University of Pisa
Faculty of Mathematical,Physical and Natural Sciences
Degree course in Computer Science
A. Y. 2001/02

Degree Thesis

Process synchronization in distributed systems via Horn clauses

Candidate
Ivan LANESE

Relator
Prof. Ugo MONTANARI

Contents

Abstract

We analyze the connections between logic programming and network models.
In particular, we extend logic programming with a restriction operator, we provide two styles of abstract semantics for this extension in the framework of tile logic and we show tile bisimilarity to be a congruence for them.
Furthermore, we show that synchronized logic programming with restriction, a subset of logic programming with restriction, is equivalent to synchronized hyperedge replacement with Hoare synchronization and we present a meta-interpreter for it.
Finally, we present Logic Process Calculus, a process calculus that behaves like logic programming with restriction and we prove the equivalence between the two formalisms.

Chapter 1

Introduction

In the last years, the importance of distributed systems has greatly increased. The first reason is the impressive development of Internet, which has today millions of users all over the world and which is fundamental for enterprises, universities, governments and single users. Moreover also local area networks are extensively employed in a lot of contexts and they support the information systems of many organizations.
Networks are a very complex and various world: in order to manage and administer a network many aspects have to be controlled: physical aspects (hardware and communication links, ...), protocol aspects (routing, safeness of communication, congestion control, ...), administrative aspects (name and address assignments,...), security aspects (access control, authentication, secrecy, ...), network aware programming (coordination, dynamic evolution and reconfiguration strategies, ...), etc.
To deal with such complex problems, a lot of models have been developed, to allow one to analyze only some aspects of a network at a time, without having to deal with all the problems of real cases. We are interested especially in models for network aware programming. They usually take the form of distributed process calculi (π-calculus [15], Ambient calculus [5], ...), of specialized program logics (e. g. Mobile Unity [17], ...) or of Linda-like coordination languages (e. g. KLAIM [16], ...).
We will concentrate on graph rewriting, an approach to this problem which utilizes hypergraphs to model networks. A hypergraph is composed by hyperedges, which represent machines or processes in the network, and nodes, which stand for ports of communication or for subnetworks.
A fundamental aspect is network reconfiguration. There are two possible approaches to this problem: a centralized one which uses rules that match a subgraph of the graph and which substitute it with another subgraph; and a distributed one where the components of the network synchronize in order to change network topology. The latter, called synchronized hyperedge replacement (SHR), allows a rule to rewrite only a single edge (local reconfiguration) but it combines many rules in a global

rewriting by imposing synchronization constraints on common nodes. We have chosen the second approach because in the world of networks (and in particular in the Internet case) centralized implementations are not realistic.
The synchronization can be done in the Hoare or Milner style. In the Hoare style all the edges which share a node must do the same action on the node and the action is observable from the outside (unless the node is restricted). In Milner synchronization we have three kinds of actions: true actions, coactions (one for each action) and internal actions. On a node we may have an action of any kind from an edge and no other actions, or a pair action-coaction from two different edges which is seen as an internal action from the outside. We have chosen the Hoare synchronization because it is apter to our purposes.
Synchronized hyperedge replacement allows also to model mobility, which is very important in order to model real world aspects such as mobile computers or telephones, or mobile applications such as Java applets. We restrict ourselves to name mobility, where a name (modeled by a node) can be transmitted from a place to another. This simplifies the mathematical aspects of the model and this is powerful enough to model also other forms of mobility.
Another important aspect is the concept of hiding: not all the pieces of information can be seen from everywhere, and not every name is meaningful everywhere. In SHR we have an hiding operator that defines the scope of names in order to model local information.
Graph rewriting is an useful approach because the modelling step is simple and suggestive, it can be used to develop graphical tools that are simple to use, yet it is powerful enough to model a lot of interesting aspects of network synchronization.
Until now, three SHR systems have been studied. They differ in the restrictions that they impose on node fusions: the first allows only communication and merging of new nodes (only for Hoare synchronization, in [10]), the second allows also merges between new and old nodes ([9]) and the third allows all kinds of fusions (only for Milner synchronization, in [7]).
Our main contribution is to analyze the relations between SHR and logic programming. This is interesting because at a first glance they are quite different models, but in fact they are analogous for many aspects, so finding relations between them allows many new insights in both models. Moreover a lot of work has been done on logic programming, so it is useful to exploit this work in the world of graph rewriting. In fact not only the properties of logic programming have been studied using a lot of different approaches, but also many implementations of logic programming languages have been created, together with tools to help the development of programs.
We do not focus on the more traditional view of logic programming for search or theorem proving, but on an interactive view ([4]) of logic programming seen as a

goal rewriting system. The idea is to understand logical predicates as (possibly open) interacting agents whose local evolution is coordinated by the unification engine. It is useful to have unification within a process calculus because a problem of many process calculi semantics is that they require to analyze the behaviour of the systems in all the possible contexts, which are infinite. Unification allows to find a single substitution (a notion dual to context) which is as general as possible, what makes the approach operationally satisfactory (see [3] for some applications of this approach).
We will show that the fundamental mechanism of computation of SHR is essentially the same of logic programming.
However between the two formalisms there are some differences. Thus mapping SHR to logic programming requires to extend logic programming at some extent.
We will present two variations of it:

Synchronized logic programming Logic programming is a very nondeterministic programming paradigm and it allows for many execution strategies. We need to restrict the possible computations in order to preserve only those meaningful for SHR. Synchronized logic programming is the result of this limitation.

Logic programming with restriction This is an extension of standard logic programming with an operator of name restriction, which has the same meaning of the restriction operator of many process calculi (CCS [14], π-calculus [15], ...) and of SHR, i. e. it hides variables. Parts of transitions involving only substitutions on hidden variables are not observable from the outside. This extension is important because it fills a gap between logic programming and network models, but it is also very natural, because also in traditional logic programming some bindings are not seen by the user. What we have done is to give an explicit and uniform syntax and semantics to something that already existed.

Another important aspect is that in logic programming one is usually interested only in completed computations (refutations) while we are intersted mainly in partial computations, because in SHR refutations would be computations that end with an empty graph, i. e. an empty network.
We will map to logic programming with restriction: 1) SHR with fusions only among new nodes; and 2) SHR with all kinds of merges. We have not provided a mapping for the intermediate SHR style because it requires some additional changes to logic programming, which are not natural, whereas the mappings of the first two styles are more straightforward.
We have used these mappings to produce further interesting results:

Meta-interpreter We have used the property of the logic programming languages of being easily meta-interpreted to build a meta-interpreter for synchronized logic programming with restriction, which is essentially a simulator of an SHR system. This is an important improvement, since no implementations of SHR are available by now.

Bisimilarity semantics We have exploited the study of logic programming in the framework of tile logic ([4]) to equip synchronized logic programming with restriction with the same semantic model (in two different styles), which is a bisimilarity semantics (a typical semantic style for process calculi). This is a suggestive semantics because it is founded on the principle that two systems are equivalent if, seen from the outside, in the same conditions they can do the same actions.

Bisimilarity is a congruence We have shown that bisimilarity is a congruence for synchronized logic programming with restriction. This is important because it means that the abstract semantics is compositional, i. e. two equivalent parts of a system can be substituted to each other without changing the behaviour of the whole system. So it is possible to study the properties of small parts of the system in isolation and to have the results to be composed to determine the properties of the system.

Logic Process Calculus As we have discovered strong analogies between logic programming with restriction and process calculi, we have created a new process calculus, the Logic Process Calculus (LPC), with the same set of behaviours as logic programming with restriction. This allows to see logic programming with restriction as a process calculus.

In chapter 2 we present an introduction to the notions and the formalisms we exploit in our work. In particular, we recall some concepts of first order logic up to define Horn clauses and SLD-resolution (i. e the basis of logic programming), we define SHR systems and we introduce tile logic (together with the required concepts of category theory). Finally, we present Prolog, the most famous logic language, and the one we have used to write our meta-interpreter.
In chapter 3 we define synchronized logic programming and logic programming with restriction, we present the tile semantics of logic programming with restriction and we show tile bisimilarity to be a congruence.
Chapter 4 is the fundamental chapter that contains the translations from SHR to logic programming with restriction, together with the proofs of correctness and completeness of the translations.
Finally, in chapter 5 and chapter 6 we present respectively the meta-interpreter for synchronized logic programming with restriction and the Logic Process Calculus.

Notation

Sets and vectors

$y \in X$		Set membership (true iff y is in X)
$X \subseteq Y$		Subset ($\forall x. x \in X \Rightarrow x \in Y$)
$\emptyset$		The empty set
$\{x_1, \ldots, x_n\}$		Set containing these elements
$X \cup Y$	$\bigcup_i X_i$	Set union
$X \uplus Y$		Disjoint set union
$X \cap Y$	$\bigcap_i X_i$	Set intersection
$X \setminus Y$		Set difference($\{x \in X \mid x \notin Y\}$)
$\vec{y}$		Vector
$\mid - \mid$		Number of elements of a vector or of a set
$\mathsf{Set}(\vec{y})$		Set of elements of a vector
$\langle\rangle$		The empty sequence

Logic

$x \wedge y$	Conjunction (x and y)
$x \vee y$	Disjunction (x or y)
$\neg x$	Negation (not x)
$x \rightarrow y$	Implication ($\neg x \vee y$) in logic formulas
$x \Rightarrow y$	Implication ($\neg x \vee y$) in proofs
$x \leftrightarrow y$	Double implication ($(x \rightarrow y) \wedge (y \rightarrow x)$) in logic formulas
$x \Leftrightarrow y$	Double implication ($(x \Rightarrow y) \wedge (y \Rightarrow x)$) in proofs
$\forall x. \chi$	Universal quantification (χ holds for all x)
$\exists x. \chi$	Existential quantification (χ holds for at least one x)

Functions

$\mathrm{Dom}(f)$	Domain of f
$\mathrm{Im}(f)$	Image of f
$f^{-1}(x)$	Counterimage of x according to f
$f(x)\uparrow$	f diverges in x
$f(x)\downarrow$	f converges in x
$f\vert_S$	Restriction of f to set S
$f\vert_{\setminus x}$	Restriction of f to the set of elements different from x
$f\vert_{\setminus X}$	Restriction of f to the set of elements not in X

Substitutions

$T\theta$	Instance of T by θ
$\theta_1\theta_2$	Composition of two substitutions
ϵ	The empty substitution
$\mathrm{eqn}(\theta)$	Set of equations associated to θ
$\mathrm{mgu}(S)$	Most general unifier of the set of equations S
$\theta^{-1}(x)$	Set of the variables a term containing x is assigned to

Categories

$f;g$	Sequential composition
id_a	Identity of a
$a\otimes b$	Tensor product
$\mathrm{dom}(f)$	Domain of f
$\mathrm{cod}(f)$	Codomain of f
$!_a$	Discharger of a
∇_a	Duplicator of a
$\gamma_{a,b}$	Permutation of a and b

Chapter 2

Background

In this chapter we present the formal tools that are needed to understand our work.
Section 2.1 starts introducing first order logic and then it defines logic programming syntax and operational semantics.
Sections 2.2 and 2.3 define graphs representation as syntactic judgements and SHR systems.
Section 2.5 introduces tile logic as double categories, thus in section 2.4 we present the required category theory definitions.
In section 2.6 we present the work of [4] on logic programming in the framework of tile logic.
Finally, section 2.7 contains an overview on Prolog, the most used logic language, and some examples of meta-interpretation.

2.1 Horn clauses, unifiers and SLD-resolution

In this section we will introduce logic programming notations and definitions useful for our purpose of expressing process synchronization.
We will use first order logic as our framework. Then an alphabet consisting of seven classes of symbols is to be used:

1. variables (referenced with x, y,...)
2. constants (referenced with a, b,...)
3. function symbols of various arities > 0 (referenced with f, g,...)
4. predicate symbols of various arities ≥ 0 (referenced with p, q,...)
5. connectives ($\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$)
6. quantifiers ($\exists$, $\forall$)

7. punctuation symbols ('(' , ')' , ',' , '.')

We will adopt the following hierarchy of precedence, where higher symbols have higher precedence:

$$\neg$$
$$\wedge, \vee$$
$$\rightarrow, \leftrightarrow$$
$$\exists, \forall$$

Definition 2.1 (Term) *A term is inductively defined as follows:*

- *a variable is a term*
- *a constant is a term*
- *if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term*

Definition 2.2 (Well-formed formula) *A (well-formed) formula is inductively defined as follows:*

- *if p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula (called atom, we will denote it by A, B,...)*
- *if F and G are formulas, then so are (F), $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ and $F \leftrightarrow G$*
- *if F is a formula and x is a variable, then $\forall x.F$ and $\exists x.F$ are formulas*

It will often be convenient to write $F \rightarrow G$ as $G \leftarrow F$.

Definition 2.3 *The scope of $\forall x$ (resp. $\exists x$) in $\forall x.F$ (resp. $\exists x.F$) is F. A bound occurrence of a variable in a formula is an occurrence which immediately follows a quantifier or an occurrence within the scope of a quantifier of the same variable which is immediately after the quantifier. Any other occurrence of a variable is free. A closed formula is a formula with no free occurrences of any variable.*

Other binders we will introduce are analogous to logic quantifiers, so the same definitions of free and bound variable and of closed formula will be used.

Definition 2.4 *If F is a formula, then $\forall F$ (resp. $\exists F$) denotes the universal (resp. existential) closure of F, which is a closed formula obtained by adding a universal (resp. existential) quantifier for every variable having a free occurrence in F.*

Definition 2.5 (Literal) *A literal is an atom or the negation of an atom. A positive literal is an atom. A negative literal is the negation of an atom.*

Definition 2.6 (Clause) *A clause is a formula of the form:*

$$\forall\ L_1 \vee \cdots \vee L_m$$

where each L_i is a literal.

Definition 2.7 (Variant) *A clause C_1 is a variant of a clause C_2 if it is identical to C_2 up to variables renaming.*

Notice that two variants have the same logical meaning because all variables are universally quantified and so they can be safely renamed.
Because clauses are so common in logic programming, it will be convenient to adopt a special clausal notation. Observe that a clause:

$$\forall\ A_1 \vee \cdots \vee A_k \vee \neg B_1 \vee \cdots \vee \neg B_n$$

is equivalent to:

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

We will write this clause as:

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

Definition 2.8 (Definite program clause)
A (definite) program clause 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".

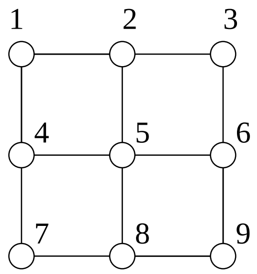
Definition 2.9 (Definite program) *A definite program is a finite set of definite program clauses.*

Definition 2.10 (Definite goal) *A (definite) goal is a clause of the form:*

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

Each B_i is called a subgoal of the goal.

Figure 2.1: Graph



Definition 2.11 (Horn clause) *A Horn clause is a clause which is either a definite program clause or a definite goal.*

Example 2.1 (Path in a graph) *Let G be the graph in figure 2.1.*
We can describe paths in this graph using the following program:
$arc(1,2) \leftarrow, arc(2,3) \leftarrow, \ldots, arc(8,9) \leftarrow$
$arc(1,4) \leftarrow, arc(4,7) \leftarrow, \ldots, arc(6,9) \leftarrow$
$arc(x,y) \leftarrow arc(y,x)$
$path(1, nil)$
$path(x, cons(y,l)) \leftarrow arc(x,y), path(y,l)$
where x, y, l are variables, $1, 2, \ldots, 9, nil$ are constants and cons is a binary function symbol. A goal for this program could be:
$\leftarrow path(5,l)$

In order to compute results using Horn clauses the concept of unification has to be introduced.

Definition 2.12 (Expression) *An expression is any syntactical object made by variables, constants and other symbols.*
For each expression E, $\mathrm{var}(E)$ is the set of variables appearing in E.
For each expression E, $\mathrm{fn}(E)$ is the set of free variables (also called free names) appearing in E.

Definition 2.13 (Substitution) *A substitution θ is a finite set of the form $\{t_1/v_1, \ldots, t_n/v_n\}$, where each t_i is a term, v_i are distinct variables and for each i $t_i \neq v_i$. Each element $[t_i/v_i]$ is called a binding for v_i. We define $\mathrm{Dom}(\theta) = \{v_1, \ldots, v_n\}$ and $\mathrm{Im}(\theta) = \{t_1, \ldots, t_n\}$.*

We will denote the empty substitution with ϵ.
Observe that a substitution is a function from variables to terms.

Definition 2.14 (Inverse of a substitution) *Given a substitution θ we define:* $\theta^{-1}(x) = \{y | x \in \mathrm{var}(y\theta)\}$

Definition 2.15 (Renaming) *A renaming is a substitution whose image is a set of variables, i. e. a function from variables to variables.*

Definition 2.16 (Name fusion) *A name fusion θ of a set of variables S is a renaming such that* $\mathrm{Dom}(\theta) = S$ *and* $\mathrm{Im}(\theta) \subseteq S$ *and* $x \in \theta^{-1}(x) \vee \theta^{-1}(x) = \emptyset$.

A name fusion of S is a function from names in S to names in S and it determines a partition of S in equivalence classes whose elements have the same image.

Definition 2.17 (Equation) *An equation is $t_1 = t_2$ where t_1 and t_2 are terms.*

Definition 2.18 *Let $\theta = \{t_1/v_1, \dots, t_n/v_n\}$ be a substitution and E be an expression. Then $E\theta$, the instance of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variables v_i in E by the corresponding terms t_i. A special rule is needed when the expression contains some binders (for example an existential quantifier): bound occurrences of variables must not be substituted, moreover if we substitute a free variable within the scope of a binder of a variable x then x must not be in the image of the substitution. If this happens we can overcome the problem by changing the name of the bound variable (bound variables can always be renamed, what is called α-conversion).*

We can apply a substitution to a set of expressions by applying the same substitution to each expression of the set.

Definition 2.19 (Composition of substitutions)
Let $\theta = \{t_1/v_1, \dots, t_n/v_n\}$ and $\sigma = \{s_1/x_1, \dots, s_m/x_m\}$ be substitutions. Then the composition $\theta\sigma$ of θ and σ is the substitution obtained from the set:

$$\{t_1\sigma/v_1, \dots, t_n\sigma/v_n, s_1/x_1, \dots, s_m/x_m\}$$

by deleting any binding $[t_i\sigma/v_i]$ for which $v_i = t_i\sigma$ and deleting any binding $[s_j/x_j]$ for which $x_j \in \{v_1, \dots, v_n\}$.

Proposition 2.1 *If θ_1, θ_2 and θ_3 are substitutions and E is an expression, we will have:*

- $\theta_1\epsilon = \theta_1 = \epsilon\theta_1$
- $(\theta_1\theta_2)\theta_3 = \theta_1(\theta_2\theta_3)$

- $(E\theta_1)\theta_2 = E(\theta_1\theta_2)$

Definition 2.20 *We say that a substitution θ is idempotent iff $\theta = \theta\theta$.*

Proposition 2.2 *A substitution θ is idempotent iff* $\mathrm{Dom}(\theta) \cap \mathrm{var}(\mathrm{Im}(\theta)) = \emptyset$.

Definition 2.21 (Unifier) *A unifier of a set of equations $S = \{s_1 = t_1, \dots, s_n = t_n\}$ is a substitution θ such that $s_1\theta \equiv t_1\theta, \dots, s_n\theta \equiv t_n\theta$ where $\equiv$ is the syntactical identity.*

Not all sets of equations have unifiers. If a set of equations has unifiers we say that it is unifiable.

Example 2.2 $S = \{f(x, z) = f(y, g(y))\}$
$\theta = \{g(u)/x, g(u)/y, g(g(u))/z\}$ *is a unifier of S.*

Definition 2.22 (Partial order on substitutions)
Let θ_1 and θ_2 be substitutions. We say that θ_1 is more general than θ_2 ($\theta_2 \le \theta_1$) if there exists a substitution θ_3 such that $\theta_2 = \theta_1\theta_3$.

Definition 2.23 (Most general unifier(mgu)) *A unifier θ_1 of a set of equations S is an mgu of S iff for each unfier θ_2 of S there exists a substitution σ such that $\theta_2 = \theta_1\sigma$.*

Example 2.3 $S = \{g(x) = g(g(z)), f(a, z) = f(a, y)\}$
The following substitutions are mgus of S:
$\{g(z)/x, z/y\}$
$\{g(w)/x, w/y, w/z, y/w\}$
It is possible to write mgus with as many bindings as we want. For instance:
$\{g(z)/x, z/y, x_2/x_1, x_3/x_2, \dots, x_1/x_n\}$

In order to avoid this problem we will consider only idempotent mgus.

Theorem 2.1 (Uniqueness of idempotent mgus) *Let S be a set of equations, θ_1 and θ_2 idempotent mgus of S. In these conditions there exists a substitution:*
$\phi = \{y_1/x_1, \dots, y_k/x_k, x_1/y_1, \dots, x_k/y_k\}$
such that
$\{y_1/x_1, \dots, y_k/x_k\} \subseteq \theta_1$ *and*
$\theta_2 = \theta_1\phi$

In other words an idempotent mgu is unique up to renaming of "equivalent" variables.
In order to compute mgus of sets of equations we transform them in a resolved form using transformations that preserve unifiers. For sets of equations in resolved form the computation of the mgu is trivial.

Definition 2.24 (Set of equations in resolved form) *A set of equations* $S = \{v_1 = t_1, \ldots, v_n = t_n\}$ *is in resolved form if all* v_i *are distinct variables which do not occur in any* t_j.

It is obvious that $\{t_1/v_1, \ldots, t_n/v_n\}$ is an idempotent mgu of S.

Algorithm 2.1 (Unification algorithm)
Choose nondeterministically an equation from the current set of equations and do one of the following actions according to the form of the equation.

1. $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$
   substitute the equation with $t_1 = s_1, \ldots, t_n = s_n$.
2. $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ *with* $f \neq g$
   end with failure.
3. $x = x$
   delete the equation.
4. $t = x$ *with* t *not variable*
   replace the equation with $x = t$.
5. $x = t$ *with* $t \neq x$ *and* x *that occurs in at least another place in the set*
   if x *occurs in* t *end with failure, otherwise replace* x *with* t *in the other equations.*

The algorithm ends with failure if stated from the rules or with success if no actions are applicable.

Example 2.4 $S = \{g(x) = g(g(z)), f(a, z) = f(a, y)\}$
Applying action 1 to equation 1:
$S = \{x = g(z), f(a, z) = f(a, y)\}$
We cannot do anything else on equation 1, applying action 1 to equation 2:
$S = \{x = g(z), a = a, z = y\}$
Applying action 1 to equation 2:
$S = \{x = g(z), z = y\}$
Applying action 5 to equation 2:
$S = \{x = g(y), z = y\}$
The algorithm ends with success.

Theorem 2.2 (Unification theorem) *Unification algorithm applied to a set of equations* S *ends with failure if* S *does not have any unifiers, otherwise it ends with success and it returns a set of equations in resolved form which has the same unifiers as the starting set.*

As a last step we can compute the trivial unifier of the set of equations in resolved form. Mgus computed in this way are idempotent and relevant (they only contain variables which occur in the starting set of equations).

Definition 2.25 (Partial SLD-derivation) *Let P be a logic program. We define a step of an SLD-resolution computation using the following rules:*

$$\frac{}{P \Vdash \Box, G \xrightarrow{\epsilon} G} \qquad \frac{}{P \Vdash G, \Box \xrightarrow{\epsilon} G} \qquad \textit{empty goal}$$

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

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

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

If $P \Vdash N \xrightarrow{\theta} N'$ and the clause variant used is C we say that N' is the resolvent of N and C with mgu θ.
We will omit $P \Vdash$ if P is clear from the context.
A partial SLD-derivation of $P \cup \{G\}$ is a sequence (eventually empty) of steps of SLD-resolution allowed by program P with initial goal G.

Theorem 2.3 (Correctness of SLD-derivation) *If $P \Vdash G \xrightarrow{\theta}{}^{*} G'$ then G' is a logical consequence of $P \cup \{G\}$.*

Definition 2.26 (SLD-refutation) *If an SLD-derivation has as its last goal the empty goal it is called an SLD-refutation.*

Theorem 2.4 (Proof by SLD-refutation) *An SLD-refutation of $P \cup \{\leftarrow A_1, \dots, A_n\}$ whose sequence of mgu is $\theta_1, \dots, \theta_k$ is a proof of: $(A_1, \dots, A_n)\theta_1 \cdots \theta_k$*

Example 2.5 *We can now compute a solution of the goal in example 2.1. We have the following clauses:*
$arc(1,2) \leftarrow, arc(2,3) \leftarrow, \dots, arc(8,9) \leftarrow$
$arc(1,4) \leftarrow, arc(4,7) \leftarrow, \dots, arc(6,9) \leftarrow$
$arc(x,y) \leftarrow arc(y,x)$
$path(1, nil)$
$path(x, cons(y,l)) \leftarrow arc(x,y), path(y,l)$

A computation could be:

Step	Goal / Variant	Unifier
	Variant	*Unifier*
1	$\leftarrow path(5,l)$	
	$path(x, cons(y,l')) \leftarrow arc(x,y), path(y,l')$	$\{5/x, cons(y,l')/l\}$
2	$\leftarrow arc(5,y), path(y,l')$	
	$arc(x',y') \leftarrow arc(y',x')$	$\{5/x', y'/y\}$
3	$\leftarrow arc(y',5), path(y',l')$	
	$arc(2,5) \leftarrow$	$\{2/y'\}$
4	$\leftarrow path(2,l')$	
	$path(x'', cons(y'',l'')) \leftarrow arc(x'',y''), path(y'',l'')$	$\{2/x'', cons(y'',l'')/l'\}$
5	$\leftarrow arc(2,y''), path(y'',l'')$	
	$arc(x''',y''') \leftarrow arc(y''',x''')$	$\{2/x''', y'''/y''\}$
5	$\leftarrow arc(y''',2), path(y''',l'')$	
	$arc(1,2)$	$\{1/y'''\}$
6	$\leftarrow path(1,l'')$	
	$path(1,nil)$	$\{nil/l''\}$

and we obtain the empty goal.
So we have proved:
path(5,cons(2,cons(1,nil)))

2.2 Hypergraphs and syntactic judgements

In this section we will introduce the notion of hypergraphs formalizing them as well-formed syntactic judgements.
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). Now, we present a definition of graphs as syntactic judgements, where nodes correspond to names (i. e. variables), 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 2.27 (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 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,* 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 occurrences of* y *(also the ones in* G*) are bound occurrences. We demand that* $\mathrm{fn}(G) \subseteq \Gamma$.

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 2.28 (Structural congruence and well-formed judgements) *We define the structural congruence* $\equiv$ *on syntactic judgements which obeys the following axioms:*

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

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

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

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

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

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

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

Syntactic rules

$$\text{(RG1)}\ \frac{x_i \in \mathcal{N}}{x_1, \ldots, x_n \vdash nil}$$

$$\text{(RG2)}\ \frac{L \in LE_m \qquad x_i \in \mathcal{N} \qquad y_i \in \{x_j\}}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}$$

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

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

The well-formed judgements for constructing graphs over LE *and* $\mathcal{N}$ *are those generated by applying the syntactic rules up to axioms of structural congruence.*

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 bounded 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 external nodes. Finally, rule (RG4) allows to hide a node from the environment.

If necessary, thanks to axiom (AG4), we will write νX, with $X = \bigcup_{i=1...n}\{x_i\}$, to abbreviate $\nu x_1\ \nu x_2 \dots \nu x_n$. 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 $\mathrm{var}(G) \subseteq \Gamma \cup X$. We can state the following correspondence theorem.

Theorem 2.5 (Correspondence of graphs and judgements)
Well-formed judgements up to structural axioms are isomorphic to graphs up to isomorphisms.

We can think that a hypergraph represents a network, where arcs represent processes or network nodes and graph nodes represent communication ports or channels.

2.3 Synchronized hyperedge replacement (SHR)

In this section we will introduce some increasingly powerful SHR systems. The difference among the systems is in the different restrictions imposed in name communications and fusions.
The first SHR system we introduce has been proposed by König and Montanari in [10].
In order to define rewriting on syntactic judgements we will introduce the notion of rewriting rule. We use a ranked set *Act* of arbitrary actions, which can be thought of as the set of signals which are allowed in a network.

Definition 2.29 (Production) *Let* $Act = \{Act_n\}_{n=0,1,...}$ *be a ranked set of actions, containing also the idle action* ϵ *of arity* 0 *(the arity indicates the number of nodes created by an action, if* a *is an action* $\mathrm{ar}(a)$ *is its arity).*
A production is of the form:

$$x_1, \dots, x_n \vdash s(x_1, \dots, x_n) \xrightarrow{\Lambda} x_1, \dots, x_n, \Gamma_\Lambda \vdash G$$

where all x_i are distinct. $\Lambda \subseteq \{x_1, \ldots, x_n\} \times Act \times \mathcal{N}^*$ such that Λ is a total function in its first argument, i. e. if $(x_i, a_i, \vec{y}_i) \in \Lambda$ we write $\Lambda(x_i) = (a_i, \vec{y}_i)$, respectively $\mathrm{act}_\Lambda(x_i) = a_i$ and $\mathrm{n}_\Lambda(x_i) = \vec{y}_i$, and we demand that $\mathrm{ar}(a_i) = |\vec{y}_i|$.
Furthermore [1] $\Gamma_\Lambda = \bigcup_{x_i \in \Lambda} \mathrm{Set}(n_\Lambda(x_i))$ *and we demand that* $\{x_1, \ldots, x_n\} \cap \Gamma_\Lambda = \emptyset$.
Productions are to be considered as schemas and so are α-convertible with respect to all the names they contain.

A production of the form given above indicates that an edge $s(x_1, \ldots, x_n)$ is rewritten, synchronizing on each node x_i with an action a_i, and during this synchronization a string $\vec{y}_i$ of new nodes is created. The set Γ_Λ contains all new nodes in the interface which are created by the rewriting step.
Observe that the last condition imposes that only new nodes can be transmitted (and so merged).
In particular a bound node cannot be transmitted to the outside and therefore it cannot become free (this is called an extrusion of the node). As this is the main characteristic of this SHR style we will refer it as SHR without extrusions.

In order to be able to define inference rules which describe how to derive more complex transitions from basic productions, we first introduce the following notion of a most general unifier which transforms a relation Λ, which does not necessarily satisfy the conditions of definition 2.29, into a function.

Definition 2.30 (Most general unifier(mgu)) *Let $\sigma : \mathcal{N} \to \mathcal{N}$ be a name substitution. If $\Lambda = \{(x_i, a_i, \vec{y}_i) | i \in \{1, \ldots, n\}\}$, then $\sigma(\Lambda) = \{(\sigma(x_i), a_i, \sigma^*(\vec{y}_i)) | i \in \{1, \ldots, n\}\}$ where σ^* is the extension of σ to strings. For any $\Lambda = \{(x_i, a_i, \vec{y}_i) | i \in \{1, \ldots, n\}\} \subseteq \mathcal{N} \times Act \times \mathcal{N}^*$ we call a name substitution ρ a unifier of Λ whenever $\rho(x_i) = x_i$ for $i \in \{1, \ldots, n\}$ and $x_i = x_j$ implies $a_i = a_j$ and $\rho^*(\vec{y}_i) = \rho^*(\vec{y}_j)$.*
The mapping ρ is called a most general unifier whenever it is a unifier with a minimal degree of non-injectivity.

The name most general unifier is justified because it is the most general unifier (in the sense of definition 2.23) of the set of equations:
$\{x_i = a_i(\vec{y}_i)\}_{i=1 \ldots n}$
where x_i are treated as constants. Note that this definition forbids fusions among old nodes even if old nodes are allowed in $\vec{y}_i$.
Even in this restricted case unifiers do not necessarily exist.

We now introduce the inference rules used to build up complex transitions from basic productions.

[1] For any string $\vec{s}$, we denote the set of its elements by $\mathrm{Set}(\vec{s})$

Definition 2.31 (Inference rules for transitions without extrusions) *All possible transitions $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ between graphs are generated by a set $\mathcal{P}$ of productions and the following inference rules.*

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

where $\sigma : \Gamma \to \Gamma$ and ρ is the most general unifier of $\sigma(\Lambda)$.

$$(par)\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma, \Gamma_{\Lambda_1} \vdash G'_1, \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Gamma_{\Lambda_2} \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\rho(\Lambda_1 \cup \Lambda_2)} \Gamma, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)} \vdash \rho(G'_1|G'_2)}$$

if $\Gamma_{\Lambda_1} \cap \Gamma_{\Lambda_2} = \emptyset$ and ρ is the most general unifier for $\Lambda_1 \cup \Lambda_2$.

$$(hide)\quad \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma \vdash \nu x\ G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash \nu x\ \nu Y\ G'}$$

where $Y = \mathrm{Set}(\vec{y}) \setminus \Gamma_\Lambda$.

$$(idle)\quad \Gamma \vdash G \xrightarrow{\Lambda} \Gamma \vdash G$$

where [2] *$\Lambda(x) = (\epsilon, \langle\rangle)$ for each $x \in \Gamma$.*

$$(new)\quad \frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'}{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, \mathrm{Set}(\vec{y}) \vdash G'}$$

We write $\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$ whenever this transition can be derived from a set $\mathcal{P}$ of productions.

In every transition, Λ assigns to each free name the action it performs and the string of new nodes it creates. Rule (ren) deals with non-injective renaming of the nodes of a graph, which is necessary in order to handle edges of the form $s(\ldots, x, \ldots, x, \ldots)$, i. e. edges which are connected several times to the same node. Parallel composition of syntactic judgements is treated in rule (par) which makes sure that whenever two edges share a node, they do compatible synchronizations on that node and that the nodes they create in doing such synchronization are merged by ρ. Rule (hide) deals with hiding of names and it hides only all the names which have been created exclusively by interaction on the hidden name. Rule (idle) guarantees that each node can always make an explicit idle step. Rule (new) allows adding an additional name to the interface which performs arbitrary actions. This is

[2] $\langle\rangle$ denote the empty sequence.

due to Hoare synchronization which requires that any number of edges, and therefore zero edges too, can synchronize on a given node.

Definition 2.32 (Grammar) *A grammar consists of an initial graph* $\Gamma_0 \vdash G_0$ *and a set* $\mathcal{P}$ *of productions. A derivation is a finite or infinite sequence of the form* $\Gamma_0 \vdash G_0 \xrightarrow{\Lambda_1} \Gamma_1 \vdash G_1 \xrightarrow{\Lambda_2} \cdots \xrightarrow{\Lambda_n} \Gamma_n \vdash G_n \ldots$, *where* $\Gamma_{i-1} \vdash G_{i-1} \xrightarrow{\Lambda_i} \Gamma_i \vdash G_i$, $i = 1, 2 \ldots$ *is a transition generated by* $\mathcal{P}$ *applying the inference rules.*

An immediate extension of this SHR system is the one proposed by Hirsch and Montanari in [9]. This extension allows transmission of both old and new nodes but it forbids fusions among old nodes. In order to do that we need to relax the condition $\{x_1, \ldots, x_n\} \cap \Gamma_\Lambda = \emptyset$ in definition 2.29 and to extend the axiom (hide):

$$\text{(hide)} \quad \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma \vdash \nu x \ G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash \nu x \ \nu Y \ G'}$$

where $Y = \mathrm{Set}(\vec{y}) \setminus (\Gamma_\Lambda \cup \Gamma)$.

All the other definitions can be preserved.
We will refer this SHR system as SHR without fusions of old names because the definition of mgu used does not allow it.

The last step is to allow fusions among old nodes.
This extension is more complicated than the previous one and in literature can only be found for Milner synchronization proposed by Ferrari, Montanari and Tuosto in [7].
We will refer this SHR style as full Milner SHR.
We recall that for Milner synchronization the set of actions Act_t is composed by three parts: a ranked set Act of actions, a set of coactions $\{\overline{a} | a \in Act\}$ and a special action τ with arity 0 that represents an internal action. First, we need to extend definition 2.29.

Definition 2.33 *A production is of the form:*

$$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda, \pi} \phi \vdash G$$

where all x_i *are distinct. Let* Γ *be* $\{x_1, \ldots, x_n\}$. $\Lambda \subseteq \{x_1, \ldots, x_n\} \times Act \setminus \{\epsilon\} \times \mathcal{N}^*$ *such that* Λ *is a partial (we do not define* Λ *for nodes in which there are no requirements) function in its first argument and we demand that* $\mathrm{ar}(a_i) = |\vec{y}_i|$. π *is a name fusion of* Γ. *We define:*

- $\mathrm{n}(\Lambda) = \{z \mid \exists x. z \in \mathrm{n}_\Lambda(x)\}$
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$
- $\phi = \pi(\Gamma) \cup \Gamma_\Lambda$

Productions are to be considered as schemas and so they are α-convertible with respect to all the names they contain.

We now need a new set of inference rules.

Definition 2.34 (Rules for full 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 $(\Gamma \cup \phi) \cap (\Gamma' \cup \phi') = \emptyset$.

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

where:

- $\Gamma_\Lambda \cap \sigma(\Gamma) = \emptyset$
- $\forall x \in \Gamma_\Lambda . \sigma(x) = x$
- $\sigma(x) = \sigma(y) \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \wedge x \neq y \Rightarrow$
  $(\forall z \in \mathcal{N} \setminus \{x, y\}. \sigma(z) = \sigma(x) \Rightarrow \Lambda(z) \uparrow)$
  $\wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w}) \wedge a \neq \tau$
- $\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) \mid \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (\overline{a}, \vec{w})\}$
  $\cup \{\sigma(x) = \sigma(y) \mid \pi(x) = \pi(y)\})$
- $\Lambda'(z) = \begin{cases} (\tau, \langle\rangle) & \textit{if } \sigma(x) = \sigma(y) = z \wedge x \neq y \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \\ \rho(\sigma(\Lambda))(z) & \textit{otherwise} \end{cases}$
- $\pi'(\sigma(x)) = \rho(\sigma(\pi(x)))$
- $U = \rho(\sigma(\phi)) \setminus \phi'$

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

where:

- $(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
- $\Lambda(x) \uparrow \vee \Lambda(x) = (\tau, \langle\rangle)$
- $\Lambda' = \Lambda \setminus \{(x, \tau, \langle\rangle)\}$
- $Z = \phi \setminus \phi'$

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

where $\Lambda(x) \uparrow$ *for each* $x \in \Gamma$.

Rule (par) simply combines together two disjoint judgements.
Rule (merge) is the rule for synchronization. The rule states that in a transition it is possible to merge two nodes x and y that offer complementary non silent actions. Here ρ is the most general unifier that fuses the corresponding names of the actions and that propagates the previous fusions (determined by π). The label Λ' takes into account all possible synchronizations and it leaves unchanged the actions offered on the other nodes up to the necessary fusions. The new fusion substitution π' acts on $\sigma(\Lambda)$ by applying to it the mgu ρ. Finally the names in ϕ after the fusion which are not present in ϕ' are restricted (these are the nodes created during the synchronization of two nodes that have been merged).
Rules (res) deals with node restriction. According to the first condition, the restricted node must not be the representative of the equivalence class induced by π when this class contains nodes different from x. Furthermore, nodes can be resticted only if either no actions or internal actions take place.
The extension of the definition of grammar is trivial.

2.4 Category theory

Definition 2.35 (Category) *A category C is:*

- *a collection Ob_C of objects, denoted by $a, b, \ldots, A, B, \ldots$*
- *a collection Mor_C of morphisms (arrows), denoted by $f, g, \ldots$*
- *two operations* dom *and* cod *assigning to each arrow f two objects respectively called domain (or source) and codomain (or target) of f*
- *an operation id assigning to each object b a morphism id_b (the identity of b) such that* $\mathrm{dom}(id_b) = \mathrm{cod}(id_b) = b$
- *an operation ; (composition) assigning to each pair f, g of arrows with* $\mathrm{cod}(f) = \mathrm{dom}(g)$ *an arrow $f;g$ such that* $\mathrm{dom}(f;g) = \mathrm{dom}(f)$ *and* $\mathrm{cod}(f;g) = \mathrm{cod}(g)$

- *identity and composition, moreover, must satisfy the following conditions:*

  identity law *: for any arrows f, g such that* $\mathrm{cod}(f) = \mathrm{dom}(g) = b$*:*
  $f; id_b = f$
  $id_b; g = g$

  associative law *: for any arrows f, g, h such that* $\mathrm{cod}(f) = \mathrm{dom}(g)$ *and* $\mathrm{cod}(g) = \mathrm{dom}(h)$
  $(f;g);h = f;(g;h)$

We use as notation $f : a \to b$ to denote a morphism f with source a and target b.

Definition 2.36 (Dual category) *The dual category C^{op} of a category C has the same objects and the same morphisms of C, $id_b^{op} = id_b$,* $\mathrm{dom}^{op}(f) = \mathrm{cod}(f)$, $\mathrm{cod}^{op}(f) = \mathrm{dom}(f)$ *and $f;^{op} g = g; f$.*

Definition 2.37 (Isomorphism) *A morphism $f : a \to b$ is called an isomorphism iff there exists $g : b \to a$ such that:*
$f; g = id_a$
$g; f = id_b$

Definition 2.38 (Isomorphic objects) *Two objects a and b are isomorphic iff there exists an isomorphism $f : a \to b$.*

Definition 2.39 (Strict monoidal category) *A category C is said to be strict monoidal iff there exist two total operations $\otimes_o$ from pairs of objects to objects and $\otimes_a$ from pairs of arrows to arrows (we will design both them with $\otimes$ if no confusion will arise) and an object 1 such that:*

$a \otimes 1 = 1 \otimes a = a$
$f \otimes id_1 = id_1 \otimes f = f$
$id_{a \otimes b} = id_a \otimes id_b$
$\frac{f:a \to b \quad g:c \to d}{f \otimes g : a \otimes c \to b \otimes d}$
$\frac{f_1:a_1 \to b_1 \quad g_1:b_1 \to c_1 \quad f_2:a_2 \to b_2 \quad g_2:b_2 \to c_2}{(f_1;g_1) \otimes (f_2;g_2) = f_1 \otimes f_2 ; g_1 \otimes g_2}$

where we have assigned higher priority to $\otimes$ with respect to ;.

Definition 2.40 (Symmetric monoidal category) *A category C is said to be symmetric monoidal iff it is strict monoidal and there exists a total operation $\gamma_{\_,\_}$ from pairs of objects to arrows such that:*

$\gamma_{a,b} : a \otimes b \to b \otimes a$
$\gamma_{1,1} = id_1$

$\gamma_{a\otimes b,c} = id_a \otimes \gamma_{b,c}; \gamma_{a,c} \otimes id_b$
$\gamma_{a,b}; \gamma_{b,a} = id_{a\otimes b}$
$\frac{f:a\to b \quad g:c\to d}{f\otimes g;\gamma_{b,d}=\gamma_{a,c};g\otimes f}$

Definition 2.41 (Cartesian category(1)) *A category C is said to be cartesian iff it is symmetric monoidal and there exist two total operations $\nabla_{\_}$ and $!_{\_}$ from objects to arrows such that:*

$\nabla_a : a \to a \otimes a$
$!_a : a \to 1$
$\nabla_{a\otimes b} = \nabla_a \otimes \nabla_b; id_a \otimes \gamma_{a,b} \otimes id_b$
$!_{a\otimes b} = !_a \otimes !_b$
$!_1 = \nabla_1 = id_1$
$\nabla_a; id_a \otimes \nabla_a = \nabla_a; \nabla_a \otimes id_a$
$\nabla_a; \gamma_{a,a} = \nabla_a$
$\nabla_a; id_a \otimes !_a = id_a$
$\frac{f:a\to b}{f;\nabla_b=\nabla_a;f\otimes f \quad f;!_b=!_a}$

Definition 2.42 (Categorical product) *Let C be a category, $a, b \in Ob_C$. The categorical product of a and b is an object $a \otimes b$ together with two morphisms $p_a : a \otimes b \to a$ and $p_b : a \otimes b \to b$ such that for any $f : c \to a$ and $g : c \to b$ there exists only one $h : c \to a \otimes b$ such that the following diagram commutes:*

$$\begin{tikzcd} & c \arrow[dl, "f"'] \arrow[d, "h"] \arrow[dr, "g"] & \\ a & a \otimes b \arrow[l, "p_a"] \arrow[r, "p_b"'] & b \end{tikzcd}$$

Definition 2.43 (Terminal object) *An object o in a category C is terminal iff for any $b \in Ob_C$ there is a unique morphism $!_b : b \to o$.*

Definition 2.44 (Cartesian category(2)) *A category C is said to be cartesian iff:*

- *it contains a terminal object t*
- *every pair of objects has a categorical product*

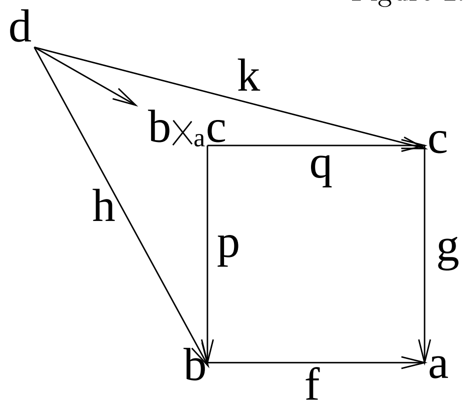
Theorem 2.6 *The definition 2.41 satisfies the definition 2.44 and it identifies isomorphic objects.*

Definition 2.45 (Pullback) *Given two arrows $f : b \to a$ and $g : c \to a$ with common target a, the pullback of f and g is an object $b \times_a c$ and two arrows $p : b \times_a c \to b$ and $q : b \times_a c \to c$, such that:*

- $p; f = q; g$
- *for every other triple $(d, h : d \to b, k : d \to c)$ such that $k; g = h; f$, there exists a unique arrow $< h, k >_a : d \to b \times_a c$ such that $< h, k >_a; p = h$ and $< h, k >_a; q = k$*

Figure 2.2 represents a typical pullback diagram.

Figure 2.2: Pullback diagram



We now analyze a special category, the algebraic theory associated to a signature Σ.
We illustrate the constructions used for one-sorted signatures.

Definition 2.46 (Algebraic theory) *The free algebraic theory associated to a signature Σ is the category $Th[\Sigma]$ defined below:*

- *its objects are 'underlined' natural numbers*
- *the arrows from $\underline{m}$ to $\underline{n}$ are n-uples of terms in the free Σ-algebra with at most m variables, and the composition of arrows is term substitution. The arrows are generated from Σ by the following inference rules, and are considered up to the axioms of cartesian category.*

$$op \ \frac{f \in \Sigma_n}{f : \underline{n} \to \underline{1}}$$

$$id \ \frac{n \in \mathbb{N}}{id_{\underline{n}} : \underline{n} \to \underline{n}}$$

$$seq\ \frac{\alpha : \underline{n} \to \underline{m}\ \beta : \underline{m} \to \underline{k}}{\alpha;\beta : \underline{n} \to \underline{k}}$$

$$mon\ \frac{\alpha : \underline{n} \to \underline{m}\ \beta : \underline{k} \to \underline{l}}{\alpha \otimes \beta : \underline{n+k} \to \underline{m+l}}$$

$$sym\ \frac{n, m \in \mathbb{N}}{\gamma_{\underline{n},\underline{m}} : \underline{n+m} \to \underline{m+n}}$$

$$dup\ \frac{n \in \mathbb{N}}{\nabla_{\underline{n}} : \underline{n} \to \underline{n+n}}$$

$$dis\ \frac{n \in \mathbb{N}}{!_{\underline{n}} : \underline{n} \to \underline{0}}$$

This is the category of finite substitutions on Σ (with canonical variables).
The rule op defines basic substitutions $[f(x_1, \ldots, x_n)/x_1]$ for all $f \in \Sigma_n$. The rule id yields identity substitutions. The rule seq applies a substitution to another substitution. The rule mon composes in parallel two substitutions. The rule sym provides permutations. The rule dup introduces sharing of variables. The rule dis defines the empty substitution.
We can extend these constructions to many-sorted signatures by considering strings of sorts instead of underlined natural numbers.

2.5 Tile logic

Tile logic is a computational model that extends SOS systems ([11]) and rewriting logic ([13]).
See also [8] for further information.
Tile logic allows the description of open components of a system, and it allows a component to make a transition iff its subcomponents make themselves transitions that trigger it.
Abstractly we can see tile logic as double categories.

Definition 2.47 (Double category) *A double category D consists of a collection of objects $o, o', \ldots$, a collection of horiziontal arrows $h, h', \ldots$, a collection of vertical arrows $v, v', \ldots$ and a collection of double cells (also called cells) $A, B, C, \ldots$.*

- *Objects and horizontal arrows form an horizontal category $\mathcal{H}$, with identity id_o for each object o and composition $- * -$.*
- *Objects and vertical arrows form also a category $\mathcal{V}$, with identity id_o for each object o and composition $- \cdot -$.*

- *Cells are assigned horizontal source and target (which are vertical arrows) and vertical source and target (which are horizontal arrows); furthermore sources and targets must be compatible, in the sense that they must satisfy the equalities on source and target represented in figure 2.3, for which we use the notation* $A : h_0 \xrightarrow[v_1]{v_0} h_1$.

Figure 2.3: Graphic representation of a cell

$$\begin{tikzcd} o_0 \arrow[r, "h_0"] \arrow[d, "v_0"'] \arrow[dr, phantom, "A"] & o_1 \arrow[d, "v_1"] \\ o_2 \arrow[r, "h_1"'] & o_3 \end{tikzcd}$$

- *cells can be composed both horizontally* $(- * -)$ *and vertically* $(- \cdot -)$ *as follows: given* $A : h_0 \xrightarrow[v_1]{v_0} h_1$, $B : h_2 \xrightarrow[v_2]{v_1} h_4$ *and* $C : h_1 \xrightarrow[v_4]{v_3} h_4$ *then* $A * B : h_0 * h_2 \xrightarrow[v_2]{v_0} h_1 * h_3$ *and* $A \cdot C : h_0 \xrightarrow[v_1 \cdot v_4]{v_0 \cdot v_3} h_4$ *are cells. Moreover, given a fourth cell* $D : h_3 \xrightarrow[v_5]{v_4} h_5$ *horizontal and vertical composition satisfy the following exchange law:*

$$(A \cdot C) * (B \cdot D) = (A * B) \cdot (C * D)$$

  Under these rules, cells form both a horizontal category $\mathcal{D}^*$ *and a vertical category* $\mathcal{D}^{\cdot}$, *with identities* $1_v : id_0 \xrightarrow[v]{v} id'_0$ *and* $1^h : h \xrightarrow[id'_0]{id_0} h$, *respectively, with* $1^{h_0 * h_1} = 1^{h_0} * 1^{h_1}$ *and* $1_{v_0 \cdot v_1} = 1_{v_0} \cdot 1_{v_1}$.

- *Furthermore we have* $1_{id_0} = 1^{id_0}$ *and we can denote this cell simply with* 1_0.

We shall use monoidal categories for horizontal and vertical categories. In both these categories we denote sequential and parallel composition with ; and $\otimes$ respectively.

Tile logic gives a computational interpretation to double categories, where:

- the objects represent the (initial/final,input/output) interfaces through which system components can be connected
- the horizontal arrows describe (initial/final) configurations, sources and targets corresponding to input and output interfaces
- the vertical arrows are the observations (trigger/effect), sources and targets corresponding to initial and final interfaces

- the cells represent the possible transformations (tiles) that can be performed by the system

Thus a cell $h_0 \xrightarrow[v_1]{v_0} h_1$ says that the state h_0 can evolve to h_1 via an action triggered by v_0 and with effect v_1. The way in which observations and configurations of a cell are connected via their interfaces expresses the locality of actions, i. e. the places where triggers are applied to and effects are produced by.
One can be interested in the cells or just in their borders. We shall concentrate only on the second alternative, which correponds to the so called flat tile logic.

Definition 2.48 (Tile system) *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 $Ob_{\mathcal{H}} = Ob_{\mathcal{V}}$, N is a set of rule names and $R : N \to Mor_{\mathcal{H}} \times Mor_{\mathcal{V}} \times Mor_{\mathcal{V}} \times Mor_{\mathcal{H}}$ is a function such that for all $\alpha \in N$, if $R(\alpha) = \langle t, u, v, s\rangle$ then $t : o_0 \to o_1$, $u : o_0 \to o_2$, $v : o_1 \to o_3$, $s : o_2 \to o_3$ for suitable objects o_0, o_1, o_2 and o_3.*
We will denote such rule by writing $\alpha : t \xrightarrow[v]{u} s$.

We say that $t \xrightarrow[v]{u} s$ is entailed by the flat tile logic associated to $\mathcal{R}$, written $\mathcal{R} : t \xrightarrow[v]{u} s$ if we can express $t \xrightarrow[v]{u} s$ as the composition of basic and auxiliary tiles (auxiliary tiles are the possible commuting squares of the category, so they depend on the chosen tile format; however identities are always auxiliary tiles).
We can apply to tile logic the well-known concept of bisimulation.

Definition 2.49 (Tile bisimulation) *Let $\mathcal{R}$ be a tile system.*
An equivalence relation $\equiv_b \subseteq Mor_{\mathcal{H}} \times Mor_{\mathcal{H}}$ is a tile bisimulation for $\mathcal{R}$ if, whenever $s \equiv_b t$ for s, t elements of $Mor_{\mathcal{H}}$, then for any sequent $s \xrightarrow[b]{a} s'$ entailed by $\mathcal{R}$ a corresponding sequent $t \xrightarrow[b]{a} t'$ is entailed by $\mathcal{R}$ with $s' \equiv_b t'$.

Proposition 2.3 *Tile bisimulations are closed under set union.*

Definition 2.50 (Tile bisimilarity) *The maximal tile bisimulation with respect to set union is called tile bisimilarity and it is denoted by $\equiv$.*

We need a condition under which tile bisimilarity yields a congruence (w. r. t. the operations of the underlying horizontal structure).

Definition 2.51 (Decomposition property) *A tile system $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ enjoys the decomposition property if for all arrows $t \in Mor_{\mathcal{H}}$ and for all sequents $t \xrightarrow[v]{u} s$ entailed by $\mathcal{R}$, then:*

1. *if $t = t_1; t_2$ then $\exists w \in Mor_{\mathcal{V}}, s_1, s_2 \in Mor_{\mathcal{H}}$ such that $\mathcal{R} : t_1 \xrightarrow[w]{u} s_1$, $\mathcal{R} : t_2 \xrightarrow[v]{w} s_2$ and $s = s_1; s_2$*

2. *if* $t = t_1 \otimes t_2$ *then* $\exists u_1, u_2, v_1, v_2 \in Mor_{\mathcal{V}}, s_1, s_2 \in Mor_{\mathcal{H}}$ *such that* $\mathcal{R} : t_1 \xrightarrow[v_1]{u_1} s_1$, $\mathcal{R} : t_2 \xrightarrow[v_2]{u_2} s_2$, $u = u_1 \otimes u_2$, $v = v_1 \otimes v_2$ *and* $s = s_1 \otimes s_2$.

Condition (1) is called sequential decomposition and condition (2) is called parallel decomposition.

Theorem 2.7 *If* $\mathcal{R}$ *enjoys the decomposition property, then tile bisimilarity is a congruence.*

When only instantiation/contextualization (i. e. adding new parts of configuration to the right/left) are considered as meaningful operations of the system, then sequential decomposition is enough for guaranteeing tile bisimilarity to be a congruence w. r. t. these closure operations.

2.6 From logic programming to tile logic

In this section we present a tile system that behaves as logic programs. For more details see [4].
First of all we have to handle mgu computation. Luckily, the mgu in the case of logic programming (in which we know that the goal and the clause variant have disjoint sets of names) corresponds to the construction of a pullback in the category of substitutions. Moreover the pullback construction can be specified, for a given signature, with a finite number of tiles. In particular, the following theorems hold:

Theorem 2.8 (Existence of a pullback tile system) *Given an op-category of substitutions on the signature* Σ *we can build a tile system* $\mathcal{R}_{PB(\Sigma)}$ *with a finite specification whose tiles are all the pullback squares of the given category.*

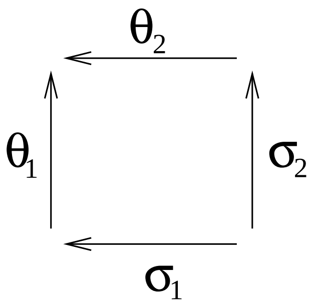
Theorem 2.9 (Pullbacks compute mgus for logic programming)
Given a category of substitutions on the signature Σ*, the tiles in the tile system* $\mathcal{R}_{PB(\Sigma)}$ *are of the form in figure 2.4 where* σ_1 *and* σ_2 *are the mgu of* $\{s_i = t_i\}_{i \in I}$ *restricted to the variables of* $\mathrm{Im}(\theta_1)$ *and* $\mathrm{Im}(\theta_2)$ *respectively, for some pair of substitution terms* $\theta_1 = \{t_i/x_i\}_{i \in I}$ *and* $\theta_2 = \{s_i/x_i\}_{i \in I}$ *where* $\mathrm{var}(\mathrm{Im}(\theta_1)) \cap \mathrm{var}(\mathrm{Im}(\theta_2)) = \emptyset$ *and for each such pair of substitution terms the corresponding tile is in* $\mathcal{R}_{PB(\Sigma)}$.

Definition 2.52 (Tile system for logic programming) *Given a logic program* P *on the alphabet* Σ*, we denote by* $\mathcal{R}_P$ *the tile system specified by the following rules:*

- *There are two basic sorts* t *(for terms) and* p *(for predicates). Correspondingly, the interfaces are elements of* $\{t, p\}^{*3}$.

[3] $\{t, p\}^*$ denotes the set of strings on the alphabet $\{t, p\}$.

Figure 2.4: Tile for mgu



- *To each functional symbol f with arity n in the alphabet, we associate an operator $f : t^n \to t$ in the signature of configurations, and to each predicate symbol p ($\Box$ can be viewed as a nullary predicate) with arity k in the alphabet, we associate an operator $p : t^k \to p$ in the signature of configurations. Then, we add the symbol $- \wedge - : p^2 \to p$ for modelling conjunction. We can define without loss of generality the conjunction operator to be associative and with unit $\Box$. The configurations are the arrows of the op-category of the free cartesian category generated by the signature of configurations.*

- *To each functional symbol f with arity n in the alphabet, we associate an operator $f : t^n \to t$ in the signature of observations (symbol f is thus overloaded). The observations are the arrows of the op-category of the free cartesian category generated by the signature of observations.*

- *To each clause*
  $c \equiv p(t_1, \ldots, t_k) \leftarrow q_1(\vec{s}_1) \wedge \cdots \wedge q_m(\vec{s}_m)$
  (over n variables $\{x_1, \ldots, x_n\}$) in the logic program we associate a basic tile T_c in our system whose initial configuration is $p : t^k \to p$ (representing the predicate symbol in the head of c), whose final configuration is $q_1(\vec{s}_1) \wedge \cdots \wedge q_m(\vec{s}_m) : t^n \to p$ (representing the body of the clause), whose trigger is the identity $id_p : p \to p$ and whose effect is the tuple $\langle t_1, \ldots, t_k \rangle : t_n \to t_k$ (representing the pattern to be matched by the arguments of predicate p). Note that the body of the clause may contain variables not appearing in the head and consequently some dischargers will be used in the effect of the tile. Moreover, since the same variable can be used more than once, duplicators can also be necessary. Since we take the op-categories the direction of all arrows is reversed w. r. t. the standard representation.

- *Finally, we add the basic tiles contained in $\mathcal{R}_{PB(\Sigma)}$ to calculate the mgus.*

Theorem 2.10 (Correspondence between SLD-derivations and tiles)
Let P be a logic program and G a goal. Then:

1. *if in* P *we have* $G \xrightarrow{\sigma} G'$ *then* $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'$ *with* $\theta = \sigma|_{\mathrm{var}(G)}$

2. *if* $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'$*, then there exists* σ *with* $\theta = \sigma|_{\mathrm{var}(G)}$ *such that in* P *we have* $G \xrightarrow{\sigma} G'$

Lemma 2.1 *For any logic program* P*, the corresponding tile system* $\mathcal{R}_P$ *enjoys the sequential decomposition property.*

Definition 2.53 *Let* P *be a logic program and* G *and* G' *two goals.*
$G \cong_P G'$ *iff* G *and* G' *are tile bisimilar in* $\mathcal{R}_P$*.*

Theorem 2.11 (Bisimilarity is a congruence)
For any logic program P*, the equivalence* $\cong_P$ *is a congruence with respect to conjunction of goals and instantiation of free variables.*

This means that $G \cong_P G'$ implies $G \wedge G'' \cong_P G' \wedge G''$ and $G\sigma \cong_P G'\sigma$ for each goal G'' and each substitution σ. Observe that sequential decomposition is enough to prove it because conjunction of goals is an operation of contextualization and instantiation of free variables is clearly an operation of instantiation.

2.7 Prolog and meta-interpretation

Prolog is the most famous logic programming language.
It provides standard SLD-resolution as well as many other features, some of which deal with logic, like $\vee$ and $\neg$ operators, while others are extra-logical. We will now introduce some of them, but first we will show Prolog syntax. Prolog has been implemented in a variety of slightly different versions. We will refer to Sicstus Prolog 3.7.1 (see also [6]).
A clause:
$A \leftarrow B_1, \ldots, B_n$
is written as:
$A : -B_1, \ldots, B_n.$
If $n = 0$ we will simply write:
$A.$
Prolog uses names starting with uppercase letters for variables and with lowercase letters for predicates, functions and constants.
It is possible to use anonymous variables (_) or also variables whose names are not meaningful (i. e. the name is used only to make the program more readable) by letting them start with _.
Prolog always expands the first atom in the goal using the first matching clause

in clauses database. If no such clauses exist, then Prolog backtracks to the last expansion done and it tries another clause. If it finds the empty goal, it reports a solution and it asks the user whether it has to search for other solutions, otherwise it ends with failure.
To reduce Prolog search the cut (!) operator is available. If we have the clause:
$A :- B_1, \ldots, B_i, !, B_{i+1}, \ldots, B_n.$
Prolog does not use the following clauses to expand A and it considers only the first solution of $B_1, \ldots, B_i$.
Moreover predicates true (which always succedes) and fail (which always fails) are available.
Prolog also provides the or operator (;).
Prolog also supplies arithmetic operators in the form $X\ is\ YopZ$ where *op* is an arithmetic operator. In this case Y and Z must be already initialized.
Moreover arithmetic predicates such as $>$, $<$ and $=$ can be used in the infix notation instead of in the prefix one.
There exists also the infix predicate $==$ that tests for the equality of terms without unifying them.
Prolog also supports lists with the syntax $[X, Y|TAIL]$ where X and Y are the first and the second element in the list and $TAIL$ is the list of the following elements.
Prolog allows a program to dynamically update clauses database with the following predicates:

assert(C) Inserts C in clauses database.

asserta(C) Inserts C as the first clause in clauses database.

assertz(C) Inserts C as the last clause in clauses database.

retract(C) Deletes the first clause in clauses database which match with C.

retractall(H) Deletes all clauses in clauses database which matches with H.

where C is a clause and H a clause head.
We can also explicitly use clauses database:

clause(H,B) Finds the first clause whose head unifies with H (B becomes the body of the clause).

call(G) Executes goal G.

Prolog offers some operators to work with terms:

var(X) Is true if X is a variable which is currently uninstantiated.

simple(X) Is true if X is currently uninstantiated or instantiated to a number or a term of ariety 0.

compound(X) Is true if X is currently instantiated to a complex term.

T=..L Translates between a term T and a list L containing as head the functor of the term and as tail the list of its arguments.

It is also possible to collect all the solutions of a goal in a list using (for example):

```
findall(Var,Goal,L)
```

which instantiates L to the list of values that the variable Var assumes in the solutions of goal Goal.
Moreover, there are some output primitives such as `write(X)` which writes the value of X and `nl` which goes to a new line and some input ones such as `read_term(Term,Options)` which reads a term from standard input.
Finally, Prolog has also a great quantity of libraries with useful predicates for many applications.

Using these facilities it is possible to simply write meta-iterpreters for Prolog.

Example 2.6 *This is a simple meta-interpreter for pure logical language.*

```
solve(true).
solve((A,B)):-solve(A),solve(B).
solve(A):-clause(A,B),solve(B).
```

We can use the same technique to add some features to the interpreter, for example with some little updates we can have a meta-interpreter which computes the length of the derivation used:

```
solve(true,0).
solve((A,B),N):-!,solve(A,N1),solve(B,N2),N is N1+N2.
solve(A,N):-clause(A,B),solve(B,N1),N is N1+1.
```

Chapter 3

Logic programming extensions

In this chapter, we introduce two variations of logic programming: logic programming with restriction, which extends logic programming with an operator of name hiding, and synchronized logic programming, which restricts logic programming to the computations that have an SHR equivalent. Then we will study some properties of logic programming with restriction and we extend the work of [4] on logic programming in the framework of tile logic to this more general case. In particular, we show that the tile semantics (we will present two forms of it) we build are compositional, i. e. tile bisimilarity is a congruence.

3.1 Synchronized logic programming

Our purpose is to define a logical goal rewriting system that can be used to simulate SHR systems. Because logic programming allows for many execution strategies and for complex interactions and data structures we need to limit it in order to keep only the computations which have an SHR correspondant. We will impose three kinds of restrictions:

- we select 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 graph rewritings (synchronized computations)

Definition 3.1 (Goal-graph) *We call goal-graph a goal which has no functional symbols and no constant symbols.*

Definition 3.2 (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*
- *each argument of the head of each rule is either a variable or a function symbol applied to variables*
- *if a variable is an argument of the head predicate then this variable also appears in the body of the clause*

Example 3.1

```
q(f(X),Y):-p(X,Y).
```

is a synchronized rule.

```
q(f(X),Y):-p(X,f(Y)).
```

is not a synchronized rule because `p(X,f(Y))` *is not a goal-graph.*

```
q(g(f(X)),Y):-p(X,Y).
```

is not a synchronized rule because it contains nested functions.

```
q(f(X),Y,f(Z)):-p(X).
```

is not a synchronized rule because Y is an argument of the head predicate but it does not appear in the body (we have not the same problem for Z because it is not a direct argument of the head predicate).

The first condition says that the result of a local rewriting must be a goal-graph. The second condition forbids complex synchronizations, which are not allowed in SHR. The last condition imposes that we cannot leave a node on which we do not perform any synchronizations. This condition has only the technical meaning of forbidding some rewritings in which an incorrect transition may not be caught 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 insert the variable in the body of the clause by adding an auxiliary predicate `u(-)` with the variable as argument and allowing it to do all the possible transitions.
For example we can rewrite last non synchronized rule as:

```
q(f(X),Y,f(Z)):-p(X),u(Y).
```

which is a synchronized rule.

Definition 3.3 (Synchronized logic programming) *We write:*
$P \Vdash G_1 \Rightarrow G_2$ *iff:*
$P \Vdash G_1 \rightarrow^* G_2$, *all steps performed in this computation expand different atoms of* G_1, G_1 *and* G_2 *are goal-graphs and* P *is a synchronized program.*
$G_1 \Rightarrow G_2$ *is called a big-step.*
All $\rightarrow$ *steps in a big-step are called small-steps.*
A synchronized logic programming computation is:
$G_1 \Rightarrow^* G_2$.

We will see later that a big-step is the logic correspondent of an SHR transition.

Example 3.2

Let P *be:*

```
q(f(X)):-p(X).
s(f(X),f(Y)):-s(X,Y).
s(g(X),g(Y)):-t(X,Y).
p(g(X)):-r(X).
```

Starting from the goal:

```
q(X),s(X,Y),q(Y)
```

we can have:

$$
\begin{aligned}
&q(X), s(X,Y), q(Y) \rightarrow \\
&\qquad \rightarrow p(X'), s(f(X'),Y), q(Y) \rightarrow \\
&\qquad \rightarrow p(X'), s(X',Y'), q(f(Y')) \rightarrow \\
&\qquad \rightarrow p(X'), s(X',Y'), p(Y') \rightarrow \\
&\qquad \rightarrow r(X''), s(g(X''),Y'), p(Y') \rightarrow \\
&\qquad \rightarrow r(X''), t(X'',Y''), p(g(Y'')) \rightarrow \\
&\qquad\qquad \rightarrow r(X''), t(X'',Y''), r(Y'')
\end{aligned}
$$

Because $p(X'), s(X',Y'), p(Y')$ *and* $r(X''), t(X'',Y''), r(Y'')$ *are goal-graphs we can rewrite this computation as:*

$$
\begin{aligned}
&q(X), s(X,Y), q(Y) \Rightarrow \\
&\qquad \Rightarrow p(X'), s(X',Y'), p(Y') \Rightarrow \\
&\qquad\qquad \Rightarrow r(X''), t(X'',Y''), r(Y'')
\end{aligned}
$$

which is obviously a synchronized logic programming computation.

3.2 Logic programming with restriction operator

An important aspect in process synchronization modelling is the distinction between what is observable from the outside of the system (global effects) and what is only visible within the system or within a part of it (local effects). In the logic programming world this distinction is less important because usually we are not modelling systems that interact with an environment. However there are two aspects of logic programming that are tightly related to this concept. Firstly, the result of the computation (computed answer substitution) is restricted to the variables of the goal, which means that other variables are 'local' and we are not interested in what happens on them. Moreover it is possible to specify that some variables are anonymous and also substitutions on these variables are not interesting. We want to deal with these problems in a precise and uniform way. In order to do that we will extend logic programming with the ν operator. This operator is of common use in the field of process models (CCS [14], π-calculus [15], SHR [10] [9] [7], ...) and it restricts a variable to be used only within its scope. We define for our language the grammar:

$$C ::= A : -G$$

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

where x is a variable, A is a logical atom and $\Box$ is the empty goal.
We assume that the restriction operator has the lowest precedence.
The restriction operator is a binder, then all the occurrences of x in $\nu x\ G$ are bound occurrences. Other occurrences of variables with the same name outside the scope of the binderer are occurrences of variables distinct from it.
We require that each free name in the body of the clause is a name of the head of the clause. In this way when a new free name appears, it has been created by some old name (i. e. assigned to it).

We consider clauses with restriction up to the following axioms:

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

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

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

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

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

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

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

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 variables that are not free (or even not present) has no influences (axioms (AL5) and (AL7)) and that restricted variables 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 2.28.

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 variables.

Before we can specify how a computation with restriction is performed we need to introduce some useful concepts.

Definition 3.4 (Old variable) *Let:*
$G \stackrel{\theta}{\rightarrow}^* G'$
be a computation. We call old variables of the computation the variables in G, new variables of the computation all the other variables used during the computation.

Definition 3.5 (Origin) *Let:*
$G \stackrel{\theta}{\rightarrow}^* G'$
be an SLD-computation.
For each variable x we define the set of origins of x with respect to θ as:
$\mathrm{or}_\theta(x) = (\theta^{-1}(x) \cup \{x\}) \cap \mathrm{var}(G)$
We will omit the subscript θ when clear from the context.

Intuitively the origins of a variable x are old variables x (or a complex term containing x) is assigned to.
Observe that because θ is idempotent we have $\mathrm{or}_\theta(x) \subseteq \{x\}$ if $x \in \mathrm{Dom}(\theta)$.

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 needs two updates. First, we have to hide substitutions on restricted variables. For this reason we define the restricted mgu as the mgu after the removal of these substitutions. Then we need to define which are the restricted variables in the new goal we compute. Because free variables are variables whose existence is known from the outside we state that a variable must be hidden if it has never been assigned to a free variable, i. e. if it has no free origins.

Definition 3.6 (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:

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

where $X = \mathrm{var}(B_1, \ldots, B_k) \setminus \mathrm{var}(H)$ *and* ρ *is an injective renaming of variables such that all variables in the clause variant* $\rho(H \leftarrow \nu X\ B_1, \ldots, B_k)$ *are new and* A *is an atom.*

$$\frac{P \Vdash G \xrightarrow[\theta]{\theta} F \qquad G, G' \textit{ do not contain restrictions}}{P \Vdash G, G' \xrightarrow[\theta]{\theta} F, G'\theta} \quad \textit{conjunctive goal}$$

$$\frac{P \Vdash G \xrightarrow[\theta]{\theta} F \qquad G \textit{ does not contain restrictions}}{P \Vdash \nu X\ G \xrightarrow[\theta]{\theta'} \nu X'\ F} \quad \textit{restricted goal}$$

where:
$X' = \{x \in \mathrm{fn}(F) | \, \mathrm{or}_\theta(x) \subseteq X\}$
$\theta' = \{[w/y] \in \theta | \exists z \in \mathrm{or}_\theta(y). z \notin X\}$
If $P \Vdash N \xrightarrow[\theta]{\theta'} N'$ *and the clause variant used is* C *we say that* N' *is the resolvent of* N *and* C *with mgu* θ *and restricted mgu* θ'*.*
We will omit $P \Vdash$ *if* P *is clear from the context.*
A partial SLD-derivation with restriction of $P \cup \{G\}$ *is a sequence (eventually empty) of such steps allowed by program* P *with* G *as first goal.*
If the sequence is empty we can assume that both the mgu and the restricted mgu are empty substitutions.

Note that thanks to the axioms of congruence we have been able to simplify the rules with respect to definition 2.25. Observe that the mgu does not depend on the restrictions we have. Restrictions affect only restricted mgu.

Definition 3.7 (Extrusion) *We have an extrusion of a variable* x *if* x *is a bound variable and* x *or a term containing it is assigned to a free variable.*

When we have an extrusion the extruded variable becomes visible from the outside. Observe that no free occurrences of the variable may exist because we have only restrictions on the whole goal, so there is not the problem of undesired merges between extruded variables and previously free variables with the same name.

Definition 3.8 (Origin) *We extend the definition 3.5 of origins to restricted computations. Let:*
$G \xrightarrow[\theta_{free}]{\theta}{}^{*} G'$
be a restricted computation.
For each variable x we define the set of origins of x with respect to θ as:
$\mathrm{or}_\theta(x) = (\theta^{-1}(x) \cup \{x\}) \cap \mathrm{fn}(G)$

Observe that if in the computation we have no restrictions this definition coincides with the previous one.

Example 3.3

Let P be:

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

Starting from the goal:

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

we can perform the following computation:

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

3.3 Properties of restricted computations

In this section we will study the properties of computations with restriction that will be useful later to prove the equivalences between operational and tile semantics of logic programs with restriction and between synchronized logic programming with restriction and synchronized hyperedge replacement.
First observe that a step starting from the goal $\nu X\ G_1, \ldots, G_n$ which unifies G_i with a clause variant $A : -B$ produces as new goal:

$\nu X'\ (G_1, \ldots, G_{i-1}, B, G_{i+1}, \ldots, G_n)\theta$
where θ is the mgu of the step.

The next lemma is a technical lemma that states properties of the use of names in computations.

Lemma 3.1 *Let:*
$G_0 \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n} G_n$ *be a computation with restriction.*
For each i, j *such that* $1 \leq i < j \leq n$ *we have:*
$\mathrm{Dom}(\theta_i) \cap \mathrm{var}(\theta_j) = \emptyset$.
Proof
$G_i = \nu X\ G'_{i-1}\theta_i$
where G'_{i-1} *is obtained from* G_{i-1} *by substituting the body of the clause variant used to the unified atom.*
Because θ_i *is idempotent we have:*
$\mathrm{Dom}(\theta_i) \cap \mathrm{var}(G_i) = \emptyset$.
We can observe that the variables in $\mathrm{Dom}(\theta_i)$ *are not reintroduced in the computation because at each step all the variables come from the previous goal (which does not contain such variables) or from the clause variant (because we use relevant mgus), which contains only new variables by definition. The thesis follows.*

We want to study the effect of restriction on multi-step computations. We start by studying the properties of the inverse of a composed substitution.

Lemma 3.2 *Let* θ_1 *and* θ_2 *be substitutions and* $x \in \mathrm{Im}(\theta_1\theta_2)$.
$(\theta_1\theta_2)^{-1}(x) = \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i)$.
Proof
We have:

$$
\begin{aligned}
& y \in (\theta_1\theta_2)^{-1}(x) \Rightarrow \\
& \Rightarrow x \in \mathrm{var}(y\theta_1\theta_2) \Rightarrow \\
& \Rightarrow (x \in \mathrm{var}(y\theta_1) \wedge x \notin \mathrm{Dom}(\theta_2)) \vee \exists x_i \in \mathrm{var}(y\theta_1).x \in \mathrm{var}(x_i\theta_2) \Rightarrow \\
& \Rightarrow (x \in \mathrm{var}(y\theta_1) \wedge x\theta_2 = x) \vee \exists x_i \in \mathrm{var}(y\theta_1).x \in \mathrm{var}(x_i\theta_2) \Rightarrow \\
& \Rightarrow \exists x_i \in \mathrm{var}(y\theta_1).x \in \mathrm{var}(x_i\theta_2) \Rightarrow \\
& \Rightarrow \exists x_i.y \in \theta_1^{-1}(x_i) \wedge x_i \in \theta_2^{-1}(x) \Rightarrow \\
& \Rightarrow y \in \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i)
\end{aligned}
$$

We also have:

$$
\begin{aligned}
& y \in \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i) \Rightarrow \\
& \qquad \Rightarrow \exists x_i \in \theta_2^{-1}(x) . y \in \theta_1^{-1}(x_i) \Rightarrow \\
& \qquad \Rightarrow \exists x_i . x_i \in \mathrm{var}(y\theta_1) \wedge x \in \mathrm{var}(x_i\theta_2) \Rightarrow \\
& \qquad \Rightarrow x \in \mathrm{var}(y\theta_1\theta_2) \Rightarrow \\
& \qquad \Rightarrow y \in (\theta_1\theta_2)^{-1}(x)
\end{aligned}
$$

We can now find a similar result for the composition of origins.

Lemma 3.3 *Let:*
$G \xrightarrow{\theta_1}{}^{*} G' \xrightarrow{\theta_2}{}^{*} G''$
be a computation with restriction and $x \in \mathrm{var}(G'')$.
$\mathrm{or}_{\theta_1\theta_2}(x) = \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i)$.

Proof

We want to show that $y \in \mathrm{or}_{\theta_1\theta_2}(x)$ *iff* $y \in \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i)$.
There are two cases: either $y = x$ *or* $y \neq x$.
In the first case y *is obviously in both sets iff* $y \in \mathrm{fn}(G)$, *in no sets otherwise.*
In the second case we have x *in* $\mathrm{Im}(\theta_1\theta_2)$ *and so for lemma 3.2:*

$$
\begin{aligned}
& y \in \mathrm{or}_{\theta_1\theta_2}(x) \Rightarrow \\
& \qquad \Rightarrow y \in (\theta_1\theta_2)^{-1}(x) \Rightarrow \\
& \qquad \Rightarrow y \in \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i) \Rightarrow \\
& \qquad \Rightarrow y \in \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i)
\end{aligned}
$$

Observe that we have $x_i \in \mathrm{var}(G')$ *because* $y \in \mathrm{var}(G)$.
Suppose now $y \in \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i)$.
Observe that either $y \neq x_i$ *or* $x_i \neq x$. *If* $y \neq x_i \neq x$ *then we can apply lemma 3.2:*

$$
\begin{aligned}
& y \in \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i) \Rightarrow \\
& \qquad \Rightarrow y \in \bigcup_{x_i \in \theta_2^{-1}(x)} \theta_1^{-1}(x_i) \Rightarrow \\
& \qquad \Rightarrow y \in (\theta_1\theta_2)^{-1}(x) \Rightarrow \\
& \qquad \Rightarrow y \in \mathrm{or}_{\theta_1\theta_2}(x)
\end{aligned}
$$

If $y = x_i$ *(and so* $x_i \neq x$*) we have* $y \in \mathrm{or}_{\theta_2}(x)$*, what means* $x \in \mathrm{var}(y\theta_2)$*. We also have* $x \in \mathrm{var}(y\theta_1\theta_2)$ *if* $y \notin \mathrm{Dom}(\theta_1)$*, but this is true because* $y \in \mathrm{var}(\theta_2)$ *and for lemma 3.1.*
If $x_i = x$ *(and so* $y \neq x_i$*) we have* $y \in \mathrm{or}_{\theta_1}(x)$*, what means* $x \in \mathrm{var}(y\theta_1)$*. We also have* $x \in \mathrm{var}(y\theta_1\theta_2)$ *if* $x \notin \mathrm{Dom}(\theta_2)$*, but this is true because otherwise* $x \notin \mathrm{var}(G'')$*.*

Now we can prove some properties of the restricted mgu.

Lemma 3.4 *Let:*
$G \xrightarrow[\theta]{\theta'}{}^{*} G'$
be a computation with restriction. Then $\theta'|_{\mathrm{fn}(G)} = \theta|_{\mathrm{fn}(G)}$ *and all the other bindings in* θ' *have as domain variable a variable that never appears neither in* G *nor in* G' *or that is an extruded variable.*
Proof
We will show it by induction on the length of the computation.
0)
We have:
$G \xrightarrow[\epsilon]{\epsilon} G$
which obviously satisfies the thesis.
Inductive step)
We have:
$G \xrightarrow[\theta_1]{\theta_1'}{}^{*} G'' \xrightarrow[\theta_2]{\theta_2'} G'$
Bindings in θ *are all bindings of the form* $[\theta_2 w/y]$ *where* $w/y \in \theta_1$ *and of the form* $[w/y] \in \theta_2$ *where* $y \notin \mathrm{Dom}(\theta_1)$*.*
If $y \in \mathrm{fn}(G)$ *then all bindings with domain variable* y *are also in the restricted mgus, moreover bindings with a domain variable* z *in* w *are also in the restricted mgus because* z *is free in* G'' *because* $y \in \mathrm{or}_{\theta_1}(z)$*.*
Other bindings of the first form have a domain variable that is not in G *for inductive hypothesis. Let* $[w/y]$ *be a binding of the second form. If* y *is in* $\mathrm{var}(G)$ *then it is also in* $\mathrm{var}(G'')$*. Moreover* y *is free in* G''*. If it is free also in* G *then we have the thesis. Otherwise it has been extruded.*
Variables in the domain of θ' *cannot appear in* G' *for idempotence.*
This proves the thesis.

Finally we can analyze how the restrictions on a goal propagate to the goals computed from it.

Lemma 3.5 *Let* P *be a program without restrictions.*
In P *we have the computation with restriction:*
$\nu Y\ G \xrightarrow[\theta]{\theta''}{}^{*} \nu Y'\ G'$

iff we have the computation with restriction:
$\nu X \cup Y\ G \xrightarrow[\theta]{\theta'}{}^{*} \nu X' \cup Y'\ G'$
where:
$X' = \{x \in \mathrm{fn}(\nu Y'\ G') |\, \mathrm{or}_{\theta''}(x) \subseteq X\}$
$Y' = \{x \in \mathrm{fn}(G') |\, \mathrm{or}_{\theta}(x) \subseteq Y\}$
Proof
Suppose firstly that $Y = \emptyset$. In this case we also have $Y' = \emptyset$ and $\theta = \theta''$. We will show the thesis in this case by induction on the length of the computation.
0)
In all P we have:
$G \xrightarrow{\epsilon} G$
$\nu X\ G \xrightarrow[\epsilon]{\epsilon} \nu X\ G$
and so the thesis holds trivially.
Inductive step)
We have:
$G \xrightarrow{\theta_1} G'' \xrightarrow{\theta_2}{}^{*} G'$
We can do the first step iff:
$\nu X\ G \xrightarrow[\theta_1]{\theta'_1} \nu X''\ G''$
where:
$X'' = \{x \in \mathrm{fn}(G'') |\, \mathrm{or}_{\theta_1}(x) \subseteq X\}$
By inductive hypothesis we can do the second step iff:
$\nu X''\ G'' \xrightarrow[\theta_2]{\theta'_2}{}^{*} \nu X'\ G'$
where:
$X' = \{x \in \mathrm{fn}(G') |\, \mathrm{or}_{\theta_2}(x) \subseteq X''\}$
We have for lemma 3.3:

$$\begin{aligned} X' = \{x \in \mathrm{fn}(G') |\, \mathrm{or}_{\theta_2}(x) \subseteq X''\} = \\ = \{x \in \mathrm{fn}(G') | \forall x_i \in \mathrm{or}_{\theta_2}(x).\, \mathrm{or}_{\theta_1}(x_i) \subseteq X\} = \\ = \{x \in \mathrm{fn}(G') |\, \bigcup_{x_i \in \mathrm{or}_{\theta_2}(x)} \mathrm{or}_{\theta_1}(x_i) \subseteq X\} = \\ = \{x \in \mathrm{fn}(G') |\, \mathrm{or}_{\theta_1 \theta_2}(x) \subseteq X\} \end{aligned}$$

and so we have the thesis in this case.
Let us consider the general case. For what already proved we have:
$\nu Y\ G \xrightarrow[\theta]{\theta''}{}^{*} \nu Y'\ G'$ *iff:*
$G \xrightarrow[\theta]{\theta}{}^{*} G'$ *where:*
$Y' = \{x \in \mathrm{fn}(G') |\, \mathrm{or}_{\theta}(x) \subseteq Y\}$
and also:

$\nu X \cup Y\ G \xrightarrow[\theta]{\theta'}{}^{*}\ \nu R'\ G'$
where:
$R' = \{x \in \mathrm{fn}(G') |\ \mathrm{or}_{\theta}(x) \subseteq X \cup Y\}$
Recall that:
$X' = \{x \in \mathrm{fn}(\nu Y'\ G') |\ \mathrm{or}_{\theta''}(x) \subseteq X\}$
For lemma 3.4:
$\theta|_{\mathrm{fn}(\nu Y\ G)} = \theta''|_{\mathrm{fn}(\nu Y\ G)}$
Suppose $x \in \mathrm{fn}(\nu Y'\ G')$. *In this case:*

$$\begin{aligned}
& y \in \mathrm{or}_{\theta}(x) \Leftrightarrow \\
& \Leftrightarrow y \in \mathrm{var}(\nu Y\ G) \wedge (x \in y\theta \vee x = y) \Leftrightarrow \\
& \Leftrightarrow (y \in \mathrm{fn}(\nu Y\ G) \vee y \in Y) \wedge (x \in y\theta \vee x = y) \Leftrightarrow \\
& \Leftrightarrow (y \in \mathrm{fn}(\nu Y\ G) \wedge (x \in y\theta'' \vee x = y)) \vee (y \in Y \wedge (x \in y\theta \vee x = y)) \Leftrightarrow \\
& \Leftrightarrow y \in \mathrm{or}_{\theta''}(x) \vee (y \in Y \wedge y \in \mathrm{or}_{\theta}(x))
\end{aligned}$$

Thus:
$X' = \{x \in \mathrm{fn}(\nu Y'\ G') |\ \mathrm{or}_{\theta}(x) \subseteq X \cup Y\}$
Clearly $x \in X' \cup Y' \Rightarrow x \in R'$.
Suppose now $x \in R'$. *If* $x \in Y'$ *then we have* $x \in X' \cup Y'$. *Otherwise* $x \in \mathrm{fn}(\nu Y'\ G')$. *But in this case* $x \in X'$. *In any case* $x \in X' \cup Y'$, *and so* $R' = X' \cup Y'$. *This proves the thesis.*

We can also analyze the effects of the restrictions that are in the program clauses.

Lemma 3.6 *Let P be a program with restrictions, P_{free} be the same program from which we have eliminated the restrictions (after choosing for the restricted variables a representative that is not an already existent free variable of the same clause). In P_{free} we have the computation:*
$\nu X\ G \xrightarrow[\theta]{\theta'}{}^{*}\ \nu X'\ G'$
where we do not rewrite atoms not in G iff in P we have the computation:
$\nu X\ G \xrightarrow[\theta]{\theta'}{}^{*}\ \nu X' \cup Y\ G'$
where Y is the set of restricted variables in the clause variants used during the computation.

Proof
We will prove it by induction on the length of the computation.
0)
The computatation:
$G \xrightarrow{\epsilon} G$
can happen in P and in P_{free}. So the thesis holds trivially.

Inductive step)
Suppose that in P_{free} we have:
$\nu X\ G \xrightarrow[\theta_1]{\theta'_1} \nu X''\ G'' \xrightarrow[\theta_2]{\theta'_2}{}^{*}\ \nu X'\ G'$
We can have the first step iff in P we have:
$\nu X\ G \xrightarrow[\theta_1]{\theta'_1} \nu X'' \cup Y''\ G''$
where Y'' is the set of restricted variables in the clause variant used.
For lemma 3.5 we have the second step iff:
$G'' \xrightarrow{\theta_2}{}^{*}\ G'$
and iff:
$\nu X'' \cup Y''\ G'' \xrightarrow[\theta_2]{\theta'_2}{}^{*}\ \nu R\ G'$
where:
$R = \{x \in \mathrm{fn}(G') | \, \mathrm{or}_{\theta_2}(x) \subseteq X'' \cup Y''\}$
Recall that:
$X' = \{x \in \mathrm{fn}(G') | \, \mathrm{or}_{\theta_2}(x) \subseteq X''\}$
and so $X' \subseteq R$.
Suppose $x \in R \wedge x \notin X'$.
This implies $\exists z \in \mathrm{or}_{\theta_2}(x).z \in Y''$ but the variables in Y'' are used only in already rewritten atoms, thus $z \notin \mathrm{var}(\theta_2)$ and $x = z$.
Because all the variables in Y'' are in R we have $R = X' \cup Y''$.
By inductive hypothesis we can have the previous step iff in P:
$\nu X'' \cup Y''\ G'' \xrightarrow[\theta_2]{\theta'_2}{}^{*}\ \nu X' \cup Y'' \cup Y'\ G'$
where Y' is the set of restricted variables in the clause variants used in the second part of the computation.
Thus $Y'' \cup Y'$ is the set of restricted variables in the clause variants used in the whole computation and so we have the thesis.

3.4 Tile semantics of logic programming with restriction

In this section we will define two styles of compositional semantics for logic programming with restriction. In order to do that we will rely on the framework of tile logic and we will extend the work of [4] for logic programming to logic programming with restriction.
The first semantic style has the advantage of requiring very little work to extend all the results of [4] we are interested in and of using standard categorical tools, but it produces a tile system in which restrictions are part of the observation and not of the configuration, what looks strange and what forbids a proof of compositionality

w. r. t. variable restriction using theorem 2.7. The second approach has complementary characteristics.
Let us start from the first approach.
We have to model the behaviour of restrictions.
Observe that in definition 2.52 the variables that appear in the head of a clause but not in its body (and so that would be restricted in logic programming with restriction) are introduced using dischargers in the observation.
Thus for the axioms of cartesian category observations on them disappear.
This is exactly what we want for substitutions on restricted variables. Thus we can say that a variable is restricted iff it is created by a discharger. This is all what we need to model restrictions in the clauses.
In order to handle restrictions in the goal we can suppose that the restricted goal:

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

is generated in a step from the unrestricted goal $g(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are all the free variables of our starting goal, using the clause:

$$g(x_1, \ldots, x_n) \to \nu X \; G_1, \ldots, G_n$$

Thus we can always suppose that the starting goal does not contain restrictions.
We can now prove that this semantics correctly handles restriction.

Theorem 3.1 (Correspondence between logic programming with restriction and tiles(1)) *Let P be a logic program with restriction and G a goal without restrictions. Then:*

1. *if we have $P \Vdash G \xrightarrow{\sigma} G'$ then $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'_{free}$ where $\theta = \sigma|_{\mathrm{fn}(G)}$, G'_{free} is equal to G' after the removal of all restrictions and all the restricted variables of G' correspond to the variables that are created by a discharger in the observation.*

2. *if $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'_{free}$, then there exists σ with $\theta = \sigma|_{\mathrm{fn}(G)}$ such that we have $P \Vdash G \xrightarrow{\sigma} G'$ where G'_{free} is equal to G' after the removal of all restrictions and all the restricted variables of G' correspond to the variables that are created by a discharger in the observation.*

Proof

1. *Observe that the translation of the clauses into tiles is the same as the one for the same clauses after the removal of all the restrictions. Thus for theorem 2.10 we have:*
   $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'_{free}$ *with* $\theta = \sigma|_{\mathrm{fn}(G)}$.
   Observe that all the variables in G' are in G'_{free}.

As $\theta = \sigma|_{\mathrm{fn}(G)}$ *all the variables bound in* G' *(and only these variables) must disappear from the observation, what happens only if they are created by a discharger.*

2. *For theorem 2.10 there exists* σ *with* $\theta = \sigma|_{\mathrm{fn}(G)}$ *such that we have* $P \Vdash G \xrightarrow{\sigma} G'$ *and if we remove all the restrictions from* G' *we have* G'_{free}*, what proves the first part of the thesis. For lemma 3.5 a variable is restricted in* G' *iff its origins are all bound, i. e. they are all created by a discharger. For the axioms of cartesian category then also this variable is created by a discharger, so the thesis follows.*

The extension of theorem 2.11 is straightforward because we have only changed the interpretation of the semantics, but not the tile system itself. Thus bisimilarity is a congruence w. r. t. goal composition and instantiation of free variables for goals without restrictions. This result is also simply extended to the general case for instantiation of free variables because the instantiated goal can be generated in one step from an instance of the original goal without restrictions. The same can be done with some little updates for goal composition only if the sets of restricted variables in the two goals are disjoint (in this case we can find a goal without restrictions that is composed by two parts, each of which generates in one step one of the two goals with restrictions). Moreover the operation of restriction of a variable is neither an instantiation nor a contextualization because restrictions do not appear in the configuration, so in this way we have not proved bisimilarity to be a congruence w. r. t. this operation.

We will now present the second approach.
First of all, we will extend the tile system of definition 2.52 to handle the restriction operator.
In this approach the restriction operator $\nu : \epsilon \to t$ will become a component of our system, and it will be introduced in the signature of configurations.
Moreover we need some tiles that specify the behaviour of this operator.

Definition 3.9 (Restriction tile) *We call restriction tiles the tiles in figures 3.1, 3.2 and 3.3.*
Observe that the tiles for constants and functions must be provided for all constants and functions in the signature.
We have no need to introduce a restriction tile for the identity operator because it coincides with the vertical identity of the ν *operator. We will however call it a restriction tile. The same holds for the restriction tiles of a permutation, which are auxiliary tiles in our model (we have represented them in figure 3.4).*

Figure 3.1: Restriction of a constant

$$\begin{tikzcd} \cdot & \cdot \arrow[l, "\nu"'] \\ \cdot \arrow[u, "a"] & \cdot \arrow[l, "id_0"] \arrow[u, "id_0"'] \end{tikzcd}$$

Figure 3.2: Restriction of a function

$$\begin{tikzcd} \cdot & \cdot \arrow[l, "\nu"'] \\ \cdot \arrow[u, "f"] & \cdot \arrow[l, "\nu^n"] \arrow[u, "id_0"'] \end{tikzcd}$$

Figure 3.3: Restriction of a duplicator

$$\begin{tikzcd} \cdot & \cdot \arrow[l, "\nu \otimes id"'] \\ \cdot \arrow[u, "\nabla"] & \cdot \arrow[l, "id"] \arrow[u, "id"'] \end{tikzcd}$$

Figure 3.4: Restrictions of a permutation

$$\begin{tikzcd} \cdot & \cdot \arrow[l, "\nu \otimes id"'] \\ \cdot \arrow[u, "\gamma"] & \cdot \arrow[l, "id \otimes \nu"] \arrow[u, "id"'] \end{tikzcd} \qquad \begin{tikzcd} \cdot & \cdot \arrow[l, "id \otimes \nu"'] \\ \cdot \arrow[u, "\gamma"] & \cdot \arrow[l, "\nu \otimes id"] \arrow[u, "id"'] \end{tikzcd}$$

We will now extend definition 2.52 to include also programs and goals with restriction.

Definition 3.10 (Tile system for logic programming with restriction) *Given a logic program with restriction P on the alphabet Σ, we denote by $\mathcal{R}_P$ the tile system specified by the following rules:*

- *The sorts are the sames of definition 2.52.*
- *We add to the signature of configurations an operator $\nu : \epsilon \to t$.*
- *We have the same observations as in definition 2.52.*
- *To each clause* $c \equiv p(t_1, \ldots, t_k) \leftarrow \nu X \;\; q_1(\vec{s}_1) \wedge \cdots \wedge q_m(\vec{s}_m)$ *(over n variables $\{x_1, \ldots, x_n\}$, r of which are unrestricted) in the logic program we associate a basic tile T_c in our system whose initial configuration is $p : t^k \to p$ (representing the predicate symbol in the head of c), whose final configuration is $res; q_1(\vec{s}_1) \wedge \cdots \wedge q_m(\vec{s}_m) : t^r \to p$ (representing the body of the clause) where res is the parallel of identities for free variables and ν operators for restricted variables, whose trigger is the identity $id_p : p \to p$ and whose effect is the tuple $\langle t_1, \ldots, t_k \rangle : t^r \to t^k$ (representing the pattern to be matched by the arguments of predicate p). Since the same variable can be used more than once, duplicators can be necessary. Since we take the op-categories the direction of all arrows is reversed w. r. t. the standard representation.*
- *Finally, we add the basic tiles contained in $\mathcal{R}_{PB(\Sigma)}$ to calculate the mgus and the restriction tiles.*

Notice that our observations are elements of the algebraic theory on our signature. These elements are like substitutions but also identity renamings are allowed and they do not keep the actual names (and so we cannot distinguish injective renamings from identities).
So we can extend the definition of origins asking that orgins are in the domain of the substitution term instead of in G (see definition 3.5). This new definition has the same properties of the old one because all the variables in the goal are in the domain of the unrestricted observation.
This means that the origins of the variables in the image of the substitution term coincide with the image of the inverse of the substitution term.

Example 3.4

Consider the program:

```
p(f(X,Y),Y,g(Z)):-q(X,Y,Z).
```

Starting from the goal:

ν`{Y,Z} p(X,Y,g(Z))`

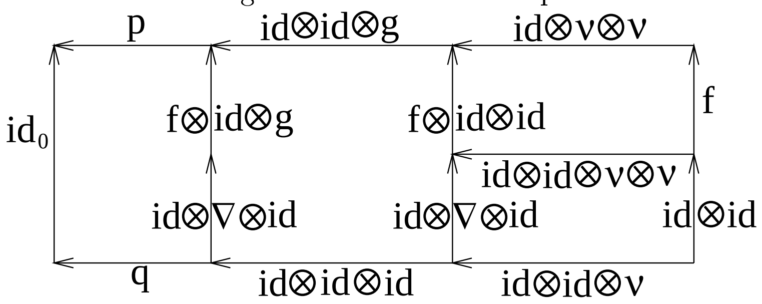
we can have the following transition:

$$\nu\{Y,Z\}\ p(X,Y,g(Z)) \xrightarrow[X=f(X',Y'),Y=Y',Z=Z']{X=f(X',Y')} \nu Z'\ q(X',Y',Z')$$

which is proved by the tile in figure 3.5.

The leftmost tile is a program tile, the central tile is an mgu tile while the two

Figure 3.5: Tile for a computation with restriction



rightmost tiles are restriction tiles. The upper one is composed by the horizontal identity of f and two vertical identities of ν. The lower one is made up of identities and the symmetric of the restriction of a duplicator. We can build this tile using the restriction of a duplicator and the restriction of a symmetry because we have $\nabla_a = \nabla_a; \gamma_{a,a}$.

We must show that the proposed tile system really models computations with restriction. We start by studying the tiles that can be built with restrictions as configurations and generic substitutions as triggers.

Lemma 3.7 *For each substitution term σ and each parallel composition of identities and restrictions res the only tile with trigger σ and initial configuration res entailed by $\mathcal{R}_P$ for any P is $res \xrightarrow[\sigma']{\sigma} res'$ where σ' is the restriction of σ to the variables not restricted by res and res' is a parallel composition of identities and restrictions where x is restricted iff $\mathrm{or}_\sigma(x)$ is a subset of the set of variables restricted by res.*

Proof

Observe that because we have as only initial configurations identities and restrictions we can only use as tiles horizontal identities and restriction tiles.

Notice, too, that given an initial configuration and a trigger we have no choices in which tile to use at each step. So, if the built tile exists, it is unique.
Observe also that for each trigger and each initial configuration we have an applicable tile:

- *if the initial configuration is an identity we can apply the horizontal identity of the trigger.*
- *if the initial configuration is a restriction we can apply a restriction tile.*
  Observe that the trigger cannot contain dischargers because all the variables are created by ν.
  Notice, too, that if the trigger uses two variables of the initial configuration, we can always suppose that one of them does an identity transition, because if we have two restrictions we can use $\nu \otimes \nu = \nu \otimes id; \nu$.
- *we can compose in the three dimensions the tiles in order to deal with compositions of triggers and of initial configurations.*

We now need to prove the correctness of the construction.
The correctness for basic tiles is obvious.
We have the correctness of parallel composition because if we have $res_1 \xrightarrow[\sigma_1']{\sigma_1} res_1'$ and $res_2 \xrightarrow[\sigma_2']{\sigma_2} res_2'$ the parallel composition is $res_1 \otimes res_2 \xrightarrow[\sigma_1' \otimes \sigma_2']{\sigma_1 \otimes \sigma_2} res_1' \otimes res_2'$.
The variables unrestricted in $res_1 \otimes res_2$ are either unrestricted in res_1 or in res_2 and so their bindings appear in either res_1' or res_2'.
The origins of variables in res_1' are all in σ_1, the origins of variables in res_2' are all in σ_2 and so a variable is restricted in $res_1' \otimes res_2'$ only if it is restricted in res_1' if it is a variable in σ_1 and in res_2' otherwise.
The correctness of horizontal composition comes from lemma 3.5.
We have the correctness of vertical composition from lemma 3.2.

We can now prove the correspondence between the proposed tile system and logic programming with restriction.

Theorem 3.2 (Correspondence between logic programming with restriction and tiles(2)) *Let P be a logic program with restriction and G a goal with restriction. Then:*

1. *if we have $P \Vdash G \xrightarrow{\sigma} G'$ then $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'$ [1] with $\theta = \sigma|_{\mathrm{fn}(G)}$*
2. *if $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'$, then there exists σ with $\theta = \sigma|_{\mathrm{fn}(G)}$ such that we have $P \Vdash G \xrightarrow{\sigma}{}^{*} G'$*

[1] Naturally here we use the categorical representation of goals.

Proof

1. *By hypothesis we have* $G \xrightarrow[\sigma_{free}]{\sigma} G'$. *This means that if we remove all restrictions we have:*
   $G_{free} \xrightarrow{\sigma_{free}} G'_{free}$.
   For theorem 2.10 we have:
   $\mathcal{R}_P : G_{free} \xrightarrow[\theta]{id} G'_{free}$ *with* $\theta = \sigma_{free}|_{\mathrm{fn}(G_{free})}$.
   For lemma 3.7 we have:
   $\mathcal{R}_P : res \xrightarrow[\theta']{\theta} res'$ *with* $\theta' = \theta|_{\mathrm{fn}(G)} = \sigma_{free}|_{\mathrm{fn}(G)} = \sigma|_{\mathrm{fn}(G)}$
   for lemma 3.4.
   We also have that res' *restricts all the variables that are restricted in* G'.
   We can now use sequential composition to prove:
   $\mathcal{R}_P : res; G_{free} \xrightarrow[\theta']{id} res'; G'_{free}$
   which is the thesis.

2. *By hypothesis we have* $\mathcal{R}_P : G \xrightarrow[\theta]{id} G'$, *and because* $G = res; G_{free}$ *and all the tiles with restrictions as initial configuration cannot have other non identity pieces of initial configuration we can decompose the tile in* $G_{free} \xrightarrow[\theta']{id} G'_{free}$ *and*
   $res \xrightarrow[\theta]{\theta'} res'$.
   For theorem 2.10 we have that there exists σ_{free} *with* $\theta' = \sigma_{free}|_{\mathrm{var}(G_{free})}$ *such that we have* $P \Vdash G_{free} \xrightarrow{\sigma_{free}} G'_{free}$.
   For lemma 3.7 we have that $\theta = \theta'|_{\mathrm{fn}(G)} = \sigma_{free}|_{\mathrm{fn}(G)} = \sigma|_{\mathrm{fn}(G)}$ *(where we used lemma 3.4) and that* res' *is the restriction of the variables whose origins are restricted by* res.
   For lemma 3.5 we have:
   $G \xrightarrow[\sigma_{free}]{\sigma} G'$
   where $G' = res'; G'_{free}$ *and the thesis.*

Now we want to show tile bisimilarity to be a congruence. We will use the criterium of theorem 2.7. We will show our system to be in the theorem hypotheses. The idea of the lemma is that each tile can be decomposed because the goal part can be decomposed for lemma 2.1 and the restriction part can be decomposed because the restriction tiles have an only non identity symbol in their configurations.

Lemma 3.8 *For any logic program with restriction* P, *the corresponding tile system* $\mathcal{R}_P$ *enjoys the sequential decomposition property.*
Proof
We have to prove that for all configurations t *and for all sequents* $t \xrightarrow[v]{u} s$ *entailed by* $\mathcal{R}_P$, *then if* $t = t_1; t_2$ *then* $\exists w, s_1, s_2$ *such that* $\mathcal{R}_P : t_1 \xrightarrow[w]{u} s_1$, $\mathcal{R}_P : t_2 \xrightarrow[v]{w} s_2$ *and*

$s = s_1; s_2$.
Here t is of the form $res; G$, s is of the form $res'; G'$.
Observe that res (resp. res') has an identity or a restriction for each object in the input interface of G (resp. G').
This means that in each decomposition $t_1; t_2$ of t we have that both t_1 and t_2 are composed by two parts, a restriction part and a goal part. Let us call res_1 and res_2 the restriction parts and G_1 and G_2 the goal parts.
For lemma 2.1 we have that $\exists \sigma_0, G'_1, G'_2$ such that $\mathcal{R}_P : G_1 \xrightarrow[\sigma_0]{id} G'_1$, $\mathcal{R}_P : G_2 \xrightarrow[\sigma]{\sigma_0} G'_2$ and $G' = G'_1; G'_2$.
Now we can attach to these tiles the res part and the restriction tiles corresponding to the symbols in σ_0. This is always possible because all the restriction tiles have an only symbol as trigger.
So we have that $\exists \sigma'_0, res'_1, res'_2$ such that:
$\mathcal{R}_P : res_1; G_1 \xrightarrow[\sigma'_0]{id} res'_1; G'_1$
$\mathcal{R}_P : res_2; G_2 \xrightarrow[\sigma']{\sigma'_0} res'_2; G'_2$.
This is the wanted decomposition.

Definition 3.11 *Let P be a logic program with restriction and G and G' two goals. $G \cong_P G'$ iff G and G' are tile bisimilar in $\mathcal{R}_P$.*

Theorem 3.3 (Bisimilarity is a congruence) *For any logic program with restriction P, the equivalence $\cong_P$ is a congruence with respect to conjunction of goals and restriction and instantiation of free variables.*
Proof
It follows from lemma 3.8 and theorem 2.7 (we need only sequential decomposition because conjunction is an operation of contextualization while restriction and clearly instantiation of free variables are operations of instantiation).

This means that $G \cong_P G'$ implies $G \wedge G'' \cong_P G' \wedge G''$,$G\sigma \cong_P G'\sigma$ and $\nu X\ G \cong_P \nu X\ G'$ for each goal G'', each substitution σ and each set of variables X.

One may be surprised that restriction is an operation of instantiation, however the fact that restrictions and bindings for the variables are both in the right part of the tile makes it clear. Intuitively we can think that asking a variable to be new (as the restriction does) is a kind of instantiation.

Chapter 4

Two translations from SHR to logic programming

In this chapter we present two translations from synchronized hyperedge replacement (SHR) systems to synchronized logic programming with restriction, together with the related results of correctness and completeness. First of all, we define the translation for SHR without extrusions (we call it in this way because in this SHR style it is forbidden to send an old name during synchronization, thus a bound name cannot become free). Then we define full Hoare SHR (only the Milner style can be found in literature [7]) and we extend our results to this more general case. We have skipped SHR without fusions among old nodes because in this SHR system there are different behaviours for new and old nodes whereas this difference does not exist in logic programming (in order to handle it we have to transform all the variables into constants at the end of each big-step).

4.1 Mapping from SHR without extrusions to logic programming

We want to use synchronized logic programming with restriction to simulate SHR systems. In order to do that we need firstly to translate hypergraphs, as we have defined them as syntactic judgements, to goals with restriction. In this translation we will lose the part Γ of syntactic judgements.

Definition 4.1 (Translation for syntactic judgements) *We define the translation operator* $[\![-]\!]$ *as:*

$$[\![\Gamma \vdash L(x_1, \dots, x_n)]\!] = L(x_1, \dots, x_n)$$

$$[\![\Gamma \vdash G_1|G_2]\!] = [\![\Gamma \vdash G_1]\!], [\![\Gamma \vdash G_2]\!]$$

$$[\![\Gamma \vdash \nu X\ G]\!] = \nu X\ [\![\Gamma, X \vdash G]\!]$$

$$[\![\Gamma \vdash nil]\!] = \Box$$

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

Definition 4.2 *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 there exists a choice of the representatives of the two graphs such that $G \equiv G'$ where $\equiv$ is the syntactical identity.*

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 4.1 (Correspondence of judgements and goal-graphs) *The operator $[\![-]\!]$ defines an isomorphism between judgements up to $\cong$ and goal-graphs.*
Proof
The proof is straightforward observing that syntactic rules for graph judgements impose only conditions on Γ, the operator $[\![-]\!]$ defines a bijection between representatives of syntactic judgements and representatives of goal-graphs and the congruence on the two structures is essentially the same.

Lemma 4.1 *Let G be a graph and θ be a substitution. Then:*

$$[\![G]\!]\theta = [\![G\theta]\!]$$

Proof
By rule induction.

We now define a translation also for rewriting rules. We start with rules in the form of definition 2.29. In order to do that we overload the $[\![-]\!]$ operator.

Definition 4.3 (Translation for rewriting rules without extrusions) *We define the translation operator $[\![-]\!]$ as:*

$$[\![L(x_1, \dots, x_n) \xrightarrow{\Lambda} G]\!] = L(a_1(x_1, \vec{y}_1), \dots, a_n(x_n, \vec{y}_n)) :- [\![G]\!]$$

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

Observe that there is no need to translate all the possible variants of the rules because when we perform computations with restriction we use variants of clauses and so the actual names we use in the program are not important.

The idea of the translation is that if an edge makes some actions and it creates some new nodes and it is rewritten into a graph G, then the corresponding predicate is matched with a clause head that has for each node a function symbol corresponding to the action (thus all the synchronizations on the node must be the sames as required for Hoare synchronization) and as arguments the node itself (otherwise it will disappear) and the string of new nodes created by the synchronization (so that they are merged with the ones created by other edges on the same node).

We have not translated explicitly the ϵ synchronization in order to avoid rewriting of subgoals not concerned by the synchronization.

In order to prove the properties of the translation we need to observe some particularity of the form of the clauses produced by the translation. We can immediately observe that these clauses are synchronized clauses.

In a synchronized computation, when a big-step is about to start the goal contains only variables. If a matching occurs, old variables are bound to new variables or to function symbols. In the first case we say that the new variable is the successor of the old variable. In the second case we say that the first argument of the function symbol is the successor of the old variable and the other arguments are the nodes created by the old variable.

After the first small-step, newly created atoms are function symbols free, whereas old atoms may have function symbols. There are two ways to eliminate these function symbols (and we have to do it in order to have a synchronized computation): by matching them with analogous function symbols (in this case the synchronization occurs), or by matching them with a variable that is not used in the body of the clause, what we have forbidden in definition 3.2.

We say that two nodes x and y are merged when either x and y are assigned the same term or x is substituted by y or viceversa.

Now, we need to introduce some lemmas about the properties of the mgu of sets of equations that will be useful when we will prove equivalences between SHR and synchronized logic programming with restiction.

To be able to do it we need to introduce the concept of set of equations associated to a substitution.

Definition 4.4 (Set of equations associated to a substitution) *Given a substitution $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ we define set of equations associated to θ:*
$\mathrm{eqn}(\theta) = \{v_1 = t_1, \ldots, v_n = t_n\}$.

Observe that if θ is idempotent then $\mathrm{mgu}(\mathrm{eqn}(\theta)) = \theta$. Moreover, $\mathrm{eqn}(\mathrm{mgu}(S))$ is a set of equations with the same unifiers as S and in resolved form.

Lemma 4.2 *Let S be a set of equations on variables. Let $\equiv_S$ be the transitive, reflexive and symmetric closure of the equalities in S. For each choice of representatives of the congruence classes of $\equiv_S$:*
$\theta = \{[[x]/x] | [x] \neq x\}$
where $[x]$ is the representative of the congruence class of x, θ is a mgu of S.
Proof
Here θ is a unifier of S because if $x = y \in S$ then x and y are in the same equivalence class and so they are mapped to the same representative. Each unifier of S must map equivalence classes to a single variable because given two variables $x \equiv_S y$ there must exist a chain of equalities in S of the form:
$x = y_0 \ y_0 = y_1 \ \ldots y_n = y$
but these equalities must be all unified and so x and y must be mapped to the same representative.
So each unifier can be obtained by composing θ with another substitution.
Finally, θ must be a mgu.

Lemma 4.3 *Let θ be a substitution and S a set of equations. Then:*
$\mathrm{mgu}(S\theta) = \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(S))\theta)$
Proof
$\mathrm{eqn}(\mathrm{mgu}(S))$ *has the same unifiers as S, then* $\mathrm{eqn}(\mathrm{mgu}(S))\theta$ *has the same unifiers as $S\theta$ and so the mgu is the same.*

Lemma 4.4 *Let θ_1 and θ_2 be substitutions and θ_1 be idempotent. We have:*

$$\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)) = \theta_1 \, \mathrm{mgu}((\mathrm{eqn}(\theta_2))\theta_1)$$

Proof
Let us prove that $\theta_1 \, \mathrm{mgu}((\mathrm{eqn}(\theta_2))\theta_1)$ *unifies* $\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))$.
It unifies $\mathrm{eqn}(\theta_1)$ *because θ_1 is idempotent.*
It unifies $\mathrm{eqn}(\theta_2)$ *iff* $\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)$ *unifies* $\mathrm{eqn}(\theta_2)\theta_1$ *but this holds trivially.*
Now let γ be a unifier of $\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)$. *Clearly γ is also a unifier of* $\mathrm{eqn}(\theta_1)$ *and because θ_1 is the* mgu *of* $\mathrm{eqn}(\theta_1)$ *we have:*
$\gamma = \theta_1\gamma'$
for some γ'.
γ is also a unifier of $\mathrm{eqn}(\theta_2)$ *and so γ' is a unifier of* $\mathrm{eqn}(\theta_2)\theta_1$ *and we have:*
$\gamma' = \mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)\gamma''$
for some γ'' and:
$\gamma = \theta_1 \, \mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)\gamma''$.
So $\theta_1 \, \mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)$ *is the mgu of* $\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)$.

Lemma 4.5 *Let θ_1 and θ_2 be idempotent substitutions such that:*
$\mathrm{Dom}(\theta_1) \cap \mathrm{Dom}(\theta_2) = \emptyset$.
$\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)) = \mathrm{mgu}(\mathrm{eqn}(\theta_1\theta_2))$.

Proof

$\mathrm{eqn}(\theta_1\theta_2)$ *is* $\{[t\theta_2/v] | [t/v] \in \theta_1, t\theta_2 \neq v\} \cup \{[t/v] \in \theta_2 | v \notin \mathrm{Dom}(\theta_1)\}$.
Because of $\mathrm{Dom}(\theta_1) \cap \mathrm{Dom}(\theta_2) = \emptyset$ *this is equal to:*
$\{[t/v]\theta_2 | [t/v] \in \theta_1, t\theta_2 \neq v\} \cup \{[t/v] \in \theta_2\}$.
Up to trivial equations (which are removed during mgu computation) this is:
$\mathrm{eqn}(\theta_1)\theta_2 \cup \mathrm{eqn}(\theta_2)$.
So we have:
$\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)) = \mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta_2 \cup \mathrm{eqn}(\theta_2)) = \mathrm{mgu}(\mathrm{eqn}(\theta_1\theta_2))$.

Lemma 4.6 *Let θ_1', θ_1'' and θ_2 be substitutions such that* $\mathrm{Dom}(\theta_1') \cap \mathrm{Dom}(\theta_1'') = \emptyset$ *and θ_1' and θ_1'' be idempotent.*
If $\mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'')\theta_2)$ *exists then also* $\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)$ *and* $\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\theta_2)$ *exist.*

Proof

Using lemma 4.3, lemma 4.5 and again lemma 4.3 we have:

$$\begin{aligned}
&\mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'')\theta_2) = \\
&= \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'')))\theta_2) = \\
&= \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_1'')))\theta_2) = \\
&= \mathrm{mgu}((\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_1''))\theta_2) = \\
&= \mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2 \cup \mathrm{eqn}(\theta_1'')\theta_2)
\end{aligned}$$

and so $\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)$ *and* $\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\theta_2)$ *exist.*

We can now use these properties of mgus to prove some properties of SLD-derivations.
First of all, we want to study the effect that applying a substitution to the starting goal of a computation has on the computation itself.

Lemma 4.7 *If we have a computation:*
$G \xrightarrow{\theta_1}{}^{*} G'$
and a substitution θ_2 such that $\mathrm{var}(\theta_2) \cap \mathrm{var}(C_i) = \emptyset$ *for each clause variant C_i of the computation and* $\mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta_2)$ *exists then we can have the following computation:*
$G\theta_2 \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta_2)}{}^{*} G' \,\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))$.

Proof

We will show it by induction on the length of the computation.
0)
$G \xrightarrow{\epsilon} G = G'$

$G\theta_2 \xrightarrow{\epsilon} G\theta_2 = G'\theta_2$
The thesis holds because:
$\mathrm{mgu}(\mathrm{eqn}(\epsilon)\theta_2) = \mathrm{mgu}(\emptyset) = \epsilon$
$\mathrm{mgu}(\mathrm{eqn}(\epsilon) \cup \mathrm{eqn}(\theta_2)) = \mathrm{mgu}(\mathrm{eqn}(\theta_2)) = \theta_2$.
Inductive step)
We have:
$G \xrightarrow{\theta_1'} G'' \xrightarrow{\theta_1''}{}^{*} G'$
with $\theta_1 = \theta_1'\theta_1''$.
We have:
$\theta_1' = \mathrm{mgu}(\{A = H\})$
$G'' = G[B/A]\theta_1'$
for some atom A of the goal and some header H and body B of a clause variant. We want to have a transition using the same clause variant (which contains only new variables by hypothesis) and the atom $A\theta_2$. We will have:

$$\begin{aligned}
&\mathrm{mgu}(\{A\theta_2 = H\}) = \\
&\qquad = \mathrm{mgu}(\{(A = H)\theta_2\}) = \\
&\qquad = \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\{A = H\}))\theta_2) = \\
&\qquad = \mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)
\end{aligned}$$

using lemma 4.3.
For lemma 3.1 we have $\mathrm{Dom}(\theta_1') \cap \mathrm{Dom}(\theta_1'') = \emptyset$ *and so, because* $\mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta_2)$ *exists, also* $\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)$ *exists for lemma 4.6.*
Therefore, we can do the transition:
$G\theta_2 \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)} G\theta_2[B/A]\,\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2) = G[B/A]\theta_2\,\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)$
because $\mathrm{var}(\theta_2) \cap \mathrm{var}(B) = \emptyset$.
Using lemma 4.4 we have:

$$\begin{aligned}
&G[B/A]\theta_2\,\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2) = \\
&\qquad = G[B/A]\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)) = \\
&\qquad = G[B/A]\theta_1'\,\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1') = \\
&\qquad = G[B/A]\theta_1'\theta_1'\,\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1') = \\
&\qquad = G[B/A]\theta_1'\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)) = \\
&\qquad = G''\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2))
\end{aligned}$$

where we used the idempotence of θ_1'.
By inductive hypothesis:

$$\begin{aligned}
&G''\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)) \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)))}{}^{*} \\
&\qquad G'\,\mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2))) \cup \mathrm{eqn}(\theta_1''))
\end{aligned}$$

if $\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)))$ *exists.*
But we have:

$$\begin{aligned}
&\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)\,\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2))) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)\,\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\theta_2\,\mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2)) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1')\theta_2 \cup \mathrm{eqn}(\theta_1'')\theta_2) = \\
&\quad = \mathrm{mgu}((\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_1''))\theta_2) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_1'')))\theta_2) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'')))\theta_2) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'')\theta_2) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta_2)
\end{aligned}$$

where we used lemma 4.4, lemma 4.3 and lemma 4.5 (we could apply it because $\mathrm{Dom}(\theta_1') \cap \mathrm{Dom}(\theta_1'') = \emptyset$ *for lemma 3.1).*
Because this mgu exists for the hypotheses also $\mathrm{mgu}(\mathrm{eqn}(\theta_1'')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2)))$ *exists.*
Using the same techniques we have:

$$\begin{aligned}
&\mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2))) \cup \mathrm{eqn}(\theta_1'')) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1') \cup \mathrm{eqn}(\theta_2) \cup \mathrm{eqn}(\theta_1'')) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1'\theta_1'') \cup \mathrm{eqn}(\theta_2)) = \\
&\quad = \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))
\end{aligned}$$

and so the thesis holds.

Now we can study the composition of goals.

Lemma 4.8 *If:*
$G_1 \xrightarrow{\theta_1}{}^{*} G_1'$
$G_2 \xrightarrow{\theta_2}{}^{*} G_2'$
and the clause variants used in the two computations have disjoint sets of variables and $\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))$ *exists then:*
$G_1, G_2 \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))}{}^{*} (G_1', G_2')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))$.
Proof
We have:
$G_1, G_2 \xrightarrow{\theta_1}{}^{*} (G_1', G_2)\theta_1$.
Moreover:
$G_1', G_2 \xrightarrow{\theta_2}{}^{*} G_1'\theta_2, G_2'$.

Now we can apply lemma 4.7 because $\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)$ *exists since exists* $\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)) = \theta_1 \,\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)$ *for lemma 4.4:*

$$(G_1', G_2)\theta_1 \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_2)\theta_1)}{}^{*} (G_1'\theta_2, G_2')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2)) =$$
$$= (G_1', G_2')\,\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2))$$

for the idempotence and for lemma 4.4.
This is the thesis.

Now we will study some properties of SHR systems. Next lemma proves that the effect of the restrictions in the rewriting rules is the same of the corresponding restrictions in the clauses (as stated in lemma 3.6). This means that we can forget all these restrictions when proving the equivalences without altering the result.

Lemma 4.9 *Let $\mathcal{P}$ be a set of productions, $\mathcal{P}_{free}$ be the same set of productions after the removal of restrictions (and after choosing for the restricted variables representatives that are not already used in the derivation).*
$\mathcal{P}_{free} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma', Y \vdash G')$ *iff:*
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash \nu Y\ G')$
where Y is the set of restricted variables in the productions used.
Proof
We will prove by rule induction a stronger lemma, which states also that $\mathrm{var}(\Lambda) \cap Y = \emptyset$.
Axioms)
Trivial.
Rule (ren))
For hypothesis we have:
$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma', Y \vdash G'$ *iff:*
$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash \nu Y\ G'$.
Observe that $Y \cap \mathrm{var}(\sigma) = \emptyset$, and so also $Y \cap \mathrm{var}(\rho) = \emptyset$. So we will have:

$$\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \rho(\sigma(\Gamma', Y)) \vdash \rho(\sigma(G')) =$$
$$= \sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \rho(\sigma(\Gamma')), Y \vdash \rho(\sigma(G'))$$

iff:

$$\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \rho(\sigma(\Gamma')) \vdash \rho(\sigma(\nu Y\ G')) =$$
$$= \sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \rho(\sigma(\Gamma')) \vdash \nu Y\ \rho(\sigma(G'))$$

Observe also that by hypothesis $\mathrm{var}(\rho(\sigma(\Lambda))) \cap Y \subseteq \mathrm{var}(\Lambda) \cap Y = \emptyset$.
Proofs for other rules are analogous.

In the next lemma we will show that we can always hide variables as the last step in the proof of an SHR transition, thus we can suppose that all the other rules are applied only to graphs without restrictions.

Lemma 4.10 *Let $\mathcal{P}$ be a set of productions. If:*
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$
then we can build a proof of this transition that never uses (hide) rule as premise of some other rules.

Proof

Suppose (hide) is the premise of a (ren) rule:

$$(ren)\ \frac{(hide)\ \frac{\Gamma,x\vdash G\xrightarrow{\Lambda\uplus\{(x,a,\vec{y})\}}\Gamma,x,\Gamma_\Lambda,Y\vdash G'}{\Gamma\vdash\nu x\ G\xrightarrow{\Lambda}\Gamma,\Gamma_\Lambda\vdash\nu x\ \nu Y\ G'}}{\sigma(\Gamma)\vdash\sigma(\nu x\ G)\xrightarrow{\rho(\sigma(\Lambda))}\sigma(\Gamma),\Gamma_{\rho(\sigma(\Lambda))}\vdash\rho(\sigma(\nu x\ \nu Y\ G'))}$$

Because $(\mathrm{var}(\sigma)\cup\mathrm{var}(\rho))\cap(Y\cup\{x\}=\emptyset)$ *we can build:*

$$(ren)\ \frac{\Gamma,x\vdash G\xrightarrow{\Lambda\uplus\{(x,a,\vec{y})\}}\Gamma,x,\Gamma_\Lambda,Y\vdash G'}{\sigma(\Gamma,x)\vdash\sigma(G)\xrightarrow{\rho(\sigma(\Lambda\uplus\{(x,a,\vec{y})\}))}\sigma(\Gamma,x),\Gamma_{\rho(\sigma(\Lambda\uplus\{(x,a,\vec{y})\}))}\vdash\rho(\sigma(G'))}$$

where we have extended σ in order to be the identity on x and ρ is unchanged because $x\notin\mathrm{var}(\Lambda)$. We have:

$$\sigma(\Gamma,x)\vdash\sigma(G)\xrightarrow{\rho(\sigma(\Lambda\uplus\{(x,a,\vec{y})\}))}\sigma(\Gamma,x),\Gamma_{\rho(\sigma(\Lambda\uplus\{(x,a,\vec{y})\}))}\vdash\rho(\sigma(G'))=$$
$$\sigma(\Gamma),x\vdash\sigma(G)\xrightarrow{\rho(\sigma(\Lambda))\uplus\{(x,a,\rho(\sigma(\vec{y})))\}}\sigma(\Gamma),x,\Gamma_{\rho(\sigma(\Lambda))},Y\vdash\rho(\sigma(G'))$$

and now:

$$(hide)\ \frac{\sigma(\Gamma),x\vdash\sigma(G)\xrightarrow{\rho(\sigma(\Lambda))\uplus\{(x,a,\rho(\sigma(\vec{y})))\}}\sigma(\Gamma),x,\Gamma_{\rho(\sigma(\Lambda))},Y\vdash\rho(\sigma(G'))}{\sigma(\Gamma)\vdash\nu x\ \sigma(G)\xrightarrow{\rho(\sigma(\Lambda))}\sigma(\Gamma),\Gamma_{\rho(\sigma(\Lambda))}\vdash\nu x\ \nu Y\ \rho(\sigma(G'))}$$

and because $Y\cap\mathrm{Dom}(\rho)=\emptyset$ *we have:*

$$\sigma(\Gamma)\vdash\nu x\ \sigma(G)\xrightarrow{\rho(\sigma(\Lambda))}\sigma(\Gamma),\Gamma_{\rho(\sigma(\Lambda))}\vdash\nu x\ \nu Y\ \rho(\sigma(G'))=$$
$$\sigma(\Gamma)\vdash\sigma(\nu x\ G)\xrightarrow{\rho(\sigma(\Lambda))}\sigma(\Gamma),\Gamma_{\rho(\sigma(\Lambda))}\vdash\ \rho(\sigma(\nu x\ \nu Y\ G'))$$

which proves the thesis.
Suppose (hide) is the premise of a (par) rule. For symmetry we can suppose that it is the left premise:

$$(par)\ \frac{(hide)\ \frac{\Gamma,x\vdash G_1\xrightarrow{\Lambda_1\uplus\{(x,a,\vec{y})\}}\Gamma,x,\Gamma_{\Lambda_1},Y\vdash G_1'}{\Gamma\vdash\nu x\ G_1\xrightarrow{\Lambda_1}\Gamma,\Gamma_{\Lambda_1}\vdash\nu x\ \nu Y\ G_1'},\quad\Gamma\vdash G_2\xrightarrow{\Lambda_2}\Gamma,\Gamma_{\Lambda_2}\vdash G_2'}{\Gamma\vdash(\nu x\ G_1)|G_2\xrightarrow{\rho(\Lambda_1\cup\Lambda_2)}\Gamma,\Gamma_{\rho(\Lambda_1\cup\Lambda_2)}\vdash\rho((\nu x\ \nu Y\ G_1')|G_2')}$$

We can build an equivalent proof using rule (new) on the second premise:

$$\text{(new)} \quad \frac{\Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Gamma_{\Lambda_2} \vdash G_2'}{\Gamma, x \vdash G_2 \xrightarrow{\Lambda_2 \uplus \{(x,a,\vec{y}_2)\}} \Gamma, x, \Gamma_{\Lambda_2}, Y_2 \vdash G_2'}$$

where:
$Y_2 = \mathrm{Set}(\vec{y}_2)$.
Now we can use rule (par):

$$\frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda_1 \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_{\Lambda_1}, Y \vdash G_1', \Gamma, x \vdash G_2 \xrightarrow{\Lambda_2 \uplus \{(x,a,\vec{y}_2)\}} \Gamma, x, \Gamma_{\Lambda_2}, Y_2 \vdash G_2'}{\Gamma, x \vdash G_1 | G_2 \xrightarrow{\Lambda'} \Gamma, x, \Gamma_{\Lambda'} \vdash \rho(G_1' | G_2')}$$

where:
$\Lambda' = \rho((\Lambda_1 \uplus \{(x, a, \vec{y})\}) \cup (\Lambda_2 \uplus \{(x, a, \vec{y}_2)\}))$
and:

$$\Gamma, x \vdash G_1|G_2 \xrightarrow{\Lambda'} \Gamma, x, \Gamma_{\Lambda'} \vdash \rho(G_1'|G_2') =$$
$$= \Gamma, x \vdash G_1|G_2 \xrightarrow{\rho((\Lambda_1 \cup \Lambda_2) \uplus \{(x,a,\vec{y})\})} \Gamma, x, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)}, Y \vdash \rho(G_1'|G_2')$$

Finally we can apply rule (hide):

$$\text{(hide)} \quad \frac{\Gamma, x \vdash G_1|G_2 \xrightarrow{\rho((\Lambda_1 \cup \Lambda_2) \uplus \{(x,a,\rho(\vec{y}))\})} \Gamma, x, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)}, Y \vdash \rho(G_1'|G_2')}{\Gamma \vdash \nu x \ G_1|G_2 \xrightarrow{\rho(\Lambda_1 \cup \Lambda_2)} \Gamma, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)} \vdash \nu x \ \nu Y \ \rho(G_1'|G_2')}$$

which proves the thesis because ρ is the identity on x and Y and $(\{x\} \cup Y) \cap (\mathrm{var}(G_2) \cup \mathrm{var}(G_2')) = \emptyset$.
Suppose (hide) is the premise of a (new) rule:

$$\text{(new)} \quad \frac{\text{(hide)} \quad \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma \vdash \nu x \ G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash \nu x \ \nu Y \ G'}}{\Gamma, x' \vdash \nu x \ G \xrightarrow{\Lambda \uplus \{(x',a',\vec{y}')\}} \Gamma, x', \Gamma_\Lambda, \mathrm{Set}(\vec{y}') \vdash \nu x \ \nu Y \ G'}$$

Then we can build:

$$\text{(hide)} \quad \frac{\text{(new)} \quad \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma, x, x' \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\} \uplus \{(x',a',\vec{y}')\}} \Gamma, x, x', \Gamma_\Lambda, Y, \mathrm{Set}(\vec{y}') \vdash G'}}{\Gamma, x' \vdash \nu x \ G \xrightarrow{\Lambda \uplus \{(x',a',\vec{y}')\}} \Gamma, x', \Gamma_\Lambda, \mathrm{Set}(\vec{y}') \vdash \nu x \ \nu Y \ G'}$$

which proves the thesis.

We will now define the relations we want to exist between a synchronization and the substitution of the logic computation that simulates it.

Definition 4.5 (Substitution associated to a transition)

Let $\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$ *be a synchronization and* $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ *a transition. We say that a substitution* θ *is associated to this transition iff* $S = \mathrm{eqn}(\theta)$ *verifies:*

- $S = S_1 \cup S_2$
- $S_1 = \{x_i = a_i(x'_i, \vec{y}'_i)\}_{x_i \in \mathrm{fn}(G), a_i \neq \epsilon} \cup \{x_i = x'_i\}_{some\ x_i \in \mathrm{fn}(G), a_i = \epsilon}$ *where all* x'_i *are distinct;* x_i *and* x'_i *are different classes of variables and the same holds for* x_i *and* $y_{i,j}$*.*
- $\mathrm{Dom}(S_2) \cap \mathrm{var}(S_1) = \emptyset$

We define inverse substitution θ^i *the substitution that replaces* y'_i *with* y_i *and* x'_i *with* x_i*.*

The idea is that the substitution contains a binding for each non trivial transition. Each of these bindings substitutes the old variable with a term having as functional symbol the action performed (this is the same for all the hyperarcs, otherwise the mgu does not exist and the transition is not allowed), as first argument a new name for the node (in SHR transitions each node keeps its name, but this is not possible in logic programming because we want idempotent substitutions) and as other arguments the nodes created during the synchronization. For ϵ transitions we may have a renaming (the name of the clause variant argument substitutes the name in the goal) or we may have nothing because the new name chosen for the node may be the old name or because trivial transitions actually do not happen in logic programming (it is the same to perform only ϵ transitions or to leave the predicate unchanged).

The substitution also contains other bindings, for example bindings for the variables in the clause variants or for variables already substituted. These bindings are in S_1 and have no correspondents in SHR.

For simplicity we suppose that the set of new variables used for logical computations and for new nodes in SHR systems are disjoint sets. This guarentees that the only common variables are the ones in the starting goal. Observe that the names in the goal change during the computation while the names in the graph do not. Observe also that the names that are created in one system and in the other for new nodes are not the sames. We keep trace of the correspondence between variables in the two systems with θ^i. In particular, applying θ^i to a goal variable gives the correspondent variable in the graph.

Lemma 4.11 *Let:*
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}^{*} T$
be a big-step of synchronized logic programming.
We cannot have two occurrences of the same variable in G where one of them is unified with a complex term while the other is not.
Proof
Let us prove it by contradiction.
Let us consider the first occurrence of the variable. Suppose that this is the one unified with the complex term.
When this substitution is made we have a binding of the form:
$[f(\dots)/x]$
so also the atom that contains the second occurrence of x must be substituted otherwise the f will be in T and we have not a big-step.
Suppose that in the second atom x is substituted by a simple variable x'. This variable will be bound to $f(\dots)$ and so cannot be in the body of the clause, but this is against the definition of synchronized clause.
The other case is analogous.

Next lemma deals with synchronizations on isolated nodes.

Lemma 4.12 *Let:*
$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$
be a transition. For each node $x \notin \mathrm{fn}(G)$ we have that $\mathrm{n}_\Lambda(x)$ is composed by nodes that do not appear in any other place of the synchronization.
Moreover for any synchronization for x we can build a transition:
$\Gamma \vdash G \xrightarrow{\Lambda'} \Gamma'' \vdash G'$
such that Λ' is equal to Λ for all the nodes but x and it performs the required synchronization on x.
Proof
The proof is by rule induction.
For all rules but (idle) and (new) we can observe that they do not introduce new names not in $\mathrm{fn}(G)$ and that if the thesis holds for the premise, then it holds also for the conclusion.
The first part of the thesis holds trivially for both (idle) and (new).
The second part holds trivially for (new) (we can choose an arbitrary synchronization for the new node).
Suppose that an isolated node x appears in rule (idle):

$$\textit{(idle)} \quad \Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,\epsilon,\langle\rangle)\}} \Gamma, x \vdash G$$

Then we can build the proof:

$$\textit{(new)}\quad \frac{\textit{(idle)}\quad \Gamma \vdash G \xrightarrow{\Lambda} \Gamma \vdash G}{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \mathrm{Set}(\vec{y}) \vdash G}$$

and so the thesis holds.

Now we state a lemma that is essentially the proof of the correctness of rule (ren): the lemma says that we can apply rule (ren) (the only condition is that the mgu exists) iff there exists a computation with the same rules starting from the goal after having applied on it the fusion substitution.
Moreover it states that if a computation is associated to the premise of (ren) then the computation obtained after the merge is associated to the consequence of (ren).

Lemma 4.13 *Let* $\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\dots n}$ *be a synchronization,* σ *be a fusion of the* x_i*,* θ *a substitution associated to a transition* $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'$*,* $\rho = \mathrm{mgu}(\sigma(\Lambda))$ *(as stated in definition 2.30).*
Then we have that ρ *exists iff exists* $\theta' = \mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$*.*
In addition if ρ *exists and we have a big-step of synchronized logic programming:*
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}{}^{*} T$
and $T\theta^i = [\![\Gamma, \Gamma_\Lambda \vdash G']\!]$*.*
then we will also have:
$[\![\sigma(\Gamma) \vdash \sigma(G)]\!] \xrightarrow{\theta'}{}^{*} T'$
where θ' *is associated to* $\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \sigma(\Gamma), \Gamma_{\rho(\sigma(\Lambda))} \vdash \rho(\sigma(G'))$ *and* $T'\theta'^i = [\![\sigma(\Gamma), \Gamma_{\rho(\sigma(\Lambda))} \vdash \rho(\sigma(G'))]\!]$*.*

Proof

Observe that for lemma 4.12 we can always suppose that if σ *merges* $x_i \notin \mathrm{fn}(G)$ *with another node* x_j *then* $a_i = a_j$*.*
Notice that for lemma 4.11 we have that if $a_i = \epsilon$ *and* $a_j \neq \epsilon$ *then* $\sigma(x_i) \neq \sigma(x_j)$*.*
$\rho = \mathrm{mgu}(\sigma(\Lambda))$ *exists iff* $\sigma(x_i) = \sigma(x_j) \Rightarrow a_i = a_j$*.*
$\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta))) = \mathrm{mgu}(\sigma(S_1 \cup S_2)) = \mathrm{mgu}(\sigma(S_1) \cup \sigma(S_2))$*.*
For our hypotheses we have:
$\sigma(S_1) = \{\sigma(x_i) = a_i(x'_i, \vec{y}'_i)\}_{x_i \in \mathrm{fn}(G), a_i \neq \epsilon} \cup \{\sigma(x_i) = x'_i\}_{some\ x_i \in \mathrm{fn}(G), a_i = \epsilon}$*.*
Let us choose for each equivalence class determined by σ *a representative that is in* $\mathrm{fn}(G)$ *(if it exists) and let us define* $r(i) = j$ *iff the representative of the class of* x_i *is* x_j*, undefined if the representative does not exist.*
In this case the preceding set of equations is equivalent to:

$$\begin{gathered}\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \textit{ is defined}, a_i \neq \epsilon\} \cup \\ \cup \{a_i(x'_i, \vec{y}'_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \textit{ is defined } \wedge i \neq r(i)\} \cup \\ \cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } x_i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G) \wedge a_i = \epsilon\}\end{gathered}$$

which for the hypothesis on σ *has no unifiers if* $\sigma(x_i) = \sigma(x_j) \not\Rightarrow a_i = a_j$ *and otherwise it is equivalent to:*

$$\begin{aligned}
&\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \textit{ is defined}, a_i \neq \epsilon\} \cup \\
&\qquad \cup \{x'_i = x'_{r(i)}\ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \textit{ is defined } \wedge i \neq r(i)\} \cup \\
&\quad \cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } x_i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G) \wedge a_i = \epsilon\}
\end{aligned}$$

In this case:

$$\begin{aligned}
&\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta))) = \\
&\qquad = \mathrm{mgu}(\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \textit{ is defined}, a_i \neq \epsilon\} \cup \\
&\qquad\quad \cup \{x'_i = x'_{r(i)}\ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \textit{ is defined } \wedge i \neq r(i)\} \cup \\
&\cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } x_i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G) \wedge a_i = \epsilon\} \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cup \sigma(S_2))
\end{aligned}$$

Observe that:

$$\begin{aligned}
&\{x'_i = x'_{r(i)}\ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \textit{ exists } \wedge i \neq r(i)\} \cup \\
&\quad \cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } x_i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G) \wedge a_i = \epsilon\}
\end{aligned}$$

is a renaming and so for lemma 4.2 we can put it in resolved form.
We can choose as representatives the $\sigma(x_i)$ *when* $a_i \neq \epsilon$ *and the* $x'_{r(i)}$*. We will call* $y''_{i,j}$ *the representatives of the* $y_{i,j}$*.*
So we can put the whole set of equations in the resolved form:

$$\begin{aligned}
&\mathrm{eqn}(\theta') = \\
&= \{w_{i,j} = a_{r(i)}(x'_{r(i)}, \vec{y}''_{r(i)}) | w_{i,j} \textit{ is in the class of } \sigma(x_i) \wedge r(i) \textit{ is defined } \wedge a_i \neq \epsilon\} \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cup (\sigma(S_2))' \cup S_3
\end{aligned}$$

where S_3 *is the resolved form of the renaming after having removed the equalities on* $w_{i,j}$ *and* $(\sigma(S_2))'$ *is* $\sigma(S_2)$ *where we have made the substitutions necessary to put the first part in this form.*
So the mgu exists.
Now let us prove part two.
Observe that we can reorder $\mathrm{eqn}(\theta')$ *in:*

$$\begin{aligned}
&\mathrm{eqn}(\theta') = \{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}''_{r(i)}) | r(i) \textit{ is defined } \wedge a_i \neq \epsilon\} \cup \\
&\cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } x_i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G) \wedge a_i = \epsilon\} \cup S'_2 = \\
&\qquad\qquad\qquad\qquad\qquad\qquad = S'_1 \cup S'_2
\end{aligned}$$

where $\mathrm{Dom}(S'_2) \cap \mathrm{var}(S'_1) = \emptyset$.
We will prove that $y'_{i,j}$ *is in the same class of* $y'_{i',j'}$ *iff* $\rho(y_{i,j}) = \rho(y_{i',j'})$.
The equalities between $y'_{i,j}$ *are determined by:*
$\{\vec{y}'_i = \vec{y}'_{r(i)} | r(i) \text{ exists } \wedge i \neq r(i)\}$
because other equations cannot merge two $y'_{i,j}$.
Here ρ *is the mgu of:*

$$\{\vec{y}_i = \vec{y}_j | \sigma(x_i) = \sigma(x_j)\} = \{\vec{y}_i = \vec{y}_j | r(i) = r(j)\} \cup \\ \cup \{\text{equations on } y_i \text{ where } r(i) \text{ is undefined}\}$$

The equations in the last set do not merge variables in the first set because when $r(i)$ *is undefined the* $y_{i,j}$ *are never used elsewhere for lemma 4.12.*
So the classes restricted to the variables for which $r(i)$ *exists are the sames.*
So $\mathrm{eqn}(\theta')$ *is associated to* $\rho(\sigma(\Lambda))$ *with inverse substitution:*
$\theta^i = \{[\sigma(x_i)/x'_{r(i)}] \text{ if the binding existed in } \theta'^i\} \cup \{[\rho(y_{i,j})/y''_{r(i),j}]\}$.
Observe that:

- $\mathrm{var}(\sigma) \cap \mathrm{var}(C_i) = \emptyset$
- $\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$ *exists because* ρ *exists*

So we can apply lemma 4.7 to prove that we have the following computation:
$\sigma([\![\Gamma \vdash G]\!]) \xrightarrow{\theta'}{}^{*} T \,\mathrm{mgu}(\mathrm{eqn}(\theta) \cup \mathrm{eqn}(\sigma))$
where:
$\sigma([\![\Gamma \vdash G]\!]) = [\![\sigma(\Gamma) \vdash \sigma(G)]\!]$
for lemma 4.1.
$T' = T \,\mathrm{mgu}(\mathrm{eqn}(\theta) \cup \mathrm{eqn}(\sigma)) = \sigma(T) \,\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$
for lemma 4.4.
Observe that $\mathrm{var}(T) \subseteq \{x_i | x_i \notin \mathrm{Dom}(\theta)\} \cup \{x'_i\} \cup \{\vec{y}'_i\}$
and so $\mathrm{var}(\sigma(T)) \subseteq \{\sigma(x_i) | \sigma(x_i) \notin \mathrm{Dom}(\theta')\} \cup \{x'_i\} \cup \{\vec{y}'_i\}$
and $\mathrm{var}(\sigma(T)\theta') \subseteq \{\sigma(x_i) | \sigma(x_i) \notin \mathrm{Dom}(\theta')\} \cup \{x'_{r(i)}\} \cup \{\vec{y}''_i\}$
As $\sigma\theta'$ *merges* v_1 *and* v_2 *iff* $\rho(\sigma(v_1\theta^i)) = \rho(\sigma(v_2\theta^i))$ *we have:*
$\sigma(T) \,\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))\theta'^i = \rho(\sigma(T\theta^i))$.

We now state a lemma useful in dealing with unions of disjoint synchronizations.

Lemma 4.14 *Let* θ_1 *and* θ_2 *be substitutions associated to* $\Gamma_1 \vdash G_1 \xrightarrow{\Lambda_1} \Gamma'_1 \vdash G'_1$ *and* $\Gamma_2 \vdash G_2 \xrightarrow{\Lambda_2} \Gamma'_2 \vdash G'_2$ *respectively. If all variables in* θ_1 *and in the first synchronization are disjoint from variables in* θ_2 *and in the second synchronization then we have that* $\theta_1\theta_2$ *is associated to* $\Gamma_1, \Gamma_2 \vdash G_1|G_2 \xrightarrow{\Lambda_1 \cup \Lambda_2} \Gamma'_1, \Gamma'_2 \vdash G'_1|G'_2$ *and the inverse substitution is the union of the two inverse substitutions.*

Proof
For our hypotheses we have:
$\mathrm{eqn}(\theta_1) = S'_1 \cup S''_1$
$\mathrm{eqn}(\theta_2) = S'_2 \cup S''_2$
according to definition 4.5. For the definition of composition of substitutions:
$\mathrm{eqn}(\theta_1\theta_2) = \mathrm{eqn}(\theta_1) \cup \mathrm{eqn}(\theta_2) = S'_1 \cup S''_1 \cup S'_2 \cup S''_2 = (S'_1 \cup S'_2) \cup (S''_1 \cup S''_2)$
and these new sets trivially verify the thesis.

We can now prove the correctness of our translation for SHR without extrusions.

Theorem 4.2 (Mapping of SHR without extrusions: correctness) *Let $\mathcal{P}$ be a set of productions of an SHR system without extrusions as defined in definitions 2.29 and 2.31. Let P be the program of logic programming with restriction obtained by translating the productions in $\mathcal{P}$ according to definition 4.3. If:*
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$
then we can have in P the following big-step of synchronized logic programming with restriction:
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}{}^{*} T$
where θ is associated to $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ with inverse substitution θ^i and $T\theta^i = [\![\Gamma' \vdash G']\!]$.
Proof
We will prove it by rule induction. Observe that thanks to lemmas 3.6 and 4.9 we can restrict ourselves to the case of productions without restrictions and thanks to lemma 4.10 we can restrict ourselves to the case of goals without restrictions for all rules but (hide).
Axioms) Suppose we have a production:
$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Gamma_\Lambda \vdash G$
where:
$\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$
Then we have in P a clause that, up to injective renaming of variables, is:
$s(a_1(x'_1, \vec{y}'_1), \ldots, a_n(x'_n, \vec{y}'_n)) : -[\![\Gamma' \vdash G]\!]\theta_v$
where $\theta_v = \{[x'_i/x_i]\ [y'_{i,j}/y_{i,j}]\}$
and where we have x'_i instead of $a_i(x'_i, \vec{y}'_i)$ if $a_i = \epsilon$.
Let us consider the goal:
$[\![s(x_1, \ldots, x_n)]\!] = s(x_1, \ldots, x_n)$
It unifies with the clause variant:
$s(a_1(x'_1, \vec{y}'_1), \ldots, a_n(x'_n, \vec{y}'_n)) : -[\![\Gamma' \vdash G]\!]\theta_v$
with mgu $\theta = \mathrm{mgu}(S)$ where:
$S = \{x_i = a_i(x'_i, \vec{y}'_i) | a_i \neq \epsilon\} \cup \{x_i = x'_i | a_i = \epsilon\}$
Observe that S is in resolved form because the set of variables of the goal and the

set of variables in the clause variant are disjoint.
So $S = \mathrm{eqn}(\theta)$.
We can see that θ *is associated to* $x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Gamma_\Lambda \vdash G$.
We have:
$\theta^i = \{[x_i/x'_i]\ [y_{i,j}/y'_{i,j}]\}$
Observe also that the result of the computation is:
$T = [\![\Gamma' \vdash G]\!]\theta_v\theta = [\![\Gamma' \vdash G]\!]\theta_v$
because $\mathrm{var}(G\theta_v) \cap \mathrm{Dom}(\theta) = \emptyset$. *So we have:*
$T\theta^i = [\![\Gamma' \vdash G]\!]\theta_v\theta^i = [\![\Gamma' \vdash G]\!]$
as required.
Rule (ren))

$$\frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'}{\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\rho(\sigma(\Lambda))} \sigma(\Gamma), \Gamma_{\rho(\sigma(\Lambda))} \vdash \rho(\sigma(G'))}$$

where $\sigma : \Gamma \to \Gamma$ *and* ρ *is the most general unifier of* $\sigma(\Lambda)$. *For inductive hypothesis we have in* P *the following big-step of synchronized logic programming with restriction:*
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}{}^{*} T$
where θ *is associated to* $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\Gamma, \Gamma_\Lambda \vdash G']\!]$.
Observe that σ *is the composition of an injective renaming* σ' *and a fusion of the* x_i σ''.
We can suppose that $\mathrm{Im}(\sigma') \cap \mathrm{var}(C_i) = \emptyset$ *for each clause variant* C_i *used in the computation (otherwise we can build a new computation by renaming the variables in the clauses), and so we can extend* σ' *to a renaming of all the variables in the computation. Because all steps are independent of the actual names of variables we have:*
$\sigma'([\![\Gamma \vdash G]\!]) \xrightarrow{\sigma'(\theta)}{}^{*} \sigma'(T)$
which is a big-step, where $\sigma'(\theta)$ *is associated to* $\sigma'(\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G')$ *with inverse substitution* $\sigma'(\theta^i)$ *and* $\sigma'(T)\sigma'(\theta^i) = \sigma'([\![\Gamma, \Gamma_\Lambda \vdash G']\!])$.
We have:
$\sigma'(\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G') = \sigma'(\Gamma) \vdash \sigma'(G) \xrightarrow{\sigma'(\Lambda)} \sigma'(\Gamma), \Gamma_{\sigma'(\Lambda)} \vdash \sigma'(G')$.
Observe that, because ρ *exists by hypothesis, if* $a_i \neq a_j$ *then* $\sigma''(\sigma'(x_i)) \neq \sigma''(\sigma'(x_j))$.
Now we can apply lemma 4.13 and have:
$[\![\sigma''(\sigma'(\Gamma)) \vdash \sigma''(\sigma'(G))]\!] \xrightarrow{\theta'}{}^{*} T'$
where $\theta' = \mathrm{mgu}(\sigma''(\mathrm{eqn}(\sigma'(\theta))))$ *is associated to:*
$\sigma''(\sigma'(\Gamma)) \vdash \sigma''(\sigma'(G)) \xrightarrow{\rho(\sigma''(\sigma'(\Lambda)))} \sigma''(\sigma'(\Gamma)), \Gamma_{\rho(\sigma''(\sigma'(\Lambda)))} \vdash \rho(\sigma''(\sigma'(G')))$
and:

$T'\theta'^i = [\![\sigma''(\sigma'(\Gamma)), \Gamma_{\rho(\sigma''(\sigma'(\Lambda)))} \vdash \rho(\sigma''(\sigma'(G')))]\!]$.
Observe that by definition of σ *we have:*
$\theta' = \mathrm{mgu}(\sigma''(\mathrm{eqn}(\sigma'(\theta)))) = \mathrm{mgu}(\sigma''(\sigma'(\mathrm{eqn}(\theta)))) = \mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$
and the thesis.
Rule (par))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma, \Gamma_{\Lambda_1} \vdash G_1' \qquad \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Gamma_{\Lambda_2} \vdash G_2'}{\Gamma \vdash G_1|G_2 \xrightarrow{\rho(\Lambda_1 \cup \Lambda_2)} \Gamma, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)} \vdash \rho(G_1'|G_2')}$$

if $\Gamma_{\Lambda_1} \cap \Gamma_{\Lambda_2} = \emptyset$ *and* ρ *is the most general unifier for* $\Lambda_1 \cup \Lambda_2$.
For inductive hypothesis we have in P *the following big-steps of synchronized logic programming with restriction:*

- $[\![\Gamma \vdash G_1]\!] \xrightarrow{\theta_1}{}^* T_1$
  where θ_1 *is associated to* $\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma, \Gamma_{\Lambda_1} \vdash G_1'$ *with inverse substitution* θ_1^i *and* $T_1\theta_1^i = [\![\Gamma, \Gamma_{\Lambda_1} \vdash G_1']\!]$.
- $[\![\Gamma \vdash G_2]\!] \xrightarrow{\theta_2}{}^* T_2$
  where θ_2 *is associated to* $\Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Gamma_{\Lambda_2} \vdash G_2'$ *with inverse substitution* θ_2^i *and* $T_2\theta_2^i = [\![\Gamma, \Gamma_{\Lambda_2} \vdash G_2']\!]$.

We can suppose that $\mathrm{var}(\theta_1) \cap \mathrm{var}(\theta_2) \subseteq \Gamma$.
Let σ *be an injective renaming that maps* Γ *to variables never used elsewhere and let* σ^{-1} *be the inverse renaming.*
We will have:
$[\![\sigma(\Gamma) \vdash \sigma(G_1)]\!] \xrightarrow{\sigma(\theta_1)}{}^* \sigma(T_1)$
where $\sigma(\theta_1)$ *is associated to* $\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\sigma(\Lambda_1)} \sigma(\Gamma), \Gamma_{\sigma(\Lambda_1)} \vdash \sigma(G_1')$ *with inverse substitution* $\sigma(\theta_1^i)$ *and* $\sigma(T_1)\sigma(\theta_1^i) = [\![\sigma(\Gamma), \Gamma_{\sigma(\Lambda_1)} \vdash \sigma(G_1')]\!]$.
For lemma 4.14 we have that $\sigma(\theta_1)\theta_2$ *is associated to* $\sigma(\Gamma), \Gamma \vdash \sigma(G_1)|G_2 \xrightarrow{\sigma(\Lambda_1) \cup \Lambda_2} \sigma(\Gamma), \Gamma_{\sigma(\Lambda_1)}, \Gamma, \Gamma_{\Lambda_2} \vdash \sigma(G_1')|G_2'$ *and the inverse substitution is the union of the two inverse substitutions.*
Observe now that σ^{-1} *is a fusion of the* x_i *and so for lemma 4.13 we have:*
$[\![\sigma^{-1}(\sigma(\Gamma), \Gamma) \vdash \sigma^{-1}(\sigma(G_1)|G_2)]\!] \xrightarrow{\theta'}{}^* T'$
where $\theta' = \mathrm{mgu}(\sigma^{-1}(\mathrm{eqn}(\sigma(\theta_1)\theta_2)))$ *is associated to:*

$$\sigma^{-1}(\sigma(\Gamma), \Gamma) \vdash \sigma^{-1}(\sigma(G_1)|G_2) \xrightarrow{\rho(\sigma^{-1}(\sigma(\Lambda_1) \cup \Lambda_2))} \sigma^{-1}(\sigma(\Gamma) \cup \Gamma), \Gamma_{\rho(\sigma^{-1}(\sigma(\Lambda_1) \cup \Lambda_2))} \vdash \rho(\sigma^{-1}(\sigma(G_1')|G_2'))$$

and:
$T'\theta'^i = [\![\sigma^{-1}(\sigma(\Gamma) \cup \Gamma), \Gamma_{\rho(\sigma^{-1}(\sigma(\Lambda_1) \cup \Lambda_2))} \vdash \rho(\sigma^{-1}(\sigma(G_1')|G_2'))]\!]$.

Observe that:
$[\![\sigma^{-1}(\sigma(\Gamma),\Gamma) \vdash \sigma^{-1}(\sigma(G_1)|G_2)]\!] = [\![\Gamma \vdash G_1|G_2]\!]$
$\rho(\sigma^{-1}(\sigma(\Lambda_1) \cup \Lambda_2)) = \rho(\Lambda_1 \cup \Lambda_2)$
$\sigma^{-1}(\sigma(\Gamma) \cup \Gamma), \Gamma_{\rho(\sigma^{-1}(\sigma(\Lambda_1)\cup\Lambda_2))} \vdash \rho(\sigma^{-1}(\sigma(G'_1)|G'_2)) = \Gamma, \Gamma_{\rho(\Lambda_1\cup\Lambda_2)} \vdash \rho(G'_1|G'_2)$
$[\![\sigma^{-1}(\sigma(\Gamma) \cup \Gamma), \Gamma_{\rho(\sigma^{-1}(\sigma(\Lambda_1)\cup\Lambda_2))} \vdash \rho(\sigma^{-1}(\sigma(G'_1)|G'_2))]\!] = [\![\Gamma, \Gamma_{\rho(\Lambda_1\cup\Lambda_2)} \vdash \rho(G'_1|G'_2)]\!]$
and so we have the thesis.
Rule (hide))

$$\frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma \vdash \nu x\ G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash \nu x\ \nu Y\ G'}$$

where $Y = \mathrm{Set}(\vec{y}) \setminus \Gamma_\Lambda$.
For inductive hypothesis we have in P *the following big-step of synchronized logic programming with restriction:*
$[\![\Gamma, x \vdash G]\!] \xrightarrow[\theta_{free}]{\theta}{}^{*}\ T$
where θ *is associated to* $\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\Gamma, x, \Gamma_\Lambda, Y \vdash G']\!]$.
For lemma 3.5 we also have:
$\nu x\ [\![\Gamma, x \vdash G]\!] \xrightarrow[\theta]{\theta'}{}^{*}\ \nu X'\ T$
where:
$X' = \{v \in \mathrm{fn}(T) | \mathrm{or}_\theta(v) \subseteq \{x\}\}$
and for lemma 3.4 we have $\theta'|_{\mathrm{fn}(\nu x\ [\![\Gamma,x\vdash G]\!])} = \theta|_{\mathrm{fn}(\nu x\ [\![\Gamma,x\vdash G]\!])}$ *and all the other bindings in* θ' *have domain variables not in* $\mathrm{var}(\nu x\ [\![\Gamma, x \vdash G]\!])$ *neither in* $\mathrm{var}(\nu X'\ T)$ *unless they refer extruded variables.*
We have:
$\mathrm{eqn}(\theta) = \{x_i = a_i(x'_i, \vec{y}'_i)\}_{x_i \in \mathrm{fn}(G), a_i \neq \epsilon} \cup \{x_i = x'_i\}_{some\ x_i \in \mathrm{fn}(G), a_i = \epsilon} \cup S_x \cup S_2$
where S_x *is* $\{x = a(x', \vec{y}')\}$ *if* $a \neq \epsilon$ *and* $x \in \mathrm{fn}(G)$, $\{x = x'\}$ *or* $\emptyset$ *if* $a = \epsilon$ *and* $\emptyset$ *otherwise.*
$\mathrm{or}_{\theta'}(y) = \mathrm{or}_{\mathrm{mgu}(S_1)}(y)$ *by definition of* S_2.
The only variables that satisfy $\mathrm{or}_{\mathrm{mgu}(S_1)}(v) \subseteq \{x\}$ *are* x,x' *and the variables in* $Y' = \mathrm{Set}(\vec{y}') \setminus \bigcup_i \mathrm{Set}(\vec{y}'_i)$.
So $X' = \{x, x'\} \cup Y'$ *and* $X'\theta^i = \{x\} \cup Y$.
$\theta'|_{\mathrm{fn}(\nu x\ [\![\Gamma,x\vdash G]\!])} = \theta|_{\mathrm{fn}(\nu x\ [\![\Gamma,x\vdash G]\!])}$, *and so* θ' *contains all the bindings in* S_1 *but the binding of* x *(if it exists) and all the other bindings in* θ' *have domain variables not in* $\mathrm{var}(\nu x\ [\![\Gamma, x \vdash G]\!])$ *(because we have no extrusions) and thus they can be put in* S_2.
This means that θ' *is associated to:*
$\Gamma \vdash \nu x\ G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash \nu x\ \nu Y\ G'$
with inverse substitution ${\theta'}^i = \theta^i|_{\mathrm{fn}(\nu X'\ T)}$. *We can change the names of all the vari-*

ables according to θ^i because the restricted variables can be α-converted.
So we have:
$(\nu X' \ T)\theta'^i = \nu x \ \nu Y \ T\theta^i = \nu x \ \nu Y [\![\Gamma', x, \Gamma_\Lambda, Y \vdash G']\!] = [\![\Gamma', \Gamma_\Lambda \vdash \nu x \ \nu Y \ G']\!]$
as required.
Rule (idle))

$$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma \vdash G$$

where $\Lambda(x) = (\epsilon, \langle\rangle)$ for each $x \in \Gamma$.
Observe that for each program P we have the transition:
$[\![\Gamma \vdash G]\!] \xrightarrow{\epsilon} [\![\Gamma \vdash G]\!]$
and that ϵ is associated to:
$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma \vdash G$
with empty θ^i because $a_i = \epsilon$ for each i and so we have the thesis.
Rule (new))

$$\frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'}{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, \mathrm{Set}(\vec{y}) \vdash G'}$$

For inductive hypothesis we have in P the following big-step of synchronized logic programming with restriction:
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}^* T$
where θ is associated to:
$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'$
with inverse substitution θ^i and $T\theta^i = [\![\Gamma, \Gamma_\Lambda \vdash G']\!]$.
But because $x \notin \mathrm{fn}(G)$ we have that θ is also associated to:
$\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Gamma, x, \Gamma_\Lambda, \mathrm{Set}(\vec{y}) \vdash G'$
with the same inverse substitution. The thesis follows.

In order to prove the completeness of our translation we need some properties of the mgus of synchronizations.

Lemma 4.15 *Let Λ and Λ' be synchronizations. If ρ is the mgu of Λ and ρ' is the mgu of $\rho(\Lambda) \cup \Lambda'$ then $\rho'\rho$ is the mgu of $\Lambda \cup \Lambda'$.*

Proof

Let us prove that $\rho'\rho$ unifies $\Lambda \cup \Lambda'$.

Suppose that $x_i = x_j$. If both are in Λ the thesis holds because ρ unifies x_i and x_j and ρ' is a function.

Otherwise suppose x_j in Λ': ρ is the identity on x_i and a_i so $a_i = a_j$ because ρ' exists.

We will also have from the definition of ρ' that the corresponding new nodes are unified.

Let ρ'' be another unifier of $\Lambda \cup \Lambda'$.

This clearly unifies Λ, thus:

$\rho'' = \rho'''\rho = \rho'''\rho\rho$

where we used the definition of mgu and the idempotence of ρ.

We have that $\rho'''\rho = \rho''$ unifies $\rho(\Lambda) \cup \Lambda'$ and therefore $\rho'''\rho = \rho''''\rho'$ and $\rho'' = \rho''''\rho'\rho$.

So $\rho'\rho$ is the mgu.

Lemma 4.16 *Let Λ_1 and Λ_2 be synchronizations. If ρ_1 is the mgu of Λ_1 and ρ_2 is the mgu of Λ_2, ρ' is the mgu of $\rho_1(\Lambda_1) \cup \rho_2(\Lambda_2)$ and ρ'' is the mgu of $\Lambda_1 \cup \Lambda_2$ then:*

$\rho''|_{\mathrm{var}(\Lambda_1)} = \rho'\rho_1|_{\mathrm{var}(\Lambda_1)}$

$\rho''|_{\mathrm{var}(\Lambda_2)} = \rho'\rho_2|_{\mathrm{var}(\Lambda_2)}$

Proof

We need only to prove the first relation because the second is symmetric.

For lemma 4.15 $\rho'\rho_1$ is the mgu of $\Lambda_1 \cup \rho_2(\Lambda_2)$.

For the same lemma we have:

$\rho'' = \rho'\rho_1\rho_2$

We have that:

$\rho'(\rho_1(\Lambda_1)) \cup \rho'(\rho_1(\rho_2(\Lambda_2)))$

is unified but we have done no more fusions than in:

$\rho'(\rho_1(\rho_2(\Lambda_1))) \cup \rho'(\rho_1(\rho_2(\Lambda_2))$

but because $\rho'\rho_1\rho_2 = \rho''$ is the mgu then the two synchronizations must be equal and so:

$\rho''|_{\mathrm{var}(\Lambda_1)} = \rho'\rho_1|_{\mathrm{var}(\Lambda_1)}$.

We can now prove the completeness of our translation.

Theorem 4.3 (Mapping of SHR without extrusions: completeness) *Let $\mathcal{P}$ be a set of productions of an SHR system without extrusions as defined in definitions*

2.29 and 2.31. Let P be the program of logic programming with restriction obtained by translating the productions in $\mathcal{P}$ according to definition 4.3. If we have in P a big-step of logic programming with restriction:
$[\![\Gamma \vdash G]\!] \xrightarrow{\theta}^{*} T$
then there exist Λ and G' such that θ is associated to $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ with inverse substitution θ^i and $T\theta^i = [\![\Gamma' \vdash G']\!]$ and:
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$

Proof

Observe that thanks to lemmas 3.6 and 4.9 we can restrict ourselves to the case of productions without restrictions.
Let us first suppose G without restrictions.
Our big-step is composed by many small-steps.
Let A_i be the atom substituted in small-step i, θ_i the resultant mgu, G_i the hyperedge corresponding to the atom, Λ_i the synchronization used in the production associated to the clause used at step i and σ_i a fusion that merges variables in the production if the corresponding variables of the atom of the goal are equal.
We will prove by induction on j that for each $j \geq 0$:
$A_1, \ldots, A_j \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1)\cup\cdots\cup\mathrm{eqn}(\theta_j))} T_1, \ldots, T_j$
where $\mathrm{mgu}(\mathrm{eqn}(\theta_1)\cup\cdots\cup\mathrm{eqn}(\theta_j))$ is associated to $\Gamma \vdash G_1|\ldots|G_j \xrightarrow{\rho(\sigma_1(\Lambda_1)\cup\cdots\cup\sigma_j(\Lambda_j))} \rho(\sigma_1(G_1'))|\ldots|\rho(\sigma_j(G_j'))$ with inverse substitution θ_j^i where ρ is the mgu of $\sigma_1(\Lambda_1)\cup\cdots\cup\sigma_j(\Lambda_j)$ and:
$(T_1, \ldots, T_j)\theta_j^i = [\![\Gamma, \Gamma_{\Lambda'} \vdash \rho(\sigma_1(G_1'))|\ldots|\rho(\sigma_j(G_j'))]\!]$
where:
$\Lambda' = \rho(\sigma_1(\Lambda_1)\cup\cdots\cup\sigma_j(\Lambda_j))$
1)
Because this is the first small-step of a big-step, the atom is function symbols free, so it is of the form:
$s(x_1, \ldots, x_n) = [\![s(x_1, \ldots, x_n)]\!]$
where the x_i are not necessarily distinct.
We can suppose that the x_i where originally distinct and they have then be merged by a fusion σ, so we can equivalently write our goal as:
$s(\sigma(x_1), \ldots, \sigma(x_n))$
where the x_i are distinct.
The clause variant used is of the form:
$s(a_1(x_1', \vec{y}_1'), \ldots, a_n(x_n', \vec{y}_n')) :- [\![\Gamma, \Gamma_\Lambda \vdash G']\!]\theta_v$
where $\theta_v = \{[x_i'/x_i]\ [y_{i,j}'/y_{i,j}]\}$ and where we have x_i' instead of $a_i(x_i', \vec{y}_i')$ if $a_i = \epsilon$.
The clause variant is associated to a rule:
$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Gamma_\Lambda \vdash G'$
where:

$\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$
The head of the clause unifies with $s(x_1, \ldots, x_n)$ *with mgu* θ*:*
$\theta = \mathrm{mgu}(\{x_i = a_i(x'_i, \vec{y}'_i) | a_i \neq \epsilon\} \cup \{x_i = x'_i | a_i = \epsilon\})$
which is associated to:
$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Gamma_\Lambda \vdash G'$
with inverse substitution θ^i*:*
$\theta^i = \{[x_i/x'_i]\ [y_{i,j}/y'_{i,j}]\}$
Observe also that the result of this computation is:
$[\![\Gamma, \Gamma_\Lambda \vdash G']\!]\theta_v\theta$
and:
$[\![\Gamma, \Gamma_\Lambda \vdash G']\!]\theta_v\theta\theta^i = [\![\Gamma, \Gamma_\Lambda \vdash G']\!]$.
The mgu of the real computation is:
$\theta' = \mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$
and let T' *be its result.*
We can now apply lemma 4.13 and have that exists the mgu ρ *of* $\sigma(\Lambda)$*.*
Furthermore we have that θ' *is associated to:*
$\sigma(\Gamma) \vdash s(\sigma(x_1), \ldots, \sigma(x_n)) \xrightarrow{\rho(\sigma(\Lambda))} \sigma(\Gamma), \Gamma_{\rho(\sigma(\Lambda))} \vdash \rho(\sigma(G'))$
and:
$T'\theta'^i = [\![\sigma(\Gamma), \Gamma_{\rho(\sigma(\Lambda))} \vdash \rho(\sigma(G'))]\!]$.
Finally observe that this rewriting step can be proved from our axiom by applying rule (ren).
Inductive step)
For the inductive hypothesis:
$A_1, \ldots, A_j \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))} T_1, \ldots, T_j$
where $\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))$ *is associated to:*
$\Gamma \vdash G_1 | \ldots | G_j \xrightarrow{\rho(\sigma_1(\Lambda_1) \cup \cdots \cup \sigma_j(\Lambda_j))} \Gamma, \Gamma_{\Lambda'} \vdash \rho(\sigma_1(G'_1)) | \ldots | \rho(\sigma_j(G'_j))$
with inverse substitution θ^i_j *and:*
$(T_1, \ldots, T_j)\theta^i_j = [\![\Gamma, \Gamma_{\Lambda'} \vdash \rho(\sigma_1(G'_1)) | \ldots | \rho(\sigma_j(G'_j))]\!]$
where ρ *is the mgu of* $\sigma_1(\Lambda_1) \cup \cdots \cup \sigma_j(\Lambda_j)$ *and:*
$\mathcal{P} \Vdash \Gamma \vdash G_1 | \ldots | G_j \xrightarrow{\rho(\sigma_1(\Lambda_1) \cup \cdots \cup \sigma_j(\Lambda_j))} \Gamma, \Gamma_{\Lambda'} \vdash \rho(\sigma_1(G'_1) | \ldots | \rho(\sigma_j(G'_j))$
Now consider the computation we obtain by applying the clause used at step $j+1$ *to* A_{j+1}*. With the same reasoning as in step 1) we have:*
$A_{j+1} \xrightarrow{\theta_{j+1}} T_{j+1}$
where θ_{j+1} *is associated to:*
$\Gamma_{j+1} \vdash G_{j+1} \xrightarrow{\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))} \Gamma_{j+1}, \Gamma_{\Lambda'_{j+1}} \vdash \rho_{j+1}(\sigma_{j+1}(G'_{j+1}))$
where ρ_{j+1} *is the mgu of* $\sigma_{j+1}(\Lambda_{j+1})$ *and* $\Lambda'_{j+1} = \rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))$*.*
We also have:
$T_{j+1}\theta^i_{j+1} = [\![\Gamma_{j+1}, \Gamma_{\Lambda'_{j+1}} \vdash \rho_{j+1}(\sigma_{j+1}(G'_{j+1}))]\!]$

$\mathcal{P} \Vdash (\Gamma_{j+1} \vdash G_{j+1} \xrightarrow{\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))} \Gamma_{j+1}, \Gamma_{\Lambda'_{j+1}} \vdash \rho_{j+1}(\sigma_{j+1}(G'_{j+1})))$
Let σ' be an injective renaming that maps variables in Γ_{j+1} to new variables. Observe that ${\sigma'}^{-1}$ is a fusion of the variables in Γ, Γ_{j+1}. So we have for lemma 4.8 that:

$$A_1, \ldots, A_j, \sigma'(A_{j+1}) \xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1})))} (T_1, \ldots, T_j, \sigma'(T_{j+1}))\, \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1})))$$

Observe that because $\mathrm{var}(\sigma'(\mathrm{eqn}(\theta_{j+1}))) \cap \mathrm{var}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j)) = \emptyset$ *we have:*

$$\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1}))) = \\ = \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))\, \mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta_{j+1})))$$

and so for idempotence:

$$(T_1, \ldots, T_j, \sigma'(T_{j+1}))\, \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1}))) = \\ = T_1, \ldots, T_j, \sigma'(T_{j+1})$$

Now we can apply lemma 4.7 and have:
${\sigma'}^{-1}(A_1, \ldots, A_j, \sigma'(A_{j+1})) \xrightarrow{\theta_n} {\sigma'}^{-1}(T_1, \ldots, T_j, \sigma'(T_{j+1}))\theta_n$
where $\theta_n = \mathrm{mgu}({\sigma'}^{-1}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1}))))))$.
Observe that ${\sigma'}^{-1}$ is the identity on variables not in $\mathrm{Im}(\sigma')$ *and so:*
${\sigma'}^{-1}(A_1, \ldots, A_j, \sigma'(A_{j+1})) = A_1, \ldots, A_j, A_{j+1}$

$$\begin{aligned} \theta_n &= \mathrm{mgu}({\sigma'}^{-1}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \sigma'(\mathrm{eqn}(\theta_{j+1})))))) = \\ &= \mathrm{mgu}({\sigma'}^{-1}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))) \cup \mathrm{mgu}(\mathrm{eqn}(\sigma'(\mathrm{eqn}(\theta_{j+1})))))) = \\ &= \mathrm{mgu}({\sigma'}^{-1}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j)))) \cup {\sigma'}^{-1}(\mathrm{mgu}(\mathrm{eqn}(\sigma'(\mathrm{eqn}(\theta_{j+1})))))) = \\ &= \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))) \cup {\sigma'}^{-1}(\sigma'(\mathrm{eqn}(\theta_{j+1})))) = \\ &= \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))) \cup \mathrm{eqn}(\theta_{j+1})) = \\ &= \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j) \cup \mathrm{eqn}(\theta_{j+1})) \end{aligned}$$

${\sigma'}^{-1}(T_1, \ldots, T_j, \sigma'(T_{j+1})) = T_1, \ldots, T_j, T_{j+1}$
Observe that for lemma 4.14 we have that θ_n, which is equal to $\mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \cdots \cup \mathrm{eqn}(\theta_j))\, \mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta_{j+1})))$ *is associated to:*

$$\Gamma, \sigma'(\Gamma_{j+1}) \vdash G_1 | \ldots | G_j | \sigma'(G_{j+1}) \xrightarrow{\rho(\sigma_1(\Lambda_1) \cup \cdots \cup \sigma_j(\Lambda_j)) \cup \sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1})))} \\ \Gamma, \Gamma_\Lambda, \sigma'(\Gamma_{j+1}), \Gamma_{\sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1})))} \vdash \\ \rho(\sigma_1(G'_1)) | \ldots | \rho(\sigma_j(G'_j)) | \sigma'(\rho_{j+1}(\sigma_{j+1}(G'_{j+1})))$$

with inverse substitution θ_n^i *and:*

$$(T_1, \dots, T_j, \sigma'(T_{j+1}))\theta_n^i = \\ = [\![\Gamma, \Gamma_\Lambda, \sigma'(\Gamma_{j+1}), \Gamma_{\sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1})))} \vdash \\ \rho(\sigma_1(G'_1))|\dots|\rho(\sigma_j(G'_j))|\sigma'(\rho_{j+1}(\sigma_{j+1}(G'_{j+1})))]\!]$$

Now for lemma 4.13 we have that θ_n *is associated to:*

$$\sigma'^{-1}(\Gamma, \sigma'(\Gamma_{j+1})) \vdash \sigma'^{-1}(G_1|\dots|G_j|\sigma'(G_{j+1})) \\ \xrightarrow{\rho'(\sigma'^{-1}(\rho(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))\cup\sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1})))))} \\ [\![\Gamma, \Gamma_{\Lambda'}, \sigma'(\Gamma_{j+1}), \Gamma_{\Lambda'_{j+1}} \vdash \\ \rho'(\sigma'^{-1}(\rho(\sigma_1(G'_1))|\dots|\rho(\sigma_j(G'_j))|\sigma'(\rho_{j+1}(\sigma_{j+1}(G'_{j+1})))))]\!]$$

with inverse substitution ${\theta'}_{j+1}^i$ *where* ρ' *is the mgu of:*
$\sigma'^{-1}(\rho(\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j)) \cup \sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))))$
and:
$\Lambda' = \rho'(\sigma'^{-1}(\rho(\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j))))$
$\Lambda' = \rho'(\sigma'^{-1}(\sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1})))))$

$$(T_1, \dots, T_j, \sigma'(T_{j+1}))\theta_n {\theta'}_{j+1}^i = \\ = [\![\Gamma, \Gamma_\Lambda, \sigma'(\Gamma_{j+1}), \Gamma_{\Lambda'_{j+1}} \vdash \rho'(\sigma'^{-1}(\rho(\sigma_1(G'_1))|\dots|\rho(\sigma_j(G'_j))| \\ |\sigma'(\rho_{j+1}(\sigma_{j+1}(G'_{j+1})))))]\!]$$

Let us call $\Gamma_n = \Gamma \cup \Gamma_{j+1}$.
We have:
$\sigma'^{-1}(\Gamma, \sigma'(\Gamma_{j+1})) = \Gamma_n$
$\sigma'^{-1}(G_1|\dots|G_j|\sigma'(G_{j+1})) = G_1|\dots|G_j|G_{j+1}$
$\Gamma, \Gamma_{\Lambda'}, \sigma'(\Gamma_{j+1}), \Gamma_{\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))} = \Gamma_n, \Gamma_{\Lambda''}$

$$\rho'(\sigma'^{-1}(\rho(\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j)) \cup \sigma'(\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))))) = \\ = \rho'(\rho(\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j)) \cup \rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))) = \\ = \rho''(\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j) \cup \sigma_{j+1}(\Lambda_{j+1}))$$

where ρ'' *is the mgu of* $\sigma_1(\Lambda_1) \cup \dots \cup \sigma_j(\Lambda_j) \cup \sigma_{j+1}(\Lambda_{j+1})$ *for lemma 4.16.*

$$\rho'(\sigma'^{-1}(\rho(\sigma_1(G'_1))|\dots|\rho(\sigma_j(G'_j))|\sigma'(\rho_{j+1}(\sigma_{j+1}(G'_{j+1}))))) = \\ = \rho'(\rho(\sigma_1(G'_1))|\dots|\rho(\sigma_j(G'_j))|\rho_{j+1}(\sigma_{j+1}(G'_{j+1}))) = \\ = \rho''(\sigma_1(G'_1)|\dots|\sigma_j(G'_j)|\sigma_{j+1}(G'_{j+1}))$$

for lemma 4.16.
We have:
$\mathcal{P} \Vdash (\Gamma \vdash G_1|\dots|G_j \xrightarrow{\rho(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))} \Gamma, \Gamma_\Lambda \vdash \rho(\sigma_1(G_1'))|\dots|\rho(\sigma_j(G_j')))$
$\mathcal{P} \Vdash (\Gamma_{j+1} \vdash G_{j+1} \xrightarrow{\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))} \Gamma_{j+1}, \Gamma_{\rho_{j+1}(\sigma_{j+1}(\Lambda))} \vdash \rho_{j+1}(\sigma_{j+1}(G_{j+1}')))$
Using rule (new) we can prove:

$$\mathcal{P} \Vdash (\Gamma_n \vdash G_1|\dots|G_j \xrightarrow{\rho(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))\cup\sigma_{j+1}(\Lambda_{j+1})|_{x\notin\Gamma}} \Gamma_n, \Gamma_{\Lambda''} \vdash \rho(\sigma_1(G_1')), \dots, \rho(\sigma_j(G_j')))$$

$$\mathcal{P} \Vdash (\Gamma_n \vdash G_{j+1} \xrightarrow{\rho_{j+1}(\sigma_{j+1}(\Lambda_{j+1}))\cup\rho(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))|_{x\notin\Gamma_{j+1}}} \Gamma_n, \Gamma_{\Lambda''} \vdash \rho_{j+1}(\sigma_{j+1}(G_{j+1}')))$$

Now we can apply rule (mer) and for the properties of ρ'' we have:

$$\mathcal{P} \Vdash (\Gamma_n \vdash G_1|\dots|G_j|G_{j+1} \xrightarrow{\rho''(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j)\cup\sigma_{j+1}(\Lambda_{j+1}))} \Gamma, \Gamma_{\Lambda''} \vdash \rho''(\sigma_1(G_1')|\dots|\sigma_j(G_j')|\sigma_{j+1}(G_{j+1}')))$$

Our goal in general contains also atoms that are not rewritten.
Let us call them $A_{j+1}, \dots, A_n$ (we can do it thanks to the commutativity axiom).
Our big-step is:
$A_1, \dots, A_n \xrightarrow{\theta} T_1, \dots, T_j, A_{j+1}\theta, \dots, A_n\theta$
where $\theta = \mathrm{mgu}(\mathrm{eqn}(\theta_1) \cup \dots \cup \mathrm{eqn}(\theta_j))$.
Observe that the only variables in $\mathrm{var}(\theta) \cap A_i$ are the x_j. These variables cannot be substituted by complex terms for lemma 4.11 because the instances of the variables in A_i are not substituted. So they can only be renamed but in any case θ^i performs the opposite renaming.
This means that:
$(T_1, \dots, T_j, A_{j+1}\theta, \dots, A_n\theta)\theta^i = [\![\Gamma, \Gamma_{\Lambda''} \vdash \rho''(\sigma_1(G_1')|\dots|\sigma_j(G_j'))|G_{j+1}, \dots|G_n]\!]$
Let us call $\Gamma_r = \Gamma_n \cup \mathrm{var}(A_{j+1}|\dots|A_n)$.
Using rule (idle) we can prove:
$\mathrm{var}(G_i) \vdash G_i \xrightarrow{\Lambda} \mathrm{var}(G_i) \vdash G_i$
for any $j < i \leq n$ with Λ containing only ϵ (note that $\mathrm{var}(A_1) = \mathrm{var}(G_i)$).
We can now use rule (new) to add variables to synchronizations that do not contain them and then we can use rule (mer) to have:

$$\mathcal{P} \Vdash (\Gamma_r \vdash G_1|\dots|G_j|G_{j+1}|\dots|G_n \xrightarrow{\rho''(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))\cup\Lambda_\epsilon} \Gamma_r, \Gamma_\Lambda \vdash \rho''(\sigma_1(G_1')|\dots|\sigma_j(G_j'))|G_{j+1}|\dots|G_n)$$

where Λ_ϵ contains the synchronization ϵ for all nodes not in Γ_n.
Observe that θ is also associated to:

$$\Gamma_r \vdash G_1|\dots|G_j|G_{j+1}|\dots|G_n \xrightarrow{\rho''(\sigma_1(\Lambda_1)\cup\dots\cup\sigma_j(\Lambda_j))\cup\Lambda_\epsilon}$$
$$\Gamma_r, \Gamma_{\Lambda''} \vdash \rho''(\sigma_1(G'_1)|\dots|\sigma_j(G'_j))|G_{j+1}|\dots|G_n$$

with the old inverse substitution and so we have the thesis in this case.
We will now prove the thesis in the general case by induction on the number of restricted variables in the goal.
The basic case has already been proved.
Observe now that for lemma 3.5 if we have the computation:
$[\![\Gamma \vdash \nu x\ G]\!] \underset{\theta}{\overset{\theta'}{\longrightarrow}}{}^{*}\ \nu X'\ T$
then we also have the computation:
$[\![\Gamma, x \vdash G]\!] \underset{\theta}{\overset{\theta''}{\longrightarrow}}{}^{*}\ T$
where:
$X' = \{y \in \mathrm{fn}(T)|\ \mathrm{or}_{\theta''}(y) \subseteq \{x\}\}$
For inductive hypothesis we have that exist Λ and G' such that θ'' is associated to $\Gamma, x \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ with inverse substitution θ^i and $T\theta^i = [\![\Gamma' \vdash G']\!]$ and:
$\mathcal{P} \Vdash (\Gamma, x \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$
We can apply rule (hide) to have:
$\mathcal{P} \Vdash (\Gamma \vdash \nu x\ G \xrightarrow{\Lambda} \Gamma' \vdash \nu X\ G')$
but for the correctness of rule (hide) we must have $X = X'$.
Moreover θ'' is associated to $\Gamma \vdash \nu x\ G \xrightarrow{\Lambda} \Gamma' \vdash \nu X'\ G'$ with inverse substitution θ^i_r and $(\nu X'\ T)\theta^i_r = [\![\Gamma' \vdash \nu X'\ G']\!]$.
This proves the theorem in the general case.

4.2 Full Hoare SHR

We want to define a set of rules for full synchronized hyperedge replacement using Hoare synchronization, following the idea of definition 2.34 for Milner synchronization.

We will call this SHR system full Hoare SHR.

Recall that our axioms are of the form:

$$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda, \pi} \phi \vdash G$$

where all x_i are distinct. Let Γ be $\{x_1, \ldots, x_n\}$.$\Lambda \subseteq \{x_1, \ldots, x_n\} \times Act \times \mathcal{N}^*$ such that Λ is a function in its first argument and we demand that $\mathrm{ar}(a_i) = |\vec{y}_i|$.$\pi$ is a name fusion of Γ and we have:

- $\mathrm{n}(\Lambda) = \{z | \exists x.z \in \mathrm{n}(x)\}$
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$
- $\phi = \pi(\Gamma) \cup \Gamma_\Lambda$

Productions are to be considered as schemas and so are α-convertible with respect to all the names they contain.

Observe that now we can impose without losing generality that Λ is a total function. It is in fact sufficient to use the special action ϵ when no synchronizations are required. This was not possible with Milner synchronization where an action ϵ on a node forbids all the other actions but $\bar{\epsilon}$ from other hyperedges on the same node.

We can now define the inference rules.

Definition 4.6 (Rules for Hoare synchronization)

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

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

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

where σ is a fusion of the variables in Γ and:
$\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$
$\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$

$\Lambda'(\sigma(z)) = \rho(\sigma(\Lambda(z)))$
$\pi'(\sigma(x)) = \rho(\sigma(\pi(x)))$
$\phi' = \pi'(\sigma(\Gamma)) \cup (\mathrm{n}(\Lambda') \setminus \sigma(\Gamma))$

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

where:
$(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
$\phi' = \pi|_\Gamma(\Gamma) \cup (\mathrm{n}(\Lambda|_\Gamma) \setminus \Gamma)$
$Z = \phi \setminus \phi'$

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

where $\Lambda(x) = (\epsilon, \langle\rangle)$ *for each* $x \in \Gamma$.

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

where $\phi' = \pi(\Gamma, x) \cup (\mathrm{n}(\Lambda \uplus \{(x, a, \vec{y})\}) \setminus (\Gamma \cup \{x\}))$

Rule (par) allows the composition of transitions which use disjoint sets of variables. Rule (merge) deals with non injective renamings (observe that we need to start with productions with disjoint sets of names in order to apply rule (par), and then to use this rule to merge names). Rule (res) restricts a name and all the names of nodes created synchronizing only on it. Rule (idle) guarantees that each node can always make an explicit idle step. Rule (new) allows adding an additional name to the interface which performs arbitrary actions. Note that rule (new) is now necessary because in Hoare synchronization also an isolated node can synchronize.

4.3 Mapping from full Hoare SHR to logic programming

We will now extend the translation of definition 4.3 to rules with fusions of old nodes.

Definition 4.7 (Translation for rewriting rules with fusions) *We define the translation operator* $[\![-]\!]$ *as:*

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

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* $\epsilon(\pi(x_i))$.

Observe that, as in definition 4.3, we have no need to translate all the possible variants of the rules.
Notice that the clauses we obtain are synchronized clauses.
We also need to extend the definition 4.5 of substitution associated to a transition:

Definition 4.8 (Substitution associated to a transition with fusions)

Let $\Lambda = \{(x_i, a_i, \vec{y_i})\}_{i=1\ldots n}$ be a synchronization and $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$ a transition. We say that a substitution θ is associated to this transition iff $S = \mathrm{eqn}(\theta)$ verifies:

- $S = S_1 \cup S_2$
- $S_1 = \{x_i = a_i(x'_i, \vec{y}'_i)\}_{x_i \in \mathrm{fn}(G), a_i \neq \epsilon} \cup \{x_i = x'_i\}_{some\ x_i \in \mathrm{fn}(G), a_i = \epsilon}$ *where the x'_i are distinct unless the corresponding x_i are merged by π; x_i and x'_i are different classes of variables and the same holds for x_i and $y'_{i,j}$. We also require that $y'_{i,j} = x'_k$ iff $y_{i,j} = x_k$. If we have an extrusion we allow S_1 to contain an additional binding for the extruded variable.*
- $\mathrm{Dom}(S_2) \cap \mathrm{var}(S_1) = \emptyset$

We define inverse substitution θ^i the substitution that replaces x'_i with $\pi(x_i)$ and $y'_{i,j}$ with $\pi(y_{i,j})$.

We have to extend the lemmas of previous section to this more general case.

Lemma 4.17 *Let $\mathcal{P}$ be a set of productions with fusions, $\mathcal{P}_{free}$ be the same set of productions after the removal of restrictions (and after choosing representatives that are not already used in the derivation).*
$\mathcal{P}_{free} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi, Y \vdash G')$ *iff:*
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash \nu Y\ G')$
where Y is the set of restricted variables in the productions used.
Proof
This is analogous to the proof of the corresponding lemma (lemma 4.9) for productions without fusions.

Lemma 4.18 *Let $\mathcal{P}$ be a set of productions with fusions. If:*
$\mathcal{P} \Vdash \Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$
then we can build a proof of this transition that never uses the rule (res) as premise of some other rules.
Proof

Suppose (res) is the premise of a (par) rule. We can suppose for symmetry that (res) is the first premise:

$$(par)\ \frac{(res)\ \dfrac{\Gamma,x\vdash G_1\xrightarrow{\Lambda,\pi}\phi\vdash G_2}{\Gamma\vdash\nu x\ G_1\xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma}\phi'\vdash\nu Z\ G_2}\qquad \Gamma'\vdash G_1'\xrightarrow{\Lambda',\pi'}\phi'\vdash G_2'}{\Gamma,\Gamma'\vdash(\nu x\ G_1)|G_1'\xrightarrow{\Lambda|_\Gamma\cup\Lambda',\pi|_\Gamma\cup\pi'}\phi,\phi'\vdash(\nu Z\ G_2)|G_2'}$$

Then we can build:

$$(res)\ \frac{(par)\ \dfrac{\Gamma,x\vdash G_1\xrightarrow{\Lambda,\pi}\phi\vdash G_2\qquad \Gamma'\vdash G_1'\xrightarrow{\Lambda',\pi'}\phi'\vdash G_2'}{\Gamma,x,\Gamma'\vdash G_1|G_1'\xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'}\phi,\phi'\vdash G_2|G_2'}}{\Gamma,\Gamma'\vdash\nu x\ G_1|G_1'\xrightarrow{(\Lambda\cup\Lambda')|_{\Gamma\cup\Gamma'},(\pi\cup\pi')|_{\Gamma\cup\Gamma'}}\phi,\phi'\vdash\nu Z\ G_2|G_2'}$$

and the thesis holds because:

$$\Gamma,\Gamma'\vdash\nu x\ G_1|G_1'\xrightarrow{(\Lambda\cup\Lambda')|_{\Gamma\cup\Gamma'},(\pi\cup\pi')|_{\Gamma\cup\Gamma'}}\phi,\phi'\vdash\nu Z\ G_2|G_2'=$$
$$=\Gamma,\Gamma'\vdash(\nu x\ G_1)|G_1'\xrightarrow{\Lambda|_\Gamma\cup\Lambda',\pi|_\Gamma\cup\pi'}\phi,\phi'\vdash(\nu Z\ G_2)|G_2'$$

because $(\Gamma\cup\{x\}\cup\phi)\cap(\Gamma'\cup\phi')=\emptyset$.
Suppose (res) is the premise of a (merge) rule:

$$(merge)\ \frac{(res)\ \dfrac{\Gamma,x\vdash G_1\xrightarrow{\Lambda,\pi}\phi\vdash G_2}{\Gamma\vdash\nu x\ G_1\xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma}\phi'\vdash\nu Z\ G_2}}{\sigma(\Gamma)\vdash\sigma(\nu x\ G_1)\xrightarrow{\Lambda|_\Gamma',\pi|_\Gamma'}\phi''\vdash\rho(\sigma(\nu Z\ G_2))}$$

where:
$\phi'=\pi|_\Gamma(\Gamma)\cup(\mathrm{n}(\Lambda|_\Gamma)\setminus\Gamma)$
$Z=\phi\setminus\phi'$
$\Lambda|_\Gamma'(\sigma(z))=\rho(\sigma(\Lambda|_\Gamma(z)))$
$\pi|_\Gamma'(\sigma(x))=\rho(\sigma(\pi|_\Gamma(x)))$
$\phi''=\pi|_\Gamma'(\sigma(\Gamma))\cup(\mathrm{n}(\Lambda|_\Gamma')\setminus\sigma(\Gamma))$
Observe that Z *contains* x *iff* $\pi(x)=x$ *and the names in* $\Lambda(x)$ *that do not occur in other places in any case.*
We can extend σ *so that it is the identity on* x *and build:*

$$(res)\ \frac{(merge)\ \dfrac{\Gamma,x\vdash G_1\xrightarrow{\Lambda,\pi}\phi\vdash G_2}{\sigma(\Gamma,x)\vdash\sigma(G_1)\xrightarrow{\Lambda',\pi'}\phi'''\vdash\rho(\sigma(G_2))}}{\sigma(\Gamma)\vdash\nu x\ \sigma(G_1)\xrightarrow{\Lambda'|_{\sigma(\Gamma)},\pi'|_{\sigma(\Gamma)}}\phi''''\vdash\nu Z'\ \rho(\sigma(G_2))}$$

where:
$\Lambda'|_{\sigma(\Gamma)}(\sigma(z))=\rho(\sigma(\Lambda|_\Gamma(z)))$
$\pi'|_{\sigma(\Gamma)}(\sigma(x))=\rho(\sigma(\pi|_\Gamma(x)))$

$\phi''' = \pi'(\sigma(\Gamma, x)) \cup (\mathrm{n}(\Lambda') \setminus \sigma(\Gamma, x))$
$\phi'''' = \pi'|_{\sigma(\Gamma)}(\sigma(\Gamma)) \cup (\mathrm{n}(\Lambda'|_{\sigma(\Gamma)}) \setminus \sigma(\Gamma)) = \phi''$
$Z' = \phi''' \setminus \phi''''$
Observe that Z' contains x iff $\pi'(x) = x$, i. e. iff $\pi(x) = x$, and the names in $\Lambda'(x)$ that do not occur in other places in any case. If ρ is the identity on these names then these names are also in $\Lambda(x)$, otherwise they are not in $\mathrm{fn}(\rho(\sigma(G_2)))$ and we can remove the corresponding restrictions.
In any case we have $Z = Z'$ as required.
Suppose (res) is the premise of a (new) rule:

$$\textit{(new)}\ \frac{\textit{(res)}\ \dfrac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \phi' \vdash \nu Z\ G_2}}{\Gamma, x' \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma \uplus \{(x',a,\vec{y})\},\pi|_\Gamma} \phi'' \vdash \nu Z\ G_2}$$

Then we can build:

$$\textit{(res)}\ \frac{\textit{(new)}\ \dfrac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\Gamma, x, x' \vdash G \xrightarrow{\Lambda \uplus \{(x',a,\vec{y})\},\pi} \phi''' \vdash G_2}}{\Gamma, x' \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma \uplus \{(x',a,\vec{y})\},\pi|_\Gamma} \phi'' \vdash \nu Z\ G_2}$$

and so the lemma holds.

Theorem 4.4 (Mapping of full Hoare SHR: correctness) *Let $\mathcal{P}$ be a set of productions of an SHR system as defined in definitions 2.33 and 4.6. Let P be the program of logic programming with restriction obtained by translating the productions in $\mathcal{P}$ according to definition 4.7. If:*
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G')$
and in the proof of the synchronization we have at most an extrusion then we can have in P the following big-step of synchronized logic programming with restriction:
$[\![\Gamma \vdash G]\!] \overset{\theta}{\rightarrow}^* T$
where θ is associated to $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$ with inverse substitution θ^i and $T\theta^i = [\![\phi \vdash G']\!]$.
Proof
We will prove it by rule induction.
Observe that thanks to lemmas 3.6 and 4.17 we can restrict ourselves to the case of productions without restrictions and thanks to lemma 4.18 we can restrict ourselves to the case of goals without restrictions for all the rules but (res).
Axioms)
Suppose we have a production:
$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \phi \vdash G$
where:

$\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\dots n}$
and π *is a fusion of the* x_i*.*
Then we have in P *a clause that, up to injective renaming of variables, is:*
$s(a_1(\pi(x_i), \pi(\vec{y}_1)), \dots, a_n(\pi(x_n), \pi(\vec{y}'_n)))\theta_v : -[\![\phi \vdash G']\!]\theta_v$
where $\theta_v = \{[x'_i/x_i]\ [y'_{i,j}/y_{i,j}] | \textit{the domain variable of the binding is in } \mathrm{Im}(\pi)\}$
and where we have $\pi(x_i)$ *instead of* $a_i(\pi(x_i), \pi(\vec{y}_i))$ *if* $a_i = \epsilon$*.*
Observe that the variables in $\mathrm{Dom}(\theta_v)$ *are the variables* v *such that* $\pi(v) = v$*, so these variables are in* $\mathrm{Im}(\pi)$ *and they are all and the only variables in* ϕ*.*
Let us consider the goal:
$[\![s(x_1, \dots, x_n)]\!] = s(x_1, \dots, x_n)$
It unifies with the clause variant:
$s(a_1(\pi(x_i), \pi(\vec{y}_1)), \dots, a_n(\pi(x_n), \pi(\vec{y}'_n)))\theta_v : -[\![\phi \vdash G']\!]\theta_v$
with mgu $\theta = \mathrm{mgu}(S)$ *where:*
$S = \{x_i = a_i(\pi(x_i)\theta_v, \pi(\vec{y}_i)\theta_v) | a_i \neq \epsilon\} \cup \{x_i = \pi(x_i)\theta_v | a_i = \epsilon\}$
Observe that S *is in resolved form because the set of variables of the goal and the set of variables in the clause variant are disjoint.*
So $S = \mathrm{eqn}(\theta)$*.*
We can see that θ *is associated to* $x_1, \dots, x_n \vdash s(x_1, \dots, x_n) \xrightarrow{\Lambda,\pi} \phi \vdash G$*.*
We have:
$\theta^i = \{[\pi(x_i)/x_i\theta_v]\ [\pi(y_{i,j})/y_{i,j}\theta_v]\}$
Observe also that the result of the computation is:
$T = [\![\phi \vdash G']\!]\theta_v\theta = [\![\phi \vdash G']\!]\theta_v$
because $\mathrm{var}(G'\theta_v) \cap \mathrm{Dom}(\theta) = \emptyset$*.*
So because π *is the identity on all the variables in* G' *we have:*
$T\theta^i = [\![\phi \vdash G']\!]\theta_v\theta^i = [\![\phi \vdash G']\!]$
as required.
Rule (par))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2 \qquad \Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \phi' \vdash G'_2}{\Gamma, \Gamma' \vdash G_1 | G'_1 \xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'} \phi, \phi' \vdash G_2 | G'_2}$$

where $(\Gamma \cup \phi) \cap (\Gamma' \cup \phi') = \emptyset$*.*
By inductive hypothesis we have:

- $[\![\Gamma \vdash G_1]\!] \xrightarrow{\theta}{}^* T$
  where θ *is associated to* $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\phi \vdash G_2]\!]$*.*
- $[\![\Gamma' \vdash G'_1]\!] \xrightarrow{\theta'}{}^* T'$
  where θ *is associated to* $\Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \phi' \vdash G'_2$ *with inverse substitution* θ'^i *and* $T'\theta'^i = [\![\phi' \vdash G'_2]\!]$*.*

We can suppose that the sets of variables used in the two computations are disjoint, and so we have:
$[\![\Gamma, \Gamma' \vdash G_1|G_1']\!] \xrightarrow{\theta\theta'} T, T'$
Observe that $\mathrm{eqn}(\theta\theta') = \mathrm{eqn}(\theta) \cup \mathrm{eqn}(\theta')$ *and that* $\theta\theta'$ *is associated to:*
$\Gamma, \Gamma' \vdash G_1|G_1' \xrightarrow{\Lambda\cup\Lambda', \pi\cup\pi'} \phi, \phi' \vdash G_2|G_2'$ *with inverse substitution* $\theta^i\theta'^i$.
So we have:
$(T, T')\theta^i\theta'^i = T\theta^i, T'\theta'^i = [\![\phi \vdash G_2]\!], [\![\phi' \vdash G_2']\!] = [\![\phi, \phi' \vdash G_2|G_2']\!]$
Rule (merge))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \phi' \vdash \rho(\sigma(G_2))}$$

where σ *is a fusion of the variables in* Γ *and:*
$\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$
$\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
$\Lambda'(\sigma(z)) = \rho(\sigma(\Lambda(z)))$
$\pi'(\sigma(x)) = \rho(\sigma(\pi(x)))$
$\phi' = \pi'(\sigma(\Gamma)) \cup (\mathrm{n}(\Lambda') \setminus \sigma(\Gamma))$
For inductive hypothesis we have:
$[\![\Gamma \vdash G_1]\!] \xrightarrow{\theta}{}^{*} T$
where θ *is associated to* $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\phi \vdash G_2]\!]$.
For lemma 4.7 we have:
$\sigma([\![\Gamma \vdash G_1]\!]) \xrightarrow{\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))}{}^{*} T\,\mathrm{mgu}(\mathrm{eqn}(\theta) \cup \mathrm{eqn}(\sigma))$
if $\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta)))$ *exists.*
In this case we have:
$\sigma([\![\Gamma \vdash G_1]\!]) = [\![\sigma(\Gamma) \vdash \sigma(G_1)]\!]$
We have:
$\mathrm{eqn}(\theta) = S_1 \cup S_2$
$S_1 = \{x_i = a_i(x_i', \vec{y}_i')\}_{x_i \in \mathrm{fn}(G_1), a_i \neq \epsilon} \cup \{x_i = x_i'\}_{some\ x_i \in \mathrm{fn}(G_1), a_i = \epsilon}$
where the x_i' *are distinct unless the corresponding* x_i *are merged by* π; x_i *and* x_i' *are disjoint classes of variables and the same holds for* x_i *and* $y_{i,j}$. *We also have that if* $y_{i,j} = x_k$ *then* $y_{i,j}' = x_k'$.
$\sigma(\mathrm{eqn}(\theta)) = \sigma(S_1) \cup \sigma(S_2)$
$\sigma(S_1) = \{\sigma(x_i) = a_i(x_i', \vec{y}_i')\}_{x_i \in \mathrm{fn}(G_1), a_i \neq \epsilon} \cup \{\sigma(x_i) = x_i'\}_{some\ x_i \in \mathrm{fn}(G_1), a_i = \epsilon}$
Let us choose for each equivalence class determined by σ *a representative that is in* $\mathrm{fn}(G_1)$ *(if it exists) and let us define* $r(i) = j$ *iff the representative of the class of* x_i *is* x_j.

The previous set of equations is equivalent to:

$$\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)})\}_{r(i) \text{ is defined}, a_i \neq \epsilon} \cup$$
$$\cup \{a_i(x'_i, \vec{y}'_i) = a_{r(i)}(x'_{r(i)}, y'_{r(i)}) | r(i) \text{ is defined}, a_i \neq \epsilon\} \cup$$
$$\cup \{\sigma(x_i) = x'_i\}_{\text{some } x_i \in \mathrm{fn}(G_1), a_i = \epsilon}$$

Observe that by hypothesis $\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$ *and so the previous set of equations is also equivalent to:*

$$\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \text{ is defined}, a_i \neq \epsilon\} \cup$$
$$\cup \{x'_i = x'_{r(i)} \ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \text{ is defined } \wedge i \neq r(i)\} \cup$$
$$\cup \{\sigma(x_i) = x'_i \text{ for some } i \text{ such that } r(i) \text{ is defined } \wedge x_i \in \mathrm{fn}(G_1) \wedge a_i = \epsilon\}$$

In this case:

$$\mathrm{mgu}(\sigma(\mathrm{eqn}(\theta))) =$$
$$= \mathrm{mgu}(\{\sigma(x_i) = a_{r(i)}(x'_{r(i)}, \vec{y}'_{r(i)}) | r(i) \text{ is defined}, a_i \neq \epsilon\} \cup$$
$$\cup \{x'_i = x'_{r(i)} \ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \text{ is defined } \wedge i \neq r(i)\} \cup$$
$$\cup \{\sigma(x_i) = x'_i \text{ for some } i \text{ such that } r(i) \text{ is defined } \wedge x_i \in \mathrm{fn}(G_1) \wedge a_i = \epsilon\} \cup \sigma(S_2))$$

Observe that:

$$\{x'_i = x'_{r(i)} \ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \text{ is defined } \wedge i \neq r(i)\} \cup$$
$$\cup \{\sigma(x_i) = x'_i \text{ for some } i \text{ such that } r(i) \text{ is defined } \wedge x_i \in \mathrm{fn}(G_1) \wedge a_i = \epsilon\}$$

is a renaming and so for lemma 4.2 we can put it in resolved form.
We can choose as representatives the $\sigma(x_i)$ *when* $a_i \neq \epsilon$. *We will call* x''_i *and* $y''_{i,j}$ *the representatives of* x'_i *and* $y'_{i,j}$. *Observe that the two classes of variables are not necessarily disjoint.*
So we can put the whole set of equations in the resolved form:

$$\mathrm{eqn}(\theta') =$$
$$= \{w_{i,j} = a_{r(i)}(x''_{r(i)}, \vec{y}''_{r(i)}) | w_{i,j} \text{ is in the class of } \sigma(x_i) \wedge r(i) \text{ is defined } \wedge a_i \neq \epsilon\} \cup$$
$$\cup (\sigma(S_2))' \cup S_3$$

where S_3 *is the resolved form of the renaming after having removed the equalities on* $w_{i,j}$ *and* $(\sigma(S_2))'$ *is* $\sigma(S_2)$ *where we have made the substitutions necessary to put the first part in this form.*

So the mgu exists.
Observe that we can reorder it in:

$$\begin{aligned}\mathrm{eqn}(\theta') = \{\sigma(x_i) = a_{r(i)}(x''_{r(i)}, \vec{y}''_{r(i)}) | r(i) \textit{ is defined } \wedge a_i \neq \epsilon\} \cup \\ \cup \{\sigma(x_i) = x''_i \textit{ for some } i \textit{ such that } r(i) \textit{ is defined} \wedge x_i \in \mathrm{fn}(G_1) \wedge a_i = \epsilon\} \cup \\ \cup \{x'_i = x'_{r(i)}\ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \textit{ is defined } \wedge i \neq r(i)\} \cup S'_2 = S'_1 \cup S'_2\end{aligned}$$

where $\mathrm{Dom}(S'_2) \cap \mathrm{var}(S'_1) = \emptyset$.
Let us prove that ρ *merges* $\sigma(v_1)$ *and* $\sigma(v_2)$ *with* v_1 *and* v_2 *in* $\mathrm{fn}(G_1)$ *iff* v'_1 *and* v'_2 *(where we use* v_1 *and* v_2 *instead of* v'_1 *and* v'_2 *if the latter ones do not exist) are the same variable or they are merged in* θ'*.*
First we need to prove that this is a well-defined statement, i. e. that if $\sigma(v_1) = \sigma(v_2)$ *then* v'_1 *and* v'_2 *are merged in* θ'*.*
This is true because v_1 *and* v_2 *must be* x_j *and* x_k *for some* j *and* k *(otherwise* σ *is the identity on them and we have* $v_1 = v_2$ *which implies the thesis). As these variables are in* $\mathrm{fn}(G_1)$*, we have in* θ'*:*
$x'_i = x'_{r(i)}$
for both variables if the synchronization is not ϵ*, otherwise we may have:*
$\sigma(x_i) = x'_i$
for both variables if the synchronization is ϵ*.*
If one of these bindings does not exist the correspondent variable does not exist and the thesis holds because x_i *is used instead of* x'_i*.*
In any case we have the well-definiteness.
$\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
The fusions in θ' *are given by:*

$$\begin{aligned}\mathrm{mgu}(\{x'_i = x'_{r(i)}\ \vec{y}'_i = \vec{y}'_{r(i)} | r(i) \textit{ is defined } \wedge i \neq r(i)\} \cup \\ \cup \{\sigma(x_i) = x'_{r(i)} \textit{ for some } i \textit{ such that } r(i) \textit{ is defined } \wedge x_i \in \mathrm{fn}(G_1) \wedge a_i = \epsilon\})\end{aligned}$$

Observe that for the hypothesis of variables in $\mathrm{fn}(G_1)$ *this is equal to:*
$\mathrm{mgu}(\{x'_i = x'_j\ \vec{y}'_i = \vec{y}'_j | \sigma(x_i) = \sigma(x_j)\}$
where we removed only equations stating equalities with other variables and where we use v_i *instead of* v'_i *if the latter does not exist.*
But we also have for the definition of substitution associated to a synchronization that $x'_i = x'_j$ *iff* $\pi(x_i) = \pi(x_j)$*.*
So we can rewrite the mgu as:
$\mathrm{mgu}(\{x'_i = x'_j\ \vec{y}'_i = \vec{y}'_j | \sigma(x_i) = \sigma(x_j)\} \cup \{x'_i = x'_j | \pi(x_i) = \pi(x_j)\})$
So the two mgus have the same form as required and we can choose $(\rho(\sigma(v_i)))'$ *as representative of the equivalence class of* v'_i*.*

Therefore θ' *is associated to* $\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \phi' \vdash \rho(\sigma(G_2))$ *with* ${\theta'}^i$ *that maps* $(\rho(\sigma(v_i)))'$ *to* $(\rho(\sigma(v_i)))$ *and so:*

$$\begin{aligned} T\,\mathrm{mgu}(\mathrm{eqn}(\theta) \cup \mathrm{eqn}(\sigma)){\theta'}^i &= \\ &= \sigma(T)\,\mathrm{mgu}(\mathrm{eqn}(\theta) \cup \mathrm{eqn}(\sigma)){\theta'}^i = \\ &= \sigma(T)\theta'{\theta'}^i = \rho(\sigma(T)\theta^i) = \\ &= \rho(\sigma(T\theta^i)) = \rho(\sigma([\![\phi \vdash G_2]\!])) = [\![\phi' \vdash \rho(\sigma(G_2))]\!] \end{aligned}$$

Rule (res))

$$\frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \phi' \vdash \nu Z\ G_2}$$

where:
$(\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x$
$\phi' = \pi|_\Gamma(\Gamma) \cup (\mathrm{n}(\Lambda|_\Gamma) \setminus \Gamma)$
$Z = \phi \setminus \phi'$
For inductive hypothesis we have:
$[\![\Gamma, x \vdash G_1]\!] \underset{\theta}{\overset{\theta'}{\longrightarrow}}{}^{*}\ T$
where θ' *is associated to* $\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\phi \vdash G_2]\!]$.
For lemma 3.5 we have:
$\nu x\ G_1 \underset{\theta}{\overset{\theta''}{\longrightarrow}}{}^{*}\ \nu X'\ T$
where:
$X' = \{y \in \mathrm{fn}(T) |\ \mathrm{or}_{\theta'}(y) \subseteq \{x\}\}$
Observe that $\mathrm{eqn}(\theta')$ *is of the form* $S_1 \cup S_2$ *where:*
$S_1 = \{x_i = a_i(x'_i, \vec{y}'_i)\}_{x_i \in \mathrm{fn}(G_1), a_i \neq \epsilon} \cup \{x_i = x'_i\}_{some\ x_i \in \mathrm{fn}(G_1), a_i = \epsilon}$
The only variables y *in* $\mathrm{fn}(G_2)$ *that can have* $\mathrm{or}_{\theta'}(y) \subseteq \{x\}$ *are* x, x' *and the variables in* $\vec{y}'$, *and this happens if these variables never appear in other bindings. The other variables have at least a binding whose domain is not* x. *If at least one of these bindings is the correspondent of a visible synchronization then these variables are in* ϕ' *and we have* $X' = \phi \setminus \phi'$ *as required.*
Otherwise a variable may appear in a binding of another extruded variable, but in this case we have two extrusions because the same variable is also extruded by x, *what we have forbidden.*
For lemma 3.4 $\theta''|_{\mathrm{fn}(\nu x\ G_1)} = \theta'|_{\mathrm{fn}(\nu x\ G_1)}$ *and all the other bindings in* θ'' *have a domain variable neither in* $\mathrm{var}(\nu x\ G_1)$ *nor in* $\mathrm{var}(\nu X'\ T)$ *unless it is an extruded variable.*
Then θ'' *is associated to* $\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \phi' \vdash \nu Z\ G_2$ *with inverse substitution*

${\theta'}^i = \theta^i|_{v \in \mathrm{fn}(\nu X'\ T)}$. *We can change the names of all the variables according to* θ^i *because the restricted variables can be* α*-converted.*
So we have:
$(\nu X'\ T){\theta'}^i = \nu Z\ T\theta^i = \nu Z\ [\![\phi \vdash G_2]\!] = [\![\phi' \vdash \nu Z\ G_2]\!]$.
Rule (idle))

$$\Gamma \vdash G \xrightarrow{\Lambda, id} \Gamma \vdash G$$

where $\Lambda(x) = (\epsilon, \langle\rangle)$ *for each* $x \in \Gamma$.
Observe that for each program P *we have the transition:*
$[\![\Gamma \vdash G]\!] \xrightarrow{\epsilon} [\![\Gamma \vdash G]\!]$
and that ϵ *is associated to:*
$\Gamma \vdash G \xrightarrow{\Lambda, id} \Gamma \vdash G$
with empty θ^i *and so we have the thesis.*
Rule (new))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda, \pi} \phi \vdash G_2}{\Gamma, x \vdash G_1 \xrightarrow{\Lambda \uplus \{(x, a, \vec{y})\}} \phi' \vdash G_2}$$

For inductive hypothesis we have in P *the following big-step of synchronized logic programming with restriction:*
$[\![\Gamma \vdash G_1]\!] \xrightarrow{\theta}{}^{*}\ T$
where θ *is associated to:*
$\Gamma \vdash G_1 \xrightarrow{\Lambda, \pi} \phi \vdash G_2$
with inverse substitution θ^i *and* $T\theta^i = [\![\phi \vdash G_2]\!]$.
But because $x \notin \mathrm{fn}(G_1)$ *we have that* θ *is also associated to:*
$\Gamma, x \vdash G_1 \xrightarrow{\Lambda \uplus \{(x, a, \vec{y})\}, \pi} \phi' \vdash G_2$
with the same inverse substitution. The thesis follows.

We need to explain better the problem of the extrusions.
This problem occurs because SHR performs all steps in parallel, so extruded names are hidden during the transition and from the outside we cannot see what happens on them. In synchronized logic programming a big-step is a sequential composition of many small-steps, so if the small-step which performs an extrusion precedes a small-step that performs a synchronization on the extruded name, then this synchronization is visible from the outside. Moreover this synchronization can extrude further names, thus the final behaviour of the two systems becomes more and more different for each extrusion.

We can now prove a completeness result for our translation.

Theorem 4.5 (Mapping of full Hoare SHR: completeness) *Let $\mathcal{P}$ be a set of productions of an SHR system as defined in definitions 2.33 and 4.6. Let P be the program of logic programming with restriction obtained by translating the productions in $\mathcal{P}$ according to definition 4.7. If we have in P a big-step of logic programming with restriction:*
$[\![\Gamma \vdash G]\!] \stackrel{\theta}{\to}^{*} T$
and in the computation we have at most an extrusion then there exist Λ, π, ϕ and G' such that θ is associated to $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$ with inverse substitution θ^i and $T\theta^i = [\![\phi \vdash G']\!]$ and:
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G')$
Proof
Observe that thanks to lemmas 3.6 and 4.17 we can restrict ourselves to the case of productions without restrictions.
Let us first suppose G without restrictions.
Our big-step is composed by many small-steps.
Let A_i be the atom substituted in small-step i, θ_i the resultant mgu, R_i the SHR axiom that corresponds to the clause used at step i and Λ_i the synchronization used in it. For the atoms that are not rewritten let R_i and Λ_i be the ones that exist for rule (idle) applied to the graph corresponding to the atom.
Let us choose the names for these axioms in order to have the first occurrence of each name in A_i to coincide with the name in the same position in the head of R_i.
Let us define σ as a variable substitution that maps all variables in R_{i+1} already used in $R_1, \ldots, R_i$ to new variables.
Observe that $\mathrm{var}(\sigma(R_i)) \cap \mathrm{var}(\sigma(R_j)) = \emptyset$ *for each i, j with $i \neq j$.*
We will use the following notation:
$\sigma(R_j) = \Gamma_j \vdash G_j \xrightarrow{\Lambda_j,\pi_j} \phi_j \vdash G'_j$
with:
$\Lambda_j = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$
We have:
$A_1, \ldots, A_j \xrightarrow{\theta_1 \cdots \theta_j} T_1, \ldots, T_j$
By applying each clause to the translation of G_j into a goal:
$[\![\Gamma_j \vdash G_j]\!] \xrightarrow{\theta'_j} T'_j\theta'_j$
where:

$$\mathrm{eqn}(\theta'_j) = \{x_i = a_i(\pi_j(x_i)\theta_v, \pi_j(\vec{y}_i)\theta_v) | a_i \neq \epsilon\} \cup$$
$$\cup \{x_i = \pi_j(x_i)\theta_v \textit{ for some } i \textit{ such that } a_i = \epsilon\}$$

$\theta_v = \{[x'_i/x_i]\ [y'_{i,j}/y_{i,j}] | \textit{the domain variable of the binding is not renamed by } \pi_j\}$

We can see that θ'_j *is associated to* $\Gamma_j \vdash G_j \xrightarrow{\Lambda_j,\pi_j} \phi_j \vdash G'_j$ *(the proof is essentially equal to the proof of the correctness for the axioms).*
We have:
$\theta'^i_j = \{[\pi_j(x_i)/x'_i]\ [\pi_j(y_{i,j})/y'_{i,j}]\}$
Observe also that the result of the computation is:
$T'_j = [\![\phi_j \vdash G'_j]\!]\theta_v\theta'_j = [\![\phi_j \vdash G'_j]\!]\theta_v$
because $\mathrm{var}(G'_j\theta_v) \cap \mathrm{Dom}(\theta'_j) = \emptyset$.
So because π_j *is the identity on all the variables in* G'_j *we have:*
$T'_j\theta'^i_j = [\![\phi_j \vdash G'_j]\!]\theta_v\theta'^i_j = [\![\phi_j \vdash G'_j]\!]$.
Observe that, because the sets of variables of these computations are disjoint by lemma 4.8, we have:
$G_1, \ldots, G_j \xrightarrow{\theta'_1\cdots\theta'_j} T'_1\theta'_1, \ldots, T'_j\theta'_j$
Let σ' *be the variable substitution that maps* G_i *into* A_i.
Observe that it is well-defined because all the variables in G_i *are distinct and that it is a name fusion because the first occurrence of each variable has not been renamed.*
Now we can apply lemma 4.7 and lemma 4.16 to have:
$\sigma'(G_1, \ldots, G_j) \xrightarrow{\mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta'_1\cdots\theta'_j)))} \sigma'(G'_1\theta'_1, \ldots, G'_j\theta'_j)\,\mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta'_1\cdots\theta'_j)))$
But $\sigma'(G_1, \ldots, G_j) = A_1, \ldots, A_j$ *by definition and because the clauses used in the two computations are the sames we have that:*
$\mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta'_1\cdots\theta'_j))) = \theta_1\cdots\theta_j$
$\sigma'(G'_1\theta'_1, \ldots, G'_j\theta'_j)\,\mathrm{mgu}(\sigma'(\mathrm{eqn}(\theta'_1\ldots\theta'_j))) = T_1, \ldots, T_j$
We can apply rule (par) $j-1$ *times to have:*
$\mathcal{P} \Vdash \Gamma_1, \ldots, \Gamma_j \vdash G_1|\ldots|G_j \xrightarrow{\Lambda_1\cup\cdots\cup\Lambda_j,\pi_1\cup\cdots\cup\pi_j} \phi_1, \ldots, \phi_j \vdash G'_1|\ldots|G'_j$
For what proved in theorem 4.4 for rule (par) we have that $\theta'_1\cdots\theta'_j$ *is associated to:*
$\Gamma_1, \ldots, \Gamma_j \vdash G_1|\ldots|G_j \xrightarrow{\Lambda_1\cup\cdots\cup\Lambda_j,\pi_1\cup\cdots\cup\pi_j} \phi_1, \ldots, \phi_j \vdash G'_1|\ldots|G'_j$
with inverse substitution composed by the union of the inverse substitutions. Let us call it θ^i_u.
We have:
$(T'_1\theta'_1, \ldots, T'_j\theta'_j)\theta^i_u = [\![\phi_1, \ldots, \phi_j \vdash G'_1|\ldots|G'_j]\!]$
We can apply rule (merge) to have:
$\mathcal{P} \Vdash \Gamma \vdash \sigma'(G_1|\ldots|G_j) \xrightarrow{\Lambda',\pi'} \phi' \vdash \rho(\sigma(G'_1|\ldots|G'_j))$
where:
$\Gamma = \Gamma_1, \ldots, \Gamma_j$
$\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
$\Lambda'(\sigma(z)) = \rho(\sigma(\Lambda(z)))$
$\pi'(\sigma(x)) = \rho(\sigma(\pi(x)))$
$\phi' = \pi'(\sigma(\Gamma)) \cup (\mathrm{n}(\Lambda') \setminus \sigma(\Gamma)) = \phi_1, \ldots, \phi_j$

For what proved in the correctness of rule (merge) $\theta_1 \cdots \theta_n$ *is associated to:*
$\Gamma \vdash A_1|\ldots|A_j \xrightarrow{\Lambda',\pi'} \phi' \vdash \rho(\sigma(G'_1|\ldots|G'_j))$[1]
with an inverse substitution θ'^i *such that:*
$(T_1,\ldots,T_n)\theta'^i = [\![\phi' \vdash \rho(\sigma(G'_1|\ldots|G'_j))]\!]$
Observe that $\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a,\vec{v}) \wedge \Lambda(y) = (a,\vec{w})$ *because for not* ϵ *transitions this is guarenteed by the existence of* $\theta_1 \cdots \theta_j$ *and for* ϵ *transitions this is guarenteed by lemma 4.11.*
Notice that ρ *exists for lemma 4.2 because it is the mgu of a renaming substitution.*
This proves the thesis in the case of goals without restrictions.
We will now prove the thesis in the general case by induction on the number of restricted variables in the goal.
The basic case has already been proved.
Observe now that for lemma 3.5 if we have the computation:
$[\![\Gamma \vdash \nu x\ G]\!] \underset{\theta}{\xrightarrow{\theta'}}{}^{*} \nu X'\ T$
then we also have the computation:
$[\![\Gamma, x \vdash G]\!] \underset{\theta}{\xrightarrow{\theta''}}{}^{*} T$
where:
$X' = \{y \in \mathrm{fn}(T) |\ \mathrm{or}_{\theta''}(y) \subseteq \{x\}\}$
For inductive hypothesis we have that there exist Λ, π, ϕ *and* G' *such that* θ'' *is associated to* $\Gamma, x \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$ *with inverse substitution* θ^i *and* $T\theta^i = [\![\phi \vdash G']\!]$ *and:*
$\mathcal{P} \Vdash (\Gamma, x \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G')$
We can apply rule (res) to have:
$\mathcal{P} \Vdash (\Gamma \vdash \nu x\ G \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \phi' \vdash \nu Z\ G')$
but for the correctness of rule (res) we must have:
$Z = X'$
Moreover θ'' *is associated to* $\Gamma \vdash \nu x\ G \xrightarrow{\Lambda|_\Gamma,\pi|_\Gamma} \phi' \vdash \nu Z\ G'$ *with inverse substitution* θ'^i *and* $(\nu Z\ T)\theta'^i = [\![\phi' \vdash \nu Z\ G']\!]$.
This proves the theorem in the general case.

[1]Here A_i is more precisely the hyperedge that corresponds to the atom.

Chapter 5

Meta-interpreter

In this chapter we present a prototype of a meta-interpreter for synchronized logic programming with restriction, which is also (thanks to the results of chapter 4) a simulator of SHR Hoare systems. Section 5.1 contains the code of the meta-interpreter written in Sicstus Prolog 3.7.1 and a short description of the project choices we have done while section 5.2 contains some sample executions.

5.1 The meta-interpreter

In this chapter we present a simple meta-interpreter for synchronized logic programming with restriction.

The starting clause of the meta-interpreter is `use`, which takes as argument the list of clauses of the synchronized program. These clauses cannot contain explicit restrictions, but all the variables in the body of each clause that are not in the head are assumed to be restricted. Notice that each restricted clause is equivalent to a clause of this form (it is sufficient to have disjoint sets of names for restricted and free variables).
The used clauses are inserted into clauses database, so their predicates must not have been already used in the code of the meta-interpreter.
If the clauses are synchronized clauses then the program asks for a goal, otherwise the program stops. This time the restriction can be supplied with a special predicate nu whose arguments are the restricted variables. This predicate must be the first predicate of the goal.
Then the program asks for the number of big-steps to perform.
The program uses a deep-first search on the tree of the possible synchronized computations of the required length and each time it finds one such computation it outputs the resulting goal and the visible substitutions.

Internally the program represents the goal as a list of subgoals and it keeps trace of free variables (and not of bound ones, because it is simpler to compute new free variables, which are all the variables that have been assigned to at least a previously free variable). Only at the end it computes the set of restricted variables.

Now we present the code of the meta-interpreter.

```
%Library inclusions.

:-use_module(library(charsio)).
:-use_module(library(lists)).
:-use_module(library(terms)).

%Clauses that perform the computation.

%Start a synchronized computation with restriction using rules Rules.
%The restricted variables in the clauses are all the variables in
%the body but not in the head.
%The predicate read_term inputs a variant of a term and Names is
%unified with a list of terms var=name where var is the variable
%in the variant and name is its original name.

use(Rules):-
    ass(Rules),
    write('Insert goal (use nu as first predicate
       to hide variables)'),
    nl,read_term(Graph,[variable_names(Names)]),nl,
    write('Number of big-steps:'),read(N),integer(N),
    getRes(Graph,Restricted,TrueGraph),reify(Names,ReifiedNames),
    apply(Restricted,TrueGraph,N,ReifiedNames),fail.

%Simulate NStep big-steps of synchronized logic programming with
%restriction on goal Graph where restricted variables are those in
%Restricted.
%Names contains the bindings between the variables in the goal and
%their names.
%At the end of the computation clean clauses database.

apply(Restricted,Graph,NStep,Names):-
    computeFree(Restricted,Graph,Free),
```

```
    computeFreeNames(Free,Names,FreeNames),!,
    (   compute(Free,Graph,NStep,Names,FreeNames);
        findall(X,toclean(X),L),
        listcall(L),retractall(toclean(X))).

%If all the required big-steps have been done, verify that the
%result is a goal-graph, compute the set of restricted variables
%and print the result.

compute(Free,Graph,0,Names,FreeNames):-
    isGraph(Graph),computeRestricted(Free,Graph,Restricted),
    output(Restricted,Graph,Free,FreeNames,Names),fail.

%Otherwise verify that the intermediate result is a goal-graph and
%then compute an additional big-step.

compute(Free,Graph,NStep,Names,FreeNames):-
    NStep>0,isGraph(Graph),
    step(Free,Graph,FreeNames,Names,NewFree,NewGraph,NewFreeNames),
    NewStep is NStep-1,
    compute(NewFree,NewGraph,NewStep,Names,NewFreeNames).

%Perform the output of a computation.

output([],Graph,Free,FreeNames,Names):-
    !,writeArgsN(Graph,Names),nl,
    writeBindings(FreeNames,Free,Names),nl.
output(Restricted,Graph,Free,FreeNames,Names):-
    OutRes=..['nu'|Restricted],writeN(OutRes,Names),write(','),
    writeArgsN(Graph,Names),nl,
    writeBindings(FreeNames,Free,Names),nl.

%Perform a big-step of the computation. If necessary after each
%small-step update the set of free variables and of their names.

step(Free,[],FreeNames,_Names,Free,[],FreeNames).
step(Free,[true|Xs],FreeNames,Names,NewFree,Xs1,NewFreeNames):-
    !,step(Free,Xs,FreeNames,Names,NewFree,Xs1,NewFreeNames).
step(Free,[X|Xs],FreeNames,Names,NewFree,L,NewFreeNames):-
    clause(X,B),update(Free,FreeNames,Names,Free1,FreeNames1),
```

```
    toGoalList(B,X1),
    step(Free1,Xs,FreeNames1,Names,NewFree,Xs1,NewFreeNames),
    append(X1,Xs1,L).
step(Free,[X|Xs],FreeNames,Names,NewFree,[X|Xs1],NewFreeNames):-
    step(Free,Xs,FreeNames,Names,NewFree,Xs1,NewFreeNames).

%Update the set of free variables and of their names.

update(Free,FreeNames,Names,NewFree,NewFreeNames):-
    term_variables(Free,Vars),
    newvars(Vars,FreeNames,Names,AddFree,AddFreeNames),
    append(Free,AddFree,NewFree),
    append(FreeNames,AddFreeNames,NewFreeNames).

%Auxiliary computing clauses

%Control the syntax of clauses and insert the program clauses in
%Prolog clauses database adding to it some pieces of information
%for the cleaning of the database.

ass([]).
ass([X|Xs]):-ass1(X),ass(Xs).

%Control a single clause and insert it clause in Prolog database.

ass1((H:-B)):-
    control((H:-B)),assert((H:-B)),assert(toclean(retractall(H))),!.
ass1((H)):-control((H)),assert((H)),assert(toclean(retractall(H))),!.

%Control the syntax of a clause.

control((H:-B)):-
    toGoalList(B,L),isGraph(L),H=..[_|Args],verify(Args,Vars),
    term_variables(B,BodyVars),areIn(Vars,BodyVars),!.
control((H)):-H=..[_|Args],verify(Args,Vars),Vars=[].

%Control if a list of terms is composed by variables and function
%symbols applied to variables and compute the set of variables at
%top level.
```

```
verify([],[]).
verify([X|Xs],[X|Ys]):-var(X),verify(Xs,Ys).
verify([X|Xs],Ys):-
    compound(X),X=..[_|Args],allVariables(Args),verify(Xs,Ys).

%Control if a list contains only variables.

allVariables([]).
allVariables([X|Xs]):-var(X),allVariables(Xs).

%Split the input into a set of restricted variables and a list of
%goals.

getRes(',' (Res,Graph),Restricted,TrueGraph):-
    Res=..L,L=[nu|Restricted],!,toGoalList(Graph,TrueGraph).
getRes(Graph,[],TrueGraph):-toGoalList(Graph,TrueGraph).

%Compute the set of free variables (variables in the goal that are
%not restricted).

computeFree(Restricted,Graph,FreeVars):-
    computeFreeVars([],Restricted,Graph,FreeVars).

computeFreeVars(Start,_Restricted,[],Start).
computeFreeVars(Start,Restricted,[X|Xs],Result):-
    X=..L,computeFreeVars1(Start,Restricted,L,Res1),
    computeFreeVars(Res1,Restricted,Xs,Result).

%Compute the set of free variables of a list of atomic items.

computeFreeVars1(Start,_Restricted,[_X],Start).
computeFreeVars1(Start,Restricted,[X,Y|Xs],Result):-
    var(Y),notIn(Y,Restricted),notIn(Y,Start),!,
    computeFreeVars1([Y|Start],Restricted,[X|Xs],Result).
computeFreeVars1(Start,Restricted,[X,_Y|Xs],Result):-
    computeFreeVars1(Start,Restricted,[X|Xs],Result).

%Transform a clause body into a list of goals.

toGoalList((A,B),[A|As]):-!,toGoalList(B,As).
```

```
toGoalList(X,[X]).

%Transform a list of goals into a clause body.

toClause([],nil).
toClause([X],X).
toClause([X,Y|Xs],(X,Rest)):-toClause([Y|Xs],Rest).

%Compute the new set of free variables (free variables whose names
%are not in FreeNames).

newvars([],_,_,[],[]).
newvars([X|Xs],FreeNames,Names,[X|AddFree1],[Name|AddFreeNames1]):-
    nameOf(X,Names,Name),notIn(Name,FreeNames),
    newvars(Xs,FreeNames,Names,AddFree1,AddFreeNames1),!.
newvars([_X|Xs],FreeNames,Names,AddFree,AddFreeNames):-
    newvars(Xs,FreeNames,Names,AddFree,AddFreeNames).

%Verify that the goal is a goal-graph.

isGraph([]).
isGraph([X|Xs]):-X=..L,isGraph1(L),isGraph(Xs).

%Verify that in a list all but the first element are variables.

isGraph1([_X]).
isGraph1([X,Y|Xs]):-var(Y),isGraph1([X|Xs]).

%Compute the set of restricted variables (variables in Graph not in
%FreeTerms (terms assigned to previously free variables)).

computeRestricted(FreeTerms,Graph,Restricted):-
    term_variables(FreeTerms,FreeVars),
    computeFreeVars([],FreeVars,Graph,Restricted).

%Transform a list of Name=Var items into a list of Name='Var' items.

reify([],[]).
reify([Name=Var|Xs],[Name=ReifiedVar|Ys]):-
    write_to_chars(Var,Ch),name(ReifiedVar,Ch),reify(Xs,Ys).
```

```
%Compute the actual names for free variables

computeFreeNames([],_Names,[]).
computeFreeNames([X|Xs],Names,[Y|Ys]):-
    nameOf(X,Names,Y),computeFreeNames(Xs,Names,Ys).

%Compute the name of a variable according to Names: reify the
%variable and search in Names if it has another name.

nameOf(X,Names,Name):-
    write_to_chars(X,Ch),name(Reified,Ch),
    nameOf1(Reified,Names,Name).

%Search Names for another name of the variable.

nameOf1(X,[],X).
nameOf1(X,[Name=Var|_L],Name):-X==Var,!.
nameOf1(X,[_|L],Name):-nameOf(X,L,Name).

%Output functions

%Write a term using a list of names as alias for variables.

writeN(X,Names):-var(X),nameOf(X,Names,Name),write(Name),!.
writeN(X,_Names):-simple(X),write(X).
writeN([X|Xs],Names):-
    write('['),writeArgsN([X|Xs],Names),write(']'),!.
writeN(X,Names):-
    compound(X),X=..[L|Ls],write(L),write('('),writeArgsN(Ls,Names),
    write(')').

%Write a list of terms using writeN.

writeArgsN([],_Names).
writeArgsN([X],Names):-writeN(X,Names).
writeArgsN([X,Y|Rest],Names):-
    writeN(X,Names),write(','),writeArgsN([Y|Rest],Names).

%Write a list of bindings using writeN.
```

```
writeBindings([],[],_Names).
writeBindings([X|Xs],[Y|Ys],Names):-
    nameOf(Y,Names,YN),X==YN,!,writeBindings(Xs,Ys,Names).
writeBindings([X|Xs],[Y|Ys],Names):-
    write(X),write('='),writeN(Y,Names),nl,
    writeBindings(Xs,Ys,Names).

%Utility functions

%Delete last element of a list.

init([],[]).
init([_X],[]).
init([X,Y|Xs],[X|Rest]):-init([Y|Xs],Rest).

%Execute a list of goals.

listcall([]).
listcall([X|Xs]):-call(X),listcall(Xs).

%Test if a list of variables is a subset of another.

areIn([],_).
areIn([X|Xs],L):-isIn(X,L),areIn(Xs,L).

%Test if a term is in a list.

isIn(X,[Y|_]):-X==Y,!.
isIn(X,[_|Ys]):-isIn(X,Ys).

%Test if a term is not in a list.

notIn(X,L):-isIn(X,L),!,fail.
notIn(_,_).

%Examples

es1:-use([(p(f(X,Y),Y,Z):-q(X,Y,Z))]).
```

```
es2:-use([(q(f(X)):-p(X)),(s(f(X),f(Y)):-s(X,Y)),
    (s(g(X),g(Y)):-t(X,Y)),(p(g(X)):-r(X))]).
```

5.2 Use of the meta-interpreter

We will now see a sample execution of `es1`.

```
| ?- es1.
Insert goal (use nu as first predicate to hide variables)
|: nu(Y),p(X,Y,Z),r(Y).

Number of big-steps:1.
q(_774,Y,Z),r(Y)
X=f(_774,Y)

nu(Y),p(X,Y,Z),r(Y)
```

We have computed all the possible computations from $\nu Y\ p(X,Y,Z),r(Y)$ composed by a single big-step.
For each computation the meta-interpreter outputs the resulting goal and the visible substitutions. Variables in the starting goal are written with their original names, new variables are written with their internal names (an underscore followed by a number).
In this example we have two possible computations: in the former p is rewritten into q and Y is extruded, so the final goal is $q(\_774,Y,Z),r(Y)$ and X is bound to $f(\_774,Y)$ where _774 is a new variable, the latter is simply an identical rewriting.
Now we will see a sample execution of `es2`.

```
| ?- es2.
Insert goal (use nu as first predicate to hide variables)
|: q(X),s(X,Y),q(Y).

Number of big-steps:2.
r(_1154),t(_1154,_1256),r(_1256)
Y=f(g(_1256))
X=f(g(_1154))
_898=g(_1154)
_986=g(_1256)

p(_898),s(_898,_986),p(_986)
```

```
Y=f(_986)
X=f(_898)

p(_929),s(_929,_1017),p(_1017)
Y=f(_1017)
X=f(_929)

q(X),s(X,Y),q(Y)
```

This time we asked for computations composed by two big-steps starting from the goal $q(X), s(X,Y), q(Y)$. The program used is the one of example 3.1. In fact the first solution is exactly the one we have computed in that example.
Notice that the second and the third solutions are essentially the same, but in one of them the idle step has been performed as first step, while in the other one it has been chosen as second step. Finally, the fourth computation is an identity computation.

Chapter 6

Logic Process Calculus

In this chapter we present Logic Process Calculus (LPC), a process calculus that behaves as logic programming with restriction. In section 6.1 we show LPC syntax and we give an operational semantics for it. In section 6.2 we show how to map a logic program with restriction, together with a goal, into LPC and we prove a correctness result and a completeness result.

6.1 Syntax and operational semantics

In this chapter we present Logic Process Calculus (LPC for short), a process calculus whose processes simulate logic programs.
The syntax of LPC processes is:

$$P ::= nil \mid \theta.P \mid P|P \mid P+P \mid \nu x\ P \mid X \mid rec\ X.P$$

where θ is a substitution, x is a data variable and X is a process name.
We assume that the restriction operator has the lowest precedence.
As in the other formalisms, the restriction operator is a binder here too, then all the occurrences of x in $\nu x\ P$ are bound occurrences.
Likely, rec is a binder for process names.
We require that all the occurrences of process variables are bound occurrences whereas we may have free occurrences of data variables.
Here nil is the process that performs no actions, $\theta.P$ is the process that performs the substitution θ and then becomes P, $P|Q$ is the parallel composition of processes P and Q, $P+Q$ is the process that can behave as P or as Q, $\nu x\ P$ is the process P where the variable x has been restricted, $rec\ X.P$ is the recursive definition of a process with name X.
We consider our processes up to the following axioms:

$$(AP1)\ (P_1|P_2)|P_3 \equiv P_1|(P_2|P_3)$$

$$(AP2)\ P_1|P_2 \equiv P_2|P_1$$

$$(AP3)\ P|nil \equiv P$$

$$(AP4)\ (P_1 + P_2) + P_3 \equiv P_1 + (P_2 + P_3)$$

$$(AP5)\ P_1 + P_2 \equiv P_2 + P_1$$

$$(AP6)\ P + nil \equiv P$$

$$(AP7)\ \nu x\ \nu y\ P \equiv \nu y\ \nu x\ P$$

$$(AP8)\ \nu x\ P \equiv P \qquad \text{if } x \notin \text{fn}(P)$$

$$(AP9)\ \nu x\ P_1|P_2 \equiv (\nu x\ P_1)|P_2 \qquad \text{if } x \notin \text{fn}(P_2)$$

$$(AP10)\ \nu x\ P_1 + P_2 \equiv (\nu x\ P_1) + P_2 \qquad \text{if } x \notin \text{fn}(P_2)$$

$$(AP11)\ \nu x\ P \equiv \nu y\ P[y/x]$$

$$(AP12)\ rec\ X.P = rec\ Y.P[Y/X]$$

The first three axioms state that the calculus has a commutative monoid structure with respect to the parallel operator, the following three axioms have the same role for the nondeterministic choice operator $+$. Then there are five axioms which state the usual rules for the restriction operator and an axiom for the α-conversion of process names.

We will define an operational semantics for our process calculus, but first we introduce an useful notation.
If f is a function (usually a substitution) and x is in the domain of f then $f|_{\backslash x}$ is f restricted to the set of all the elements in its domain but x; in the same way if X is a set of variables then $f|_{\backslash X}$ is the restriction of f to all the elements not in X.

We use for transitions labels denoted by L, which are pairs whose first component is a substitution and whose second component is a set of variable names that contains the names that have been extruded. The second part of the label is necessary because if a variable has been extruded by assigning it to a free variable and this free variable is then restricted, also the extruded variable must be bound (this is analogous to the bound output notation for π-calculus [15]). When we do not write an explicit synchronization label we intend it to be $(\epsilon, \emptyset)$.

$$\text{prefix} \qquad \frac{}{\theta.P \xrightarrow{\theta,\emptyset} P}$$

$$\text{choice} \qquad \frac{P \xrightarrow{L} Q}{P + R \xrightarrow{L} Q}$$

$$\text{res} \qquad \frac{P \xrightarrow{\theta,E} Q}{\nu x\ P \xrightarrow{\theta|_{\setminus x},E\setminus Y} \nu x\ \nu Y\ Q} \text{ if } x \notin \mathrm{var}(\theta|_{\setminus x})$$

where $Y = \{y \in E | \theta^{-1}(y) \cap \mathrm{fn}(P) \subseteq \{x\}\}$.

$$\text{open} \qquad \frac{P \xrightarrow{\theta,E} Q}{\nu x\ P \xrightarrow{\theta|_{\setminus x},E\cup\{x\}} Q} \text{ if } x \in \mathrm{var}(\theta|_{\setminus x})$$

We require that if we have an extrusion no previously existent free variables have the same name of the extruded variable (we can overcome the problem by renaming the restricted variable before the extrusion).

$$\text{rec} \qquad \frac{P[rec\ X.P/X] \xrightarrow{L} Q}{rec\ X.P \xrightarrow{L} Q}$$

$$\text{par} \qquad \frac{P \xrightarrow{\theta,E} Q}{P|R \xrightarrow{\theta,E} P|\theta(R)}$$

where, for each substitution θ, $\theta(R)$ is inductively defined as follows:

$$\theta(nil) = nil$$

$$\theta(\theta_1.P) = \mathrm{mgu}(\mathrm{eqn}(\theta_1)\theta).\theta(P)$$

$$\theta(P_1|P_2) = \theta(P_1)|\theta(P_2)$$

$$\theta(P_1 + P_2) = \theta(P_1) + \theta(P_2)$$

$$\theta(\nu x\ P) = \nu x\ \theta|_{\setminus x}(P) \text{ where } x \notin \mathrm{Im}(\theta)$$

$$\theta(X) = X$$

$$\theta(rec\ X.P) = rec\ X.\theta(P)$$

6.2 Mapping from logic programming with restriction to LPC

We want to define a mapping from logic programming with restriction to LPC. In particular, we want to build a process for each pair (goal, program).
In the following discussion, we will write a process definition for each predicate and then a process definition for the goal process.

We will define processes using other process names. This is only meta-syntax. We need to expand each process name in order to have the real term of the process calculus. If this creates an infinite process we need to introduce a recursive definition (using rec operator) in order to have a finite term.
A generic clause with restriction has the form:
$p(t_1, \dots, t_k) : -\nu X\ q_1(t_{1,1}, \dots, t_{1,k_1}), \dots, q_n(t_{n,1}, \dots, t_{n,k_n})$.
We associate to this clause the process:
$p = \nu V\ \{t_i/p_i\}_{i=1\dots k}.nil|\nu X\ |_i\ \nu\{q_{i,j}\}\ (\{t_{i,j}/q_{i,j}\}_{j=1\dots k_i}.nil|q_i)$
where V is the set of free names of the clause.
If the body of the clause is empty we have the parallel composition of 0 processes which is equal to nil.
We associate to each predicate a process that is the sum of all the processes associated to clauses with this predicate as head.
Given a goal we can define a process that solves it.
A generic goal with restriction has the form:
$\nu X\ p_1(t_{1,1}, \dots, t_{1,k_1}), \dots, p_n(t_{n,1}, \dots, t_{n,k_n})$
We associate to this goal the process:
$g = \nu X\ |_i\ \nu\{p_{i,j}\}\ (\{t_{i,j}/p_{i,j}\}_{j=1\dots k_i}.nil|p_i)$

Let us see some examples.

Example 6.1 *Let us consider the following program:*
$p(X) : -\nu Y\ q(X, f(Y)), r(f(X))$
$q(X, X)$
$r(f(f(a)))$
We can define the following processes:
$p = \nu X\ \{X/P_1\}.nil|\nu Y\ (\nu Q_1\ \nu Q_2\ \{X/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X)/R_1\}.nil|r)$
$q = \nu X\ \{X/Q_1, X/Q_2\}.nil$
$r = \{f(f(a))/R_1\}.nil$
Suppose we want to solve the goal $p(f(X))$.
The associated process is:
$g = \nu P_1\ \{f(X)/P_1\}.nil|p$
The process p can do the transition:

$$\begin{aligned}&\nu X'\ \{X'/P_1\}.nil|\nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X')/R_1\}.nil|r)\\ &\qquad \xrightarrow{X'/P_1,\{X\}} \nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X')/R_1\}.nil|r)\end{aligned}$$

so g can do:

$$\begin{aligned}&g \rightarrow \nu X'\ \{f(X)/X'\}.nil|\\ &\qquad\qquad |\nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X')/R_1\}.nil|r)\end{aligned}$$

$\nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X')/R_1\}.nil|r)$ can do the transition:

$$\begin{aligned}&\nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|(\nu R_1\ \{f(X')/R_1\}.nil|r) \rightarrow\\ &\qquad \nu Y\ (\nu Q_1\ \nu Q_2\ \nu X''\ \{f(Y)/X'', f(Y)/X'\}.nil)|(\nu R_1\ \{f(X')/R_1\}.nil|r)\end{aligned}$$

and so:

$$\begin{aligned}&\nu X'\ \{f(X)/X'\}|\nu Y\ (\nu Q_1\ \nu Q_2\ \{X'/Q_1, f(Y)/Q_2\}.nil|q)|\\ &\qquad\qquad |(\nu R_1\ \{f(X')/R_1\}.nil|r) \rightarrow\\ &\qquad \nu X'\ \{f(X)/X'\}|\nu Y\ (\nu Q_1 \nu Q_2\ \nu X''\ \{f(Y)/X'', f(Y)/X'\}.nil)|\\ &\qquad\qquad\qquad |(\nu R_1\ \{f(X')/R_1\}.nil|r)\end{aligned}$$

Now we can delete the restrictions on Q_1 and Q_2 so:
$\nu Y\ (\nu X''\ \{f(Y)/X'', f(Y)/X'\}.nil)|(\nu R_1\ \{f(X')/R_1\}.nil|r)$
can do the transition:

$$\begin{aligned}&\nu Y\ (\nu X''\ \{f(Y)/X'', f(Y)/X'\}.nil)|(\nu R_1\ \{f(X')/R_1\}.nil|r) \xrightarrow{f(Y)/X', \{Y\}}\\ &\qquad\qquad (\nu R_1\ \{f(f(Y))/R_1\}.nil|r)\end{aligned}$$

and so we can have:

$$\begin{aligned}&\nu X'\ \{f(X)/X'\}|\nu Y\ (\nu X''\ \{f(Y)/X'', f(Y)/X'\}.nil)|\\ &\qquad |(\nu R_1\ \{f(X')/R_1\}.nil|r) \rightarrow \nu X'\ \nu Y\ \{X/Y\}|(\nu R_1\ \{f(f(Y))/R_1\}.nil|r)\end{aligned}$$

Then:
$\nu X'\ \nu Y\ \{X/Y\}|(\nu R_1\ \{f(f(Y))/R_1\}.nil|r) \rightarrow \nu X'\ \nu Y\ \{X/Y\}|(\nu R_1\ \{a/Y\}.nil)$
Now we can do:
$\nu Y\ \{X/Y\}|(\nu R_1\ \{a/Y\}.nil) \rightarrow \nu Y\ \{a/Y, a/X\}$
and finally:
$\nu Y\ \{a/Y, a/X\} \xrightarrow{a/X, \emptyset} nil$
The only visible transition label is $(a/X, \emptyset)$, whose substitution is $\{a/X\}$ which is exactly the computed answer substitution for the chosen goal.

Now we will show an example of a recursive program.

Example 6.2 *Let us consider the following program:*
$p(Y) : -q(Y)$
$q(a)$
$q(f(X)) : -p(X)$
The process definitions associated to this clauses are:

$q = \{a/Q_1\}.nil + \nu X\ \{f(X)/Q_1\}.nil|\nu P_1\ \{X/P_1\}.nil|p$
$p = \nu Y\ \{Y/P_1\}.nil|\nu Q_1\ \{Y/Q_1\}.nil|q$
Suppose we want to solve the goal $q(Z)$.
The associated process is:
$g = \nu Q_1\ \{Z/Q_1\}.nil|q$
By eliminating process names we have:

$$g = \nu Q_1\ \{Z/Q_1\}.nil|rec\ n.\{a/Q_1\}.nil + \\ + \nu X\ \{f(X)/Q_1\}.nil|\nu P_1\ \{X/P_1\}.nil|\nu Y\ \{Y/P_1\}.nil|\nu Q_1\ \{Y/Q_1\}.nil|n$$

$q[rec\ n.q/n] \xrightarrow{a/Q_1,\emptyset} nil$
thus also $rec\ n.q$ *can do it and:*
$g \rightarrow \{a/Z\}.nil \xrightarrow{a/Z,\emptyset} nil$
This is exactly what we wanted, because $\{a/Z\}$ *is a computed answer substitution for the chosen goal.*
Another possible computation is:
$q[rec\ n.q/n] \xrightarrow{a/Q_1,\emptyset} nil$

$$\begin{array}{l} \nu Q_1\ \{Z/Q_1\}.nil|rec\ n.\{a/Q_1\}.nil + \\ \quad + \nu X\ \{f(X)/Q_1\}.nil|\nu P_1\ \{X/P_1\}.nil|\nu Y\ \{Y/P_1\}.nil|\nu Q_1\ \{Y/Q_1\}.nil|n \rightarrow \\ \nu Q_1\ \{Z/Q_1\}.nil|\nu X\ \{f(X)/Q_1\}.nil|\nu P_1\ \{X/P_1\}.nil|\nu Y\ \{Y/P_1\}.nil|\{a/Y\}.nil \rightarrow \\ \qquad \nu Q_1\ \{Z/Q_1\}.nil|\nu X\ \{f(X)/Q_1\}.nil|\nu P_1\ \{X/P_1\}.nil|\{a/P_1\}.nil \rightarrow \\ \qquad\qquad \nu Q_1\ \{Z/Q_1\}.nil|\nu X\ \{f(X)/Q_1\}.nil|\{a/X\}.nil \rightarrow \\ \qquad\qquad\qquad \nu Q_1\ \{Z/Q_1\}.nil|\{f(a)/Q_1\}.nil \rightarrow \\ \qquad\qquad\qquad\qquad\qquad \{f(a)/Z\}.nil \xrightarrow{f(a)/Z,\emptyset} nil \end{array}$$

Also $\{f(a)/Z\}$ *is a computed answer substitution, as we expect.*

We will now show that the proposed mapping is correct.

Theorem 6.1 (Mapping from logic programming with restriction to LPC: correctness) *Let* P *be a program with restriction,* G *a goal with restriction and* p *the process associated to* G *and* P. *If* $P \Vdash G \xrightarrow{\theta}{}^{*} G'$ *then* $p \xrightarrow{\sigma,E}{}^{*} p'$ *where* $\sigma = \theta|_{\mathrm{fn}(G)}$, $E = \mathrm{var}(\mathrm{Im}(\sigma)) \setminus \mathrm{fn}(G)$ *and* p' *is the process associated to goal* G' *and program* P.

Proof

Let us prove it by rule induction.
Observe first that the translation of equivalent goals and clauses gives equivalent processes.

$$\frac{H \leftarrow \nu X\ B_1, \dots, B_k \in P \qquad \theta = \text{mgu}(A, \rho(H))}{P \Vdash A \xrightarrow[\theta]{\theta} \rho(\nu X\ B_1, \dots, B_k)\theta} \quad \textit{Atomic goal}$$

where $X = \text{var}(B_1, \dots, B_k) \setminus \text{var}(H)$ *and* ρ *is an injective renaming of variables such that all the variables in the clause variant* $\rho(H \leftarrow \nu X\ B_1, \dots, B_k)$ *are new and* A *is an atom.*
Let:
$A = p(p_1, \dots, p_n)\sigma$
$\rho(H) = p(p_1, \dots, p_n)\sigma'$
$\rho(B_i) = q_i(q_{i,1}, \dots, q_{i,n})\sigma_i$
From the existence of the clause we have that p *can behave as:*
$\nu\, \text{var}(\rho(H))\ \sigma'.nil | \nu X\ |_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i)$
The goal A *corresponds to the process:*
$g = \nu\{p_i\}_{i=1\dots n}\ \sigma.nil | \nu\, \text{var}(\rho(H))\ \sigma'.nil | \nu X\ |_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i)$
We can have the following transition:
$g \rightarrow \nu\{p_i\}_{i=1\dots n}\ \nu\, \text{var}(\rho(H))\, \text{mgu}(\sigma, \sigma').nil | \nu X\ |_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i)$
where the label is empty because $\text{Dom}(\sigma) = \{p_{i,j}\}$.
Note that $\text{mgu}(\sigma, \sigma')$ *is the mgu of the set of equations obtained by matching the corresponding terms of* σ *and* σ'*, so it is equal to* θ. *Moreover applying* σ *has no other effects because the variables in* $\text{Dom}(\sigma)$ *do not appear elsewhere.*
Observe that we can eliminate the restriction on $\{p_i\}_{i=1\dots n}$ *because these variables no more appear in the term.*
Notice, too, that $\text{mgu}(\sigma, \sigma')$ *contains only variables in* $\text{var}(\rho(H))$ *and in* $\text{var}(A)$.
Then we can do:

$$\nu\, \text{var}(\rho(H))\ \theta.nil | \nu X\ |_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i) \xrightarrow{\theta|_{\text{var}(A)}, E} \nu X'\ \nu X\ \theta(|_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i))$$

where:
$X' = \{y \in \text{var}(\rho(H)) |\, \text{or}_\theta(y) \cap \text{var}(A) = \emptyset\}$
$E = \{y \in \text{var}(\rho(H)) |\, \text{or}_\theta(y) \cap \text{var}(A) \neq \emptyset\}$
Observe that the variables of the clause variant are always assigned to at least a goal variable unless they are in the domain of the substitution, but in this case they disappear from the names of the term, and so the corresponding restriction can be eliminated. This means that we can delete X'. *Notice, too, that the only occurrences of variables in* $\text{var}(\theta)$ *in* $|_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i)$ *are the ones in* $\text{Im}(\sigma_i)$ *so:*
$\nu X\ \theta(|_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i)) = \nu X\ (|_i\ (\nu\{q_{i,j}\}\ \sigma_i.nil | q_i))\theta$
The goal:
$\rho(B_1, \dots, B_k)\theta$
corresponds to the process:

$\nu X \ (|_i \ (\nu\{q_{i,j}\} \ \sigma_i.nil|q_i))\theta$
as needed.
Observe that:
$E = \{y \in \mathrm{var}(\rho(H))|\, \mathrm{or}_\theta(y) \cap \mathrm{var}(A) \neq \emptyset\} = \mathrm{var}(\mathrm{Im}(\theta|_{\mathrm{var}(A)})) \setminus \mathrm{fn}(A)$
because $\mathrm{var}(\mathrm{Im}(\theta)) \subseteq \mathrm{var}(\rho(H)) \cup \mathrm{var}(A)$ *and* $\mathrm{or}_\theta(y) \cap \mathrm{var}(A) \neq \emptyset$ *implies that exists* $z \in \mathrm{Dom}(\theta) \cap \mathrm{var}(A)$ *such that* $y \in \mathrm{var}(z\theta)$ *and thus* $y \in \mathrm{Im}(\theta|_{\mathrm{var}(A)})$ *(y cannot be the only origin of itself because otherwise it is not extruded).*
This proves the thesis.

$$\frac{P \Vdash G \xrightarrow[\theta]{\theta} F \qquad G, G' \textit{ do not contain restrictions}}{P \Vdash G, G' \xrightarrow[\theta]{\theta} F, G'\theta} \qquad \textit{conjunctive goal}$$

Let p be the process associated to G and P, p' the one associated to G' and P.
The process associated to G, G' and P is $p|p'$.
We have for inductive hypothesis:
$p \xrightarrow{\sigma,E}{}^{*} q$ *where* $\sigma = \theta|_{\mathrm{fn}(G)}$, $E = \mathrm{var}(\mathrm{Im}(\sigma)) \setminus \mathrm{fn}(G)$ *and q is the process associated to goal F and program P.*
Thus:
$p|p' \xrightarrow{\sigma,E}{}^{*} q|\sigma(p')$
but because the variables in σ occur only in the image of the head substitution of p' we have:
$q|\sigma(p') = q|p'\sigma$
Moreover because $\mathrm{var}(\theta) \cap \mathrm{fn}(p') \subseteq \mathrm{fn}(G)$ *we have:*
$q|p'\sigma = q|p'\theta.$
Observe that:
$E = \mathrm{var}(\mathrm{Im}(\sigma)) \setminus \mathrm{fn}(G) = \mathrm{var}(\mathrm{Im}(\sigma)) \setminus \mathrm{fn}(G, G')$
because all the variables not in $\mathrm{fn}(G)$ *come from the clause variants (and so are new) or from extrusions (and so we can suppose their names not in* $\mathrm{fn}(G)$*). The thesis is so proved.*

$$\frac{P \Vdash G \xrightarrow[\theta]{\theta} F \qquad G \textit{ does not contain restrictions}}{P \Vdash \nu X \ G \xrightarrow[\theta]{\theta'} \nu X' \ F} \qquad \textit{restricted goal}$$

where:
$X' = \{x \in \mathrm{fn}(F)|\, \mathrm{or}_\theta(x) \subseteq X\}$
$\theta' = \{[w/y] \in \theta | \exists z \in \mathrm{or}_\theta(y).z \notin X\}$
Let p be the process associated to G and P.
The process associated to $\nu X \ G$ and P is $\nu X \ p$.
We have for inductive hypothesis:

$p \xrightarrow{\sigma,E}{}^{*}$ *q where* $\sigma = \theta|_{\mathrm{fn}(G)}$, $E = \mathrm{var}(\mathrm{Im}(\sigma)) \setminus \mathrm{fn}(G)$ *and q is the process associated to goal F and program P.*
Thus:
$\nu X\ p \xrightarrow{\sigma|_{\setminus X},E'}{}^{*} \nu X'\ q$ *where:*
$X' = \{x \in X \cup E | \,\mathrm{or}_\sigma(x) \subseteq X\}$.
$E' = \{x \in X \cup E | \,\mathrm{or}_\sigma(x) \not\subseteq X\}$.
Note that:
$\sigma|_{\setminus X} = \theta|_{\mathrm{fn}(\nu X\ G)} = \theta'|_{\mathrm{fn}(\nu X\ G)}$
for lemma 3.4.
Observe that for a variable x we have $\mathrm{or}_\sigma(x) \subseteq X$ *iff we have* $\mathrm{or}_\theta(x) \subseteq X$ *because only the variables in* $\mathrm{fn}(G)$ *can be origins.*
Moreover if $\mathrm{or}_\sigma(x) \subseteq X$ *then* $x \in \mathrm{fn}(F)$ *iff* $x \in \mathrm{fn}(G) \vee x \in \mathrm{Im}(\sigma)$. *In the first case because x is an origin of itself we must have* $x \in X$. *In the second case* $x \in E = \mathrm{Im}(\sigma) \setminus \mathrm{fn}(G)$.
So the two definitions of X' *are equivalent.*
Moreover $y \in E'$ *iff* $y \in (\mathrm{Im}(\sigma) \setminus \mathrm{fn}(G)) \cup X = \mathrm{Im}(\sigma) \setminus \mathrm{fn}(\nu X\ G)$ *unless* $\mathrm{or}_\sigma(x) \subseteq X$ *but this condition means that y is not in* $\mathrm{var}(\mathrm{Im}(\sigma|_{\setminus X}))$ *thus* $E' = \mathrm{var}(\mathrm{Im}(\sigma|_{\setminus X})) \setminus \mathrm{fn}(\nu X\ G)$ *which proves the thesis.*
Observe that when the recursion operator is needed the proof is the same because a process name inside a recursive definition behaves exactly as the whole process it references.

We will now prove a confluence theorem, i. e. a theorem that guarentees that a process without the + operator that can make two different transitions can do further transitions to continue the two computations in order to reach a common result.

Theorem 6.2 (Confluence) *For each process p without the + operator, if* $p \xrightarrow{\theta',E'} p'$ *and* $p \xrightarrow{\theta'',E''} p''$ *then exist* θ''',θ'''',E''' *and* E'''' *such that:*
$p' \xrightarrow{\theta''',E'''} q$
$p'' \xrightarrow{\theta'''',E''''} q$
$\theta\theta''' = \theta''\theta''''$
$E \cup E''' = E'' \cup E''''$

Proof

We can suppose p without recursion (we can imagine to unfold our processes as necessary).
Thus we can put the process in the following normal form:
$p = \nu X\ |_i \sigma_{i,1}.\ldots.\sigma_{i,n_i}.nil = \nu X\ q$
All the possible transitions for q are of the form:
$q \xrightarrow{\sigma_{j,1},E} \sigma_{j,2}.\ldots.\sigma_{j,n_j}.nil|\ |_{i \neq j}\ \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{j,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,n_i})\sigma_{j,1}).nil$

If it is instead k the first term that acts we have:
$q \xrightarrow{\sigma_{k,1},E} \sigma_{k,2}.\ldots.\sigma_{k,n_k}.nil|\ |_{i\neq k}\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{k,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,n_i})\sigma_{k,1}).nil$
But in the first case:

$$
\begin{aligned}
&\sigma_{j,2}.\ldots.\sigma_{j,n_j}.nil|\ |_{i\neq j}\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{j,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,n_i})\sigma_{j,1}).nil \\
&\qquad\qquad\xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\sigma_{k,1})\sigma_{j,1}),E} \\
&\qquad\mathrm{mgu}(\mathrm{eqn}(\sigma_{j,2})\sigma_{k,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{j,n_j})\sigma_{k,1}).nil| \\
&\qquad\mathrm{mgu}(\mathrm{eqn}(\sigma_{k,2})\sigma_{j,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{k,n_k})\sigma_{j,1}).nil| \\
&\quad|_{i\neq j,k}\, \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{j,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{k,1})\sigma_{j,1})).\ldots \\
&\qquad\qquad.\,\mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,k_i})\sigma_{j,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{k,1})\sigma_{j,1})).nil
\end{aligned}
$$

Analogously:

$$
\begin{aligned}
&\sigma_{k,2}.\ldots.\sigma_{k,n_k}.nil|\ |_{i\neq j}\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{k,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,n_i})\sigma_{k,1}).nil \\
&\qquad\qquad\xrightarrow{\mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1})\sigma_{k,1}),E} \\
&\qquad\mathrm{mgu}(\mathrm{eqn}(\sigma_{k,2})\sigma_{j,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{k,n_k})\sigma_{j,1}).nil| \\
&\qquad\mathrm{mgu}(\mathrm{eqn}(\sigma_{j,2})\sigma_{k,1}).\ldots.\mathrm{mgu}(\mathrm{eqn}(\sigma_{j,n_j})\sigma_{k,1}).nil| \\
&\quad|_{i\neq j,k}\, \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{k,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1})\sigma_{k,1})).\ldots \\
&\qquad\qquad.\,\mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,n_i})\sigma_{k,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1})\sigma_{k,1})).nil
\end{aligned}
$$

which proves the thesis because:

$$
\begin{aligned}
&\mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{k,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1})\sigma_{k,1})) = \\
&\qquad= \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{k,1}\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1})\sigma_{k,1})) = \\
&\qquad= \mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{j,1}) \cup \mathrm{eqn}(\sigma_{k,1})) = \\
&\qquad\qquad= \mathrm{mgu}(\mathrm{eqn}(\mathrm{mgu}(\mathrm{eqn}(\sigma_{i,1})\sigma_{j,1}))\, \mathrm{mgu}(\mathrm{eqn}(\sigma_{k,1})\sigma_{j,1}))
\end{aligned}
$$

for lemma 4.4 and because if a process performs two sequences of transitions with the same final result the same holds for the same process where some variables have been restricted.

We want now to show that all the (complete, i. e. that termines with *nil* as process) computations of an LPC process obtained from the mapping of a goal and a program correspond to refutations of logic programming with restriction. We cannot show a similar result for each step because in LPC the computation can start from any clause used in the logic computation, also from clauses not in the starting goal.

Theorem 6.3 (Mapping from logic programming with restriction to LPC: completeness) *Let P be a program with restriction, G a goal with restriction and g the process associated to G and P. If $g \xrightarrow{\sigma,E}{}^* nil$ then $P \Vdash G \xrightarrow{\theta}{}^* \Box$ where $\theta|_{\mathrm{fn}(G)} = \sigma$.*

Proof

We can suppose p without sums (it is equal to choose in advance which alternative to follow) and without recursion (we can imagine to unfold our processes as necessary). Let us prove the thesis by induction on the number of steps in the computation.

The only computation in 0 steps is:

$nil \to^* nil$

which satisfies the thesis because for each program P we have:

$P \Vdash \Box \xrightarrow{\epsilon}{}^* \Box$.

We now need to prove the inductive step.

Let $\rho(H \leftarrow \nu X\ B_1, \ldots, B_k)$ be a clause variant that corresponds to one of the subprocesses that are effectively executed (i. e. not discarded in a nondeterministic choice in the considered computation) and θ_1 the corresponding mgu.

Our goal can do the transition:

$G \xrightarrow{\theta_1} G'$

but for theorem 6.1 we have:

$p \xrightarrow{\sigma_1, E} p'$

where $\sigma_1 = \theta_1|_{\mathrm{fn}(G)}$, $E = \mathrm{Im}(\sigma_1) \setminus \mathrm{fn}(G)$ and p' is associated to G' and P.

For theorem 6.2 we have:

$p' \xrightarrow{\sigma_2, E'}{}^* nil$

and $\sigma_1\sigma_2 = \sigma$.

For inductive hypothesis:

$P \Vdash G' \xrightarrow{\theta_2} \Box$

and $\theta_2|_{\mathrm{fn}(G')} = \sigma_2$.

Thus:

$P \Vdash G \xrightarrow{\theta_1\theta_2} \Box$

and $(\theta_1\theta_2)|_{\mathrm{fn}(G)} = \theta_1|_{\mathrm{fn}(G)}\theta_2|_{\mathrm{fn}(G')} = \sigma_1\sigma_2$.

This proves the thesis.

Chapter 7

Conclusion

In this work we have analyzed logic programming in relation to network models.
In particular, we have extended logic programming with the restriction operator ν and studied its properties. Moreover we have built two tile semantics for it and proved bisimilarity to be a congruence in both the cases.
Note that the first style of tile semantics is simply a reinterpretation of the tile semantics of standard logic programming, so the correspondence between tile semantics and operational semantics justifies the latter.
We have shown that a subset of logic programming with restriction, synchronized logic programming with restriction, is strongly related to Hoare synchronized hyperedge replacement.
We have also written a meta-interpreter for our logic formalism.
We have also demonstrated that logic programming with restriction can be seen as a process calculus, which we have called Logic Process Calculus.
In the future this can be utilized by applying process calculus methods and instruments to logic programming and viceversa.
Another possible extension of the results of this work is the study of Milner synchronization (and of other types of synchronization too), what would be useful in order to allow process calculi such as π-calculus and Ambient calculus to be mapped into SHR (see for example [7]) and to model synchronizations between only two processes, what has many applications in the study of network security.
We think that this is not possible through logic programming, because in this framework there are some problems in making Milner synchronization and mobility coexist, but it is possible to skip this step and to study directly the synchronizations in the framework of tile logic.

Thanks

I thank Prof. Ugo Montanari for the time he spent discussing with me and for his ability to show me each time the right perspective to observe the problems in. I also thank Prof. Roberto Barbuti, Prof. Pierpaolo Degano, Prof. Giorgio Levi, Mirco Nanni and Sonia Fagorzi for the support they gave to me, expecially in the research of materials. Moreover I thank Marco Fantozzi, Gianni Ceccarelli and Andrea Gambassi for the help they gave to me with LaTeX in many occasions and Fabiana Rosi and Alessandro Mazzucca for having helped me with English language. I also thank all the people that had to stand me during these months, first of all my parents. Finally, I thank Roberto Zunino for the big help he has given to me in many occasions.

Bibliography

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

[2] A. Asperti and G. Longo. Categories, types and structures. The MIT Press, 1991.

[3] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1995.

[4] Roberto Bruni, Ugo Montanari, and Francesca Rossi. An interactive semantic of logic programming. Submitted, 2000.

[5] Luca Cardelli and Andrew 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.

[6] Mats Carlsson et al. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, 1995.

[7] GianLuigi Ferrari, Ugo Montanari, and Emilio Tuosto. A lts semantics of ambients via graph synchronization with mobility. In Springer Verlag, editor, *Proc. of ICTCS'01*, LNCS, 2001. To appear.

[8] F. Gadducci and U. Montanari. The tile model. Technical Report TR-96-27, 30, 1996.

[9] Dan Hirsch and Ugo Montanari. Synchronized hyperedge replacement with name mobility. In Springer Verlag, editor, *Proc. of CONCUR'01*, LNCS, 2001. To appear.

[10] Barbara König and Ugo Montanari. Observational equivalence for synchronized graph rewriting. In Springer Verlag, editor, *Proc. TACS'01*, LNCS, 2001. To appear.

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

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

[13] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[14] R. Milner. Communication and concurrency, 1989.

[15] R. Milner. Communicating and mobile systems: the pi-calculus, 1999.

[16] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *Software Engineering*, 24(5):315–330, 1998.

[17] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.