound in $(x)P$. Similarly rec is a binder for agent variables. Note that this syntax allows only agents with guarded summation. Furthermore we only consider agents which are closed w.r.t. agent variables and where in $\mathrm{rec}\, X.P$ each occurrence of X is within a sequential agent (guarded recursion).

We define the functions fn, bn and n that given an agent P compute the sets $\mathrm{fn}(P)$, $\mathrm{bn}(P)$ and $\mathrm{n}(P)$ of respectively free, bound and all the names in P (the

$$\frac{\alpha.P \xrightarrow{\alpha} P}{-}\ PREF$$

$$\frac{P + Q \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P'}\ SUM$$

$$\frac{P|Q \xrightarrow{\alpha} P'|Q}{P \xrightarrow{\alpha} P'}\ PAR \qquad \frac{P|Q \xrightarrow{\{\vec{x}=\vec{y}\}} P'|Q'}{P \xrightarrow{u\vec{x}} P', Q \xrightarrow{\overline{u}\vec{y}} Q', |\vec{x}| = |\vec{y}|}\ COM$$

$$\frac{(z)P \xrightarrow{\phi \backslash z} P'\{x/z\}}{P \xrightarrow{\phi} P', z\phi x, z \neq x}\ SCOPE$$

$$\frac{(z)P \xrightarrow{\alpha} (z)P'}{P \xrightarrow{\alpha} P', z \notin \mathrm{n}(\alpha)}\ PASS \qquad \frac{(z)P \xrightarrow{(z\vec{y})a\vec{x}} P'}{P \xrightarrow{(\vec{y})a\vec{x}} P', z \in \vec{x} \setminus \vec{y}, a \notin \{z, \overline{z}\}}\ OPEN$$

$$\frac{[x = x]P \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P'}\ MATCH \qquad \frac{[x \neq y]P \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P', x \neq y}\ MISMATCH$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \xrightarrow{\alpha} P' \quad P \equiv Q \quad P' \equiv Q'}\ STRUCT$$

Table 2.1: SOS semantics for Fusion Calculus.

names in a fusion are the names in non singleton equivalence classes). Analogous functions can be defined for actions.

Processes are agents considered up-to structural axioms defined as follows.

Definition 9 *The structural congruence* $\equiv$ *between agents is the least congruence satisfying the* α*-conversion law (both for names and for agent variables), the abelian monoid laws for summation and composition (associativity, commutativity and 0 as identity), the scope laws* $(x)0 \equiv 0$, $(x)(y)P \equiv (y)(x)P$, *the scope extension law* $P|(z)Q \equiv (z)(P|Q)$ *where* $z \notin \mathrm{fn}(P)$ *and the recursion law* $\mathrm{rec}\, X.P \equiv P\{\mathrm{rec}\, X.P/X\}$.

Note that fn can be trivially extended to processes.

Definition 10 *A bound action is of the form* $(\vec{z})a\vec{x}$ *where* $|\vec{z}| > 0$ *and all elements in* $\vec{z}$ *are also in* $\vec{x}$. *Names in* $\vec{z}$ *are bound names. The actions consist of the free actions and the bound actions.*

For convenience we define $\phi \setminus z$ to mean $\phi \cap (\mathcal{N} \setminus \{z\})^2 \cup \{(z, z)\}$. The SOS semantics of Fusion Calculus is in Table 2.1.

When only closed agents are considered one can use the simpler reduction semantics [Vic98]. We need the following auxiliary definition.

Definition 11 (Substitutive effect) *A substitutive effect of a fusion ϕ is any idempotent substitution $\sigma : \mathcal{N} \rightarrow \mathcal{N}$ having ϕ as its kernel. In other words $\sigma(x) = \sigma(y)$ iff $x\phi y$ and σ sends all members of each equivalence class of ϕ to one representative in the class.*

The reduction semantics for Fusion Calculus (we present here just the rules for Fusion Calculus without match and mismatch) is the least relation satisfying the following rules.

Definition 12 (Reduction semantics for Fusion Calculus)

$$(\vec{z})(R|(\cdots + u\vec{x}.P)|(\overline{u}\vec{y}.Q + \dots)) \rightarrow (\vec{z})(R|P|Q)\sigma$$

where $|\vec{x}| = |\vec{y}|$ and σ is a substitutive effect of $\{\vec{x} = \vec{y}\}$.

$$(\vec{z})(R|(\cdots + \phi.P)) \rightarrow (\vec{z})(R|P)\sigma$$

where σ is a substitutive effect of ϕ.

$$\frac{P \equiv P', P' \rightarrow Q', Q' \equiv Q}{P \rightarrow Q}$$

The relation between SOS semantics and reduction semantics is shown in the following theorem.

Theorem 2 *Let P be a closed fusion process without match and mismatch. Then $P \xrightarrow{1} Q$ iff $P \rightarrow Q$.*

We show here a useful decomposition that can be defined for fusion agents.

Definition 13 *The standard decomposition of an agent P is defined as:*

$$P = \sigma_P \hat{P}$$

where σ_P is the standard substitution and $\hat{P}$ is the standard agent of P.
This decomposition satisfies:

$$P = \sigma Q \text{ implies } \hat{P} = \hat{Q} \wedge \sigma_P = \sigma\sigma_Q$$

We denote with $\mathrm{fnarray}(P)$ the array of the free name occurrences in P, ordered according to some fixed order dictated by the structure of P. In particular $\sigma_P = \{\mathrm{fnarray}(P)/\mathrm{fnarray}(\hat{P})\}$.

The standard decomposition can be easily extended to processes.

2.3 Logic programming

For a complete presentation of the topic see [Llo93].

In this work we are not interseted in logic computations as refutations of goals for problem solving or artificial intelligence, but we consider logic programming in a more interactive way as a *goal rewriting mechanism*. We can consider logic goals as concurrent processes that evolves according to the rules defined by the clauses and using unification as the fundamental interacting primitive. This kind of use of logic programming can be found in [BMR01].

In order to stress the similarities between logic programming and process-calculi we present a semantics of logic programming based on a labelled transition system. We suppose to have finite sets of predicates and function symbols and a countable set of variables on which clauses are defined.

Definition 14 (Clause) *A clause is a formula of the form:*

$$\forall\ A_1 \vee \cdots \vee A_k \leftarrow B_1 \wedge \cdots \wedge B_n$$

where each A_i and B_i are atoms (predicates applied to terms).

We will write clauses in the usual logic programming notation as:

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$$

Definition 15 *We have for clauses and goals the following grammar:*

$$C ::= A \leftarrow G$$

$$G ::= G, G \mid A \mid \Box$$

where A is a logical atom, "," is the AND conjunction and $\Box$ is the empty goal.
We can suppose "," to be associative and commutative and with unit $\Box$.

Definition 16 (Definite program clause)
A (definite) program clause is in the syntactic category C and is a clause of the form:

$$A \leftarrow B_1, \ldots, B_n$$

A is called the head and $B_1, \ldots, B_n$ the body of the program clause.

The informal semantics of $A \leftarrow B_1, \ldots, B_n$ is "for each assignment of each variable if $B_1, \ldots, B_n$ are all true, then A is true".

Definition 17 (Definite program) *A definite program is a finite set of definite program clauses.*

$$\begin{tikzcd} \textit{initial input interface}\quad x \arrow[r, "s"] \arrow[d, "a"'] \arrow[dr, phantom, "\alpha"] & y \quad \textit{initial output interface} \arrow[d, "b"] \\ \textit{final input interface}\quad z \arrow[r, "t"'] & w \quad \textit{final output interface} \end{tikzcd}$$

Figure 2.4: A generic TL sequent α (whence the name 'tile').

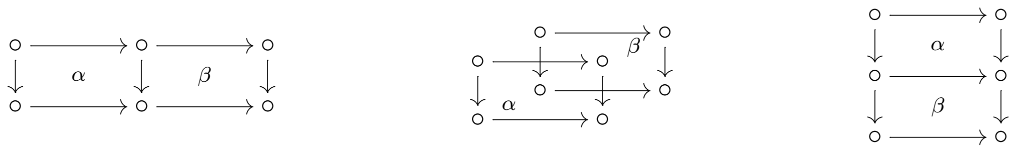


Figure 2.5: Horizontal, parallel and vertical tile compositions.

Definition 18 (Definite goal) *A (definite) goal is in the syntactic category G and is a clause of the form:*

$$\leftarrow B_1, \ldots, B_n$$

Each B_i is called a subgoal of the goal. We may drop the $\leftarrow$ symbol.

Definition 19 (Partial SLD-derivation) *Let P be a logic program. We define a step of an SLD-resolution computation using the following rules:*

$$\frac{H \leftarrow B_1, \ldots, B_k \in P \qquad \theta = \mathrm{mgu}(A, \rho(H))}{P \Vdash A \xrightarrow{\theta} \rho(B_1, \ldots, B_k)\theta} \qquad \textit{atomic goal}$$

where ρ is an injective renaming of variables such that all the variables in the clause variant $\rho(H \leftarrow B_1, \ldots, B_k)$ are fresh.

$$\frac{P \Vdash G \xrightarrow{\theta} F}{P \Vdash G, G' \xrightarrow{\theta} F, G'\theta} \qquad \textit{conjunctive goal}$$

We will omit $P \Vdash$ if P is clear from the context. A partial SLD-derivation using P on a goal G is a sequence (eventually empty) of steps of SLD-resolution that uses clauses in P with initial goal G. If G' is the final goal then we write $P \Vdash G \xrightarrow{\theta}{}^{*} G'$ where θ is the composition of all the mgus.

2.4 The Tile Model

The Tile Model [GM00] relies on rewrite rules with side effects, called *basic tiles*, which extend SOS rules and rewriting logic [Mes92] (RL). As for RL, the Tile Model admits a logical presentation, called tile logic (TL), where tiles are seen as sequents subject to inference rules. TL extends RL by handling rewrites with side effects and synchronization.

A TL sequent α has the graphical representation in Figure 2.4, also written $\alpha : s \xrightarrow[b]{a} t$, stating that the *initial configuration* s can evolve to the *final configuration* t via α, producing the *effect* b, which can be observed by the environment; but the step is allowed only if the 'arguments' of s can contribute producing a, which acts as the *trigger* of α. Triggers and effects are called *observations*; tile vertices are called *interfaces*. Configurations and observations are represented by arrows to show that they can be composed via their interfaces.

TL is designed for systems that are *compositional in space* (coordination of components) and in *time*: tiles can be composed horizontally, in parallel, and vertically to generate larger steps (see Figure 2.5). Horizontal composition $(\alpha;\beta)$ coordinates the evolution of the initial configuration of α with the evolution of the environment (the initial configuration of β) yielding the 'synchronization' of the two rewrites. The monoidal parallel composition $(\alpha \otimes \beta)$ builds concurrent steps. Vertical composition $(\alpha * \beta)$ is sequential composition of computations. Given a set of basic tiles, the associated TL is obtained by adding some canonical 'auxiliary' tiles and then closing by all kinds of composition. The exact set of auxiliary tiles depends on the TL format under consideration.

The main prerequisite for expressing the operational semantics of concurrent systems via tiles is that system configurations form a monoidal category (see [AHS90] for a general presentation of category theory) $\mathcal{H}$, such that a suitable monoidal category $\mathcal{V}$ of observations can be selected with the same set of underlying objects. In fact, tiles exploit this property to establish the correspondence between the description of states and their evolution in all basic steps that can be performed.

Definition 20 *A* tile system *is a tuple $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ where $\mathcal{H}$ and $\mathcal{V}$ are monoidal categories with the same set of objects $O_{\mathcal{H}} = O_{\mathcal{V}}$, N is the set of rule names and $R\colon N \to \mathcal{H} \times \mathcal{V} \times \mathcal{V} \times \mathcal{H}$ is a function such that for all $\alpha \in N$, if $R(\alpha) = \langle s, a, b, t\rangle$, then the arrows s, a, b, t can form a tile like in Figure 2.4.*

We focus on tile systems where $\mathcal{H}$ and $\mathcal{V}$ are *categories of substitutions*. Substitutions and their composition $\_;\_$ form a cartesian category $\mathbf{Subs}_\Sigma$. An alternative presentation of $\mathbf{Subs}_\Sigma$ is obtained by resorting to Lawvere's *algebraic theories* [Law63], where cartesianity is expressed by imposing a symmetric monoidal structure $(\otimes, \underline{0}, \gamma)$ enriched with two natural transformations, a duplicator $\nabla = \{\nabla_n : \underline{n} \to \underline{2n}\}$ and a discharger $! = \{!_n : \underline{n} \to \underline{0}\}$. The result is a cartesian category $\mathbf{Th}[\Sigma]$ whose objects are 'underlined' natural numbers; whose arrows from $\underline{m}$ to $\underline{n}$ are in one-to-one correspondence with n-tuples of terms of the free Σ-algebra over m variables; and whose arrow composition is term substitution. In particular, $\mathbf{Th}[\Sigma]$ is equivalent to $\mathbf{Subs}_\Sigma$, and the arrows from $\underline{0}$ to $\underline{1}$ are in bijective correspondence with the closed terms over Σ. We assume the standard naming $x_1, ..., x_m$ of the m input variables. For example, $f \in \Sigma_2$ defines an arrow $f(x_1, x_2)\colon \underline{2} \to \underline{1}$ in $\mathbf{Th}[\Sigma]$. When composing $\vec{s}\colon \underline{m} \to \underline{k}$ and $\vec{t}\colon \underline{k} \to \underline{n}$, the result $\vec{s};\vec{t}$ is obtained by replacing each occurrence of x_i in $\vec{t}$ by the i-th term of the tuple $\vec{s}$, for $i \in [1, k]$.

$$\frac{R(\alpha) = \langle s, a, b, t\rangle}{s \xrightarrow[b]{a} t}\ (\texttt{bas}) \qquad \frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[c]{b} f}{s; h \xrightarrow[c]{a} t; f}\ (\texttt{hor}) \qquad \frac{t\colon x \to y \in \mathcal{H}}{t \xrightarrow[id_y]{id_x} t}\ (\texttt{vid})$$

$$\frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[d]{c} f}{s \otimes h \xrightarrow[b\otimes d]{a\otimes c} t \otimes f}\ (\texttt{par}) \qquad \frac{s \xrightarrow[b]{a} t \quad t \xrightarrow[d]{c} h}{s \xrightarrow[b;d]{a;c} h}\ (\texttt{ver}) \qquad \frac{a\colon x \to z \in \mathcal{V}}{id_x \xrightarrow[a]{a} id_z}\ (\texttt{hid})$$

Figure 2.6: Inference rules for (flat) tile logic.

The identity arrow $\langle x_1, ..., x_n\rangle$ for the object $\underline{n}$ (with brackets denoting term tupling) is written $id_{\underline{n}}$. The empty substitution $\langle\rangle = id_{\underline{0}}$ is the unique arrow from $\underline{0}$ to $\underline{0}$.

In this work we deal with substitutions on multi-sorted terms, thus instead of natural numbers we have monoids on the set of sorts. For instance if a and b are sorts we have $\gamma_{a,b} : a \otimes b \to b \otimes a$ and $\nabla_a : a \to a \otimes a$.

We always have the horizontal and vertical identities as auxiliary tiles (vertical identities model idle components, while horizontal identities serve for propagation of effect through identity substitutions). All this is defined in Figure 2.6. Depending on the chosen tile format, $\mathcal{H}$ and $\mathcal{V}$ must satisfy certain constraints and some other auxiliary tiles are added and composed with basic tiles and identities. In particular in this work we use permutations, which need the auxiliary tiles $\gamma_{a,b} \xrightarrow[id_b\otimes id_a]{id_a\otimes id_b} \gamma_{a,b}$ and $id_a \otimes id_b \xrightarrow[\gamma_{a,b}]{\gamma_{a,b}} id_b \otimes id_a$. The set of resulting tiles (*flat sequents*) defines the *flat tile logic* associated with $\mathcal{R}$. We say that any flat sequent $s \xrightarrow[b]{a} t$ is entailed by the logic, written $\mathcal{R} \vdash s \xrightarrow[b]{a} t$. Note that we are only concerned with the existence of a tile with a certain border, as opposed to the theory of decorated sequents, based on rule names and derivation proofs.

By taking $\langle\text{trigger}, \text{effect}\rangle$ pairs as labels one can see tiles as forming a labelled transition system. In this context the usual notion of bisimilarity is called *tile bisimilarity.*

Definition 21 *Let $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ be a tile system. A symmetric relation $\sim_{\mathrm{t}}$ on configurations is called* tile bisimulation *if whenever $s \sim_{\mathrm{t}} t$ and $\mathcal{R} \vdash s \xrightarrow[b]{a} s'$, then t' exists such that $\mathcal{R} \vdash t \xrightarrow[b]{a} t'$ and $s' \sim_{\mathrm{t}} t'$. The maximal tile bisimulation is denoted by $\simeq_{\mathrm{t}}$, and two configurations s and t are* tile bisimilar *if $s \simeq_{\mathrm{t}} t$.*

Definition 22 *A tile system $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ enjoys the* decomposition property *if for all $s \in \mathcal{H}$ and for all $\mathcal{R} \vdash s \xrightarrow[b]{a} t$:*

1. *if $s = s_1; s_2$ then there exist $c \in \mathcal{V}$ and $t_1, t_2 \in \mathcal{H}$ such that $\mathcal{R} \vdash s_1 \xrightarrow[c]{a} t_1$, $\mathcal{R} \vdash s_2 \xrightarrow[b]{c} t_2$ and $t = t_1; t_2$;*

2. if $s = s_1 \otimes s_2$ *then there exist* $a_1, a_2, b_1, b_2 \in \mathcal{V}$ *and* $t_1, t_2 \in \mathcal{H}$ *such that* $\mathcal{R} \vdash s_1 \xrightarrow[b_1]{a_1} t_1$*,* $\mathcal{R} \vdash s_2 \xrightarrow[b_2]{a_2} t_2$*, with* $a = a_1 \otimes a_2$*,* $b = b_1 \otimes b_2$ *and* $t = t_1 \otimes t_2$*.*

The format-independent *tile decomposition property* ensures that $\simeq_{\mathrm{t}}$ is a congruence (w.r.t. monoidal and sequential arrow compositions) and is thus fundamental for compositionality: it amounts to say that any evolution α of a complex system s can be expressed as the composition of the evolution of any subsystem s_1 of s with the evolution of the rest. Since in $\mathbf{Subs}_\Sigma$ arrows composition is substitution application, ordinary *congruences* are easily recovered as an instance of Proposition 2 when $\mathcal{H}$ is freely generated by Σ.

Proposition 2 (cfr. [GM00]) *If the tile system* $\mathcal{R}$ *enjoys the decomposition property, then tile bisimilarity is a congruence.*

A syntactic property on tiles guaranteeing the decomposition property is the so-called *basic source format*, where each basic tile is required to carry a single operator as initial configuration.

2.5 CommUnity

CommUnity is a parallel program design language in the style of Unity [CM88] but based on *action sharing* and synchronization. It was initially proposed in [FM97] to show how programs fit into Goguen's categorical[1] approach to General Systems Theory [Gog73] and, hence, has a categorical semantics. Since then, the language and the design framework have been extended and have evolved into an architectural description language, capitalizing on the fact that CommUnity takes to an extreme the separation between "computation" and "coordination" concerns. In CommUnity, the definition of individual components of a system is completely separated from the mechanisms that are made available for coordinating their joint behaviour.

The individual components of a system can be defined in terms of *channels* and actions organized in designs. For the sake of simplicity, in this work, we consider only a special class of CommUnity designs, called *programs*, which are particular instances, closer to parallel programming languages, of the more general following form (for even more expressive CommUnity designs see [FLW03]).

design P **is**		
in		in(V)
out		out(V)
do	$[]_{g\in\Gamma}$	g[D(g)]: G(g) $\rightarrow$ R(g)

V is the set of channels of design P. There are *input channels* and *output channels*. Input channels are read-only: the component has no control on their values. Output

[1] See [AHS90] for an introduction to category theory.

```
design  counter is
in      x: nat
out     y: int
do      inc: true → y:=y+x
[]      dec: y>MIN → y:=y-x
```

Figure 2.7: The "counter" design.

channels are controlled locally by the component, i.e. they cannot be modified by the environment. Each channel v is typed with a sort $sort(v)$ in a set S. This set takes part of a fixed data signature $\langle S, \Omega\rangle$, where S is a set (of sorts) and Ω is a $S^* \times S$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations.

Γ is the set of *action names* of design P. Actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. For each action name g:

- $D(g)$, the write frame of g, consists of the output channels that can be affected by executions of the action named by g. For simplicity, we omit the explicit reference to the write frame when $R(g)$ is a multiple assignment (see below), in which case $D(g)$ can be inferred from the assignments. Given an output channel v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$.

- $G(g)$ is the enabling condition of g.

- $R(g)$ is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local channels from the write frame of g. These primed channels account for references to the values that the channels take after the execution of the action. $R(g)$ is a specification of the effects of the action g on the state of the component. When $R(g)$ is such that the primed version of each local channel in the write frame of g is fully determined, we obtain a multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame $D(g)$ is empty, $R(g)$ is tautological, which we denote by *skip*.

As an example, consider the design in Figure 2.7.

It models a component that calculates and stores (and makes available to the environment) an integer value. It can decrease or increment this value of x units (received from the environment) through the execution of actions *inc* and *dec*, respectively. However, decreasing is only possible provided a minimum value (represented by constant MIN) has not been reached.

From a mathematical point of view, these CommUnity designs are structures defined as follows.

$$
\begin{tikzcd}
 & C \arrow[dl] \arrow[dr] & \\
P & & Q
\end{tikzcd}
$$

Figure 2.8: Shape of a configuration diagram.

Definition 23 *A design is a pair* $\langle \theta, \Delta \rangle$ *where:*

- θ*, the signature of the design, is a tuple* $\langle V, \Gamma_\perp, tv, ta, D \rangle$ *where:*
  - V *is an* S*-indexed family of mutually disjoint finite sets; if* $v \in V_s$*, then we use* $sort(v)$ *to denote* s*;*
  - $\Gamma_\perp$ *is a finite pointed set; we use* $\perp$ *to designate its point and* Γ *to designate the set with all the elements but* $\perp$*;*
  - $tv : V \to \{out, in\}$ *is a total function; we shall use* $out(V)$ *to denote* $\{v \in V : tv(v) = out\}$ *(and similarly for* $in(V)$*);*
  - $D : \Gamma \to 2^{out(V)}$ *is a total function.*
- Δ*, the body of the design, is a pair* $\langle R, G \rangle$ *where:*
  - R *assigns to every* $g \in \Gamma$*, a proposition over* $V \cup D(g)'$*;*
  - G *assigns to every* $g \in \Gamma$*, a proposition over* V*.*

When, for every $g \in \Gamma$, the relation $R(g)$ defines a multiple assignment (as in the example), the design is called a program. Notice that a program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in the context of a configuration in which these inputs have been instantiated with channels of other components.

The behaviour of a closed program is as follows. At each execution step, one of the actions whose enabling condition holds is selected, and its assignments are executed atomically.

The model of interaction between CommUnity designs is based on action synchronization and interconnection of input and output channels. Such interactions have to be made explicit and external to the components by providing the corresponding name bindings in well identified connectors. More concretely, name bindings of designs P and Q are established trough (configuration) diagrams of the form in Figure 2.8 in a category of designs with the morphisms defined as follows. In fact, as shown in [FLW03], we may consider that the design C in the middle is simply a program with no computational behaviour, i.e., a program whose actions are all associated to the guarded command $true \to skip$.

Definition 24 *A morphism of designs* $\sigma : P_1 \to P_2$ *consists of a total function* $\sigma_{var} : V_1 \to V_2$ *and a mapping* $\sigma_{ac} : \Gamma_{2_\perp} \to \Gamma_{1_\perp}$ *that maps the designated element of* $\Gamma_{2_\perp}$ *to that of* $\Gamma_{1_\perp}$ *such that:*

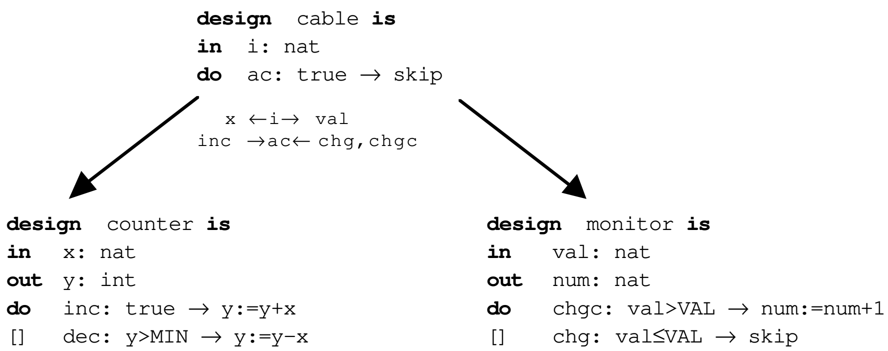


Figure 2.9: A sample diagram.

1. for every $v \in V_1, o \in out(V_1)$:

$$sort_2(\sigma_{var}(v)) = sort_1(v)$$
$$\sigma_{var}(o) \in out(V_2)$$

2. for every $g \in \Gamma_2$ such that $\sigma_{ac}(g) \neq \perp$:

$$\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$$
$$\sigma_{ac}(D_2(\sigma_{var}(v))) \subseteq D_1(v) \text{ for every } v \in out(V_1)$$
$$\Phi \models (R_2(g) \Rightarrow \sigma(R_1(\sigma_{ac}(g))))$$
$$\Phi \models (G_2(g) \Rightarrow \sigma(G_1(\sigma_{ac}(g))))$$

where we also have used σ to denote the extension of the morphism to the language of expressions and conditions. Designs and composition morphisms define a category **c-DSGN***.*

A morphism $\sigma : P_1 \rightarrow P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that it can be considered as having been obtained from P_1 through the superposition of additional behaviour, namely the interconnection of one or more components. In other words, σ identifies P_1 as a component of P_2.

Example 2 *The diagram depicted in Figure 2.9 defines a system with two components, counter and monitor.*

The program monitor can be used to count how many times an action is executed in situations in which a given input channel has a value greater than VAL. There are two actions accounting for the execution of the monitored action: chg accounts for the executions that take place when the value of val is not greater than VAL and chgc for the others.

In this configuration, it monitors the execution of action inc and the value of the input channel x of counter. This is established by the interconnection of channels x of counter and val of monitor and the synchronization of action inc

```
design  mon-counter is
in   x: nat
out  y: int, num: nat
do   incc: val>VAL → y:=y+x || num:=num+1
[]   inc: val≤VAL → y:=y+x
[]   dec: y>MIN → y:=y-x
```

Figure 2.10: The colimit of the diagram in Figure 2.9.

of counter with both chg or chgc of monitor. The fact that, in this configuration, the execution of action dec of counter is not under the control of design monitor is represented by the mapping of this action to the distinguished action of cable. In the diagram, this is implicitly represented: the action of cable to which dec is mapped is omitted.

Through a universal construction of category theory, the *colimit* construction, it is possible to internalize the interactions explicitly described in a configuration diagram and obtain a design for the system as a whole. Colimits in **c-DSGN** capture a generalized notion of parallel composition in which the designer makes explicit what interconnections are used between components. For instance, the colimit of the configuration in Figure 2.9 returns, up to an isomorphism, the design in Figure 2.10. For each diagram **dia** let $\mathsf{colim}(\mathbf{dia})$ be its colimit.

Not every diagram in **c-DSGN** reflects a meaningul configuration. In particular, it does not make sense to interconnect components by connecting their output channels. The diagrams that define correct configurations are defined as follows.

Definition 25 *A diagram* **dia** *is a* configuration diagram *if, letting* $\mathcal{C} = \mathsf{colim}(\mathbf{dia})$, *then for every* $o \in out(V_{\mathcal{C}})$ *there exists a unique design* $\mathcal{D}$ *in* **dia** *such that* $\mu_{\mathcal{D}}^{-1}(o) \cap out(V_{\mathcal{D}}) \neq \emptyset$, *where* $\mu_{\mathcal{D}}\colon \mathcal{D} \to \mathcal{C}$ *is the colimit injection relative to* $\mathcal{D}$.

To conclude this overview of CommUnity, it is important to mention that glue-cables-roles configurations (see Figure 2.11) play an important role in the process of structuring systems. They can be used to represent architectural connectors as defined in [AG97]: the design in the center is the glue and the designs in the vertices are the roles of the connector. These roles can be instantiated with specific components of the system under construction. The use of a connector in the construction of a particular system is achieved by the instantiation of its roles with specific components of the system. The glue of the connector defines how the activities of the role instances are to be coordinated.

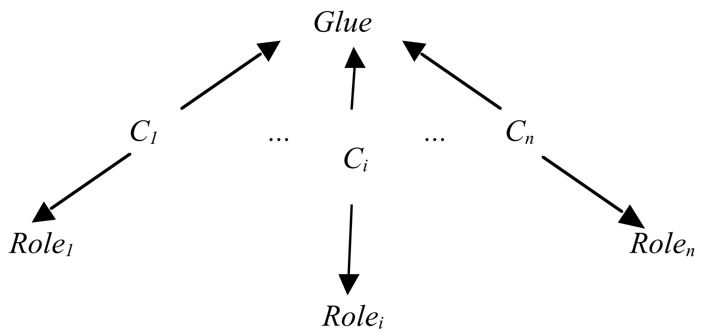


Figure 2.11: Glue-cables-roles configuration.

Chapter 3

Name fusions and binding operators from Fusion Calculus to logic programming

In this chapter we present the main results we have produced so far, that is a comparative analysis of three formalisms, namely logic programming, SHR in both the Hoare and the Milner style and logic programming. This work has multiple aims:

- we look for similarities between the different formalisms: we think that features found in such different models are good candidates to be features necessary in any GC model;
- we look for differences among the models, since different choices w.r.t. a particular aspect probably form a dimension in a "space of GC models" thus allowing some classification of the different approaches;
- we want to compare the expressiveness of different models in the GC scenario, that is in order to model synchronization and coordination in systems with (at least) mobility;
- we want to export theory and features from a model to the others, thus enriching existing models.

The choice of the three models under analysis has different motivations:

- Fusion Calculus is an important calculus for mobility, furthermore we are interested in studying the power of fusions;
- SHR allows to use graphs which are intuitive for modeling systems, and the SHR approach is well-suited for modeling GC systems because of its compositional nature and because it allows multiple synchronizations;

- logic programming is an important paradigm with efficient implementations and easily extendable: even if it is not yet completely satisfactory as a GC model we think it is a good starting point for building new models.

3.1 Mapping Fusion Calculus into Milner SHR

In this section, we present a mapping from Fusion Calculus to SHR that extends the work of [HM01] for the π-calculus and maps Fusion Calculus (without match and mismatch) into a particular kind of SHR which uses Milner synchronization and has special rules in order to force an interleaving behaviour. This is necessary since the semantics of Fusion Calculus is itself interleaving. This mapping has the advantage of being natural (there is a one-to-one mapping that maps parallel composition into parallel composition and scope into restriction) and of realizing a complete correspondence between the two models.

This material is the subject of [LM03].

In order to define the inference rules for Milner interleaving SHR we need to restrict ourselves to two kinds of transitions:

- communication transitions: they have $\pi = id$ and $\Lambda(x) = (\epsilon, \langle\rangle)$ for each x but one (we denote this kind of Λ with act);
- fusion transitions: they have $\Lambda = \Lambda_\epsilon$.

We present now the particular set of inference rules used in this case.

Definition 26 (Rules for interleaving Milner synchronization)

$$(ren)\quad \frac{\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'}{\Gamma', \sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\Lambda',\pi'} \Gamma', \rho(\sigma(\Phi)) \vdash \rho(\sigma(G'))}$$

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

- $\rho = \mathrm{mgu}(\{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
- $\Lambda'(z) = \begin{cases} \rho(\sigma(\Lambda(x))) & \textit{if } \sigma(x) = z \wedge \Lambda(x) \neq (\epsilon, \langle\rangle) \\ (\epsilon, \langle\rangle) & \textit{otherwise} \end{cases}$
- $\pi' = \rho|_{\sigma(\Gamma)}$

$$(com/close)\quad \frac{\Gamma \vdash G_1 \xrightarrow{act_1,id} \Phi \vdash G_1' \qquad \Gamma \vdash G_2 \xrightarrow{act_2,id} \Phi' \vdash G_2'}{\Gamma \vdash G_1|G_2 \xrightarrow{\Lambda_\epsilon,\pi} \rho(\Gamma) \vdash \nu U\ \rho(G_1'|G_2')}$$

where:

- $act_1 = (x, a, \vec{y}_1)$ *and* $act_2 = (x, \overline{a}, \vec{y}_2)$ *for some node* x *and pair action-coaction* $(a, \overline{a})$
- $\rho = \mathrm{mgu}(\{\vec{y}_1 = \vec{y}_2\})$
- $\pi = \rho|_\Gamma$
- $U = \rho(\Phi \cup \Phi') \setminus \rho(\Gamma)$

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

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

where:

- $(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
- $\Lambda(x) = (\epsilon, \langle\rangle)$
- $Z = \{x\}$ *if* $x \notin \mathrm{n}(\Lambda|_\Gamma), Z = \emptyset$ *otherwise*

These rules are a modification of rules for Milner synchronization. In particular rule (ren) extsends (merge) by also allowing to add isolated nodes to the interface. It is simpler than (merge) since we can never have a synchronization between two actions (each transition contains at most one non trivial action). Rule (com/close) performs synchronization, thus it is applied to two communication productions to derive a fusion production. Note that here the result of the synchronization is ϵ instead of τ. This simplifies the correspondence with Fusion Calculus. Rule (par) allows to add an idle part of the graph. Note that π must be applied also to the new part. Rule (ren) is essentially the same of the standard model.

Note that all the transitions that we obtain starting from a set of communication and fusion productions are either communication ones or fusion ones, that is they all have just one effect. This makes the semantics interleaving.

We write $\mathcal{P} \Vdash_{iM} (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ to stress the fact that the transition is derived using interleaving semantics for the Milner synchronization model.

We can state the following correspondence theorem between Milner SHR and interleaving Milner SHR.

Theorem 3 *Given a set of communication and fusion productions* $\mathcal{P}$ *and a starting graph* $\Gamma \vdash G$, $\mathcal{P} \Vdash_{iM} \Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ *iff* $\mathcal{P} \Vdash_M \Gamma \vdash G \xrightarrow{\Lambda',\pi} \Phi \vdash G'$ *where* $\Lambda(x) = (\epsilon, \langle\rangle)$ *if* $\Lambda'(x) = (\tau, \langle\rangle)$ *and* $\Lambda(x) = \Lambda'(x)$ *otherwise, and the second transition is derived using one production or two communication productions which synchronize on some node.*

We define now the translation between Fusion Calculus and interleaving Milner SHR.

Definition 27 (Translation of actions) *We define now the relation between the actions of Fusion Calculus and the synchronizations of SHR. This relation is a function that we denote with* $[\![-]\!]$ *in the case of communication actions (we have a free action when* $\vec{y} = \emptyset$*):*
$[\![(\vec{y})u\vec{x}]\!] = (u, in_n, \vec{x}), id$ *where* $n = |\vec{x}|$
$[\![(\vec{y})\overline{u}\vec{x}]\!] = (u, out_n, \vec{x}), id$ *where* $n = |\vec{x}|$
We define in_n *and* out_n *as complementary actions, i.e.* $in_n = \overline{out_n}$ *and* $out_n = \overline{in_n}$*.*

A ϕ *action of Fusion Calculus corresponds in the SHR setting to any substitution* π *which is a substitutive effect of* ϕ*.*

Note that in the translation the distinction between free names and bound names vanishes, but for any transition $P \xrightarrow{\alpha} P'$ we have $\mathrm{bn}(\alpha) = \mathrm{n}(\alpha) \setminus \mathrm{fn}(P)$, thus we can retrieve the set of bound names when necessary.

Since we want a finite number of productions that describe the evolution of each process we use edges and productions only for sequential processes in standard form. In particular, each edge has a label which encapsulates the corresponding process.

Definition 28 (Translation of agents)
We define now the translation on agents:
$[\![0]\!] = nil$
$[\![S]\!] = L_{\hat{S}}(\mathrm{fnarray}(S))$ *where* $\hat{S}$ *is the process that corresponds to the standard agent for* S
$[\![P_1|P_2]\!] = [\![P_1]\!]|[\![P_2]\!]$
$[\![(x)P]\!] = \nu x\ [\![P]\!]$
$[\![\mathrm{rec}\, X.P]\!] = [\![P\{\mathrm{rec}\, X.P/X\}]\!]$

We have no need to translate agent variables since they occur only inside sequential processes.

The translation is well-defined also on equivalence classes, i.e. on processes, since if $P_1 \equiv P_2$ then $[\![P_1]\!] \equiv [\![P_2]\!]$. Note that if a congruence rule is applied inside a sequential agent then the translation of the two agents is the same since the edge is labelled by the process itself. Furthermore abelian monoid laws for $|$ correspond to rules (AG1), (AG2) and (AG3), $(x)0 = 0$ corresponds to an instance of (AG5), $(x)(y)P = (y)(x)P$ corresponds to (AG4) and the scope extension law corresponds to (AG7). Finally α-conversion for names is (AG6). The recursion law holds since the translation of a recursively defined agent coincides with the translation of its expansion using recursion law. Note that even if this part of the definition is not by structural induction, still the termination is granted because we use only guarded recursion. Thus, after one application of the rule, we get a sequential agent, whose translation is directly defined. Finally laws for $+$ and α-conversion for agent variables can be applied only inside sequential agents. Note also that rule (AG5) corresponds to a rule that can be obtained from $P|0 = P$, the scope extension law and $(x)0 = 0$. Thus both formalisms have the same underlying structure, that is a parallel composition operator and a binder for names, ruled by equivalent congruences.

As a last step we need to show the productions used in the SHR system.

Definition 29 *We define productions only for standard sequential agents $\sum_i \alpha_i.P_i$. Let Γ be $\mathrm{fn}(\llbracket\sum_i \alpha_i.P_i\rrbracket)$. The productions are of the following forms:*

$$\Gamma \vdash \llbracket\sum_i \alpha_i.P_i\rrbracket \xrightarrow{\llbracket\alpha_i\rrbracket} \Gamma \vdash \llbracket P_i\rrbracket$$

if α_i is a communication action (note that this is a communication production) and

$$\Gamma \vdash \llbracket\sum_i \alpha_i.P_i\rrbracket \xrightarrow{\pi} \pi(\Gamma) \vdash \llbracket\pi(P_i)\rrbracket$$

if $\alpha_i = \phi$ and π is a fusion effect of ϕ (note that this is a fusion production).

Note that since, in SHR computations, nodes are never deleted, the graphs that we obtain during the computation have some more unused nodes.

One of the differences between Fusion Calculus and SHR is that in one case fusions are represented by equivalence relations, whereas in the other one substitutions are used. The correspondence between the different methods used for computing fusions in the two systems is given by the following lemma.

Lemma 1 *Let S be a set of equalities representing a fusion ϕ and σ be a substitution. Then the following two propositions are equivalent:*

- *σ is a substitutive effect of ϕ;*
- *σ is an mgu of S.*

An important characteristic of graphs obtained from Fusion Calculus is that we have no generation of new names as shown by the following lemma (but we can have extrusions of bound names), we can however have infinitely many names since the axiom for recursion allows a finite representation of an infinite agent (which may contain infinitely many names).

Lemma 2 *For each process P if $P \xrightarrow{\alpha} P'$ for some action α then all the free names that appear in α are also free names of $\llbracket P\rrbracket$.*

We can now state the two theorems that show the correctness and the completeness of the translation.

Theorem 4 (Correctness) *For each fusion process P and each $\Gamma \supseteq \mathrm{fn}(\llbracket P\rrbracket)$ if $P \xrightarrow{\alpha} P'$ and $\Gamma \cap \mathrm{bn}(\alpha) = \emptyset$ then:*

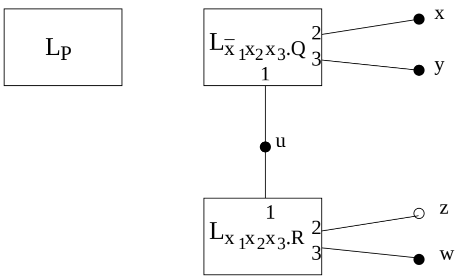


Figure 3.1: Graphical representation of a fusion process.

1. $\Gamma \vdash [\![P]\!] \xrightarrow{[\![\alpha]\!]} \Gamma, \Gamma_E \vdash [\![P']\!]$ *where* $\Gamma_E = \mathrm{bn}(\alpha)$ *if* α *is a communication action;*
2. $\Gamma \vdash [\![P]\!] \xrightarrow{\pi} \pi(\Gamma) \vdash [\![\pi(P')]\!]$ *for each substitutive effect* π *of* α *if* α *is a fusion action.*

Theorem 5 (Completeness) *For each fusion process* P *if* $\Gamma \vdash [\![P]\!] \xrightarrow{\Lambda,\pi} \Gamma \vdash G$ *for some* $\Gamma \supseteq \mathrm{fn}([\![P]\!])$ *then there exists* P' *such that:*

1. *either* $P \xrightarrow{\alpha} P'$, $[\![\alpha]\!] = (\Lambda, id)$, $\pi = id$, $[\![P']\!] = \Gamma' \vdash G$ *and* $\Gamma' = \Gamma, \Gamma_E$ *where* $\Gamma_E = \mathrm{bn}(\alpha)$*;*
2. *or* $P \xrightarrow{\phi} P'$, π *is a substitutive effect of* ϕ, $\Lambda = \Lambda_\epsilon$, $[\![\pi(P')]\!] = G$ *and* $\Gamma' = \pi(\Gamma)$.

We can also handle the match operator by distributing it on sequential agents and then adding to SHR a special rule for dealing with edges that correspond to sequential processes with match.

We show an example on how the translation can be used.

Example 3 *Let us consider the fusion transition:*
$(z)(P|\overline{u}xy.Q|uzw.R) \xrightarrow{\{y=w\}} (P|Q|R)\{x/z\}$.
The translation of the starting process into a graph is:
$\nu z\ L_P|L_{\overline{x_1}x_2x_3.Q}(u,x,y)|L_{x_1x_2x_3.R}(u,z,w)$ *and we can choose* $\Gamma = u, x, y, w$ *which is graphically represented in Figure 3.1. Furthermore we have the following productions:*

$$x_1, x_2, x_3 \vdash L_{\overline{x_1}x_2x_3.Q}(x_1,x_2,x_3) \xrightarrow{(x_1, out_2, \langle x_2, x_3\rangle)} x_1, x_2, x_3 \vdash L_Q$$
$$x_1, x_2, x_3 \vdash L_{x_1x_2x_3.R}(x_1,x_2,x_3) \xrightarrow{(x_1, in_2, \langle x_2, x_3\rangle)} x_1, x_2, x_3 \vdash L_R$$

By applying rule (ren) we can derive:

$$u, x, y, z, w \vdash L_{\overline{x_1}x_2x_3.Q}(u,x,y) \xrightarrow{(u, out_2, \langle x, y\rangle)} u, x, y, z, w \vdash L_Q$$
$$u, x, y, z, w \vdash L_{x_1x_2x_3.R}(u,z,w) \xrightarrow{(u, in_2, \langle z, w\rangle)} u, x, y, z, w \vdash L_R$$

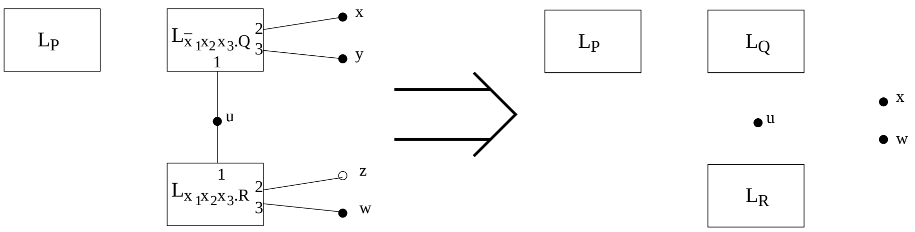


Figure 3.2: Graphical representation of a fusion transition.

We can then apply rules (com/close):

$$u, x, y, z, w \vdash L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w) \xrightarrow{\{x/z,y/w\}} u, x, y \vdash (L_Q|L_R)\{x/z, y/w\}$$

(par):

$$u, x, y, z, w \vdash L_P|L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w) \xrightarrow{\{x/z,y/w\}} u, x, y \vdash (L_P|L_Q|L_R)\{x/z, y/w\}$$

and (res) to have:

$$u, x, y, w \vdash \nu z\ L_P|L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w) \xrightarrow{\{y/w\}} u, x, y \vdash (L_P|L_Q|L_R)\{x/z, y/w\}$$

as desired since $\{y/w\}$ is a substitutive effect of $\{y = w\}$. The transition is graphically represented in Figure 3.2

Here we show how SHR can be used to execute many Fusion Calculus transitions in one step.

Example 4 *Let us consider the fusion process $(xy)(ux.P|yx.Q|\overline{y}z.R)$.*
We can have the two following computations:

$$\begin{array}{lcl} (xy)(ux.P|yx.Q|\overline{y}z.R) & \xrightarrow{(x)ux} & (y)(P|yx.Q|\overline{y}z.R) \\ & \xrightarrow{\{x=z\}} & (y)(P|Q|R) \\ (xy)(ux.P|yx.Q|\overline{y}z.R) & \xrightarrow{1} & (y)((ux.P|Q|R)\{z/x\}) \\ & \xrightarrow{uz} & (y)((P|Q|R)\{z/x\}) \end{array}$$

Using the translation and the inference rules of Definition 26 we have the two corresponding sequences of transitions:

$$\begin{array}{ll} \multicolumn{2}{l}{u, z \vdash \nu x, y\ L_{x_1x_2.P}(u, x)|L_{x_1x_2.Q}(y, x)|L_{\overline{x_1}x_2.R}(y, z)} \\ \xrightarrow{(u,in_1,x)} & u, z, x \vdash \nu y\ L_P|L_{x_1x_2.Q}(y, x)|L_{\overline{x_1}x_2.R}(y, z) \\ \xrightarrow{\{z/x\}} & u, z \vdash \nu y\ (L_P|L_Q|L_R)\{z/x\} \end{array}$$

$$
\begin{array}{rl}
u, z \vdash \nu x, y\ (L_{x_1 x_2.P}(u, x) | L_{x_1 x_2.Q}(y, x) | L_{\overline{x_1} x_2.R}(y, z)) & \\
\rightarrow & u, z \vdash \nu y\ (L_{x_1 x_2.P}(u, x) | L_Q | L_R)\{z/x\} \\
\xrightarrow{(u, in_1, z)} & u, z \vdash \nu y\ (L_P | L_Q | L_R)\{z/x\}
\end{array}
$$

Note that in this case the two results are equal, even if the observations are different because of the different ordering on the transitions. Furthermore if we choose the set of inference rules of Definition 6 we can also have the transition:

$$
\begin{array}{rl}
u, z \vdash \nu x, y\ (L_{x_1 x_2.P}(u, x) | L_{x_1 x_2.Q}(y, x) | L_{\overline{x_1} x_2.R}(y, z)) & \\
\xrightarrow{(u, in_1, z)} & u, z \vdash \nu y\ (L_P | L_Q | L_R)\{z/x\}
\end{array}
$$

which corresponds to the concurrent execution of the two transitions.

We end this section with a simple schema on the correspondence between the two models.

Fusion	SHR	Fusion	SHR
Process	Graph	Transition	Transition
Sequential process	Edge	Parallel comp.	Parallel comp.
Name	Node	Scope	Restriction
Prefix execution	Production	0	Nil

3.2 Comparing Hoare and Milner synchronization in the SHR setting

In this section we want to compare the expressive power of Hoare and Milner synchronization in a framework with mobility and fusions. We choose to this end SHR, with the aim of translating Milner SHR and thus Fusion Calculus, into Hoare SHR which, as we will see later, is strictly related to logic programming.

Some of the ideas used here are presented in [LM04].

We firstly present the dual approach, namely the translation from Hoare SHR into Milner SHR, since it is a bit simpler and shows most of the issues.

We want to take a graph whose evolution is described using productions to be used in the Hoare synchronization model and derive a translation of that graph that has “the same” possible evolutions, using a translation of the productions and the Milner synchronization. The idea is that edges are translated into themselves, while nodes are implemented using complex graphs called *amoeboids*. Let us consider a node with, for instance, five attached edges. We want to translate it into a structure, that we call (Milner[1]) amoeboid, with five external nodes. Thus, when translating, we must take each node, count the number of edges connected to that node, create a new node for each of these edges, and finally connect these

[1]The name denotes the framework where the amoeboid is supposed to be used.

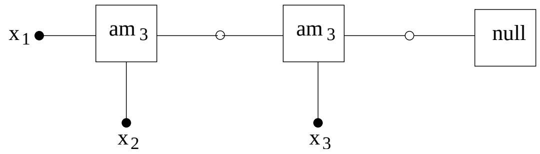


Figure 3.3: A complex amoeboid.

new nodes using an amoeboid. Amoeboids essentially perform the broadcast of the common action. The main problem is that amoeboids must be compositional, i.e. one must be able to merge a set of amoeboids into one structure, which must still be an amoeboid. This is done by connecting amoeboids using other amoeboids, that is in order to merge nodes x and y we create a new amoeboid that connects the amoeboids for x and y.

We assume the existence of edge labels am_i and $null$ of ranks $i = 2, 3, \dots$ and 1 respectively. Edges labelled by am_i implement broadcast among i nodes, while $null$ edges "close" nodes which are not used any more.

Definition 30 (Amoeboid) *Given a set of nodes S an amoeboid for S is a connected graph composed by am and null edges that satisfies the following properties:*

- *the nodes are either internal or external;*
- *S is the set of external nodes;*
- *external nodes are connected to exactly one edge of the amoeboid;*
- *internal nodes are connected to exactly two edges of the amoeboid.*

We consider *equivalent* all the amoeboids with the same set S of external nodes.

Note that an am_i connector is a correct amoeboid for a set S of i nodes. A more complex amoeboid for the set $\{x_1, x_2, x_3\}$ is shown in Figure 3.3.

We show here the productions for Milner amoeboids. Since Milner synchronization is asymmetric (we have actions and coactions) while Hoare synchronization is not we define $\overline{a} = a$ for each a. In this way we drop the differences between actions and coactions and we make also Milner synchronization symmetric.

Definition 31 (Milner auxiliary productions) *The set Mil of auxiliary productions for simulating Hoare synchronization in the Milner model contains a production of the form:*

$$x_1, \dots, x_n \vdash am_n(x_1, \dots, x_n) \xrightarrow{(x_1,a,\vec{y}_1)\dots(x_n,a,\vec{y}_n)} \\ x_1, \dots, x_n, \Gamma_y \vdash am_n(x_1, \dots, x_n)|(|_{i=1\dots|\vec{y}_1|} am_n(y_{1,i}, \dots, y_{n,i}))$$

for each n and each action a such that $\mathrm{ar}(a) = |\vec{y}_1|$. Here $\vec{y}_i, i = 1, \dots, n$ are vectors of fresh names of the same length. Finally, Γ_y is the concatenation of the vectors $\vec{y}_i, i = 1, \dots, n$.

We have no productions for connector null.

Next lemma shows that any amoeboid has the same behaviours as an *am* connector with the same set of external nodes.

Lemma 3 *If $AM(x_1, \ldots, x_n)$ is an amoeboid for the set of nodes $\{x_1, \ldots, x_n\}$ then the only possible transitions for $AM(x_1, \ldots, x_n)$ are of the form:*

$$x_1, \ldots, x_n \vdash AM_n(x_1, \ldots, x_n) \xrightarrow{(x_1,a,\vec{y}_1)\ldots(x_n,a,\vec{y}_n)} \\ x_1, \ldots, x_n, \Gamma_y \vdash AM_n(x_1, \ldots, x_n)|(|_{i=1\ldots|\vec{y}_1|}AM_n(y_{1,i}, \ldots, y_{n,i}))$$

for some action a such that $\mathrm{ar}(a) = |\vec{y}_1|$. Here $\vec{y}_i, i = 1, \ldots, n$ are vectors of fresh names of the same length. Finally, Γ_y is the concatenation of the vectors $\vec{y}_i, i = 1, \ldots, n$.

The definition of amoeboids is essentially motivated by the requirement of having graphs where each node is attached to exactly two edges (we call them 2-shared graphs), since in this case Hoare and Milner synchronizations are quite similar. In fact, if we consider the symmetric version of Milner synchronization, both in the Hoare and in the Milner model, two edges must interact by exposing the same action. Note that this similarity disappears if we consider nodes with more than two attached edges, since in the Hoare model all the edges must expose the same action, while in the Milner one just two of them interact. Similarly, when we have just one edge, in the Hoare model it can perform any synchronization, while in the Milner model it can just perform ϵ (if the node is restricted).

Since we can not identify for each node in the starting graph a unique node in the translation that represents it (we may choose one, but fusions can not agree on one representative) and since observations are located at nodes, observations are not meaningful in this setting. Thus we will work only on closed graphs (we however need to deal with open graphs for productions).

We present here a decomposition for graphs which is essentially analogous to the standard decomposition for Fusion Calculus processes of Definition 13, but where the set of standard names is explicitly given as a parameter.

Definition 32 *The standard decomposition of a graph term G is defined w.r.t. a vector of names $\vec{w}$, which are the names to be used in the standard graph term. We denote with $\mathrm{fnarray}(G)$ the array of the free name occurrences in G, ordered according to some fixed order dictated by the structure of G. The decomposition is defined iff $|\vec{w}| = |\,\mathrm{fnarray}(G)|$ and has the form:*

$$G = \sigma_{G,\vec{w}}\hat{G}_{\vec{w}}$$

where $\sigma_{G,\vec{w}}$ is the standard substitution and $\hat{G}_{\vec{w}}$ is the standard graph term of G. This decomposition satisfies:

$$G = \sigma G' \textit{ implies } \hat{G}_{\vec{w}} = \hat{G}'_{\vec{w}} \wedge \sigma_{G,\vec{w}} = \sigma\sigma_{G',\vec{w}}$$

We require $\mathrm{fnarray}(\hat{G}_{\vec{w}}) = \vec{w}$. Note that $\sigma_{G,\vec{w}} = \{\mathrm{fnarray}(G)/\vec{w}\}$.

Definition 33 (Translation for graphs) *Let $\Gamma \vdash G$ be a graph. Using the axioms we can write $G = \nu X\ G'$ where G' is a parallel composition of edges. Consider the standard decomposition $\sigma_{\vec{w}} G''$ of G' with a vector $\vec{w}$ of fresh names such that $|\vec{w}| = |\operatorname{fnarray}(G')|$.*

We define the translation w.r.t. a function uses *that associates to each node a set of nodes containing its uses, that is nodes that must be in the same amoeboid.*
$[\![\Gamma \vdash G]\!] = \Gamma \vdash \nu \vec{w}\ G''|(|_{x \in X \cup \Gamma} AM(\text{uses}(x)))$
We will need more complex definitions for $\text{uses}(x)$ *when translating productions, but for the moment we just have:*
$\text{uses}(x) = \sigma_{\vec{w}}^{-1}(x)$ *if* $x \in X$
$\text{uses}(x) = \sigma_{\vec{w}}^{-1}(x) \cup \{x\}$ *if* $x \in \Gamma$

Since we want that the graph obtained by applying concurrently a set of compatible productions is a 2-shared graph we must control how many times we use each node. In particular, the number of edges attached to a node can be influenced by a production iff the node is attached to the rewritten edge.

A node x is used:

- once for each time an edge in the RHS is attached to x (since the edge need to be connected to x);
- once for each time the node is transmitted in a synchronization (in fact x is then attached to the amoeboid that receives the node reference);
- once for each node $y \in \Gamma$ such that $\pi(y) = x$ (in fact the other edge attached to y in the starting graph becomes then attached to the amoeboid for x).

Now we define the translation for productions.

Definition 34 (Milner translation for productions) *Given a set of productions $\mathcal{P}$, its Milner translation $Mil(\mathcal{P})$ contains a translation for each production $x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$ in $\mathcal{P}$.*

$$[\![x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G]\!] = \\ x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda',id} [\![\Phi \vdash G]\!]_{\text{uses}}$$

Here Λ' is obtained by changing each occurrence of a name in $\operatorname{Im}(\Lambda)$ *into a fresh name. Let σ_Λ be the substitution such that $\sigma_\Lambda(\Lambda'(x)) = \Lambda(x)$ for each $x \in \{x_1, \ldots, x_n\}$. Then we define:*
$\text{uses}(x) = \sigma_{\vec{w}}^{-1}(x) \cup \sigma_{\Lambda}^{-1}(x) \cup \pi^{-1}(x)$
where $\sigma_{\vec{w}}$ is the one used in the translation of $\Phi \vdash G$ (see Definition 33). Note that the three components of the definition of uses *correspond exactly to the three kinds of possible uses for nodes.*

We can now present the results on the correctness and completeness of our translation.

Theorem 6 (Correctness) *Let $\mathcal{P}$ be a set of productions. For each closed graph G, if $\mathcal{P} \models_H \vdash G \rightarrow \vdash G'$ then $Mil(\mathcal{P}) \cup Mil \models_M [\![\vdash G]\!] \rightarrow [\![\vdash G']\!]$ up-to equivalence of amoeboids.*

Theorem 7 (Completeness) *Let $\mathcal{P}$ be a set of productions. For each closed graph G if $Mil(\mathcal{P}) \cup Mil \models_M [\![\vdash G]\!] \rightarrow \vdash G''$ then $\mathcal{P} \models_H \vdash G \rightarrow \vdash G'$ and $\vdash G''$ is equal to $[\![\vdash G']\!]$, up-to equivalence of amoeboids.*

Suitable productions can be added to simplify the structure of the amoeboids. Anyway it is impossible to transform an amoeboid in its simplest form, since this requires to delete cycles, what is not possible in the general case.

Example 5 *Let us consider the final transition of Example 1, where we close also node x. We do not write explictly restrictions in the graph since all nodes are restricted. We have:*

$$\begin{aligned}&[\![C(x,y)|C(y,z)|C(z,v)|C(v,x)]\!] = \\ &\qquad = C(x_1,y_1)|C(y_2,z_1)|C(z_2,v_1)|C(v_2,x_2)| \\ &\qquad\qquad am_2(x_1,x_2)|am_2(y_1,y_2)|am_2(z_1,z_2)|am_2(v_1,v_2)\end{aligned}$$

where we used $\vec{w} = \langle x_1, y_1, y_2, z_1, z_2, v_1, v_2, x_2\rangle$. The production to be used is:

$$\begin{aligned}&x,y \vdash C(x,y) \xrightarrow{(x,r,\langle w_1\rangle)(y,r,\langle w_2\rangle)} \\ &\qquad x,y,w_1,w_2 \vdash \nu y_1, w_3\ S(y_1,w_3)|null(x)|am_2(y,y_1)|am_3(w_1,w_2,w_3)\end{aligned}$$

Note that since y_1 is the only use of y in the RHS we can simplify the production by collapsing the two variables y and y_1 and use:

$$\begin{aligned}&x,y \vdash C(x,y) \xrightarrow{(x,r,\langle w_1\rangle)(y,r,\langle w_2\rangle)} \\ &\qquad x,y,w_1,w_2 \vdash \nu w_3\ S(y,w_3)|null(x)|am_3(w_1,w_2,w_3)\end{aligned}$$

Thus we can have the computation:

$$\begin{aligned}&C(x_1,y_1)|C(y_2,z_1)|C(z_2,v_1)|C(v_2,x_2)| \\ &\qquad am_2(x_1,x_2)|am_2(y_1,y_2)|am_2(z_1,z_2)|am_2(v_1,v_2) \rightarrow \\ &S(y_1,w_{3,1})|null(x_1)|am_3(w_{1,1},w_{2,1},w_{3,1})|S(z_1,w_{3,2})|null(y_2)|am_3(w_{1,2},w_{2,2},w_{3,2})| \\ &S(v_1,w_{3,3})|null(z_2)|am_3(w_{1,3},w_{2,3},w_{3,3})|S(x_2,w_{3,4})|null(v_2)|am_3(w_{1,4},w_{2,4},w_{3,4})| \\ &\qquad am_2(x_1,x_2)|am_2(w_{1,1},w_{2,4})|am_2(y_1,y_2)|am_2(w_{1_,2},w_{2,1})| \\ &\qquad\qquad am_2(z_1,z_2)|am_2(w_{1,3},w_{2,2})|am_2(v_1,v2)|am_2(w_{1,4},w_{2,3})\end{aligned}$$

Note that edges

$$\begin{aligned}&am_3(w_{1,1},w_{2,1},w_{3,1})|am_3(w_{1,2},w_{2,2},w_{3,2})| \\ &\qquad am_3(w_{1,3},w_{2,3},w_{3,3})|am_3(w_{1,4},w_{2,4},w_{3,4})| \\ &\qquad am_2(w_{1,1},w_{2,4})|am_2(w_{1_,2},w_{2,1})|am_2(w_{1,3},w_{2,2})|am_2(w_{1,4},w_{2,3})\end{aligned}$$

compose a unique amoeboid that corresponds to node w and thus gives the star structure to the net (note also that since all nodes are restricted their actual names are not important).

We consider now the dual problem, that is mapping Milner synchronization in the Hoare setting. Now amoeboids (whose structure is not changed) are essentially routers that connect an action with the corresponding coaction.

We show here productions for Hoare amoeboids.

Definition 35 (Hoare auxiliary productions) *The set Hoa of auxiliary productions for simulating Milner synchronization in the Hoare model contains a production of the form:*

$$x_1, \ldots, x_n \vdash am_n(x_1, \ldots, x_n) \xrightarrow{(x_i, a, \vec{y}_1)(x_j, \overline{a}, \vec{y}_2)} \\ x_1, \ldots, x_n, \Gamma_y \vdash am_n(x_1, \ldots, x_n) | (|_{i=1 \ldots |\vec{y}_1|} am_2(y_{1,i}, y_{2,i}))$$

for each n, each pair of nodes x_i and x_j in $x_1, \ldots, x_n$ and each pair action-coaction $(a, \overline{a})$. Here $\vec{y}_1$ and $\vec{y}_2$ are two vectors of fresh names such that $|\vec{y}_1| = |\vec{y}_2|$. Finally, Γ_y is the concatenation of the two vectors $\vec{y}_1$ and $\vec{y}_2$.

Again, we have no productions for connector null.

We show now a lemma that states that also Hoare amoeboids behave as simple am_n edges, but now the lemma holds only if no more than three external synchronizations are available (otherwise complex amoeboids can accept multiple action-coaction pairs at the same time). Note that a stricter condition has to be applied in any case to the translation of Fusion Calculus processes, in order to preserve their interleaving behaviour.

Lemma 4 *If $AM(x_1, \ldots, x_n)$ is an amoeboid for $x_1, \ldots, x_n$ then the only possible transitions for $AM(x_1, \ldots, x_n)$ which are non idle and involve no more than three external synchronizations are of the form:*

$$x_1, \ldots, x_n \vdash AM(x_1, \ldots, x_n) \xrightarrow{(x_i, a, \vec{y}_1)(x_j, \overline{a}, \vec{y}_2)} \\ x_1, \ldots, x_n, \Gamma_y \vdash AM(x_1, \ldots, x_n) | (|_{i=1 \ldots |\vec{y}_1|} AM'(y_{1,i}, y_{2,i}))$$

for some pair of nodes x_i and x_j in $x_1, \ldots, x_n$ and some pair action-coaction $(a, \overline{a})$. Here $\vec{y}_1$ and $\vec{y}_2$ are two vectors of fresh names such that $|\vec{y}_1| = |\vec{y}_2|$ and AM' is a binary amoeboid. Finally, Γ_y is the concatenation of the two vectors $\vec{y}_1$ and $\vec{y}_2$.

The translation for productions is the same of Definition 34, what means that the translation is needed only to have 2-shared graphs, which are required in both the cases. When the translation is used from Milner synchronization to Hoare synchronization we denote it with $Hoa(-)$.

We can now present the results on the correctness and completeness of our translation.

Theorem 8 (Correctness) *Let $\mathcal{P}$ be a set of productions. For each closed graph G, if $\mathcal{P} \vDash_M \vdash G \to \vdash G'$ then $Hoa(\mathcal{P}) \cup Hoa \vDash_H [\![\vdash G]\!] \to [\![\vdash G']\!]$ up-to equivalence of amoeboids.*

Theorem 9 (Completeness) *Let $\mathcal{P}$ be a set of productions. For each closed graph G if $Hoa(\mathcal{P}) \cup Hoa \vDash_H [\![\vdash G]\!] \to \vdash G''$ with a transition where at most three external synchronizations for each amoeboid are used then $\mathcal{P} \vDash_M \vdash G \to \vdash G'$ and $\vdash G''$ is equal to $[\![\vdash G']\!]$ up-to equivalence of amoeboids.*

3.3 Extending logic programming with a restriction operator

A first presentation of this topic is discussed in [Lan02].

An important aspect in GC systems is the distinction between what is observable from outside (part of) the system (global effects) and what is only visible within the system or within a part of it (local effects). This kind of distinction allows to build more compositional models since local resources can not be modified or used outside their scope, i.e. their space of visibility.

In the logic programming world, this distinction is less important because usually logic programming is not used to model systems that interact with an environment. However in some Prologs it is possible to specify that some variables are anonymous and substitutions on these variables are not interesting, what means that these variables are local. We want to deal with this kind of problems in a precise way.

In order to do that we extend logic programming with the ν operator. This operator is of common use in the field of process models (CCS, π-calculus, SHR, ...) and it restricts a variable to be used only within its scope. We define for our language the grammar:

$$C ::= A \leftarrow G$$

$$G ::= \nu x\ G \mid G, G \mid A \mid \Box$$

where x is a variable, A is a logic atom and $\Box$ is the empty goal. We assume that the restriction operator has the lowest precedence. The restriction operator is a binder, thus all the occurrences of x in $\nu x\ G$ are bound occurrences. We require that each free variable in the body of the clause is a variable of the head of the clause. Since we want variables to represent channels, and assigning a term containing a variable x to a variable y has the meaning of sending name x on y, we want x to be hidden if it has not been sent on any free channel. Thus new variables not in the head, which are not assigned to any variable when mgu is applied, must be hidden. For uniformity, we call names the variables and we define also for goals and clauses the functions n and fn that computes the sets of names and of free names respectively.

We consider clauses and goals with restriction up to the following axioms:

$$(AL1)\ (G_1, G_2), G_3 \equiv G_1, (G_2, G_3)$$

$$(AL2)\ G_1, G_2 \equiv G_2, G_1$$

$$(AL3)\ G, \Box \equiv G$$

$$(AL4)\ \nu x\ \nu y\ G \equiv \nu y\ \nu x\ G$$

$$(AL5)\ \nu x\ G \equiv G \qquad \text{if } x \notin \mathrm{fn}(G)$$

$$(AL6)\ \nu x\ G \equiv \nu y\ G[y/x]$$

$$(AL7)\ \nu x\ G_1, G_2 \equiv (\nu x\ G_1), G_2 \qquad \text{if } x \notin \mathrm{fn}(G_2)$$

The first three axioms state properties (associativity, commutativity and existence of the neutral element $\Box$) that hold also for standard logic programming. The other axioms are the usual rules of restriction, which state that the order of restrictions is not important (axiom (AL4)), that restricting names that are not free (or even not present) has no influence (axioms (AL5) and (AL7)) and that restricted names can be α-converted (axiom (AL6)). We have chosen to state these properties as axioms (instead of let them as provable theorems) so that they induce a congruence which is essentially identical to the structural congruence on syntactic judgements for graphs of Definition 2.

Because of the axiom (AL4) we can write νX where $X = \{x_1, \ldots, x_n\}$ instead of $\nu x_1 \ldots \nu x_n$ without any ambiguity. We can always transform a goal in the following normal form:

$$\nu X\ G_1, \ldots, G_n$$

where X is a set of names.

An useful concept when dealing with restricted computations is the concept of origin of a variable with respect to a substitution and a goal.

Definition 36 (Set of ancestors) *Given a substitution θ and a variable x we define the set of the ancestors of x with respect to θ as $\theta^{-1}(x) = \{y | x \in \mathrm{n}(y\theta)\}$.*

Definition 37 (Origin) *Let G be a goal and θ a substitution.*
For each variable x we define the set of origins of x with respect to θ and G as: $\mathrm{orig}_{\theta,G}(x) = \theta^{-1}(x) \cap \mathrm{fn}(G)$

We can find also in this framework the concept of extrusion.

Definition 38 (Extrusion) *We have an extrusion of a name x if x is a bound name and x or a term containing it is assigned to a free name.*

When we have an extrusion the extruded name becomes visible from the outside.

We can now extend the semantics of logic programming to goals and programs with restriction. Fundamentally, we have to add the semantics of the restriction operator. This requires two updates. First, we have to hide substitutions on restricted names. We also hide substitutions on names not used in any goal but only in the clauses. Then we need to define which are the restricted names in the new goal we compute. Because free names are names whose existence is known from the outside, we state that a name must be hidden if it has never been assigned to a free name.

Definition 39 (Partial SLD-derivation with restriction)
Let P be a logic program with restriction. We define a step of a computation with restriction using the following rules:

$$\frac{H \leftarrow \nu X\ B_1, \ldots, B_k \in P \qquad \theta = \mathrm{mgu}(A, \rho(H))}{P \Vdash A \xrightarrow{\theta|_{\mathrm{n}(A)}}_r (\nu X\ B_1, \ldots, B_k)\rho\theta} \qquad \textit{atomic goal}$$

where ρ is an injective renaming such that $\mathrm{n}(\rho(H))$ contains only fresh names and A is an atom.

$$\frac{P \Vdash G \xrightarrow{\theta}_r F}{P \Vdash G, G' \xrightarrow{\theta}_r F, G'\theta} \qquad \textit{conjunctive goal}$$

$$\frac{P \Vdash G \xrightarrow{\theta}_r F}{P \Vdash \nu x\ G \xrightarrow{\theta'}_r \nu X'\ F} \qquad \textit{restricted goal}$$

where:
$\theta' = \theta|_{\mathrm{fn}(G)\setminus x}$
$X' = \{y \in \mathrm{fn}(F) | \operatorname{orig}_{\theta,G}(y) \subseteq \{x\}\}$

We also require that, when an extrusion happens, the representative chosen for the extruded name is a fresh name of the computation (otherwise substitutions on this name will affect substitutions on other occurrences of the same name).

The observed substitution is called the restricted mgu. We omit $P \Vdash$ if P is clear from the context.

A partial SLD-derivation with restriction using P on goal G is a sequence (eventually empty) of steps using clauses in P with G as first goal. If G' is the final goal then we write $P \Vdash G \xrightarrow{\theta}{}^{*}_r G'$ where θ is the composition of all the restricted mgus.

Note that, even for goals without restriction, we have as observation not the mgu but the mgu restricted to names in the goal (this is analogous to what done in usual SLD-refutation, where the computed answer substitution is restricted to names in the starting goal).

Example 6
Let P be:

$p(X,Y) \leftarrow \nu Z\ q(X,Y,Z), r(Z).$
$q(X,X,Z) \leftarrow p(X,Z).$
$r(f(X,Y)) \leftarrow p(X,Y).$

Starting from the goal:

$\nu X\ p(X,Y).$

we can perform the following computation:

$$\nu X\ p(X,Y) \xrightarrow{Y=Y'}_r$$
$$\nu X', Z'\ q(X',Y',Z'), r(Z') \xrightarrow{Y'=X''}_r$$
$$\nu Z''\ p(X'',Z''), r(Z'') \rightarrow_r$$
$$\nu X''', Y'''\ p(X'', f(X''',Y''')), p(X''',Y''')$$

We present now some lemmas that show properties of restricted computations. We start with the extension of some properties that hold also for standard logic programming.

Lemma 5 *Let $G \xrightarrow{\theta}_r G'$ be a restricted computation. If $x \in \mathrm{fn}(G')$ then $x \notin \mathrm{dom}(\theta)$.*

Lemma 6 *Let:*
$G_0 \xrightarrow{\theta_1}_r \cdots \xrightarrow{\theta_n}_r G_n$ be a computation with restriction.
For each i, j such that $1 \leq i < j \leq n$ we have $\mathrm{dom}(\theta'_i) \cap \mathrm{n}(\theta'_j) = \emptyset$.

We will now state a lemma that deals with multiple restrictions.

Lemma 7 *The following rule is equivalent to rule (restricted goal) of Definition 39:*

$$\frac{P \Vdash G \xrightarrow{\theta}_r F}{P \Vdash \nu X\ G \xrightarrow{\theta'}_r \nu X'\ F} \qquad \textit{multi restricted goal}$$

where:
$\theta' = \theta|_{\mathrm{fn}(G)\setminus X}$
$X' = \{y \in \mathrm{fn}(F) | \, \mathrm{orig}_{\theta,G}(y) \subseteq X\}$

We present now a more direct characterization of steps in computations with restriction.

Lemma 8 *Let $\nu X\ A_1, \ldots, A_n$ be a goal. All the steps that can be obtained from this goal rewriting atom A_1 using a clause variant $H \rightarrow \nu Y\ B_1, \ldots, B_k$ are of the form:*

$$\nu X\ A_1, \ldots, A_n \xrightarrow{\theta|_{fn(\nu X\ A_1,\ldots,A_n)}}_r \nu X'\ ((\nu Y\ B_1, \ldots, B_k), A_2, \ldots, A_n)\theta$$
where θ is an mgu of $\{A_1 = H\}$ and
$X' = \{x \in \mathrm{fn}(((\nu Y\ B_1, \ldots, B_k), A_2, \ldots, A_n)\theta) | \operatorname{orig}_{\theta,(A_1,\ldots,A_n)}(x) \subseteq X\}$.

We present here some lemmas useful to deal with multi-steps computations.

Lemma 9 *Let θ_1 and θ_2 be substitutions.*
$(\theta_1\theta_2)^{-1}(x) = \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i)$.

Lemma 10 *Let:*
$G \xrightarrow{\theta_1}{}^{*} G' \xrightarrow{\theta_2}{}^{*} G''$
be a computation with restriction and $x \in \mathrm{n}(G'')$.
$\operatorname{orig}_{\theta_1\theta_2,G}(x) = \bigcup_{x_i \in \operatorname{orig}_{\theta_2,G'}(x)} \operatorname{orig}_{\theta_1,G}(x_i)$.

Lemma 11 *Let P be a program. In P we have the computation with restriction:*
$G \xrightarrow{\theta}{}_r^{*} G'$
iff we have the computation with restriction:
$\nu X\ G \xrightarrow{\theta'}{}_r^{*} \nu X'\ G'$
where:
$\theta' = \theta|_{\mathrm{fn}(\nu X\ G)}$
$X' = \{x \in \mathrm{fn}(G') | \operatorname{orig}_{\theta,G}(x) \subseteq X\}$

We will now give a characterization of the restricted mgu.

Theorem 10 *Let $G \xrightarrow{\theta'}{}_r^{*} G'$ be a restricted computation. We have $\theta' = \theta|_D$ where θ is the composition of actual mgu used and D is the set of all names that are free in some goal of the computation.*

3.4 Mapping HSHR to logic programming

In this section we present the translation of HSHR into Synchronized Logic Programming (SLP), a variation of logic programming where clause applications are grouped, i.e. synchronized, into big-steps.

A first version of the translation is presented in [Lan02]. More refined versions, but only dealing with logic programming without restriction, are discussed in [LM02, LM04].

SLP has been introduced because logic programming allows for many execution strategies and for complex interactions and data structures and so it is necessary to limit it in order to keep only the computations which have an HSHR correspondent.

We impose three kinds of restrictions:

- we use only goals that correspond to graphs (goal-graphs)
- we allow in the program only clauses that correspond to graph synchronizations (synchronized clauses)

- we choose only computations that model HSHR transitions (synchronized computations)

Definition 40 (Goal-graph) *We call goal-graph a goal which has no functional symbols.*

Definition 41 (Synchronized program) *A synchronized program is a finite set of synchronized rules, i. e. definite program clauses such that:*

- *the body of each rule is a goal-graph;*
- *the head of each rule is $A(t_1, \ldots, t_n)$ where t_i is either a variable or a single function symbol applied to variables. If it is a variable then it also appears (as a free name) in the body of the clause.*

Example 7

$q(f(x), y) \leftarrow p(x, y)$	*synchronized rule;*
$q(f(x), y) \leftarrow p(x, f(y))$	*not synchronized because $p(x, f(y))$ is not a goal-graph;*
$q(g(f(x)), y) \leftarrow p(x, y)$	*not synchronized because it contains nested functions;*
$q(f(x), y, f(z)) \leftarrow p(x)$	*not synchronized because y is an argument of the head predicate but it does not appear in the body;*
$q(f(x), f(z)) \leftarrow p(x)$	*synchronized rule, even if z does not appear in the body.*

In our approach, a clause $A(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_n$ represents a production where the head predicate A is the label of the edge in the left hand side, and the body $B_1, \ldots, B_n$ is the graph in the right hand side. Term t_i in the head represents the action occurring in x_i, if $A(x_1, \ldots, x_n)$ is the edge matched by the production. Intuitively, the first condition of Definition 41 states that the result of a local rewriting must be a goal-graph. The second condition forbids synchronizations with structured actions, which are not allowed in HSHR. Furthermore, it imposes that we cannot disconnect from a node without synchronizing on it[2].

Now we define the subset of computations we are interested in.

Definition 42 (Synchronized logic programming) *Given a synchronized program P we write:*

$$G_1 \stackrel{\theta}{\Rightarrow} G_2$$

[2]This condition has only the technical meaning of making impossible some big-steps in which an incorrect small-step may not be forbidden because its only effect is on the discarded variable. Luckily, we can impose this condition without altering the power of the formalism, because we can always perform a special foo action on the node we leave and make sure that all the other edges can freely do the same action. For example we can rewrite $q(f(x), y, f(z)) \leftarrow p(x)$ as $q(f(x), foo(y), f(z)) \leftarrow p(x)$, which is an allowed synchronized rule.

iff:
$G_1 \xrightarrow{\theta'}{}^{*} G_2$, *all steps performed in this computation expand different atoms of* G_1, $\theta'|_{\mathrm{fn}(G_1)} = \theta$ *(we use* $\mathrm{n}(G_1)$ *instead of* $\mathrm{fn}(G_1)$ *if we use logic programming without restriction) and both* G_1 *and* G_2 *are goal-graphs.*
We call $G_1 \stackrel{\theta}{\Rightarrow} G_2$ *a big-step and all the* $\rightarrow$ *steps in a big-step small-steps.*
A SLP computation is:
$G_1 \Rightarrow^* G_2$ *i. e. a sequence of 0 or more big-steps.*

We want to use SLP with restriction to model HSHR systems. In order to do that we need as a first step to translate hypergraphs, as we have defined them as syntactic judgements, to goal-graphs with restriction. In this translation we lose the part Γ of syntactic judgements.

Definition 43 (Translation for syntactic judgements) *We define the translation operator* $[\![-]\!]$ *as:*

$$[\![\Gamma \vdash L(x_1, \ldots, x_n)]\!] = L(x_1, \ldots, x_n)$$
$$[\![\Gamma \vdash G_1|G_2]\!] = [\![\Gamma \vdash G_1]\!], [\![\Gamma \vdash G_2]\!]$$
$$[\![\Gamma \vdash \nu x\ G]\!] = \nu x\ [\![\Gamma, x \vdash G]\!]$$
$$[\![\Gamma \vdash nil]\!] = \Box$$

Sometimes we will omit the Γ part of the syntactic judgement. We can do this because it does not influence the translation.

Definition 44 *Let* $\Gamma \vdash G$ *and* $\Gamma' \vdash G'$ *be graphs. We define the equivalence relation* $\cong$ *in the following way:* $\Gamma \vdash G \cong \Gamma' \vdash G'$ *iff* $G \equiv G'$.

Observe that if two judgements are equivalent then they can be written as:
$\Gamma, \Gamma_{unused} \vdash G$
$\Gamma, \Gamma'_{unused} \vdash G$
where $\Gamma = \mathrm{fn}(G)$.

Theorem 11 (Correspondence of judgements and goal-graphs) *The operator* $[\![-]\!]$ *defines an isomorphism between judgements (defined up to* $\cong$*) and goal-graphs.*

We now define the translation from HSHR productions to definite clauses.

Definition 45 (Translation from productions to clauses)
We define the translation operator $[\![-]\!]$ *as:*

$$[\![L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} G]\!] = L(a_1(\pi(x_1), \vec{y}_1), \ldots, a_n(\pi(x_n), \vec{y}_n)) \leftarrow [\![G]\!]$$

if $\Lambda(x_i) = (a_i, \vec{y}_i)$, $i = 1, \ldots, n$ *and* $a_i \neq \epsilon$. *If* $a_i = \epsilon$ *we write simply* $\pi(x_i)$ *instead of* $\epsilon(\pi(x_i))$.

The idea of the translation is that the condition given by an action $(x, a, \vec{y})$ is represented by using the term $a(\pi(x), \vec{y})$ as argument in the position that corresponds to x. Thus during unification x is bound to that term and, when other instances of x are met, the corresponding term must have the same functional symbol (as required by Hoare synchronization) in order to be unifiable. Furthermore, the corresponding tuples of transmitted names are unified. Since x disappears, we need another name to represent the node that corresponds to x. We use the first argument of a to this purpose. If two nodes are merged by π then their successors are the same as required.

Observe that we have no need to translate all the possible variants of the rules since variants with fresh names are automatically built when clauses are applied. Notice that the clauses we obtain are synchronized clauses.

The observable substitution contains information on Λ and π. Thus, given a transition, we can associate to it a substitution θ. We have different choices for θ according to where we map names. In fact, in HSHR, nodes are mapped to their representatives according to π, while, in SLP, θ cannot do the same, since the names of the clause variant must be all fresh. These choices correspond to changing by an injective renaming the names in the result of the big-step.

Definition 46 (Substitution associated to a transition)
Let $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ be a transition. We say that the substitution θ_ρ associated to this transition is:

$$\theta_\rho = \{a(\pi'(x'), \vec{y}')/x | \Lambda(x) = (a, \vec{y}), a \neq \epsilon, x \textit{ free in } G\} \cup \\ \cup \{\pi'(x')/x | \Lambda(x) = (\epsilon, \langle\rangle), x \textit{ free in } G\}$$

for some injective renaming $\rho(x) = x'$, $x \in \Gamma \cup \Gamma_\Lambda$. Here $\pi'(\rho(x)) = \rho(\pi(x))$. Note that if $\pi'(x') = x$ and $\Lambda(x) = (\epsilon, \langle\rangle)$ then the substitution for x disappears.

In order to prove the correspondence between HSHR and SLP we need some lemmas. We start characterizing mgus of sets of equations on variables.

Lemma 12 *Let S be a set of equations on variables. Let $\equiv_S$ be the equivalence relation defined by S. For each choice of representatives of the equivalence classes of $\equiv_S$:*
$\theta = \{[x]/x | [x] \neq x\}$
where $[x]$ is the representative of the equivalence class of x, θ is a mgu of S.

We present now an explicit characterization of big-steps.

Lemma 13 *Let $\nu X\ A_1, \ldots, A_n$ be a goal with restriction. We want to characterize the big-steps where the clause unified with A_i is $H_i \leftarrow B_i$ (if we want to substitute A_i). Let us number the name occurrences in $A_1, \ldots, A_n$. We denote with $\mathrm{gl}(i)$ the name that corresponds to the i-th occurrence, and with $\mathrm{cl}(i)$ the term to be unified* *with* $\mathrm{gl}(i)$*, i.e. if* $\mathrm{gl}(i) \in A_j$*, then* $\mathrm{cl}(i)$ *is the term in the corresponding position in* H_j *(if the atom is not substituted we define* $\mathrm{cl}(i) = \mathrm{gl}(i)$*). Let* R *be the equivalence on names defined by the following set of equations:*

$$\{x_i = x_j, \vec{y}_i = \vec{y}_j | \, \mathrm{gl}(i) = \mathrm{gl}(j), \mathrm{cl}(i) = a_i(x_i, \vec{y}_i), \mathrm{cl}(j) = a_j(x_j, \vec{y}_j)\} \cup \\ \cup \{\mathrm{cl}(i) = \mathrm{gl}(i) | cl(i) \text{ is a variable}\}$$

We denote with $[x]_R$ *the representative of the equivalence class of* x*. We extend the notation also to terms, atoms and goals:* $[t]_R$ *is the term* t *where each occurrence of a variable has been substituted by its representative.*

We have a big-step of the form:
$\nu X \; A_1, \ldots, A_n \overset{\theta|_{\mathrm{fn}(\nu X \; A_1,\ldots,A_n)}}{\Rightarrow} \nu X' \; G_1, \ldots, G_n$
iff
$\mathrm{gl}(i) = \mathrm{gl}(j) \Rightarrow \mathrm{cl}(i) = a_i(\ldots), \mathrm{cl}(j) = a_j(\ldots), a_i = a_j \vee cl(i) = x_i, cl(j) = x_j$
Furthermore we have:
$\theta = \{a_i([x_i]_R, [\vec{y}_i]_R)/gl(i) | \, \mathrm{cl}(i) = a_i(x_i, \vec{y}_i)\} \cup \{[x]_R/x | [x]_R \neq x\}$
$G_1, \ldots, G_n = [B_1, \ldots, B_n]_R$
$X' = \{x \in \mathrm{n}(G_1, \ldots, G_n) | \, \mathrm{orig}_{\theta,(A_1,\ldots,A_n)}(x) \subseteq X\}$

We can now present the correctness and completeness results for our translation.

Theorem 12 (Correctness) *Let* $\mathcal{P}$ *be a set of productions. Let* P *be the program of logic programming with restriction obtained by translating the productions in* $\mathcal{P}$ *according to Definition 45. If:*
$\mathcal{P} \Vdash_H (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$
then we can have using clauses in P *the following big-step of SLP with restriction:*
$[\![\Gamma \vdash G]\!] \overset{\theta_\rho}{\Rightarrow} T$
for every ρ *such that* $\rho(x)$ *is a fresh name unless* $x \in \Gamma \wedge \Lambda(x) = (\epsilon, \langle\rangle)$ *(in that case* $\rho(x) = x$*,* $\rho(x)$ *fresh is still possible if an edge attached to* x *is rewritten). Furthermore the substitution* θ_ρ *is associated to* $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ *and* $T\rho^{-1} = [\![\Phi \vdash G']\!]$*.*

Theorem 13 (Completeness) *Let* $\mathcal{P}$ *be a set of productions. Let* P *be the program of logic programming with restriction obtained by translating the productions in* $\mathcal{P}$ *according to Definition 45. If we have in* P *a big-step of SLP with restriction:*
$[\![\Gamma \vdash G]\!] \overset{\theta}{\Rightarrow} T$
then there exist ρ*,* θ'*,* Λ*,* π*,* Φ *and* G' *such that* $\theta = \theta'_\rho$ *is associated to* $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$*. Furthermore* $T\rho^{-1} = [\![\Phi \vdash G']\!]$ *and* $\mathcal{P} \Vdash_H (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$*.*

Example 8 *We present here the translation into SLP with restriction of Example 1. The starting graph,* $x \vdash C(x, x)$*, becomes the goal* $C(x, x)$*. We have the two* *following productions:*

$$x, y \vdash C(x,y) \xrightarrow{(x,\epsilon,\langle\rangle),(y,\epsilon,\langle\rangle)} x, y \vdash \nu z\ C(x,z)|C(z,y)$$

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

whose translation is:
$C(x,y) \leftarrow \nu z\ C(x,z), C(z,y)$
$C(r(x,w), r(y,w)) \leftarrow S(y,w)$
Thus we have the following computation:

$$\begin{aligned} C(x,x) \Rightarrow \nu y\ C(x,y), C(y,x) \Rightarrow \\ \Rightarrow \nu y, z\ C(x,y), C(y,z), C(z,x) \Rightarrow \\ \Rightarrow \nu y, z, v\ C(x,y), C(y,z), C(z,v), C(v,x) \overset{r(x',w)/x}{\Rightarrow} \\ \Rightarrow \nu y', z', v'\ S(x',w), S(y',w), S(z',w), S(v',w) \end{aligned}$$

Let us consider in more detail the last big-step. It is associated to:

$$\begin{aligned} x \vdash \nu y, z, v\ C(x,y)|C(y,z)|C(z,v)|C(v,x) \xrightarrow{(x,r,\langle w\rangle)} \\ \rightarrow x, w \vdash \nu y, z, v\ S(x,w)|S(y,w)|S(z,w)|S(v,w) \end{aligned}$$

and the substitution $\theta = \{r(x',w)/x\}$ *can be expressed as* θ'_ρ *where* $\rho = x'/x$*. In fact* $(\nu y', z', v'\ S(x',w), S(y',w), S(z',w), S(v',w))\rho^{-1}$ *is* α*-convertible to:*
$\nu y, z, v\ S(x,w), S(y,w), S(z,w), S(v,w)$
as required.

We end this section with a simple schema on the correspondence between the two models.

HSHR	SLP	HSHR	SLP
Graph	Goal	Transition	Big-step
Edge	Atomic goal	Node	Name
Parallel comp.	And comp.	Restriction	Restriction
Production	Clause	Nil	$\square$

3.5 Putting the steps together

In this section we present a complete example, which translates a Fusion Calculus computation into an SLD-resolution in SLP with restriction. In order to make it possible we must start from a closed process.

Let us consider the fusion transition:
$(uxyzw)(P|\overline{u}xy.Q|uzw.R) \rightarrow (uxy)(P|Q|R)\{x/z, y/w\}$.
The translation of the starting process into a graph is:

$\nu u, x, y, z, w\ L_P|L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w)$.
Furthermore we have the following productions:

$$x_1, x_2, x_3 \vdash L_{\overline{x_1}x_2x_3.Q}(x_1, x_2, x_3) \xrightarrow{(x_1, out_2, \langle x_2, x_3 \rangle)} x_1, x_2, x_3 \vdash L_Q$$
$$x_1, x_2, x_3 \vdash L_{x_1x_2x_3.R}(x_1, x_2, x_3) \xrightarrow{(x_1, in_2, \langle x_2, x_3 \rangle)} x_1, x_2, x_3 \vdash L_R$$

By applying rule (ren) we can derive:

$$u, x, y, z, w \vdash L_{\overline{x_1}x_2x_3.Q}(u, x, y) \xrightarrow{(u, out_2, \langle x, y \rangle)} u, x, y, z, w \vdash L_Q$$
$$u, x, y, z, w \vdash L_{x_1x_2x_3.R}(u, z, w) \xrightarrow{(u, in_2, \langle z, w \rangle)} u, x, y, z, w \vdash L_R$$

We can then apply rules (com/close):

$$u, x, y, z, w \vdash L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w)$$
$$\xrightarrow{\{x/z, y/w\}} u, x, y \vdash (L_Q|L_R)\{x/z, y/w\}$$

(par):

$$u, x, y, z, w \vdash L_P|L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w)$$
$$\xrightarrow{\{x/z, y/w\}} u, x, y \vdash (L_P|L_Q|L_R)\{x/z, y/w\}$$

and finally 5 times rule (res):

$$\vdash \nu u, x, y, z, w\ L_P|L_{\overline{x_1}x_2x_3.Q}(u, x, y)|L_{x_1x_2x_3.R}(u, z, w)$$
$$\rightarrow \vdash \nu u, x, y\ (L_P|L_Q|L_R)\{x/z, y/w\}$$

getting the desired transition. We can now translate this transition into the SHR with Hoare synchronization framework. The starting graph becomes:

$$\vdash \nu u_1, u_2, x, y, z, w\ L_P|L_{\overline{x_1}x_2x_3.Q}(u_1, x, y)|L_{x_1x_2x_3.R}(u_2, z, w)|$$
$$am_2(u_1, u_2)|null(x)|null(y)|null(z)|null(w)$$

Using for productions the shorter form shown in Example 5 we have:

$$x_1, x_2, x_3 \vdash L_{\overline{x_1}x_2x_3.Q}(x_1, x_2, x_3) \xrightarrow{(x_1, out_2, \langle x_2, x_3 \rangle)} x_1, x_2, x_3 \vdash L_Q|null(x_1)$$
$$x_1, x_2, x_3 \vdash L_{x_1x_2x_3.R}(x_1, x_2, x_3) \xrightarrow{(x_1, in_2, \langle x_2, x_3 \rangle)} x_1, x_2, x_3 \vdash L_R|null(x_1)$$

Thus, by using also the auxiliary production for am_2 connector, we have:

$$\vdash \nu u_1, u_2, x, y, z, w\ L_P|L_{\overline{x_1}x_2x_3.Q}(u_1, x, y)|L_{x_1x_2x_3.R}(u_2, z, w)|$$
$$am_2(u_1, u_2)|null(x)|null(y)|null(z)|null(w) \rightarrow$$
$$\vdash \nu u_1, u_2, x, y, z, w\ L_P|L_Q|null(u_1)|L_R|null(u_2)|$$
$$am_2(u_1, u_2)|am_2(x, z)|am_2(y, w)|null(x)|null(y)|null(z)|null(w)$$

Finally we can move to the SLP setting, where productions are transalted into:

$$L_{\overline{x_1}x_2x_3.Q}(out_2(x_1,x_2,x_3),x_2,x_3) \leftarrow L_Q, null(x_1)$$
$$L_{x_1x_2x_3.R}(in_2(x_1,x_2,x_3),x_2,x_3) \leftarrow L_R, null(x_1)$$

and the transtion becomes the following big-step:

$$\begin{array}{c} \nu u_1,u_2,x,y,z,w\ L_P, L_{\overline{x_1}x_2x_3.Q}(u_1,x,y), L_{x_1x_2x_3.R}(u_2,z,w), \\ am_2(u_1,u_2), null(x), null(y), null(z), null(w) \Rightarrow \\ \nu u_1,u_2,x,y,z,w\ L_P, L_Q, null(u_1'), L_R, null(u_2'), \\ am_2(u_1',u_2'), am_2(x',z'), am_2(y',w'), null(x'), null(y'), null(z'), null(w') \end{array}$$

which is equal to the wanted one up-to α-conversion and up-to equivalence of amoeboids.

Chapter 4

Comparing strategies for composing system: categorical approach vs algebraic approach

In this chapter we present the first results of an analysis on strategies for composing systems. In particular, we compare the categorical approach, and in particular the CommUnity approach, with the algebraic approach, as realized by the Tile Model.

The main results of our investigation are:

- while modeling CommUnity programs and diagrams with tiles, we identified a standard decomposition for each program, which clearly separates all the aspects involved: channel managers, guard managers, coordination;
- a new algebra of connectors has been developed and applied to tile configurations thanks to the coordination mechanism of CommUnity;
- a suitable axiomatization of the above mentioned algebra of connectors has been defined which captures the colimit construction in CommUnity;
- the separation of concerns has been exported from CommUnity to the Tile Model by achieving a clean separation between state and functionalities in the latter.

4.1 Standard decomposition of CommUnity programs

In this section we present a decomposition for CommUnity programs. This decomposition transforms a complex program in a structure of the form glue-cables-roles with simpler components. This is useful in order to emphasize the algebraic structure behind a CommUnity program and this constitutes the first step of our mapping from CommUnity into the Tile Model.

Given a CommUnity program we decompose it in a diagram with four kinds of components:

- a *glue*, which is the starting program from which we remove all information on the actual actions. The glue has as many actions as the number of actions in the starting program, but each action in the glue has the form $true \rightarrow skip$. Furthermore the glue has one input channel for each input/output channel of the starting program (with the same sort);

- a *channel manager* for each output channel in the starting program. The channel manager for channel x has exactly one action for each action in the starting program, with $true$ guard and as body the assignment (if any) in the action that assigns x. To some extent, each action in the channel manager is the projection on x of the corresponding action in the starting program. The channel manager has as channels x (as output channel) and all the channels needed by its assignments as input channels;

- a *guard manager* for each action in the starting program. The guard manager has exactly one action in the form $p \rightarrow skip$ where p is the guard of the corresponding action in the starting program. The guard manager has exactly the channels needed for evaluating its guard, all as input channels;

- *cables* to connect each channel manager and each guard manager to the glue. Each cable has one action for each action of the corresponding role, mapped to that action and to the corresponding action in the glue, and one input channel for each channel of the corresponding role, mapped to that channel and to the corresponding channel in the glue.

Roughly, letting n and m be respectively the number of output channels in the starting program $\mathcal{D}$ and the number of actions in $\mathcal{D}$, then the standard decomposition of $\mathcal{D}$ has n channel managers, m guard managers, $n + m$ cables and one glue. The only morphisms in the decomposition are the ones that connect each cable to its role and to the glue. Note that all the morphisms are injective. Given a program $\mathcal{D}$, we denote by $\mathsf{DS}(\mathcal{D})$ its standard decomposition diagram. We write $\mathcal{D}_1 \cong \mathcal{D}_2$ when programs $\mathcal{D}_1$ and $\mathcal{D}_2$ are isomorphic.

Figure 4.1 shows a sample decomposition. In the figure we have not explicitly represented morphisms' details, but we have used as notational convention the same name for denoting corresponding actions and corresponding channels (this is only a convention, since all names are local). We have avoided a guard manager for the action *inc* since the guard is true (adding the guard manager does not change the operational behaviour of the system).

The correctness of the decomposition is shown in the following theorem.

Theorem 14 *For each program $\mathcal{D}$, we have* $\mathsf{colim}(\mathsf{DS}(\mathcal{D})) \cong \mathcal{D}$.

We can extend the notion of standard decomposition also to diagrams. The standard decomposition of a diagram **dia** is a diagram obtained by substituting each non cable program with its standard decomposition. The morphisms entering a program become morphisms entering the glue of its standard decomposition (note that the glue has all actions and all channels, thus this can be done).

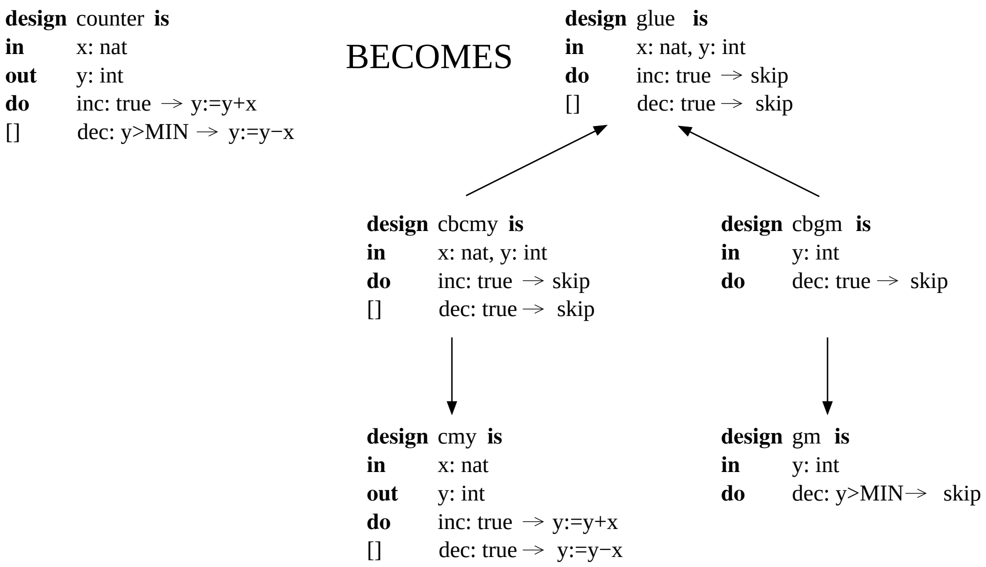


Figure 4.1: Standard decomposition of a CommUnity program.

Corollary 1 *For each diagram* **dia** *we have* $\mathsf{colim}(\mathsf{DS}(\mathbf{dia})) \cong \mathsf{colim}(\mathbf{dia})$.

4.2 Mapping CommUnity into the Tile Model

In this section we define the operational and abstract semantics of CommUnity programs (and diagrams) by exploiting a compositional encoding in the Tile Model.

The mapping maps a CommUnity diagram into a tile system together with an initial configuration for the system. Computations of the CommUnity diagram correspond to the tiles derivable in the tile system that have this initial configuration.

In order to have a clear separation between functionalities of the system and the state of the system, each configuration is the horizontal composition of two parts: one that corresponds to the state (in order to build this part we need to know the actual values used for each channel) and the other one that corresponds to the *unanchored* system, i.e. the system without state. The unanchored system has more behaviours w.r.t. the anchored system, but these behaviours are not compatible with any tile that corresponds to an evolution of the state. For instance, when considering unanchored systems, it is possible to compose steps that make unsatisfiable assumptions on the values of channels.

To define the tile system associated to a CommUnity diagram we must first fix a total ordering over the programs in the diagram. Then we translate each role/glue separately, we compose them in parallel and finally we translate each cable (and its associated span of morphisms) by suitably connecting the interfaces of the two components which are instances of the role and the glue connected to the cable. A main result ensures that the behaviour of the resulting tile system is independent from the choice of the total ordering over programs, so that the choice is immaterial.

As usual, to define the tile system we must explain which are its objects, its configurations, its observations and its tiles.

4.2.1 Tile objects

The objects of our tile system are tuples that contains the following elements:

- channels: these are specified by a sort (boolean, integer, ...) and a modality (input or output) exactly as in CommUnity;
- a special boolean object b that is attached to the evaluation of guards;
- placeholders for actions, which play the role of synchronization objects.

We denote tuples of channels (and possibly special boolean objects) with *chs* with an optional subscript to denote their cardinality and/or sorts, furthermore we write *ins* instead of *chs* to specify that all channels in the tuple are input channels (or possibly special boolean objects) and similarly *outs* for outputs. We denote a synchronization object with 1 and a tuple of n synchronization objects with n.

4.2.2 Tile configurations

We present here the horizontal category of our tile system.

For the moment we use as category the symmetric strict monoidal category freely generated by the basic arrows below (we will add further axioms in Section 4.3). Note that symmetries allow for permuting in all possible ways the (order of the) objects defining the interface of any configuration. The basic arrows can be thought of as the basic building blocks of a configuration: they can be composed sequentially and in parallel to build larger configurations, and any configuration can be expressed as the (parallel and sequential) composition of basic arrows. For each arrow we specify also input and output interfaces.

- $state[values : sorts] : 0 \to chs$ where $values$ is a tuple of values whose sorts are in $sorts$ and chs a tuple of channels with these sorts (note that the order of the elements in the tuple is meaningful); arrows of this form model the actual state of one anchored configuration;
- $\nabla_{chs,ins \otimes chs} : chs \to ins \otimes chs$ where ins and chs have the same number of elements and where corresponding elements have the same sort (but some channels may be output channels in chs); arrows of this form are used to connect components linked by cables;

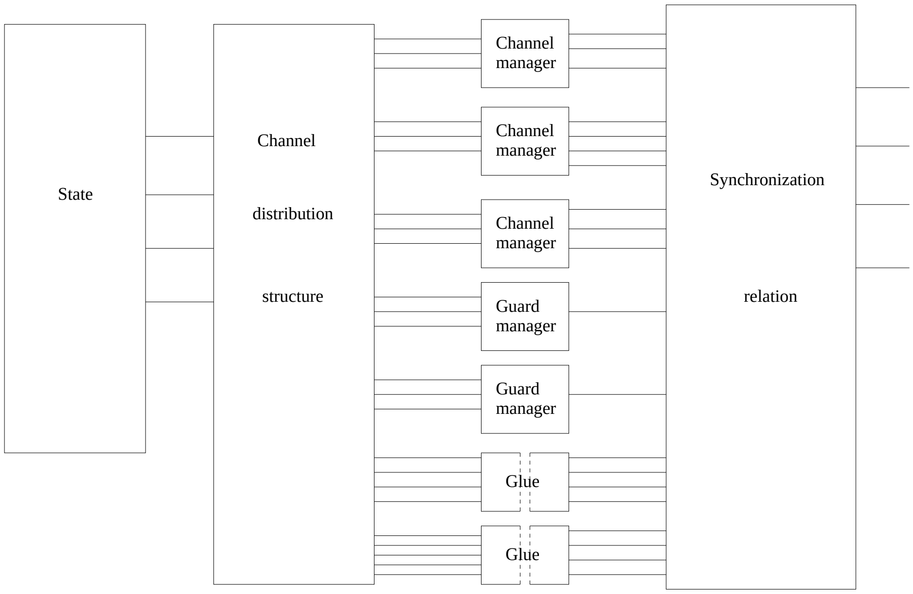


Figure 4.2: Initial configuration for a Community program.

- $cm[\langle f_i \rangle_{i=1,\ldots,n}] : chs \otimes b \to n$ where chs contains exactly one output channel and the f_is are functions on channels in chs (to the output channel); arrows of this form model channel managers;

- $gm[p] : ins \otimes b \to 1$ where p is a predicate on the channels in ins; arrows of this form model guard managers;

- $\dot{\nabla}_n : n \to 2n$ which is a mutual exclusion connector; arrows of this form are used to model glues and to connect components linked by cables;

- $\nabla_n : n \to 2n$ and $\Delta_n : 2n \to n$ which are synchronization connectors; arrows of this form are used to connect components linked by cables;

- $!_n : n \to 0$ which is an hiding connector; arrows of this form are used to model glues and to connect components linked by cables;

- $\overline{\mathbf{1}} : 0 \to 1$ which is a clock connector that forces some actions to be performed; it is used for modeling glues.

These connectors are enough to specify the starting configuration that is part of the result of the translation of a CommUnity diagram into the Tile Model. The structure of the initial configuration is shown in Figure 4.2.

Given a well-formed connected diagram on which we have applied the standard decomposition we want to build such an arrow in a compositional way. Note that

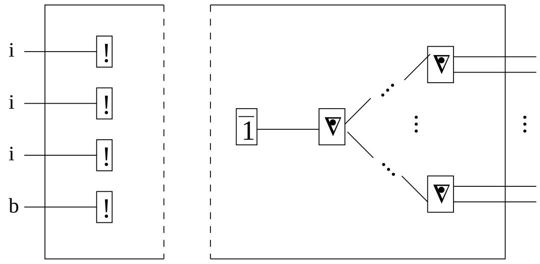


Figure 4.3: A glue as an arrow.

all the non-cable programs in the diagram are connected through cables. We first show how to translate each non-cable program into an horizontal arrow and then we use as basic operation to build up the system the "parallel composition through one cable". Thanks to the hypothesis on the CommUnity diagram, a sequence of operations of this kind allows to translate the whole diagram. At the end we can possibly add the state (defining an anchored configuration).

During the translation we have to remember which are the channels that correspond to each channel object and which is the action that corresponds to each synchronization object.

The translation of a channel manager is a basic arrow $cm[\langle f_i \rangle_{i=1,\ldots,n}] : chs \otimes b \to n$ where chs contains the channels used by the channel manager and f_i is the function of the input channels that is assigned to the output channel during the i-th action. The i-th action corresponds to the i-th synchronization object in the interface.

The translation of a guard manager is a basic arrow $gm[p] : ins \otimes b \to 1$ where p is the guard of the only action of the guard manager. This also corresponds to the only synchronization object in the interface.

The translation of a glue with n actions is a tree composed by $\bar{\nabla}$ connectors with n leaves and with a $\overline{\mathbf{1}}$ connector as root together with the ! for all its channels (plus one for a special boolean object), see e.g. Figure 4.3.

Note that we have a bijective correspondence between channels in a program and channels in the left interface of its translation (where furthermore we have exactly a special boolean object). Similarly, there is a bijective correspondence between actions of a program and synchronization objects in its right interface.

We show here how the operation of "parallel composition through one cable" is performed. Suppose we have a cable with channels $x_1, \ldots, x_n$ and actions $a_1, \ldots, a_n$. Each channel is mapped through morphisms to two groups of channels, one for each of the diagrams to be composed. Each action is the image of zero or more actions from the diagrams to be composed.

The resulting arrow is obtained by taking the parallel composition of the translations of the two diagrams to be composed. On the left we merge, using trees of ∇ connectors (and possibly some permutations in order to have the corresponding

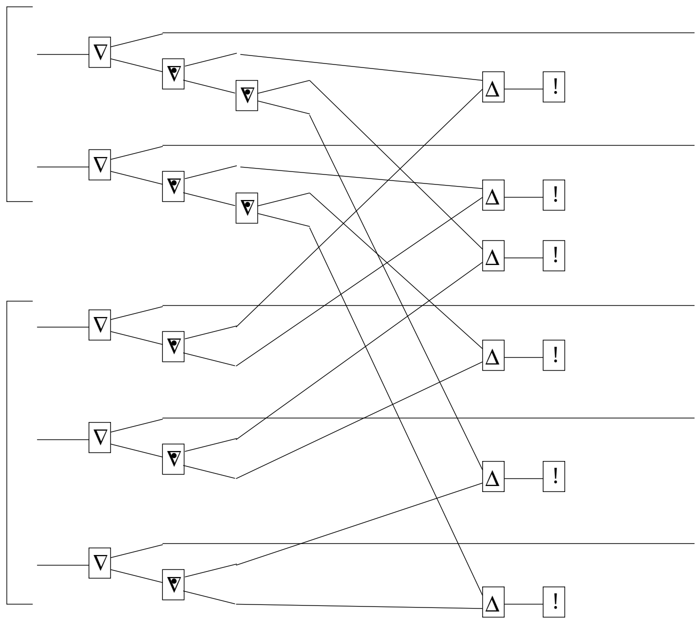


Figure 4.4: Connecting actions.

channel objects adjacent), the two special boolean objects and all the channels that are mapped to the same channel in the cable (for each channel in the cable the set of the channels to be merged is the union of two sets, that are the sets of channels in the two programs that are mapped to the considered channel in the cable).

On the right we have to synchronize tuples of actions that are mapped to the same action in the cable. In order to synchronize a tuple t_1 of n actions with a tuple t_2 of m actions we have to duplicate each action. Then we create using ∇ connectors n copies of each object in t_2 and m copies of each object in t_1. Then we merge using Δ connectors each action in the first group with each action in the second group and we close the resulting objects using ! connectors. See Figure 4.4 for an example (groups of 2 and 3 actions respectively).

The resulting arrow has in the left interface all the channels modulo equivalence (when further composing, if we refer twice the same channel we do nothing) and in the right interface all the actions in the composed diagrams (that thus can still be referenced when further composing).

When the whole diagram has been mapped we can add the state to the left, connecting a value of the correct type to each channel object, and, in order to compute the colimit, we also need to close all the synchronization objects using ! connectors.

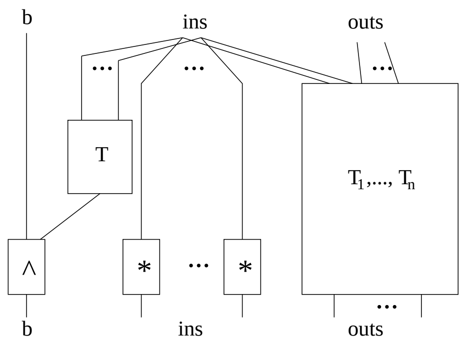


Figure 4.5: Structure of functional terms.

4.2.3 Tile observations

We use as vertical category for our tile system the cartesian category freely generated from the following basic arrows:

- tick : $1 \to 1$, which expresses that the action associated with the initial interface is taking place;
- untick : $1 \to 1$, which expresses that the action associated with the initial interface is inhibited;
- $* : in \to in$ where in is an input channel; this arrow stands for a guess on the real function and it will be instantiated when the actual assignment is found (note that we can not use all possible assignments since they depend on all the possible channels and we can not connect to them);
- $f : chs \to out$ where f is a functional symbol with domain corresponding to the types in chs (chs does not contain special boolean objects) and out is an output channel of the right type (if the type is boolean we can have b for the special boolean object instead of out);
- $\wedge : b \otimes b \to b$ (conjunction).

When building tiles we always have as observations (for a single step) either tuples of tick and untick or functional terms composed using the other connectors.

Functional terms are of the form $Term : b \otimes ins \otimes outs \to b \otimes ins \otimes outs$ and have the structure in Figure 4.5.

The form of functional terms is constrained because our system has a static structure, thus the initial and the final interfaces must be equal (only values in the state can change). In figure, T and $T_1, \ldots, T_n$ are normal terms generated using functional symbols and auxiliary structures. For instance, a term $f(x, x)$ where x is a natural number is represented by $\nabla_{Nat}; f$. The whole structure associates the condition expressed by T to the special boolean object, a $*$ observation to each input (the intuitive meaning is that the input is controlled by the environment and so it can change in any way) and a normal term, namely terms $T_1, \ldots, T_n$, to each output.

4.2.4 Tiles

We present now in detail the rules defining the behaviour of the configurations in terms of allowed observations. Since, as we have already said, the structure of our system is static, we usually have tiles with equal initial and final configurations, except for the values in the state which can change. Some tiles do not respect this constraint, but they can only be used, in order to build a tile with a well-formed initial configuration, combined in such a way that the condition is satisfied by the whole tile.

We have the following basic tiles:

- $state[values : sorts] \xrightarrow[Term]{id_0} state[values' : sorts]$ where $Term$ is as described above (see Figure 4.5) and the condition T is satisfied by $values$ (this means that the special boolean object is always true) and $values'$ is obtained evaluating the terms on $values$ (any value of the right type can be an evaluation of $*$);

- for value synchronization we have three different kinds of rules
  $\nabla_i \xrightarrow[*\otimes *]{*} \nabla_i$
  $\nabla_b \otimes id_{b\otimes b}; id_b \otimes \gamma_{b,b\otimes b}; \gamma_{b,b} \otimes id_{b\otimes b} \xrightarrow[\wedge\otimes\wedge]{id_b\otimes\wedge;\wedge} \nabla_b$
  $\nabla_{o,i\otimes o} \otimes id_n \xrightarrow[*\otimes f]{f} \nabla_{o,i\otimes o}$
  where the last tile is replicated for each functional symbol $f : o \otimes ins_n \to o$;

- there are n possible tiles for each channel manager with n actions, of the form $cm[\langle f_i\rangle_{i=1,\ldots,n}] \xrightarrow[\mathsf{untick}^{i-1}\otimes\mathsf{tick}\otimes\mathsf{untick}^{n-i}]{f_i} cm[\langle f_i\rangle_{i=1,\ldots,n}]$
  where untick^{i-1} denotes the monoidal product of $i-1$ instances of untick (similarly for untick^{n-i}), and the term f_i has the usual structure with an only output channel (whose assigned value is f_i) and with $true$ as predicate;

- there are two possible tiles for each guard manager:

  - $gm[p] \xrightarrow[\mathsf{tick}]{p} gm[p]$ where p is a term with the usual structure with no output channels and with p as T (see Figure 4.5);
  - $gm[p] \xrightarrow[\mathsf{untick}]{true} gm[p]$ where $true$ is coded in the usual form;

- there are three possible evolutions for the $\boldsymbol{\nabla}\!\!\!\!\cdot\;$ connector:

$$\boldsymbol{\nabla}\!\!\!\!\cdot\; \xrightarrow[\mathsf{tick}\otimes\mathsf{untick}]{\mathsf{tick}} \boldsymbol{\nabla}\!\!\!\!\cdot\; \qquad \boldsymbol{\nabla}\!\!\!\!\cdot\; \xrightarrow[\mathsf{untick}\otimes\mathsf{tick}]{\mathsf{tick}} \boldsymbol{\nabla}\!\!\!\!\cdot\; \qquad \boldsymbol{\nabla}\!\!\!\!\cdot\; \xrightarrow[\mathsf{untick}\otimes\mathsf{untick}]{\mathsf{untick}} \boldsymbol{\nabla}\!\!\!\!\cdot\;$$

The first two rules express that trees of mutual exclusion connectors $\nabla\!\!\!\!\nabla$ can propagate an activity from the root to exactly one leaf, and that any leaf can be selected for that; the third rule states that if activities are inhibited for a node x, then the same holds for all nodes in the whole subtree rooted at x;

- there are two tiles for the ∇_1 connector:

$$\nabla_1 \xrightarrow[\mathsf{tick}\otimes\mathsf{tick}]{\mathsf{tick}} \nabla_1 \qquad \nabla_1 \xrightarrow[\mathsf{untick}\otimes\mathsf{untick}]{\mathsf{untick}} \nabla_1$$

  that model synchronization of shared actions;

- similarly, there are two tiles for the Δ_1 connector:

$$\Delta_1 \xrightarrow[\mathsf{tick}]{\mathsf{tick}\otimes\mathsf{tick}} \Delta_1 \qquad \Delta_1 \xrightarrow[\mathsf{untick}]{\mathsf{untick}\otimes\mathsf{untick}} \Delta_1$$

- there are two tiles for the $!_1$ connector:

$$!_1 \xrightarrow[id_0]{\mathsf{tick}} !_1 \qquad !_1 \xrightarrow[id_0]{\mathsf{untick}} !_1$$

  Since there are only two possible observations that can take place on action placeholders and both are let through by $!_1$, we have that $!_1$ does not constrain the behaviour of the system, but just hides part of the interface to the environment;

- finally there is exactly one tile for the $\overline{\mathbf{1}}$ connector: $\overline{\mathbf{1}} \xrightarrow[\mathsf{tick}]{id_0} \overline{\mathbf{1}}$, that forces any component $\overline{\mathbf{1}}$ to be always active.

As an example we show in Figure 4.6 a state tile (the term has been represented in compact form).

4.2.5 The encoding and its properties

The tile system for CommUnity is entirely determined by all the objects, configurations, observations and tiles discussed so far. Of course, there are many ill-formed configurations that find no correspondence with CommUnity diagrams, so we restrict the attention to configurations which are images of some CommUnity diagram.

The operational semantics is then given by taking as transitions the tiles whose initial configuration is the image of a CommUnity diagram. The abstract semantics is given by tile bisimilarity.

Although the details of the encoding are omitted here, the formal definition is given inductively on the size of the diagram. As we have anticipated, the induction takes place by assuming a total ordering of programs. Let **dia** be a CommUnity configuration diagram, let $\mathsf{DS}(\mathbf{dia})$ be its standard decomposition, let $\leq$ be a total

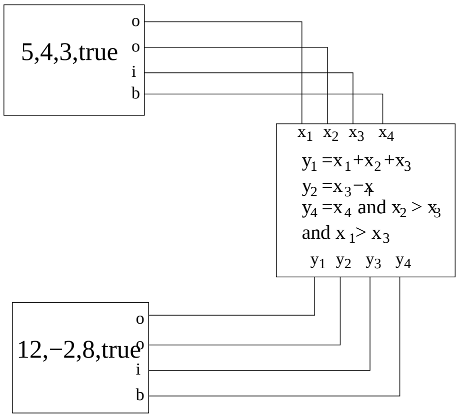


Figure 4.6: A sample state tile.

ordering on the programs in $\mathsf{DS}(\mathbf{dia})$, and let *values* denote the initial state (i.e. a consistent assignment of values to the channels in all programs); then we denote by $\mathsf{TS}(\mathbf{dia}, \leq, values)$ the resulting initial anchored configuration, where all synchronization objects have been closed using $!_1$ connectors (and by extension the tile system associated with it). We denote by $\mathsf{TS}(\mathbf{dia}, \leq)$ the corresponding unanchored configuration. Note that the configuration $\mathsf{TS}(\mathbf{dia}, \leq, values)$ has empty input and output interfaces, while the input interface of $\mathsf{TS}(\mathbf{dia}, \leq)$ has the form $b \otimes chs$, for chs the list of typed channels in $\mathsf{colim}(\mathbf{dia})$ (all channels that are mapped to the same channel in the colimit are merged using ∇ when encoding cables). When **dia** is a single program, the total order is fixed and we write just $\mathsf{TS}(\mathbf{dia}, \bullet, values)$ and $\mathsf{TS}(\mathbf{dia}, \bullet)$.

We denote with $\mathbf{dia}[values]$ the configuration that corresponds to **dia** with a state containing *values* as values; note that only channels in the colimit need to be assigned a value since all the channels in the diagram are mapped into them.

Proposition 3 *If a tile has initial configuration* $\mathsf{TS}(\mathbf{dia}, \leq, values)$*, then there exist an assignment of values values′ such that the final configuration is* $\mathsf{TS}(\mathbf{dia}, \leq, values')$*.*

Since the operational semantics of CommUnity is not defined in the literature, we state the correctness result w.r.t. the intended behaviour of the program obtained as colimit of the diagram.

Theorem 15 (Correctness) *We have a tile with initial configuration* $\mathsf{TS}(\mathbf{dia}, \leq, values)$ *and final configuration* $\mathsf{TS}(\mathbf{dia}, \leq, values')$ *iff there exists a sequence of enabled actions from* $\mathsf{colim}(\mathbf{dia}[values])$ *that arrives in the state* $\mathsf{colim}(\mathbf{dia}[values'])$*.*

It follows that the ordering considered in the encoding is immaterial.

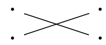
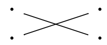
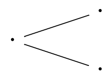
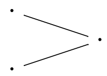
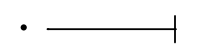
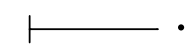
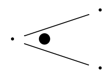
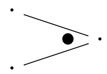
Ordinary structure		Dual structure	
symbolic notation	graphical	symbolic notation	graphical
$\gamma\colon 2 \to 2$		$\gamma\colon 2 \to 2$	
$\nabla\colon 1 \to 2$		$\Delta\colon 2 \to 1$	
$!\colon 1 \to 0$		$¡\colon 0 \to 1$	
$\nabla\!\!\!\!\!\cdot\;\colon 1 \to 2$		$\Delta\!\!\!\!\!\cdot\;\colon 2 \to 1$	
$\mathbf{0}\colon 1 \to 0$		$\overline{\mathbf{0}}\colon 0 \to 1$	
$\mathbf{1}\colon 1 \to 0$		$\overline{\mathbf{1}}\colon 0 \to 1$	

Figure 4.7: Syntax.

Proposition 4 *Let* **dia** *be a CommUnity configuration diagram, let* $\mathsf{DS}(\mathbf{dia})$ *be its standard decomposition, let* $\leq$, $\leq'$ *be two total orderings on the programs in* $\mathsf{DS}(\mathbf{dia})$*, and let values denote the initial state. Then,* $\mathsf{TS}(\mathbf{dia}, \leq, values) \simeq_{\mathrm{t}} \mathsf{TS}(\mathbf{dia}, \leq', values)$*. Moreover, there exists a symmetry* ρ *such that* $\mathsf{TS}(\mathbf{dia}, \leq) \simeq_{\mathrm{t}} (id_b \otimes \rho); \mathsf{TS}(\mathbf{dia}, \leq')$*.*

The symmetry ρ is needed to rearrange the input interface $\mathsf{TS}(\mathbf{dia}, \leq')$ so that channel objects having the same position in $\mathsf{TS}(\mathbf{dia}, \leq)$ and $(id_b \otimes \rho); \mathsf{TS}(\mathbf{dia}, \leq')$ are associated to the same channel in $\mathsf{colim}(\mathbf{dia})$.

Finally we have the easy corollary below.

Corollary 2 $\mathsf{TS}(\mathsf{colim}(\mathbf{dia}), \bullet, values) \simeq_{\mathrm{t}} \mathsf{TS}(\mathbf{dia}, \leq, values)$*. Moreover, there exists a symmetry* ρ *such that* $\mathsf{TS}(\mathsf{colim}(\mathbf{dia}), \bullet) \simeq_{\mathrm{t}} (id_b \otimes \rho); \mathsf{TS}(\mathbf{dia}, \leq)$*.*

4.3 Axiomatization of colimits

In this section we show that we can establish a much stronger link between the configurations $\mathsf{TS}(\mathbf{dia}, \leq)$ and $\mathsf{TS}(\mathsf{colim}(\mathbf{dia}), \bullet)$, which is not just behavioural, but structural. The idea is to axiomatize the algebra of connectors employed in the tile system for modeling linking through cables in such a way that all diagrams with the same colimit yield exactly the same configuration (modulo connectors' axioms). To this aim, it is useful to consider a richer algebra of connectors than the one strictly necessary for (behaviourally) encoding CommUnity diagrams.

Therefore, the algebra of connectors we consider is freely generated by composing in all possible ways, sequentially and in parallel, the basic connectors in Figure 4.7 and identities.

id	0	1
0	✓	
1		✓

γ	00	01	10	11
00	✓			
01			✓	
10		✓		
11				✓

∇	00	01	10	11
0	✓			
1				✓

Δ	0	1
00	✓	
01		
10		
11		✓

!	$\emptyset$
0	✓
1	✓

¡	0	1
$\emptyset$	✓	✓

$\dot{\nabla}$	00	01	10	11
0	✓			
1		✓	✓	

$\dot{\Delta}$	0	1
00	✓	
01		✓
10		✓
11		

$\mathbf{0}$	$\emptyset$
0	✓
1	

$\overline{\mathbf{0}}$	0	1
$\emptyset$	✓	

$\mathbf{1}$	$\emptyset$
0	
1	✓

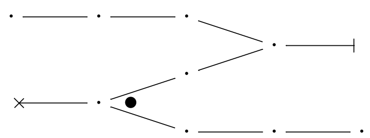
$\overline{\mathbf{1}}$	0	1
$\emptyset$		✓

Figure 4.8: Intuitive semantics.

Each connector can be seen as constraining the behaviours of the connections in its interface. Assuming that each attach point can receive two different signals (either tick expressing an ongoing activity on that object or untick expressing forced inactivity), then the abstract behaviour of a generic connector $\alpha\colon n \to m$ can be represented as a $2^n \times 2^m$ table whose cells have boolean values. Each row represents a combination of tick/untick values for the n inputs, while each column represents a combination of tick/untick values for the m outputs. If a cell is true (i.e. marked), then the corresponding combination of inputs and outputs is admissible, otherwise (the cell is false, i.e. empty, unmarked) the corresponding combination of inputs and outputs is forbidden. We call tick-table any table of this form.

The tables for basic constructors are in Figure 4.8. We shall abbreviate by $T(\alpha)$ the table representing the abstract behaviour of the connector α (note that tables of dual operators are related by transposition). Rather interestingly, we have that $T(\alpha;\beta)$ corresponds to the product matrix $T(\alpha) \times T(\beta)$, i.e. $T(\alpha;\beta)[i,j] = \bigvee_k (T(\alpha)[i,k] \wedge T(\beta)[k,j])$. Instead, $T(\alpha \otimes \beta)$ is obtained by refining each marked entry of $T(\alpha)$ by a copy of the table $T(\beta)$, and each unmarked entry of $T(\alpha)$ by the empty table with the same dimension as $T(\beta)$.

Thus, while the *concrete semantics* of $\alpha\colon n \to m$ is the solution of its associated network of constraints S, i.e. the set of assignments to tick/untick values to all the nodes appearing in the the graph denoted by α, we can assume that the *abstract semantics* of α is the solution of the network, where all the internal nodes (i.e. neither inputs nor outputs) have been existentially quantified.



neg	0	1
0		✓
1	✓	

Figure 4.9: Negation.

Proposition 5 *If we consider the subalgebra of connectors which are built using neither* $\mathbf{1}$ *nor* $\overline{\mathbf{1}}$*, then each table has at least one marked entry.*

The connectors $\mathbf{1}$ and $\overline{\mathbf{1}}$ are fundamental because they allow to complement the value of an entry, i.e. to represent negation: see the table of the term $\mathbf{neg} = (id \otimes (\mathbf{1}; \nabla)); ((\Delta; !) \otimes id) \colon 1 \to 1$ in Figure 4.9.

Theorem 16 (Completeness) *For each abstract semantics, i.e. for each table that has at least one marked entry, there is a term* α *implementing it.*

4.3.1 Axioms

Several axioms for connectors have been proposed, studied and applied in the literature, see e.g. [CG99, BGM02, BM02]. Axioms over connectors are usually aimed to characterize a suitable category of *links between objects* as the equational term algebra freely generated from a restricted set of basic connectors, like those in Figure 4.7. Usually the axioms have just to consider the few possible ways in which two or three of the basic connectors can be composed together. However, the algebra used for encoding CommUnity diagrams is very rich and therefore a few more complex patterns need to be considered. The overall aim is to axiomatize tile bisimilar connectors, in such a way that the axioms can be used to represent any connector in normal form, with equivalence classes in bijective correspondence with the class of tick-tables.

The consistency of all the axioms we are going to present w.r.t. the abstract semantics can be easily shown by just looking at the tables associated to each term. This also means that all axioms equate bisimilar connectors.

We also show some inequalities, when axioms of that form hold for similar algebras of connectors considered in the literature but not in our case. Again, it is easy to see that the same distinctions arise when considering the algebra of tick-tables.

The axioms are presented in five different blocks, corresponding to three classes already presented in the literature, one block for novel axioms for normalization, and the last block for axioms related to the somewhat spurious connectors $\mathbf{1}$ and $\overline{\mathbf{1}}$.

All structures that we are going to present are based on the symmetric monoidal structure given by symmetries γ, tensor product $\otimes$ and unit id_0, for which the ordinary coherence and naturality axioms hold.

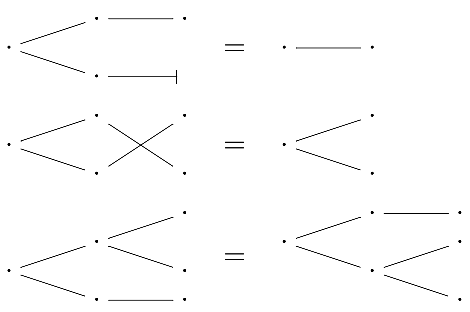


Figure 4.10: Gs-monoidal axioms for ∇ and $!$.

Gs-monoidal structures.

The axioms of gs-monoidal categories have been proposed in [CG99]. We can select four different gs-monoidal structures (two of them are cogs-monoidal, in the sense that, in their case, the gs-monoidal structure is opposite to the direction of composition).

Gs-monoidal axioms for ∇ and !. There are four (schemes of) axioms expressing connectors defined for structured objects in terms of connectors defined for smaller objects:

$$\begin{array}{ll} \nabla_{n+m} = (\nabla_n \otimes \nabla_m); (id_n \otimes \gamma_{n,m} \otimes id_m) & \nabla_0 = id_0 \\ !_{n+m} = !_n \otimes !_m & !_0 = id_0 \end{array}$$

Other three axioms express the "associativity", "commutativity" and "unit" for the duplicator (with discharger as "unit"). The quoted terminology can be straightforwardly understood by looking at the graphical representaton of the axioms in Figure 4.10.

$$\nabla; (id \otimes !) = id \qquad \nabla; \gamma = \nabla \qquad \nabla; (\nabla \otimes id) = \nabla; (id \otimes \nabla)$$

Gs-monoidal axioms for $\odot\!\!\!\!\nabla$ and 0. Omitted, they are analogous to the ones above (just replace ∇ by $\odot\!\!\!\!\nabla$ and ! by **0** in all the equations).

Cogs-monoidal axioms for Δ and ¡. A cogs-monoidal structure is just a gs-monoidal structure in the opposite category. Therefore the axioms are obtained by reversing the order of composition. For instance, the coherence axioms become:

$$(id \otimes ¡); \Delta = id \qquad \gamma; \Delta = \Delta \qquad (\Delta \otimes id); \Delta = (id \otimes \Delta); \Delta$$

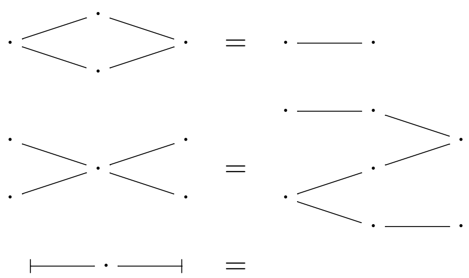


Figure 4.11: P-monoidal axioms.

Cogs-monoidal axioms for $\dot{\Delta}$ and $\overline{\mathbf{0}}$. Omitted, they are analogous to the ones above (just replace Δ by $\dot{\Delta}$ and $¡$ by $\overline{\mathbf{0}}$ in all the equations).

P-monoidal structure.

The axioms of p-monoidal categories have been proposed in [BGM02], where the free algebra of p-monoidal connectors has been shown to model partition relations between source and target objects. The p-monoidal axioms express a particular way to relate a gs-monoidal structure and a cogs-monoidal structure. In our case, the p-monoidal structure is given by $(\nabla, !)$ and $(\Delta, ¡)$.

P-monoidal axioms for ∇, Δ, ! and ¡. There are three p-monoidal axioms (see also Figure 4.11):

$$\nabla; \Delta = id \qquad \Delta; \nabla = (id \otimes \nabla); (\Delta \otimes id) \qquad ¡; ! = id_0$$

The leftmost axiom essentially says that the multiplicity of connections between two objects is not important (but just the fact that they are connected or not). The central axiom says that the path connecting two object is not important (but just the fact that they are connected or not). The third axiom garbage collects isolated empty partitions.

Other axioms that can be easily derived from those above are e.g. (see also Figure 4.12):

$$\Delta; \nabla = (\nabla \otimes id); (id \otimes \Delta) \qquad \Delta; \nabla = (\nabla \otimes \nabla); (id \otimes \gamma \otimes id); (\Delta \otimes \Delta)$$

The first is just a symmetric variant of the second p-monoidal axiom. The second is a weaker axiom that hold e.g. for r-monoidal structures discussed below.

Note that in the initial model the two inequalities below hold (see Figure 4.13):

$$\Delta; ! \neq ! \otimes ! \qquad ¡; \nabla \neq ¡ \otimes ¡$$

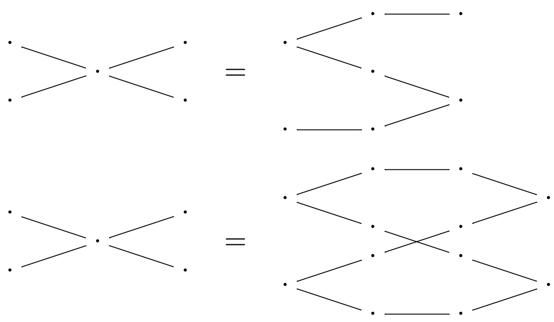


Figure 4.12: Derivable p-monoidal equalities.

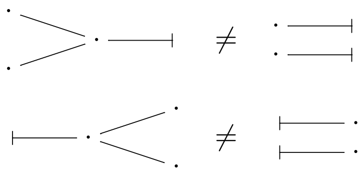


Figure 4.13: Some p-monoidal inequalities.

R-monoidal structure.

The axioms of r-monoidal categories have been proposed in [BGM02], where the free algebra of r-monoidal connectors has been shown to model relations between source and target objects. Similarly to the p-monoidal case, also the r-monoidal axioms express a particular way to relate a gs-monoidal structure and a cogs-monoidal structure. In our case, the r-monoidal structure is given by $(\boldsymbol{\nabla}, \mathbf{0})$ and $(\boldsymbol{\Delta}, \overline{\mathbf{0}})$.

R-monoidal axioms for $\boldsymbol{\nabla}$, $\boldsymbol{\Delta}$, $\mathbf{0}$ and $\overline{\mathbf{0}}$. There are five r-monoidal axioms (see also Figure 4.14):

$$\boldsymbol{\nabla}; \boldsymbol{\Delta} = id \qquad \boldsymbol{\Delta}; \boldsymbol{\nabla} = (\boldsymbol{\nabla} \otimes \boldsymbol{\nabla}); (id \otimes \gamma \otimes id); (\boldsymbol{\Delta} \otimes \boldsymbol{\Delta})$$

$$\boldsymbol{\Delta}; \mathbf{0} = \mathbf{0} \otimes \mathbf{0} \qquad \overline{\mathbf{0}}; \boldsymbol{\nabla} = \overline{\mathbf{0}} \otimes \overline{\mathbf{0}} \qquad \overline{\mathbf{0}}; \mathbf{0} = id_0$$

It can be readily seen that, in general, p-monoidal and r-monoidal structures are incomparable. In fact, the axiom for $\boldsymbol{\Delta}; \boldsymbol{\nabla}$ is weaker than the analogous axiom for $\Delta; \nabla$: we have that in the initial r-monoidal model the following two inequalities hold (see Figure 4.15):

$$\boldsymbol{\Delta}; \boldsymbol{\nabla} \neq (id \otimes \boldsymbol{\nabla}); (\boldsymbol{\Delta} \otimes id) \qquad \boldsymbol{\Delta}; \boldsymbol{\nabla} \neq (\boldsymbol{\nabla} \otimes id); (id \otimes \boldsymbol{\Delta})$$

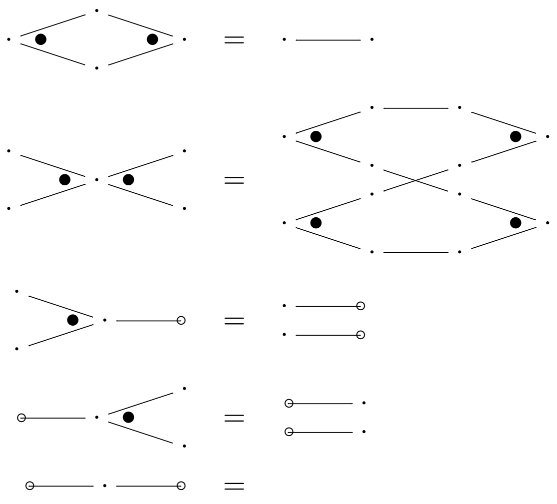


Figure 4.14: R-monoidal axioms.

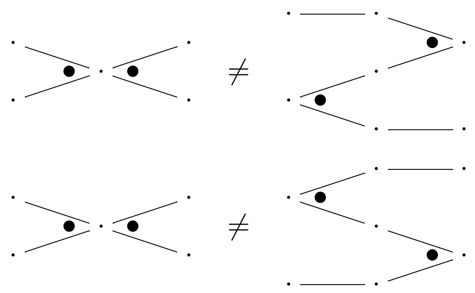


Figure 4.15: Some r-monoidal inequalities.

On the other hand, the r-monoidal axioms for $\dot{\Delta}\,;\mathbf{0}$ and $\overline{\mathbf{0}};\dot{\nabla}$ impose equalities that find no correspondence in p-monoidal categories.

Pr-monoidal structure.

The challenge is here to combine the p-monoidal and the r-monoidal structures presented above, in order to define a complete axiomatization of tick-tables. To this aim, it is convenient to see how e.g. the gs-monoidal structure $(\nabla, !)$ mixes with the cogs-monoidal structure $(\dot{\Delta}, \overline{\mathbf{0}})$, and so on for the other combinations. However, we can find out soon that this is not enough, as more complex patterns involving three or even four mixed structures must be taken into account.

The axiomatization we propose is called *pr-monoidal*. Note that by self-duality of the p-monoidal and r-monoidal structures, all axioms must always come together

$$\nabla ; \dot{\Delta} \quad = \quad \mathbf{0} ; \overline{\mathbf{0}} \tag{4.1}$$

$$\dot{\Delta} ; \nabla \quad = \quad (\nabla \otimes \nabla) ; (id \otimes \gamma \otimes id) ; (\dot{\Delta} \otimes \dot{\Delta}) \tag{4.2}$$

$$\nabla ; (\dot{\nabla} \otimes id) ; (id \otimes \gamma) \quad = \quad \dot{\nabla} ; (\nabla \otimes \nabla) ; (id \otimes \dot{\Delta} \otimes id) \tag{4.3}$$

$$\nabla ; (id \otimes \mathbf{0}) \quad = \quad \mathbf{0} ; \overline{\mathbf{0}} \tag{4.4}$$

$$\Delta ; \mathbf{0} \quad = \quad \mathbf{0} \otimes \mathbf{0} \tag{4.5}$$

$$\dot{\nabla} ; (! \otimes !) \quad = \quad ! \tag{4.6}$$

$$\text{¡} ; \dot{\nabla} ; (id \otimes !) \quad = \quad \text{¡} \tag{4.7}$$

$$\dot{\nabla} ; (id \otimes !) ; (id \otimes \text{¡}) ; \dot{\Delta} \quad = \quad ! ; \text{¡} \tag{4.8}$$

$$\overline{\mathbf{0}} ; ! \quad = \quad id_0 \tag{4.9}$$

$$\nabla ; \dot{\nabla} \otimes id ; id \otimes \Delta \quad = \quad \overline{\mathbf{0}} \otimes id \tag{4.10}$$

Figure 4.16: Novel pr-monoidal axioms.

with their opposites, which are here omitted for simplicity.

Novel pr-monoidal axioms. The pr-monoidal axioms are collected in Figure 4.16 (see also their graphical representations in Figure 4.17).

They must be interpreted as a mean to reduce any complex connector to a suitable normal form. A main property of the normal form is that any arrow α can be written as $\alpha_1 ; \alpha_2 ; \alpha_3 ; \alpha_4 ; \beta_4 ; \beta_3 ; \beta_2 ; \beta_1$, where:

- α_1 involves only identities, ¡, $\overline{\mathbf{0}}$;
- α_2 involves only identities and $\dot{\nabla}$;
- α_3 and β_3 involve only identities and γ;
- α_4 involves only identities and Δ;
- β_4 involves only identities and ∇;
- β_2 involves only identities and $\dot{\Delta}$;
- β_1 involves only identities, $!$, $\mathbf{0}$.

The second property is that the number of $!$, $\mathbf{0}$, ¡, and $\overline{\mathbf{0}}$ should be kept minimal.

Axioms for 1 and $\overline{1}$

There is one additional constructor that has been used in the mapping from CommUnity diagrams to tiles, namely $\overline{\mathbf{1}}$, whose task is to enforce a (mutual exclusion) choice to take place and some activity to be performed. In particular, it allows for modeling glues just as suitable connectors (instead of computational entities).

Unfortunately, we have already seen that the presence of $\overline{\mathbf{1}}$ (and of its dual $\mathbf{1}$) can introduce inconsistent connectors (e.g. via negation). However this is not the case for well-behaved configuration diagrams.

Again, we omit all dual axioms for simplicity.

$$
\begin{aligned}
\dot{\nabla}; (\mathbf{1} \otimes id) &= \mathbf{1}; \overline{\mathbf{0}} && (4.11)\\
\nabla; (\mathbf{1} \otimes id) &= \mathbf{1}; \overline{\mathbf{1}} && (4.12)\\
\Delta; \mathbf{1} &= \mathbf{1} \otimes \mathbf{1} && (4.13)\\
\overline{\mathbf{1}}; \mathbf{1} &= id_0 && (4.14)\\
¡; \mathbf{1} &= id_0 && (4.15)
\end{aligned}
$$

For dealing with inconsistent connectors, we assume the existence of an absorbing *error* element $\perp$, with

$$\overline{\mathbf{1}}; \mathbf{0} = \overline{\mathbf{0}}; \mathbf{1} = \perp$$

4.3.2 Normalization

Proposition 6 *We have a tile of the form* $cm[S] \xrightarrow[effect]{trigger} cm[S]$ *where* $cm[S]$ *is a channel manager with* n *actions iff we have a tile of the form* $cm[S] \otimes (\overline{\mathbf{1}}; \dot{\nabla}{}^{n}_{1}); \Delta_n \xrightarrow[effect]{trigger} cm[S] \otimes (\overline{\mathbf{1}}; \dot{\nabla}{}^{n}_{1}); \Delta_n$.

The intuitive meaning of this proposition is that the behaviour of channel managers is sequential, so adding mutual exclusion to the actions does not change the behaviour.

Theorem 17 (Axiomatization of the colimit) *Let* **dia** *be a CommUnity diagram, and let* $Conf$ *be the tile configuration associated to (the standard decomposition of)* **dia**. *Then we can prove that* $Conf$ *is isomorphic up-to the axioms of the algebra of connectors to the tile configuration associated to (the standard decomposition of) the colimit of* **dia**.

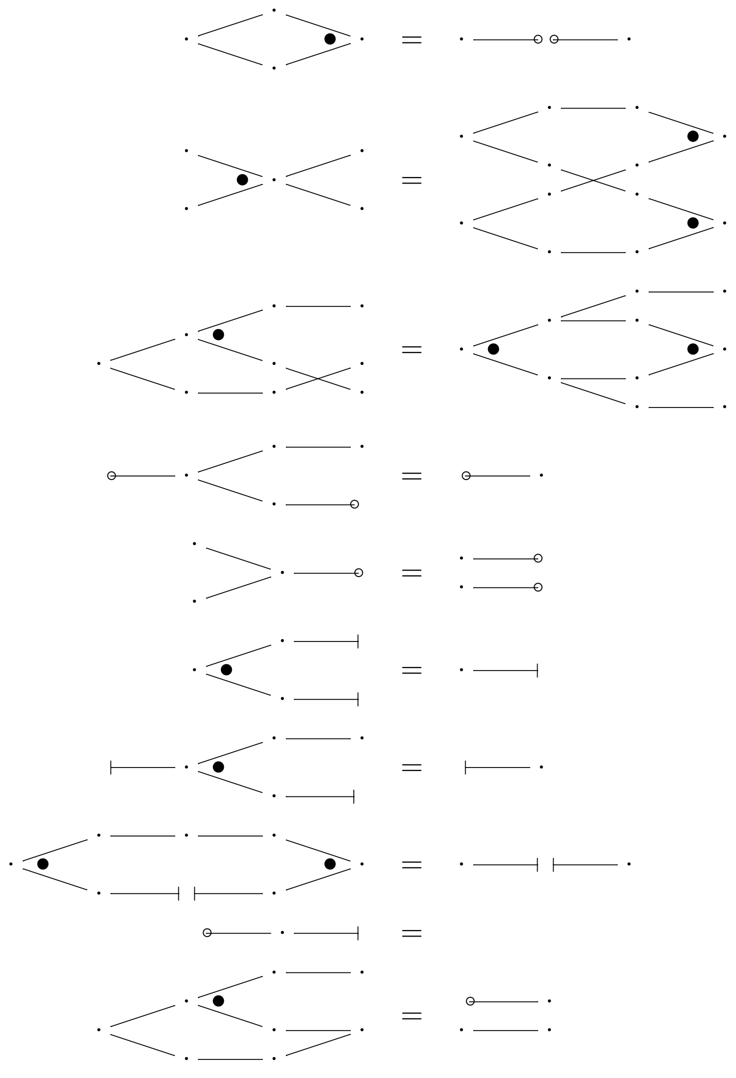


Figure 4.17: Novel pr-monoidal axioms.

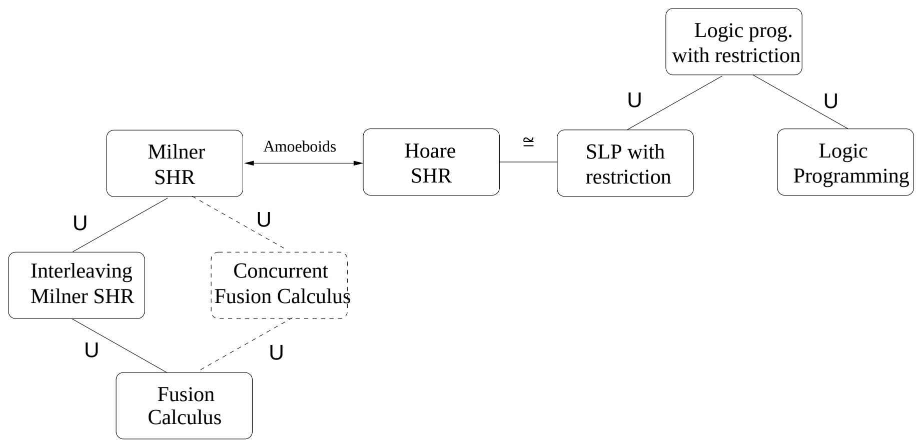


Figure 5.1: Relations between models.

Chapter 5

Final remarks and future work

In general we aim at a better understanding of synchronization strategies, in particular in relation to other features of interest in GC systems, such as, e.g. , concurrency, mobility, resource-handling and visibility, fusions,...

Until now, our main results have been the comparisons between the operational behaviour of Fusion Calculus, SHR and logic programming. A final view of these results is shown in Figure 5.1.

We have represented concurrent Fusion Calculus (but only dashed) since some hints on it can be retrieved using the mapping from Fusion Calculus into Milner SHR (see Example 4). We think that in this framework, an important result has been the development of logic programming with restriction, since it presents good features of both the process-calculi and the logic programming approach. Note that in figure we mainly have relations of "submodel" (denoted with $\cup$), and an "isomorphism" relation between HSHR and SLP with restriction. Finally, between Hoare and Milner SHR we have two implementations based on the concept of amoeboid (see Definition 30).

Disconnected from this figure, since driven by a different aim, that is comparing strategies for composing systems, we have the mapping from CommUnity into the Tile Model.

In the following sections we present the general idea that will drive our future work, followed by some more precise items to be developed in the future.

5.1 An equational approach to coordination

As we have already stated, an important aim when modeling GC systems, is the choice of the right level of abstraction. In fact, while it is true that in some steps of the development process one has to deal with low-level features such as asynchronous communications, faults, communication protocols and others, in other steps, mainly at an early stage of development, one wants to concentrate on the overall structure of the system and on behaviours involving a large number of components. This is clearly a complex problem.

We want to face the problem using an approach suggested by the following physical analogy. In physics, one is usually interested in systems where multiple objects interact, for instance fluid molecules in a tube or antennas transmitting in the sky. One can use physical equations to analyze all the interactions between the components, but usually this approach is by far too complex to be carried on. Thus one considers small parts of the space and describes each of them by using partial derivative equations. When suitable boundary conditions are added, one gets (if he is able to solve the system of equations) a solution that describes the possible evolutions of the system.

We want to apply a similar approach to GC systems. Thus, instead of directly considering all the interactions between each pair of components of our system, we want to have some *equations* describing, for each "part of the space", which are the *constraints*. Components' descriptions give the boundary conditions.

Such an approach is partly realized by SHR and tile logic. In the former, nodes are characterized by equations defining the possible synchronizations, for instance, in the case of Hoare synchronization, all actions must be equal. The solution of this system of equations, with the boundary conditions corresponding to the set of productions associated to each edge, gives the possible steps. Similarly, in the Tile Model, the equations state that each trigger must equate the effects of the subsystems connected to it.

It is clear that all the equations described so far are very simple. We can continue with our physical analogy, applying it on Hoare and Milner synchronizations: Hoare synchronization reminds electric power, since all the connected components must "agree" on its value. Similarly, Milner synchronization has some analogies with electric current, since it is directional and "ingoing actions" must be in correspondence with "outgoing actions", that is coactions. This suggests that some kind of duality between the two forms of interactions may exist, and also that these two synchronization models are quite basic. Thus one of our aims will be to look for more complex forms of synchronization.

5.2 Synchronization algebras for mobility

If many useful synchronization strategies can be found, the approach of *synchronization algebras* [Win86], which gives a classification of different synchronization models, will be useful. However, it must be extended in order to deal with GC systems. To be precise, we want to build an algebra of synchronizations with mobility and name generation and extrusion (we think that the approach of name mobility can still be carried on, by analogy with what happens in π-calculus). This algebra must be able to express at least Hoare, Milner and broadcast (this is a model where an action that models a broadcast is synchronized with a tuple of coactions, one for each receiver) synchronizations since they can be expressed (even if without mobility) by the original proposal. A more challenging aim from the mobility point of view is to handle multiple Milner synchronization, where different Milner synchronizations happen at the same time and in the same place, between pairs of different components. This is a more difficult case since names sent by different components must be handled differently. Note also that this is closer to electric current in the physical analogy, since in electric circuits one may have many ingoing and outgoing currents. More complex synchronization strategies to be developed for GC systems may also be included in the synchronization algebra model.

5.3 Observational and abstract semantics

When studying synchronization, we must consider not only the operational aspect of systems, as done so far, for instance in the case of Fusion Calculus and SHR, but also the *observational and abstract semantics.* This is important since one usually wants to abstract from differences that are not visible from the outside, thus considering as equivalent those systems which have the same set of behaviours w.r.t. some methods of observing them (traces, bisimulation, ...). To this aim, we think that *tile logic* is a good framework to work in, since it is particularly well-suited for analyzing observational properties of open systems. In particular, in literature *conditions* exist that guarentee that bisimilarity is a congruence, that is the system is *compositional.* This kind of criteria for compositionality proofs is quite useful since it allows to prove compositionality results by considering only a finite number of cases, i.e. the basic tiles, instead of infinitely many, i.e. all possible systems and behaviours. However, conditions found in literature require to have *only unary operators* in the vertical signature, which is too restrictive when one wants to model synchronizations that depend on multiple components (that is on multiple objects in the interfaces) or wants observable fusions. We think that these conditions can be weakened by considering a different axiomatization for the horizontal structure of the system, which in particular relies *only on the horizontal composition* and not on the monoidal one. After that we just need to model systems of interest using such an axiomatization for configurations.

5.4 Exploring "hybrid" models

Another aim is to exploit the work done so far in comparing models in order to export synchronization strategies, and also other interesting features of models such as name-handling, from one model to the other. This will both extend the understanding of the relationships between different features of the same model and suggest some new *"hybrid" models*. In fact, we can derive a new model by taking an existing one and plugging into it a feature from another model. It will be interesting to analyze the expressive power and the properties of these new models, both in general and in the particular context of GC systems.

We think that Figure 5.1 can be made symmetric, with one half that uses Hoare synchronization and the other one that uses Milner synchronization. Furthermore analogous schemes can be possibly obtained for other synchronization models.

We think thus that the following "hybrid" models can be analyzed:

- concurrent Fusion Calculus: as already said, this can be obtained from SHR, but the interesting point is the comparison between this kind of concurrent semantics and the semantics obtained using standard approaches such as shift equivalence [EKMR99]. We think this is interesting in order to understand the relationships between concurrency, synchronizations and fusions;

- Fusion Calculus with non-Milner synchronization: this can be easily obtained since SHR is already (to some extent) parametric w.r.t. the synchronization model, and Fusion Calculus can be seen as a subcalculus of it. However, the fact that Hoare synchronization is symmetric suggests that Hoare Fusion Calculus is not expressive enough. Anyway this has not been yet analyzed in detail, and Hoare synchronization is not the only other possible synchronization mechanism;

- logic programming with "non-Hoare unification": this possibility is worth some analysis, but it is not so clear how synchronization models from SHR can be extended to a complete unification algorithm, able to deal not only with synchronizations, i.e. functional symbols, but also with general terms. If such a model can be developed, then its expressiveness, its properties and its relations with preexisting models have to be analyzed;

- logic programming with restriction: we think that this is a very promising field, since logic programming is a good framework both from the theoretical and the implementative point of view, but it requires to deal with observations in order to become applicable in the GC field. The first results are interesting since they allow to map into logic programming the restriction operator of SHR, but the properties of restriction, when general logic computations are performed, have not yet been fully analyzed.

5.5 Models with general or multiple synchronization strategies

A further aim consists in applying the synchronization algebras approach, extended with mobility and name-handling, to existing formalisms. In this way, we will create a model which is parametric w.r.t. the choice of the synchronization algebra. We think that, in order to start this kind of work, SHR would be a good model, since it already exists in different flavours (such as Milner and Hoare), but until now they have been tackled separately, instead of being considered as different instances of a more general model. From an expressiveness point of view, it would be interesting to develop a general way to define amoeboids, given the synchronization algebras of the source and of the target model. In fact, such a result would allow to separate the choice of the synchronization algebra used when modeling the system from the choice of the synchronization algebra used when analyzing it (for instance used in the implementation of a tool) since a standard translation would be available.

A further step is the analysis of models where different synchronization strategies coexist. In this case too, we think that SHR is a good framework for preliminary analysis. A possible approach is to have nodes typed with their synchronization algebra. At least two issues have still to be analyzed:

1. the interactions between different types of nodes, that is what happens when nodes of different types are merged;

2. the creation of new nodes, that is who decides the type of a newly created node.

5.6 A last word

We think that our work can, at least, clarify the relationships between existing models and which aspects of the models are more suitable for modeling GC systems. Anyway our hope is that the investigation would allow to develop some new useful models, that is models devised for the analysis of huge distributed systems at an high level of abstraction.

Bibliography

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[AHS90] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley, 1990.

[BGM02] R. Bruni, F. Gadducci, and U. Montanari. Normal forms for algebras of connections. *Theoretical Computer Science*, 286(2):247–292, 2002.

[BM00] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2):46–89, 2000.

[BM02] R. Bruni and U. Montanari. Dynamic connectors for concurrency. *Theoretical Computer Science*, 281(1-2):131–176, 2002.

[BMR01] R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(6):647–690, 2001.

[CDM85] A. Corradini, P. Degano, and U. Montanari. Specifying highly concurrent data structure manipulation. In G. Bucci and G. Valle, editors, *COMPUTING 85: A Broad Perspective of Current Developments*. Elsevier Science, 1985.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, Berlin, Germany, 1998.

[CG99] A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7:299–331, 1999.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design - A Foundation*. Addison-Wesley, 1988.

[de 85] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37(3):245–267, December 1985.

[DFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *Software Engineering*, 24(5):315–330, 1998.

[DM87] P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *Journal of the ACM (JACM)*, 34(2):411–449, 1987.

[EHKPP91] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *Proceedings IEEE Conference on Automata and Switching Theory*, pages 167–180, 1973.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[FLW03] J. L. Fiadeiro, A. Lopez, and M. Wermelinger. A mathematical semantics for architectural connectors. In R. Backhouse and J. Gibbons, editors, *Generic Programming*, number 2793 in LNCS, pages 190–234. Springer-Verlag, 2003.

[FM97] J. L. Fiadeiro and T. S. E. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2-3):111–138, 1997.

[FMT01] G. Ferrari, U. Montanari, and E. Tuosto. A LTS semantics of ambients via graph synchronization with mobility. In S. Ronchi Della Rocca A. Restivo and L. Roversi, editors, *Proc. of ICTCS'01*, volume 2202 of *LNCS*, pages 1–16. Springer, October 2001.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[GM00] F. Gadducci and U. Montanari. The tile model. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[Gog73] J. Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research, Transcripta Books*, pages 121–130. 1973.

[GS89] H. Gaifman and E. Y. Shapiro. Fully abstract compositional semantics for logic programs. In *Proceedings of POPL'89*, pages 134–142. ACM, 1989.

[GW00] P. Gardner and L. Wischik. Explicit fusions. In *Mathematical Foundations of Computer Science*, pages 373–382, 2000.

[HIM00] D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In *Proceedings of the 4th International Conference on Coordination Languages and Models, Coordination 2000*, volume 1906 of *LNCS*, February 2000.

[HM01] D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In Springer Verlag, editor, *Proc. of CONCUR'01*, LNCS, 2001.

[Hoa85] C.A.R. Hoare. *CSP – Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, Nj, 1985.

[Lan02] I. Lanese. Process synchronization in distributed systems via Horn clauses. Master's thesis, University of Pisa, Computer Science Department, 2002. Downloadable from `http://www.di.unipi.it/~lanese/tesi.ps`.

[Law63] F.W. Lawvere. Functorial semantics of algebraic theories. *Proc. National Academy of Sciences*, 50:869–872, 1963.

[Llo93] J. W. Lloyd. *Foundations of Logic Programming, Second Extended Edition*. Springer-Verlag, 1993.

[LM02] I. Lanese and U. Montanari. Software architectures, global computing and graph transformation via logic programming. In L. Ribeiro, editor, *Proc SBES'2002 - 16th Brazilian Symposium on Software Engineering*, pages 11–35. Anais, 2002.

[LM03] I. Lanese and U. Montanari. A graphical fusion calculus. In *Proceedings of CoMeta Final Workshop*, Electronic Notes in Theoretical Computer Science, 2003. To appear.

[LM04] I. Lanese and U. Montanari. Mapping fusion and synchronized hyperedge replacement into logic programming. *Theory and Practice of Logic Programming, Special Issue on Multiparadigm Languages and Constraint Programming*, 2004. Submitted.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1&2):181–224, 1993.

[LX90] K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M. S. Paterson, editor, *Proceedings ICALP '90*, volume 443 of *LNCS*, pages 526–539, Warwick U., July 1990. Springer-Verlag.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mil89] R. Milner. A calculus of communicating systems. In *Lect. Notes in Comp. Sci.*, volume 92. Springer, Berlin, 1989.

[MM90] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, October 1990.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inform. and Comput.*, 100:1–77, 1992.

[MZ03] M. Merro and F. Zappa Nardelli. Bisimulation proof techniques for mobile ambients. In *Proceedings of the 30th "International Colloquium on Automata, Languages, and Programming" (ICALP 2003)*, volume 2719 of *LNCS*, pages 584–598. Springer, 2003.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PV98] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS '98*. IEEE, Computer Society Press, June 1998.

[Rei85] W. Reisig. *Petri Nets*. Springer-Verlag, 1985.

[Roz97] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, vol. 1: Foundations*. World Scientific, 1997.

[Rut00] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[San01] D. Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253:311–350, 2001.

[Vic98] B. Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, June 1998.

[Win86] G. Winskel. Event structures. In G. Goos and J. Hartmanis, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer Verlag, 1986.