A Fine-Grained Concurrent Completion Procedure

Claude Kirchner, Christopher Lynch, Christelle Scharff

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A fine-grained Concurrent Completion Procedure

Claude KIRCHNER , Christopher LYNCH , Christelle SCHARFF

N ° 2990

September 1996

THÈME 2

ISSN 0249-6399

A fine-grained Concurrent Completion Procedure

Claude KIRCHNER , Christopher LYNCH , Christelle SCHARFF*

Thème 2 — Génie logiciel
et calcul symbolique
Projet PROTHEO

Rapport de recherche n ° 2990 — September 1996 — 56 pages

Abstract: We present a concurrent Completion procedure based on the use of a SOUR graph as data structure. The procedure has the following characteristics. It is asynchronous, there is no need for a global memory or global control, equations are stored in a SOUR graph with maximal structure sharing, and each vertex is a process, representing a term. Therefore, the parallelism is at the term level. Each edge is a communication link, representing a (subterm, ordering, unification or rewrite) relation between terms. Completion is performed on the graph as local graph transformations by cooperation between processes. We show that this concurrent Completion procedure is sound and complete with respect to the sequential one, provided that the information is locally time stamped in order to detect out of date information.

Key-words: Rewriting, Completion, SOUR Graphs, Concurrency

(Résumé : tsvp)

*Email:{Claude.Kirchner,Christopher.Lynch,Christelle.Scharff}@loria.fr, http://www.loria.fr/equipe/protheo.html

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Une procédure de complétion concurrente de fine granularité

Résumé : Nous présentons une nouvelle procédure de Complétion des systèmes de réécriture basée sur l'utilisation des graphes SOUR comme structure de donnée. Cette procédure a les caractéristiques suivantes. Les équations sont représentées sous forme de graphe avec le maximum de partage de structure. Chaque sommet de ce graphe est un processus représentant un terme. Les arcs sont des canaux de communication représentant les relations de sous-terme, d'ordre, d'unification et de réécriture. La Complétion est réalisée de façon asynchrone par coopération entre les processus en opérant des transformations locales sur le graphe. Il n'y a ni besoin d'une mémoire globale ni d'un contrôle global. Nous montrons que cette procédure est correcte et complète par rapport à la version séquentielle, à condition que l'information soit localement datée pour détecter les objets périmés.

Mots-clé : Réécriture, Complétion, Graphes SOUR, Concurrence

Contents

1 Introduction

Parallelization is an attractive way for improving efficiency of automated deduction, and the main approaches are surveyed in [BH94] and [SS93]. We present in this paper a new approach to term rewriting completion which is based on fine grain concurrency, and which relies on a novel approach to completion.

The resolution (and paramodulation) inference systems are theorem proving procedures for first-order logic (with equality) that can run exponentially long for subclasses which have polynomial time decision procedures, as in the case of the Knuth-Bendix ground completion procedure. D. Kozen [Koz77] developed a method based on congruence closure to solve word problem in ground equational theories. J. Gallier, P. Narandran, D. Plaisted, S. Raatz [GNP+93] extend Kozen's techniques to generate convergent equational system equivalent to a ground set of equations. W. Snyder improves the running time of this algorithm to $nlog(n)$ [Sny93]. D. Plaisted and A. Sattler-Klein [PSK96] show that, with a particular strategy and with structure sharing, Knuth-Bendix completion runs in polynomial time for a ground set of equations. Recently C. Lynch has shown [Lyn95] that a special form of paramodulation which does not need to copy terms or literals runs in polynomial time in ground cases that include ground completion. This can be implemented in an elegant way using the notion of SOUR graph [LS95]. These graphs represent in a very convenient way a state of the completion, where all the basic ingredients (i.e. orientation, unification and rewriting) are made explicit at the object level. This makes explicit the fundamental operations of completion. A SOUR graph has its edges labelled by S when representing a subterm relation, by O when representing an orientation, by U when representing a unification problem and by R when representing a rewrite rule. The nodes of the graph are labelled by function symbols, and edges are labelled by constraints and renamings.

The properties of SOUR graphs are useful to study the parallelization and implementation of automated deduction on parallel distributed memory machines. There is no duplication of work since there is no copying. There is also no need of a consistency check. The explicit representation of basic operations allows a direct implementation of the inference rules as transition rules and thus to get completion as the result of independent (and asynchronous) operations as well as an easier way to describe and prove soundness and completeness of the graph transformations. This property is always desirable, but the increased complexity of concurrent processes makes it even more important.

Thus this paper presents a new fine-grained concurrent completion procedure based on the notion of distributed SOUR graphs and it is proved to be sound and complete, for simplicity in the case of ground completion, although everything can be extended to non-ground completion.

We consider each node of the SOUR graph as a process and the edges as communication links between processes. Each process is in charge of detecting particular configurations corresponding to critical pairs, unification problems or ordering of terms. A node acts only in response to a message, independently of the other nodes. The processes are thus completely asynchronous. When a successful configuration is found, the corresponding operation is performed, typically directing a subterm or a rewrite edge from one node to another. In the

first design of the approach we thought that this was enough to ensure correctness of the process. But after investigations, we discovered that since all these detections and actions are performed asynchronously, we need to keep account of only the newest information arriving from a given node, in order to ensure the global consistency of the SOUR graph. This is performed via the use of local time stamps, so that the system still works asynchronously, but old information is ignored.

The paper describes the principles of this approach and its current implementation on a network of processors. We show that this implements ground completion using fine-grain concurrency. This relies in particular on an original parallelization of the detection of the satisfiability of unification and LPO orientation constraints, detection that runs concurrently with the standard completion process. Because it is fine-grained and completely asynchronous, our approach is quite different from the PaReDuX work [BGK95] and the clause diffusion method [BH95] but it also allows backward contraction. It is also quite different from the discount approach [ADF95] or the work on Partheo [LS90]. Let us finally emphasise that we think our approach quite promising since as opposed to the unsuccessful attempts to parallelize prolog on fine-grained architectures, it is backed by a very simple concept, SOUR graphs, no synchronization is needed, and each process gets enough work because of the internalization of unification and ordering constraint satisfiability.

The paper is structured as follows. Assuming the reader familiar with term rewriting (see [DJ90]), we summarize in section 2 the notion of SOUR graph. In section 3, we present the principle of concurrent completion on SOUR graphs. We show how critical pairs are detected, how unification and LPO ordering constraints are checked satisfiable, why time stamps are needed and how they are used. We also show how to detect the termination of the completion process and we prove that this model of completion is sound and complete.

The short version of this paper appeared in the proceedings of RTA'96 (Rewriting Techniques and Application) [KLS96].

2 Preliminaries

To simplify our presentation of inference rules, we will present them for ground terms. We refer the reader to [LS95] to see how the definitions and the inference rules are lifted to non-ground terms.

The symbol $\approx$ is a binary symbol, written in infix notation, representing semantic equality. Let EQ be a set of equations. We define a function Sub so that $Sub(EQ)$ is the set of subterms of EQ. If $t = f(t_1, \cdots, t_k)$ with $k \geq 0$, then $Sub(t) = \{t\} \cup \bigcup_{1 \leq i \leq k} Sub(t_i)$. We define $Sub(s \approx t) = Sub(s) \cup Sub(t)$ and $Sub(EQ) = \bigcup_{eq \in EQ} Sub(eq)$.

In this paper $\preceq$ will refer to the *lexicographic path ordering* ($\prec$ in its strict version)which is a reduction order total on ground terms. It is defined based on a total ordering $>$ on the function symbols. Let $s = f(s_1, \cdots, s_n)$ and $t = g(t_1, \cdots, t_m)$. Then $s \succeq t$ if (i) there exists an i such that $s_i \succeq t$, (ii) $f > g$ and $s \succ t_j$ for all t_j, or (iii) $f = g$ and either $s = t$ or there exists a k such that $s_k \succ t_k$, $s_i = t_i$ for all $i < k$ and $s \succ t_j$ for all $j > k$. If $u[s]$ is a ground term, and EQ is a set of ground equations, we write $u[s] \Rightarrow u[t]$ and say that $u[s]$ *rewrites*

in one step to $u[t]$ if there is an equation $s \approx t \in EQ$ such that $s \succ t$. We write $u_0 \stackrel{*}{\Rightarrow} u_n$ and say that u_0 *rewrites* to u_n if there is a a set of terms $\{u_1, \cdots, u_{n-1}\}$ such that for all i, $1 \leq i \leq n$, $u_{i-1} \Rightarrow u_i$. A ground set of equations is *convergent* if it is terminating and confluent.

A *SOUR* graph is a compact dag representation of a set of equations. Let EQ be a set of equations. The SOUR graph of EQ is the graph which has one node associated with each element of $t \in Sub(EQ)$, labelled with the root symbol of t. For each element $f(t_1 \cdots, t_k) \in Sub(EQ)$, with $k > 0$, for each i, $1 \leq i \leq k$, there is a directed edge called a *subterm edge* in the SOUR Graph from the node associated with $f(t_1, \cdots, t_k)$ to the node associated with t_i labelled with i, its index. For each equation $s \approx t \in EQ$ with $s \succ t$, there is a directed edge called a *rewrite edge* from the node associated with s to the node associated with t.

We define a semantic function $Term$ from the nodes of the graph to the set of terms. If v is a node in the graph labelled with f such that $arity(f) = k$, then there are k subterms edges from v labelled $1, \cdots, k$ to nodes $v_1 \cdots, v_k$ respectively. Then $Term(v) = f(Term(v_1), \cdots, Term(v_k))$. We define a function $Rule$ from the rewrite edges in the graph to the set of equations. If e is a rewrite edge from v_1 to v_2, then $Rule(e)$ is $Term(v_1) \approx Term(v_2)$. Graph G represents $\{Rule(e) \mid e \text{ is a rewrite edge in } G\}$.

We also add other kinds of edges to the SOUR graph. Some terms have an undirected edge called a *unification edge* between them, including between a node and itself. For now, we assume these edges are placed between every pair of nodes v_1 and v_2 such that $Term(v_1) = Term(v_2)$. There is a directed edge called an *orientation edge* between some pairs of nodes v_1 and v_2 such that $Term(v_1) \succ Term(v_2)$. These edges are used to perform inferences. The graph is called a *SOUR graph* because of **S**ubterm, **O**rientation, **U**nification and **R**ewrite edges.

The inference rule that we are coding with SOUR Graphs in the ground case is the *critical pair* inference rule.

$$\text{Critical Pair} \qquad \frac{s \approx t \qquad u[s] \approx v}{u[t] \approx v} \quad \text{if } s \succ t \text{ and } u[s] \succ v$$

We simulate the completion inference rules by searching for patterns (or configurations) in the graph and performing a transformation of the graph whenever we find one. A transformation consists of removing a subterm or rewrite edge and adding a new subterm or rewrite edge. Afterwards, unification and orientation edges are re-calculated. There are three graph transformations.

The first transformation is called an *SUR transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4 such that there is a subterm edge e_S from v_1 to v_2, a unification edge e_U between v_2 and v_3, and a rewrite edge v_R from v_3 to v_4. Then e_S is removed, and a new subterm edge e is added from v_1 to v_4, labelled with the same index as e_S (see Figure 1). This simulates the critical pair rule, because it unifies a subterm of a term with the larger side of an equation and replaces it with the smaller side of the equation. Figure 1 shows a critical pair between $f(a) \approx c$ and $a \approx b$.

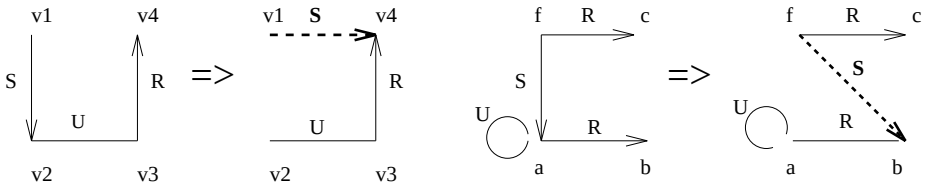

Figure 1: *SUR* transformation and an example

The second transformation is called an *RUR transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4, such that there is a rewrite edge e_{R_1} from v_1 to v_2, a unification edge e_U between v_1 and v_3, and a rewrite edge v_{R_2} from v_3 to v_4. Also $Term(v_2) \succ Term(v_4)$. Then e_{R_1} is removed, and a new rewrite edge e is added from v_2 to v_4 (see Figure 2). This simulates the critical pair rule, because it unifies the larger side of two equations and adds a new equation between the two smaller sides.

The third transformation is called an *RUR − rhs transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4, such that there is a rewrite edge e_{R_1} from v_1 to v_2, a unification edge e_U between v_2 and v_3, and a rewrite edge v_{R_2} from v_3 to v_4. Then e_{R_1} is removed, and a new rewrite edge e is added from v_1 to v_4 (see Figure 2). This simulates a simplification of the smaller side of an equation, because is unifies the smaller side with the larger side of another equation and replaces it with the smaller side of the other equation.

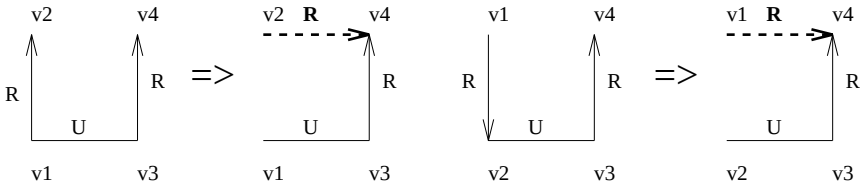

Figure 2: *RUR* and *RUR-rhs* transformation

The completeness result from [LS95] says that if a SOUR graph is created from a set of equations EQ and the above transformations are performed in any order, until no longer possible, then the resulting SOUR graph represents a convergent system, logically equivalent to EQ.

We describe briefly how this is lifted to the non-ground case, and refer the reader to [LS95] for more details. For the non-ground case, the initial SOUR graph is constructed in exactly the same way. However, the transformations are handled differently. We simulate Basic Completion. Therefore, each inference is performed by renaming the variables in one of the premises, and by applying a constraint to the conclusion of the inference, representing the unification problem. In the SOUR graph, the new edge that is added to the graph in a transformation is labelled with the constraint and renaming associated with the inference, which is a combination of the constraints and renamings labelling the edges which caused the transformation to be performed. Initial edges can be considered to have constraints and renamings which are trivial. As opposed to the ground case, each inference does not

represent a simplification. It is only possible to delete an edge when the inference represents a simplification.

Since edges are not deleted when new ones are added, a node represents a set of terms, instead of a single term. The semantics of SOUR graphs must also be modified to accommodate the constraints and renamings on the edges. Each dag of subterm edges still represents a term. But the constraints on the dag must be conjoined with each other, and the renamings on the edges must be applied to everything that appears underneath it in the term. Also, instead of performing transformations among edges with a unification constraint of true, we now perform transformations where the unifications constraints must be satisfiable, and the constraint representing the orientation is satisfiable. These constraints are passed along to to the new edge in a transformation, along with the constraints on the old edges.

In summary, the algorithm for completion of SOUR graphs is the same in the non-ground case as in the ground case, except for the fact that an old edge might not be deleted in a transformation, and the new edge is labelled by a renaming inherited from the old edges, and a constraint inherited from the old edges and the unification constraint determined by the unification problem. For simplicity, we present the completion procedure in the ground case, but the algorithm for the non-ground case is the same, except that messages will contain constraints and nodes will try to satisfy these constraints.

3 Concurrent Completion using SOUR Graphs

We now present our concurrent method for performing completion using SOUR graphs. The set of equations is represented as a SOUR graph. The number of vertices of the initial SOUR graph gives the number of independent children processes which will perform completion. A *root* process launches all these children processes. Completion is performed using edges as communication links. At the beginning of the concurrent completion process, the *root process* has knowledge of the whole SOUR graph. For each node, it launches a process. To each process is associated its identification number called its *tid*. The preliminary information that is initially loaded by each process, is the symbol of the node and a dictionary containing for each symbol its arity and its place in the precedence. Then, for each node, unification edges, subterm edges, and rewrite edges are sent in this order to the associated process using messages called *INITU*, *INITS* and *INITR* respectively. These messages contain characteristics of an edge. The root process creates a unification edge between every pair of nodes with the same symbol, with an initial value of false. One node adjacent to each unification edge is designated as being the node in charge of calculating the unification constraint. An *INITU* message is sent to the nodes on both ends of each unification edge. The *INITU* message contains the *tid* of the process at the other end of the edge, the initial false unification constraint, a notification whether that node is in charge of calculating the unification problem and an another notification whether that node needs to calculate the unification problem. *INITS* is sent to each node adjacent to a subterm edge. It contains the *tid* and the symbol of the process at the other end of the edge, a boolean indicating if the edge is incoming or outgoing, and the index of the subterm edge. *INITR* is sent to each

node adjacent to a rewrite edge. It contains the *tid*, the symbol of the process at the other end of the edge and a boolean indicating if the rewrite edge is incoming or outgoing. We call this phase the *Initialization phase*.

The set of information stored in a node is called its *state*. It is composed of its symbol *symb*, its dictionary *dico_order_arity* and its unification edges, its outgoing and incoming subterm edges and its outgoing and incoming rewrite edges, which are saved in data structures. Unification edges are saved in a list of unification edges *U_list*. Initially, the *U_list* of a process contains the unary cycle between the node and itself, which forms a unification edge with a true unification constraint. Incoming subterm edges are saved in a list of subterm edges *S_in_list*, outgoing subterm edges are saved in a list of subterm edges called *S_out_list*. In the same manner, rewrite edges are saved in *R_in_list* and *R_out_list*. So for now, the *state* of a child process is a record whose fields are its identification number *tid*, its symbol *symb*, its dictionary *dico_order_arity*, its list of edges *U_list*, *S_in_list*, *S_out_list*, *R_in_list*, *R_out_list*. This record will be extended with fields concerning semi-configurations *semiconfig_list* and fields concerning Unification: *list_infou*, *list_requestu* and *list_answeru* and fields concerning Orientation: *O_out_list*, *O_in_list*, *O_eq_list*, *calculo_list*, *requesto1_list*, *requesto2_list*, *answero1_list* and *answero2_list*, that we will describe later.

Completion, including the following different phases: configuration creation, configuration processing and Unification and Orientation computation, concerns all but the *root process*. A process works only by *reaction to messages*, which allows the whole process to be fully asynchronous.

Actions of a process implied by a received message are formalized using the *transition rules*.

Definition 1

- *An* α**-transition** *is the transformation of a* state *of a process to another* state*, when it receives a message. It is denoted by:* $(\{Mesg\}, State) \xrightarrow{\alpha} (Mesgset, State')$*, where* α *is a phase of completion like the creation of configurations, the processing of configurations, calculation of Unification and calculation of Orientation,* $Mesg$ *is the received message,* $State$ *is the* state *of the process before receiving the message* $Mesg$*,* $Mesgset$ *is the set of all messages that are consequences of message* $Mesg$*, which must be sent and* $State'$ *is the* state *of the same process with the consequences of receiving* $Mesg$*. In the set* $Mesgset$*, each message* $mesg$ *is characterized by its destination* $dest$*, and denoted* $mesg[dest]$*.*

- *A* **C-transition** *is an* α*-transition for the creation of configurations.*

  A **P-transition** *is an* α*-transition for the processing of configurations.*

  A **U-transition** *is an* α*-transition for Unification calculation.*

  An **O-transition** *is an* α*-transition for Orientation calculation.*

- *The binary + operator has got two arguments: a list and an element to add to this list. It is defined by: $L + x = L$ if x belongs to the list L else + pushes x at the end of L. The + operator can be overloaded such that its two arguments are lists. In this case for $L + L'$, + appends elements of the list L' to the list L without repetition.*

  The binary − operator has got two arguments: a list and an element to delete from this list. It is defined by: $L - x = L$ if x does not belong to the list L else − deletes x from L. The − operator can be overloaded such that its two arguments are lists. In this case for $L - L'$, − deletes all elements of L' from L.

  The binary $\otimes$ operator combines elementary computations on a node : Starting from a state, it adds the new information while updating the old. It is defined by: if ({Mesg}, State) $\xrightarrow{\alpha}$ (Mesgset1, State1) and ({Mesg}, State) $\xrightarrow{\beta}$ (Mesgset2, State2) then ({Mesg}, State) $\xrightarrow{\alpha\otimes\beta}$ (Mesgset1 $\cup$ Mesgset2, State3) where State3 is defined by:

  - $State3.tid = State.tid$
  - $State3.symb = State.symb$
  - $State3.dico\_order\_arity = State.dico\_order\_arity$
  - $State3.X = State.X + (State1.X - State.X) - (State.X - State1.X) + (State2.X - State.X) - (State.X - State2.X)$
    for $X \in \{U\_list, R\_in\_list, R\_out\_list, S\_in\_list, S\_out\_list, semiconfig\_list, infou\_list, requestu\_list, answeru\_list, O\_in\_list, O\_out\_list, O\_eq\_list, calculo\_list, requesto1\_list, requesto2\_list, answero1\_list, answero2\_list\}$.

  The $\otimes$ operator is commutative and associative.

  An **A-transition** *is an α-transition ({Mesg}, State) $\xrightarrow{A}$ (Mesgset, State'), which processes all* D, P, U *and* O-transitions *that can be applied to the state State. $A = \alpha_1 \otimes \alpha_2 \ldots \otimes \alpha_n$ where $\alpha_i \in \{D, U, P, O\}$.*

3.1 Implementation of Inference Rules for Completion

Completion, i.e. local graph transformation, is performed by cooperation between processes by message passing, because each process has a local view of the graph. In this section, we present the implementation of the inference rules of Completion, as the creation and processing of configurations.

3.1.1 Creation of configurations

For the creation of a configuration, we use the fact that a process can detect a sequence of two adjacent edges forming a configuration: SUR (see Figure 1), RUR-*rhs*, RUR (see Figure 2) with a unification edge between a node and itself, or two adjacent edges forming a semi-configuration UR, (a unification edge and an outgoing rewrite edge). A semi-configuration

or a configuration is defined by edges, therefore by the *tids* of processes at the ends of these edges. These patterns of two adjacent edges are detected when a new subterm or a new rewrite edge is added or when the unification constraint of a unification edge becomes true.

When a semi-configuration UR is detected, a message called *SEMICONF* is sent to the process at the other end of the unification edge. When a configuration of type SUR, *RUR-rhs* or RUR is detected, a message called *CONFIG* containing this configuration is sent to only one process, since we want to avoid redundant work. *SUR* and *RUR-rhs* configurations are sent to the process at the top left with respect to figures 1 and 2. An RUR configuration is sent to the process of the outgoing end of the rewrite edge with the maximal *tid*.

When a semi-configuration UR is detected, a message called *SEMICONF* is sent. All received messages *SEMICONF* are stored in a list called *semiconfig_list*.

Consider the creation of an SUR configuration and look at the example of figure 1. The process representing term a detects an SUR configuration containing the rewrite edge from itself to process with term b and the unification edge between itself and process with term f. This configuration is sent to process with term f to be processed.

Creation of configurations can be defined in terms of *C-transitions*. We only mention here data structures and messages, which are useful for the creation of semi-configurations and configurations. Also, states are represented as sets.

Notation : We define a function *Constraint* such that for a unification edge e_u, $Constraint(e_u) = True$ if the unification constraint of e_u is true, otherwise $Constraint(e_u) = False$ (for the ground case). For a message $Mesg$ of type *CONFIG*, we write *CONFIG(T, S)* to show that the configuration contained in $Mesg$ is of type T and contains the elements of S. T can be *SUR*, *RUR-rhs*, or *RUR*, and S contains the enumeration of the edges or the edge and the semi-configuration forming the configuration contained in the message. We generalize this notation such that for a message $Mesg$, $Mesg(S)$ means that the message $Mesg$ contains the elements of S. This notation can also be extended for describing the contents of a field, of an edge or of a semi-configuration. We denote subterm edges by e_s, e'_s and e''_s, rewrite edges by e_r, e'_r and e''_r, unification edges by e_u and semi-configurations contained in a *SEMICONF* message by $semiconf$.

Let St be the state of the process which receives the message. The creation of configurations can be expressed by the following *C-transitions*.

The first seven *C-transitions* show how an *INITR* message, indicating the addition of a new rewrite edge, causes the creation of semi-configurations and *SUR*, *RUR* and *RUR-rhs* configurations.

(1) $(\{INITR(e_r)\},\ \{St.R\_out\_list,\ St.U\_list(e_u(tid))\})$
$\xrightarrow{C}$
$(\{SEMICONF(e_r, e_u)[tid]\},\ \{St.R\_out\_list + e_r,\ St.U\_list(e_u)\})$
if $Constraint(e_u) = True$.

(2) $(\{INITR(e_r)\},\ \{St.R\_out\_list,\ St.R\_in\_list(e'_r(tid))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs}, e_r, e'_r)[tid]\},\ \{St.R\_out\_list + e_r,\ St.R\_in\_list(e'_r)\})$

(3) $(\{INITR(e_r(tid1))\},\ \{St.R\_out\_list,\ St.R\_out\_list(e'_r(tid2))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR}, e_r, e'_r)[tidmax]\},\ \{St.R\_out\_list + e_r,\ St.R\_out\_list(e'_r)\})$
To prevent two messages *CONFIG* containing the same configuration from being sent, *CONFIG* message is sent to the process with the tid $tidmax$ such that $tidmax = max(tid1, tid2)$.

(4) $(\{INITR(e_r)\},\ \{St.R\_out\_list,\ St.S\_in\_list(e_s(tid, index))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR}, e_s, e_r)[tid]\},\ \{St.R\_out\_list + e_r,\ St.S\_in\_list(e_s)\})$

(5) $(\{INITR(e_r(tid))\},\ \{St.R\_in\_list,\ St.R\_out\_list(e'_r)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs}, e_r, e'_r)[tid]\},\ \{St.R\_in\_list + e_r,\ St.R\_out\_list(e'_r))\})$

(6) $(\{INITR(e_r(tid1))\},\ \{St.R\_out\_list,\ St.semiconfig\_list(semiconf(e'_r(tid2), u(tid3)))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR}, e_r, semiconf)[tidmax]\},\ \{St.R\_out\_list + e_r, St.semiconfig\_list(semiconf)\})$
To prevent two messages *CONFIG* containing the same configuration from being sent, if the tid of the process receiving *INITR* is larger than $tid3$ then the message *CONFIG* is sent to the process with tid $tid1$, otherwise to the process with tid $tid2$.

(7) $(\{INITR(e_r(tid))\},\ \{St.R\_in\_list,\ St.semiconfig\_list(semiconf)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs}, e_r, semiconf)[tid]\},\ \{St.R\_in\_list + e_r,\ St.semiconfig\_list(semiconf)\})$

The following *C-transition* indicates that a semi-configuration can be formed when there exists an outgoing rewrite edge and when the unification constraint of a unification edge becomes $True$, after a *INITU* message has been received.

(8) $(\{INITU(e_u(tid))\},\ \{St.U\_list,\ St.R\_out\_list(e_r)\})$
$\xrightarrow{C}$
$(\{SEMICONF(e_u, e_r)[tid]\},\ \{St.U\_list + e_u,\ St.R\_out\_list(e_r)\})$
if $Constraint(e_u) = True$

The following two *C-transitions* express the creation of *SUR* configurations due to the addition of an incoming subterm edge.

(9) $(\{INITS(e_s(tid, index))\},\ \{St.S\_in\_list,\ St.semiconfig\_list(semiconf)\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR}, e_s, semiconf)[tid]\},\ \{St.S\_in\_list + e_s,\ St.semiconfig\_list(semiconf)\})$

(10) $(\{INITS(e_s(tid, index))\},\ \{St.S\_in\_list,\ St.R\_out\_list(e_r))\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR}, e_s, e_r)[tid]\},\ \{St.S\_in\_list + e_s,\ St.R\_out\_list(e_r)\})$

The following three *C-transitions* show how a received *SEMICONF* message is used to form *SUR*, *RUR-rhs* and *RUR* configurations respectively.

(11) $(\{SEMICONF(semiconf)\},\ \{St.S\_in\_list(e_s(tid, index)),\ St.semiconfig\_list\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{SUR}, e_s, semiconf)[tid]\},\ \{St.S\_in\_list(e_s),\ St.semiconfig\_list + semiconf\})$

(12) $(\{SEMICONF(semiconf)\},\ \{R\_in\_list(e_r(tid)),\ semiconfig\_list\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR-rhs}, e_r, semiconf)[tid]\},\ \{St.R\_in\_list(e_r(tid)),$
$St.semiconfig\_list + semiconf\})$

(13) $(\{SEMICONF(semiconf(e_u(tid1), e_r(tid1, tid2)))\},\ \{St.R\_out\_list(e'_r(tid3)),$
$St.semiconfig\_list\})$
$\xrightarrow{C}$
$(\{CONFIG(\text{RUR}, e'_r, semiconf)[tid]\},\ \{St.R\_out\_list(e'_r),\ St.semiconfig\_list + semiconf\})$
In the same manner as before, the configuration contained in message *CONFIG* is only sent once to process with tid $tid3$ if the tid of the process receiving the message *SEMICONF* is larger than $tid1$, otherwise the message *CONFIG* is sent to the process with tid $tid2$, the process at the outgoing extremity of the rewrite edge of $semiconf$. So tid is equal to $tid2$ or to $tid3$.

3.1.2 Processing of configurations

The processing of configurations is set off when a *CONFIG* message is received by a process. The message then releases a sequence of actions depending on the type of the received

configuration. The process receiving the message processes the configuration i.e. performs the associated transformation, and sends messages which will be used to update information of other processes. The two main steps of the processing of a configuration are the addition and the deletion of an edge. The completion rules of figures 1 and 2 show these two steps and can be implemented using *P-transitions* and *O-transitions* (see section 3.3).

Messages used for the processing of configurations are *INITS* and *INITR* to announce that a new edge is added and *UPDATES*, *UPDATER* and *UPDATESEMICONF* to delete respectively a subterm edge, a rewrite edge and a semi-configuration from the state of a process. An *INITR* or *INITS* message can be ignored, because the rewrite edge or the subterm edge contained in the message already exists in the state of the process receiving this message. Consequences of this messages are already done. This remark is valid for *D, P, U* and *O-transitions*.

Let St be the state of the process receiving a message. The processing of configurations can be expressed by the following *P-transitions*.

The following two *P-transitions* show the processing of an *SUR* configuration i.e. the addition and the deletion of a subterm edge.

(14) $(\{CONFIG(\mathrm{SUR}, e_s(tid1, index), e_r(tid2))\},\ \{St.S\_out\_list(e_s(tid1, index))\})$
$\xrightarrow{P}$
$(\{INITS(e'_s(tid2, index))[tid2],\ UPDATES(e_s)[tid1]\},\ \{St.S\_out\_list + e'_s - e_s\})$
where e'_s is the new outgoing subterm edge and $tid2$ is the tid of the process at the outgoing extremity of e_r.

(15) $(\{UPDATES(e_s(tid, index))\},\ \{St.S\_in\_list(e_s(tid, index))\})$
$\xrightarrow{P}$
$(\{\},\ \{St.S\_in\_list - e_s\})$

The following four *P-transitions* implement the post-processing of *RUR* and *RUR-rhs* configurations i.e. the deletion of a rewrite edge. The first part of this processing i.e. the addition of a rewrite edge is linked to the Orientation calculation and so is postponed to section 3.3.

(16) $(\{UPDATER(e_r)\},\ \{St.R\_out\_list(e_r)\})$
$\xrightarrow{P}$
$(\{\},\ \{St.R\_out\_list - e_r\})$

(17) $(\{UPDATER(e_r)\},\ St.R\_out\_list(e_r),\ St.U\_list(e_u(tid))\})$
$\xrightarrow{P}$
$(\{UPDATESEMICONF(e_r, u(tid))[tid]\},\ \{St.R\_out\_list - e_r,\ St.U\_list(e_u)\})$
if $Constraint(e_u) = True$.

(18) $(\{UPDATER(e_r)\},\ \{St.R\_in\_list(e_r)\})$
$\xrightarrow{P}$
$(\{\},\ \{St.R\_in\_list - e_r\})$

(19) $(\{UPDATESEMICONF(semiconf)\},\ \{St.semiconfig\_list(semiconf)\})$
$\xrightarrow{P}$
$(\{\},\ \{St.semiconfig\_list - semiconf\})$

3.2 Concurrent Unification

Although concurrency cannot improve the worst-case behavior of unification [DKM84], we give an algorithm that reduces the amount of processing by a single node. Unification is evaluated by a request-answer method. In a previous implementation, we used only a bottom-up method for computing Unification, but a lots of calculated constraints were never used. So we decided to mix a bottom-up and a top-down method. A process sends a request to each child to ask if there is a unification edge with a $True$ constraint between itself and a given process. The child responds only if the constraint is $True$.

A Unification problem, in the ground case, is to determine if two terms are equal i.e. if the unification constraint of the unification edge between the two processes representing the two terms is $True$. Consider the problem represented by figure 3. U_0 is the unification edge between the process representing term $f(s_1, \ldots, s_n)$ and the process representing term $f(t_1, \ldots, t_n)$. Computation of the unification constraint c_0 of unification edge U_0 depends on the calculation of unification constraints c_1,. . .,c_n of unification edges U_1,. . .,U_n. We have $c_0 = c_1 \wedge \ldots \wedge c_n$. Each constraint c_i represents the unification problem $s_i \stackrel{?}{=} t_i$.

Unification must be re-calculated when subterm edges are added or deleted.

Using figure 3, we will explain the principles of the calculation of the unification constraint c_0 concurrently:

- Initially, all unification constraints are false except unification constraints of unary cycle unification edges.
- Suppose process p represents $f(s_1, \ldots, s_n)$ and p' represents $f(t_1, \ldots, t_n)$, such that p' is in charge of computing c_0. Process p sends to p' the *tids* of processes represen-

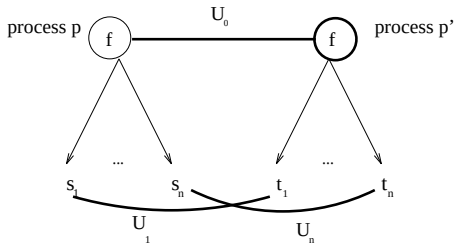

Figure 3: Calculation of the unification constraint c_0 of the unification edge U_0

ting terms $s_1, \ldots, s_n$, so that p' knows on which children's unification constraints c_0 depends.

This is done with *INFOU* messages. An *INFOU* message contains the unification edge to calculate, the *tid* of the process at the extremity of the outgoing subterm edge, and the index of this subterm edge. *INFOU* received messages are saved in *infou_list*.

- When p' receives this information, it asks the processes representing terms t_1, ..., t_n, if their unification constraints c_1, ..., c_n respectively are $True$. This calculation is top-down. A request is sent with a *REQUESTU* message. *REQUESTU* received messages are saved in *requestu_list*.

- If a unification constraint c_i $(i > 0)$ becomes $True$ and has a saved request concerning the computation of c_0, then the process representing term t_i informs process p' that c_i became $True$. This calculation is bottom-up.

  An answer is announced with an *ANSWERU* message. *ANSWERU* received messages are saved in *answeru_list*.

- If a process detects that a unification constraint becomes $True$ i.e. for the corresponding edge, the number of *ANSWERU* messages is equal to the arity of the symbol of the process, then it must inform the other extremity of the unification edge.

Two observations permit us to optimize this request-answer method for the calculation of unification. Indeed, it is useless to send the same request several times (*REQUESTU* messages) and the same information (*INFOU* messages) and it is useless to calculate all unification constraints of all unification edges. A request can be sent only once, because requests are saved by processes receiving them and when an answer to a request is available, the answer will be sent. The unification constraint of a unification edge must be calculated, only if the unification edge forms a semi-configuration with an adjacent outgoing rewrite edge. This observation is implemented by the following principals. The *Initialization Phase* can be considered as a pre-processing phase. If there exist an outgoing rewrite edge and a unification edge for one node of the initial graph, then both processes at the extremity of this unification edge must be informed that they need to calculate the unification constraint, but only one process will calculate it. In this case, the process not in charge of computing

the unification constraint of this unification edge is allowed to send *INFOU* information messages to the process at the other end of the edge to permit it to calculate the constraint. The optimization resides in the fact that unification is calculated only if it is needed. During the *Completion* process, if an outgoing rewrite edge is added to the state of a process and if this process has got a unification edge, the unification constraint of this unification edge must be calculated and the other extremity of the unification edge is notified of this fact using either a message *INFOU* if the process is in charge of computing the unification edge or a message *NEEDU* otherwise.

Notations We define a function *Needu* such that for a unification edge e_u, $Needu(e_u)$ returns $True$ if the unification constraint of e_u must be calculated (as described before), otherwise it returns $False$. We denote the information contained in *INFOU* messages by $infou$ and $infou'$, the information contained in *REQUESTU* messages by $requestu$ and $requestu'$, and the the information contained in a *ANSWERU* message by $answeru$ and $answeru'$.

Let St be the state of the process which receives the message. The preceding algorithm can be expressed using the following *U-transitions*.

The following four *U-transitions* explain how the data structures *U_list*, *infou_list*, *requestu_list* and *answeru_list* are updated.

(20) $(\{INITU(e_u(tid))\},\ \{St.U\_list(e_u)\})$
$\xrightarrow{U}$
$(\{\},\ \{St.U\_list(e_u) - e'_u(tid) + e_u(tid)\})$
if e'_u is the unification edge e_u with a False unification constraint and if $Constraint(e_u) = True$ in the *INITU* message.

(21) $(\{INFOU(infou(e_s(tid, index)), e_u(tid))\},\ \{St.infou\_list,\ St.U\_list(e_u)\})$
$\xrightarrow{U}$
$(\{\},\ \{St.infou\_list - infou'(index, e_u) +$
$infou(e'_s, e_u),\ St.U\_list(e_u)\})$
where tid is the outgoing extremity of the subterm edge e'_s and $infou'$ is the old unification information concerning the calculation of the unification of e_u using the index $index$.
If initially, $Needu(e_u) = False$, this value is changed to $True$.

(22) $(\{REQUESTU(requestu(e_u(tid), e'_u(tid2, tid1), e'_s(tid1, tid, index)))\},\ \{St.U\_list(e_u),$
$S\_in\_list(e'_s(tid2, index)),\ St.requestu\_list\})$
$\xrightarrow{U}$
$(\{\},\ \{St.U\_list(e_u),\ St.S\_in\_list(e'_s),\ St.requestu\_list + requestu - requestu'(e'_u, e_s)\})$
where $requestu'$ is due to the old *REQUESTU* message concerning e_u and the index $index$.

(23) $(\{ANSWERU(e'_u(tid1, tid2), e_u(St.tid, tid), index)\},\ \{St.U\_list(e_u(tid)),$
$St.S\_out\_list(e'_s(tid1, index))\})$
$\xrightarrow{U}$
$(\{\},\ \{St.U\_list(e_u),\ St.S\_out\_list(e'_s),\ St.answeru\_list - answeru' + answeru\})$
where $answeru'$ is the old received $ANSWERU$ message concerning e_u and $index$.

The following four *U-transitions* shows how an explicit addition of an outgoing subterm edge (*INITR* message) or an implicit addition of an outgoing subterm edge (processing of an *SUR* configuration) causes messages for the calculation of Unification to be sent. These messages are either information (*INFOU* messages) or requests.

(24) $(\{INITS(e_s)\},\ \{St.S\_out\_list,\ St.U\_list(e_u(tid))\})$
$\xrightarrow{U}$
$(\{INFOU(e_s, e_u)[tid]\},\ \{St.S\_out\_list + e_s,\ St.U\_list(e_u)\})$
if $Constraint(e_u) = False$, $Needu(e_u) = True$ and the process that receives *INITS* is not in charge of calculating the unification constraint of e_u.

(25) $(\{INITS(e_s(tid, index))\},\ \{St.S\_out\_list,\ St.U\_list(e_u),\ St.infou\_list(infou(e_u, index))\})$
$\xrightarrow{U}$
$(\{REQUESTU(infou, e_s)[tid]\},\ \{St.S\_out\_list + e_s,\ St.U\_list(e_u),\ St.infou\_list(infou)\})$
if $Constraint(e_u) = False$ and the process that receives *INITS* is in charge of calculating the unification constraint of e_u.

(26) $(\{CONFIG(\text{SUR}, e_s(index), e_r(tid'))\},\ \{St.S\_out\_list(e_s(index)),\ St.U\_list(e_u(tid)\})$
$\xrightarrow{U}$
$(\{INFOU(e'_s(tid', index), e_u)[tid]\},\ \{St.S\_out\_list - e_s + e'_s,\ St.U\_list(e_u)\})$
if $Constraint(e_u) = False$, $Needu(e_u) = True$ and the process that receives *CONFIG* is not in charge of calculating the unification constraint of e_u.
Also tid' is the tid of the process at the outgoing extremity of e_r. e'_s is the new outgoing subterm edge.

(27) $(\{CONFIG(\mathrm{SUR}, e_s(index), e_r(tid'))\},\ \{St.S\_out\_list(e_s(index)),\ St.U\_list(e_u(tid)),\ St.infou\_list(infou(index, e_u(tid)))\})$
$\xrightarrow{U}$
$(\{REQUESTU(e'_s(index, tid'), infou)[tid']\},\ \{St.S\_out\_list - e_s + e'_s,\ St.U\_list(e_u),\ St.infou\_list(infou)\})$
if $Constraint(e_u) = False$ and the process that receives $CONFIG$ is in charge of the calculation of the unification constraint of e_u.
Also tid' is the tid of the process at the outgoing extremity of e_r. e'_s is the new outgoing subterm edge.

The following four *U-transitions* show the up-propagation of unification information, in particular $ANSWERU$ messages.

(28) $(\{INITS(e_s(tid, index))\},\ \{St.S\_in\_list,\ St.U\_list(e_u),\ St.requestu\_list(requestu(e_u, index))\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u, e_s)[tid]\},\ \{St.S\_in\_list + e_s,\ St.U\_list(e_u),\ St.requestu\_list(requestu)\})$
if $Constraint(e_u) = True$.

(29) $(\{INITU(e_u(tid))\},\ \{St.U\_list(e_u(tid)),\ St.S\_in\_list(e_s(tid', index)),\ St.requestu\_list(requestu(e_u, e_s)\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u, e_s)[tid']\},\ \{St.U\_list(e_u),\ St.S\_in\_list(e_s),\ St.requestu\_list(requestu)\})$
if $Constraint(e_u) = False$ initially and $Constraint(e_u) = True$ in the $INITU$ message.

(30) $(\{REQUESTU(requestu(e_u, e_s(tid, index)))\},\ \{St.U\_list(e_u),\ S\_in\_list(e_s(tid, index)),\ St.requestu\_list\})$
$\xrightarrow{U}$
$(\{ANSWERU(e_u, e_s)[tid]\},\ \{St.U\_list(e_u),\ St.S\_in\_list(e_s),\ St.requestu\_list(requestu)\})$
if $Constraint(e_u) = True$.

(31) $(\{ANSWERU(e_u(tid))\},\ \{St.U\_list(e_u(tid)),\ St.requestu\_list(e'_u(tid1), e_u(tid), e_s(index, tid1)),\ St.S\_in\_list(e_s))\})$
$\xrightarrow{U}$
$(\{ANSWERU(e'_u, e_s))[tid1]\},\ \{St.U\_list(e_u),\ St.requestu\_list(e'_u, e_u, e_s),\ St.S\_in\_list(e_s))\})$
if the number of $ANSWERU$ received messages for calculating the unification constraint of e_u, initially False is equal to the arity of the symbol of the process which receives $ANSWERU$, the unification constraint of e_u is then changed to $True$.

The following *U-transition* indicates that if unification information is received by a process, this process is able to send requests asking if there exists a particular unification constraint $True$ between two processes.

(32) $(\{INFOU(infou(e'_s(tids, index)), e_u)\},\ \{St.S\_out\_list(e_s(tid, index)),\ St.infou\_list,\ St.U\_list(e_u)\})$
$\xrightarrow{U}$
$(\{REQUESTU(infou, e_s)[tid]\},\ \{St.S\_out\_list(e_s),\ St.infou\_list(infou),\ St.U\_list(e_u)\})$
where $tids$ is the outgoing extremity of the subterm edge e'_s and $infou$ is the old unification information concerning the calculation of the unification of e_u using the index $index$.
If initially, $Needu(e_u) = False$, this value is changed to $True$.

The following two *U-transitions* mean that, if a received request has no answer available, requests and information must be sent to make this answer available later.

(33) $(\{REQUESTU(requestu(e_u, e_s)\},\ \{St.U\_list(e_u),\ St.S\_in\_list(e_s),\ S\_out\_list(e'_s(tid)),\ St.infou\_list(infou(e_u, e''_s))\})$
$\xrightarrow{U}$
$(\{REQUESTU(e_u, e'_s, e''_s)[tid]\},\ \{St.U\_list(e_u),\ St.S\_in\_list(e_s),\ St.S\_out\_list(e'_s),\ St.infou\_list(infou)\})$
if $Constraint(e_u) = False$ and if the process which receives $REQUESTU$ is in charge of the calculation of the unification constraint of e_u.

(34) $(\{REQUESTU(requestu(e_u(tid), e_s)\},\ \{St.U\_list(e_u(tid)),\ St.S\_in\_list(e_s),\ S\_out\_list(e'_s)\})$
$\xrightarrow{U}$
$(\{INFOU(e_u, e'_s)[tid]\},\ \{St.U\_list(e_u),\ St.S\_out\_list(e'_s),\ St.S\_in\_list(e_s)\})$
if $Constraint(u) = False$ and if the process which receives $REQUESTU$ is not in charge of the calculation of the unification constraint of e_u.
$requestu'$ is due to the old $REQUESTU$ message concerning e_u and the index $index$.

This following *U-transition* is used to decide when a unification constraint becomes $True$.

(35) $(\{ANSWERU(e_u(tid))\},\ \{St.U\_list(e_u(tid))\})$
$\xrightarrow{U}$
$(\{INITU(e_u)[tid]\},\ \{St.U\_list(e_u)\})$
if the number of $ANSWERU$ received messages for calculating the unification constraint of e_u, initially False is equal to the arity of the symbol of the process which receives $ANSWERU$, the unification constraint of e_u is then changed to $True$.

The following *U-transitions* optimize of the Unification calculation. This optimization, as explained above, is based on the criterion that Unification needs to be calculated only when a process has got a unification edge adjacent to an outgoing rewrite edge. So when an outgoing rewrite edge is added explicitly using a *INITR* message or implicitly when a configuration is processed, Unification must be calculated. For this last case, we use *O-transitions*, that will be expressed later in section 3.3.

(36) $(\{INITR(e_r)\},\ \{St.R\_out\_list,\ St.U\_list(e_u(tid)),\ St.S\_out\_list(e_s(tid1))\})$
$\xrightarrow{U}$
$(\{INFOU(e_u, e_s)[tid]\},\ \{St.R\_out\_list + e_r,\ St.U\_list(e_u),\ St.S\_out\_list(e_s)\})$
if $Constraint(e_u) = False$ and the process receiving *INITR* is not in charge of the calculation of the constraint of e_u and $Needu(e_u) = False$ in the initial state of the process receiving *INITR*. After the processing of the message *INITR*, the value of $Needu(e_u)$ is changed to $True$.

(37) $(\{INITR(e_r)\},\ \{St.tid,\ St.R\_out\_list,\ St.U\_list(e_u(tid)),\ St.S\_out\_list(e_s(tid1))\})$
$\xrightarrow{U}$
$(\{NEEDU(St.tid)[tid]\},\ \{St.tid,\ St.R\_out\_list + e_r,\ St.U\_list(e_u),\ St.S\_out\_list(e_s)\})$
if $Constraint(e_u) = False$ and the process receiving *INITR* is in charge of the calculation of the constraint of e_u and $Needu(e_u) = False$ in the initial state of the process receiving *INITR*. After the processing of the message *INITR*, the value of $Needu(e_u)$ is changed to $True$.

(38) $(\{NEEDU(e_u)\},\ \{St.S\_out\_list(s(tid, index)),\ St.U\_list(e_u(tid1))\})$
$\xrightarrow{U}$
$(\{INFOU(e_u, e_s)[tid1]\},\ \{St.S\_out\_list(s),\ St.U\_list(e_u)\})$
if the process receiving the message *NEEDU* is not in charge of the calculation of the unification constraint of e_u. After processing of the message, $Needu(e_u) = True$.

The soundness and completeness results of this set of rules for this concurrent Unification algorithm are given in section 3.6.

3.3 Concurrent Orientation

An Orientation problem is to determine if a term s is bigger than a term t with respect to LPO. If it is, we create an outgoing orientation edge from the process representing term s to the process representing term t. We use a request-answer method as we did for Unification.

Each process saves its orientation edges in three lists: *O_in_list*, *O_out_list* and *O_eq_list* depending on whether the orientation edge is incoming, outgoing or 'equal'. This last case means that there exists a unification edge with a true unification constraint between these two processes. So, an orientation edge is characterized by the *tid* and the symbol of the process at the other extremity of the orientation edge.

3.3.1 Graphic Implementation of *LPO*

The concurrent calculation of Orientation is based on the simplification ordering LPO (see definition in section 2). We will now see how to pass from a textual definition of LPO to a graphic definition adapted to our concurrent calculation of Orientation.

Let p_1 be the process representing the term $s = f(s_1, \ldots, s_n)$ and p_2 the process representing the term $t = g(t_1, \ldots, t_m)$. We want to know if $s \succ_{lpo} t$ i.e. if we can add an orientation edge O_o going from the process of term s to the process of term t. The three conditions of the definition of LPO can be expressed by the following schemas respectively.

1. There is an outgoing orientation edge O_0 from p_1 to p_2, if the symbol of process p_1 is bigger than the symbol of process p_2 and if process p_1 is linked to each child process of p_2 by an outgoing orientation edge (see figure 4).

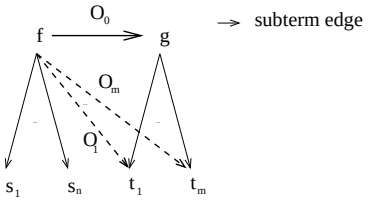

Figure 4: Implementation of LPO - Case 1

2. There is an outgoing orientation edge O_0 from p_1 to p_2, if there is an outgoing or 'equal' orientation edge from one child process of p_1 to p_2 (see figure 5).

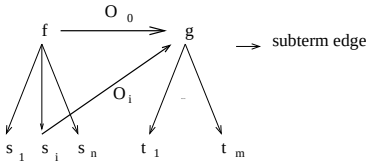

Figure 5: Implementation of LPO - Case 2

3. There is an outgoing orientation edge O_0 from p_1 to p_2, if the symbol of process p_1 is equal to the symbol of process p_2 and if process p_1 is linked to each child process of p_2 by an outgoing orientation edge, and there exists a child process of p_1 (linked with p_1 with a subterm edge of index i) which has got an outgoing orientation edge to a child process of p_2 (linked with p_2 with a subterm edge of index i), and all orientation edges from p_1 (linked with p_1 with a subterm edge of index $j < i$) to a child process of p_2 (linked with p_2 with a subterm edge of index j) have as direction 'equal'. Figure

6 presents the case where the n^{th} child of p_1 is linked to the n^{th} child process of p_2 with an outgoing orientation edge and $O_{n+1},\ldots,O_{2n-1}$ have as direction 'equal'.

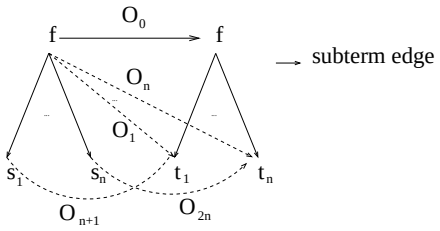

Figure 6: Implementation of LPO - Case 3

3.3.2 Principles of the concurrent calculation of Orientation

Orientation is evaluated by a request-answer method. The request consists of a process asking each of its children if there is an incoming orientation edge between itself and another process. The child process answers if such an edge exists.

The algorithm of concurrent calculation of Orientation works according to the following principles:

- The concurrent calculation of Orientation is based on the knowledge of the symbols of the two processes at the extremities of the orientation edge to orient.
- The Orientation calculation takes place when:
  - an *RUR-rhs* or an *RUR* configuration is received, to orient the new rewrite edge to add. If there is no orientation information available, requests are made to ask how to orient this edge and the process records that this information of orientation will serve to orient the corresponding rewrite edge. We will not insist on this in what is following for reasons of readability.
  - an *SUR* configuration is received. In this case, the direction of outgoing orientation edges are no longer sure, new orientation request messages are sent.
- 1. Assume that a process p_1 needs to calculate the orientation edge o between itself and the process p_2, such that the symbol of p_1 is $symb_1$ and the symbol of p_2 is $symb_2$ and $symb_1 > symb_2$ with respect to the precedence. Then we proceed as follows:
     - If $symb_2$ is a constant, an outgoing orientation edge can be added from p_1 to p_2 in p_1's *O_out_list* and in p_2's *O_in_list*.
     - Otherwise, process p_1 sends a message to process p_2 to tell it that it must calculate the orientation edge o. This message contains characteristics of child processes of p_1 and is called $CALCULO$.

- When process p_2 receives this message, it saves it in the data structure called *calculo_list* and sends to each of its children a request, to ask them if there is an orientation edge from p_1 to them. Messages used for doing that are *REQUESTO1* messages. For optimization, a request is not re-sent, if the same request has already been sent.
- When p_3, child process of p_2, receives this request, it does not automatically process it. The consequences of a request are processed only if there still exists an outgoing subterm edge from p_2 to p_3.
  If there still exists an outgoing subterm edge from p_2 to p_3, when p_3 receives this request, it saves it in *requesto1_list* and the following tests are made.
- If p_3's *O_in_list* contains an orientation edge from p_1 to p_3, p_3 sends to p_2 an answer saying that there is an orientation edge from p_1 to p_3. This answer is sent using an *ANSWERO1* message. This message will be saved in p_2's *answero1_list*.
- If p_3's *O_out_list* contains an outgoing orientation edge from p_3 to p_1, process p_3 sends a message to process p_2 to tell it that the orientation edge o is going from p_2 to p_1 using a message $OUTO$ containing a flag equal to 2. When process p_2 receives this message, if there still exists an outgoing subterm edge from p_2 to p_3, p_2 will tell process p_1 that the orientation edge o is going from p_2 to p_1 using a message *INO*.
- If there is no orientation edge between p_3 and p_1, then either an orientation edge can be easily added, because p_1 or p_3 is a constant, or requests must be sent considering the symbols of p_1 and p_3.
- If process p_2 has received n answers messages *ANSWERO1*, one from each of its n child process p_k saying that there is an outgoing edge from p_1 to p_k, p_2 can decide that the orientation edge o is going from p_1 to p_2 and tells p_1 using a message $OUTO$, and adds the new orientation edge o to its *O_in_list*. If o exists in p_1's *O_in_list*, this information is updated, because the state of a process can not contain contradictory information. If this orientation information serves to add a rewrite edge, the corresponding rewrite edge is added. We will not insist on this in what is following for reasons of readability.

2. Assume that process p_1 needs to calculate the orientation edge o between itself and process p_2, such that symbols $symb_1$ and $symb_2$ of p_1 and p_2 are the same and are function symbols.
   - Processes p_1 and p_2 must calculate the direction of the orientation edge o. Process p_1 sends a message to process p_2 to tell it that it must calculate the orientation edge o, with a $CALCULO$ message.
   - When process p_2 receives this message, it saves it and sends to each of its children a message called *REQUESTO1* and a message called *REQUESTO2*, to ask them if there is an orientation edge from p_1 to them and if there is an 'equal' or incoming orientation edge from child processes of p_1 to children processes of p_2 with the same index.

Furthermore, process p_2 sends a message called *CALCULO1* to process p_1 to tell it that it must calculate the orientation edge o too. This message *CALCULO1* is a consequence of message *CALCULO* and contains the tids and the indexes of children processes of p_2. Consequences of message *CALCULO1* are the same as consequences of message *CALCULO* (except that a message *CALCULO* is no longer sent). Received messages *CALCULO1* are saved in *calculo_list*.

- When p_3, child process of p_2, receives a message *REQUESTO1*, it saves this message in its *requesto1_list* and makes the following test.
  If p_3's *O_in_list* or *O_out_list* contains an orientation edge between p_1 and p_3, p_3 has the same behavior as in the previous point.
  If there is no orientation edge between p_3 and p_1, then either an orientation edge can be easily added, because p_1 or p_3 is a constant, or requests must be sent considering the symbols of p_1 and p_3.
- When p_3, i^{th} child process of p_2, receives a message *REQUESTO2*, it saves this message in its *requesto2_list* and makes the following test.
  If p_3's *O_in_list* (respectively *O_eq_list*) contains an orientation edge from (respectively between) a child process p_4 of p_1 with index i to p_3 (child process of p_2) sends to process p_2 a message called *ANSWERO2* containing the flag $>$ (respectively the flag $=$).
  If there is no orientation edge between p_4 and p_3, then either an orientation edge can be easily added, because p_4 or p_3 is a constant, or requests must be sent considering the symbols of p_4 and p_3.
- If process p_2 has received n messages *ANSWERO1* one from each of its n children processes and if process p_2 has received k messages *ANSWERO2*, containing 'equal' orientation edge (flag $=$), from its k first children processes and a message *ANSWERO2*, containing an incoming edge (flag $>$), from its $k+1^{st}$ child process such that $k \in \{0, \ldots, n-1\}$ and if all these messages concern the calculation of the orientation edge o, process p_2 can decide that the orientation edge o is going from p_1 to p_2. Process p_2 sends to process p_1 a message *OUTO* , to tell it the direction of o and add the new orientation edge o to its *O_in_list*. We will not insist on this in what is following for reasons of readability.
  Process p_1 makes the same test and can decide that the orientation edge o is going from p_2 to p_1.

Notation : We denote orientation edges by e_o, e'_o and e''_o, the information contained in a *REQUESTO1* message by $requesto1$ and $requesto1'$, the information contained in a *REQUESTO2* message by $requesto2$ and $requesto2'$, the information contained in a *ANSWERO1* message by $answero1$ and $answero1'$, the information contained in a *ANSWERO2* message by $answero2$ and $answero2'$, the information contained in a *CALCULO* or *CALCULO1* message by $calculo$ and $calculo'$.

Let St be the state of the process which receives the message. The preceding algorithm can be expressed more specifically using the following *O-transitions*.

The following five *O-transitions* explain how the data structures *calculo_list*, *requesto1_list*, *requesto2_list* *answero1_list* and *answero2_list* are updated.

(39) $(\{CALCULO(calculo(e_o(tid1, symb1), e_s(tid1, tid2, index))\},\ \{St.calculo\_list\})) \xrightarrow{O} (\{\},\ \{St.calculo\_list - calculo'(e_o, e'_s(tid1, index)) + calculo\})$
where $calculo'$ is due to an old saved $CALCULO$ message concerning e_o and $index$.

(40) $(\{REQUESTO1(requesto1(e_o(tid1, tid2), e'_o(St.tid, tid1), index))\},\ \{St.tid,\ St.S\_in\_list(e_s(tid2, index)),\ St.requesto1\_list\}) \xrightarrow{O} (\{\},\ \{St.tid,\ St.S\_in\_list(e_s),\ St.requesto1\_list + requesto1\})$

(41) $(\{REQUESTO2(requesto2(e_o(tid1, tid2), e'_o(St.tid, tid3), index))\},\ \{St.tid,\ St.S\_in\_list(e_s(tid2, index)),\ St.requesto2\_list\}) \xrightarrow{O} (\{\},\ \{St.tid,\ St.S\_in\_list(e_s),\ St.requesto2\_list - requesto2'(e_o, index) + requesto2\})$
where $requesto2'$ is due to an old saved *REQUESTO2* message concerning e_o and $index$ and $tid3$ is the tid of $index^{th}$ child process of process with tid $tid1$.

(42) $(\{ANSWERO1(answero1(e_o(tid1, St.tid), e'_o(tid2, tid1), index))\},\ \{St.tid,\ St.S\_out\_list(e_s(tid2, index),\ answero1\_list)\}) \xrightarrow{O} (\{\},\ \{St.tid,\ St.S\_out\_list(e_s),\ St.answero1\_list + answero1\})$

(43) $(\{ANSWERO2(answero2(e_o(tid2, tid1), e'_o(St.tid, tid3), index))\},\ \{St.tid,\ St.S\_out\_list(e_s(tid2, index)),\ St.answero2\_list\}) \xrightarrow{O} (\{\}, \{St.tid,\ St.S\_out\_list(e_s),\ St.answero2\_list - answero2'(e_o, index) + answero2\})$
where $answero2'$ is due to an old received *ANSWERO2* message concerning e_o and $index$ and $tid3$ is the tid of $index^{th}$ child process of process with tid $tid1$.

The following five *O-transitions* show the processing of an RUR configuration, in particular when an outgoing rewrite edge can be added, because an outgoing orientation edge is present or can be easily added.

(44) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2))\},\ \{St.R\_in\_list(e_r(tid1)),\ St.O\_out\_list(e_o(tid2)),\ St.R\_out\_list\})$
$\xrightarrow{O}$
$(\{INITR(St.tid)[tid2]\},\ \{St.R\_in\_list - e_r,\ St.O\_out\_list(e_o),\ St.R\_out\_list + e'_r(tid2)\})$
where $tid2$ is the tid of the process at the outgoing extremity of the rewrite edge e''_r. e'_r is the new rewrite edge.

(45) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2)\},\ \{St.O\_out\_list(e_o(tid2)),\ St.R\_in\_list(e_r(tid1))\})$
$\xrightarrow{O}$
$(\{UPDATER(e_r)[tid1]\},\ \{St.O\_out\_list(e_o),\ St.R\_in\_list - e_r\})$
The same assertions as in (44).

(46) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.tid,\ St.O\_out\_list,\ St.R\_in\_list(e_r(tid1)),\ St.R\_out\_list\})$
$\xrightarrow{O}$
$(\{INO(St.tid)[tid2], INITR(St.tid)[tid2]\},\ \{St.symb,\ St.tid,\ St.O\_out\_list + e_o(tid2, symb2),\ St.R\_in\_list - e_r,\ St.R\_out\_list + e'_r(tid2, symb2)\})$
if $symb2$ is a constant, $St.symb$ is either a constant or a function symbol, $St.symb > symb2$ there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid2$ (the tid of the process at the outgoing extremity of e''_r). e_o is a new orientation edge. e'_r is the new rewrite edge.

(47) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.tid,\ St.R\_in\_list(e_r(tid1))\})$
$\xrightarrow{O}$
$(\{UPDATER(St.tid)[tid2]\},\ \{St.symb,\ St.tid,\ St.R\_in\_list - e_r\})$
The same assertions as in (46).

(48) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.tid,\ St.S\_in\_list(e_s(tid, index)),\ St.R\_in\_list(e_r)\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(tid, tid2), St.tid, index, 2)[tid]\},\ \{St.symb,\ St.tid,\ St.S\_in\_list(s_s),\ St.R\_in\_list - e_r\})$
The same assertions as in (46).

The following four *O-transitions* show the processing of an RUR configuration, in particular when an incoming orientation edge used to orient the corresponding rewrite edge is present or can be easily added.

(49) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, tid3), e_u(tid1, tid2))\},\ \{St.O\_in\_list(e_o(tid3)),$
$St.R\_in\_list(e_r(tid1))\})$
$\xrightarrow{O}$
$(\{INITR(St.tid)[tid3]\},\ \{St.O\_in\_list(e_o),\ St.R\_in\_list + e'_r(tid3)\})$
where $tid3$ is the tid of the process at the outgoing extremity of e''_r. e'_r is the new rewrite edge.

(50) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, tid3), e_u(tid1, tid2)\},\ \{St.O\_in\_list(e_o(tid3)),$
$St.R\_in\_list(e_r(tid1))\})$
$\xrightarrow{O}$
$(\{UPDATER(e''_r(tid3))[tid2], UPDATER(e''_r(tid2))[tid3]\},\ \{St.O\_in\_list(e_o),$
$St.R\_in\_list + e'_r(tid3)\})$
where e'_r is the new rewrite edge and $tid3$ is the tid of the process at the outgoing extremity of e''_r and both messages $UPDATER$ contain the rewrite edge e''_r as outgoing and the rewrite edge e''_r as incoming, respectively.

(51) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, tid3, symb3), e_u(tid1, tid2))\},\ \{St.symb, St.tid, St.O\_in\_list,$
$St.R\_in\_list(e_r)\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(St.tid), 1)[tid3], INITR(St.tid)[tid3]\},\ \{St.symb, St.tid, St.O\_in\_list + e_o(tid3),$
$St.R\_in\_list + e'_r(tid3)\})$
if $St.symb$ is a constant, $symb3$ is either a constant or a function symbol, $symb3 > St.symb$ and there is no orientation edge between the process with tid $St.tid$ and process with tid $tid3$ (the tid of the process at the outgoing extremity of e''_r). e_o is a new orientation edge. e'_r is a new rewrite edge.

(52) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, tid3, symb3), e_u(tid1, tid2))\},\ \{St.symb,\ St.tid,$
$R\_in\_list(e_r(tid1))\})$
$\xrightarrow{O}$
$(\{UPDATER(e''_r(tid3))[tid2], UPDATER(e''_r(tid2))[tid3]\},\ \{St.symb,\ St.tid,\ St.R\_in\_list+$
$e'_r(tid3)\})$
The same assertions as in (51).

The following two *O-transitions* show the processing of an RUR configuration, in particular when a rewrite edge can not be added, because orientation information is not available. In this case, requests are sent to ask for orientation information.

(53) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.tid,\ St.R\_in\_list(e_r)\})$
$\xrightarrow{O}$
$(\{CALCULO(e_o(St.tid, St.symb), list\_sons)[tid2]\},\ \{St.symb, St.tid,\ St.R\_in\_list(e_r)\})$
if $St.symb$ and $symb2$ are function symbols, $St.symb \geq symb2$ and there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid2$ (the tid of the process at the outgoing extremity of e''_r). $list\_sons$ is the list of the children of the process with tid $St.tid$.

(54) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, tid3, symb3), e_u(tid1, tid2))\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s(tid, index)),\ St.R\_in\_list(e_r)\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid, tid3), e'_o(tid3, tid), index)[tid]\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s)),\ St.R\_in\_list(e_r)\})$
if $St.symb$ and $symb3$ are function symbols, $St.symb < symb3$ and there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid3$, tid of the process at the outgoing extremity of e''_r.

The following four *O-transitions* described one part of the Unification calculation. The case, where an outgoing rewrite edge is added because of the processing of an RUR configurations and Unification needs to be calculated, can not be described using *U-transitions*.

(55) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2))\},\ \{St.R\_in\_list(e_r(tid1)),\ St.U\_list(e_u(tid3))\})$
$\xrightarrow{O}$
$(\{NEEDU[tid3]\},\ \{St.R\_in\_list - e_r,\ St.U\_list(e_u)\})$
if $Constraint(e_u) = False$ and the process receiving $CONFIG$ is in charge of the calculation of the constraint of e_u and if $Needu(e_u) = False$ in the initial state of the process receiving $CONFIG$. After the processing of the message $CONFIG$, the value of $Needu(e_u)$ is changed to $True$.

(56) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2))\},\ \{St.R\_in\_list(e_r(tid1)),\ St.U\_list(e_u(tid)),\ St.S\_out\_list(e_s)\})$
$\xrightarrow{O}$
$(\{INFOU(e_u, e_s)[tid]\},\ \{St.R\_in\_list - e_r,\ St.U\_list(e_u),\ St.S\_out\_list(e_s)\})$
if $Constraint(e_u) = False$ and the process receiving $CONFIG$ is not in charge of the calculation of the constraint of e_u and $Needu(e_u) = False$ in the initial state of the process receiving $CONFIG$. After the processing of the message $CONFIG$, the value of $Needu(e_u)$ is changed to $True$.

(57) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.O\_out\_list,\ St.R\_in\_list(e_r),\ St.U\_list\})$
$\xrightarrow{O}$
$(\{NEEDU[tid]\},\ \{St.symb,\ St.O\_out\_list + e_o(tid2, symb2),\ St.R\_in\_list - e_r,\ St.U\_list(e_u)\})$
if the following Unification conditions are true: $Constraint(e_u) = False$, the process receiving $CONFIG$ is in charge of the calculation of the constraint of e_u and $Needu(e_u) = False$ in the initial state of the process. The following Orientation conditions must be true too: $symb2$ is a constant, $St.symb > symb2$ and there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid2$ (the tid of the process at the outgoing extremity of e''_r). e_o is a new orientation edge. After the processing of the message $CONFIG$, the value of $Needu(e_u)$ is changed to $True$.

(58) $(\{CONFIG(\text{RUR}, e_r(tid1), e''_r(tid2, symb2))\},\ \{St.symb,\ St.O\_out\_list,\ St.R\_in\_list(e_r),$
$St.U\_list(e_u(tid)),\ St.S\_out\_list(e_s)\})$
$\xrightarrow{O}$
$(\{INFOU(e_s, e_u)[tid]\},\ \{St.O\_out\_list + e_o(tid2), St.R\_in\_list - e_r, St.U\_list,\ St.S\_out\_list\})$
if the following Unification conditions are true: $Constraint(e_u) = False$, the process receiving $CONFIG$ is not in charge of the calculation of the constraint of e_u and $Needu(e_u) = False$ in the initial state of the process. The following Orientation conditions must be true: $symb2$ is a constant, $St.symb > symb2$ and there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid2$, the tid of the process at the outgoing extremity of e''_r. e_o is a new orientation edge.

O-transitions are similar for the processing of RUR-rhs configurations, and we do not express them.

The following two *O-transitions* illustrate the orientation part of the SUR configuration processing. Orientation requests are made via the new added subterm edge.

(59) $(\{CONFIG(\text{SUR}, e_s(tid, index), e_r(tid1, symb1))\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s(index, tid)),$
$St.calculo\_list(e_o(tid2, symb2))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid, St.symb, tid2, symb2), e'_o(tid1, tid2), index)[tid]\},\ \{St.symb, St.tid,$
$St.S\_out\_list - e_s + e'_s(tid1, symb1),\ St.calculo\_list(e_o)\})$
if $St.symb = symb2$ or $St.symb < symb2$. e'_s is the new subterm edge and o is an edge to calculate.

(60) $(\{CONFIG(\text{SUR}, e_s(index, tid), e_r(tid1, symb1))\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s(index, tid)),$
$St.calculo\_list(e_o(tid2, symb2), index, tidsontid2, symbsontid2)\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o(St.tid, St.symb, tid2, symb2), e'_o(tid1, tidsontid2, symbsontid2), index)[tid]\},$
$St.symb,\ St.tid,\ St.S\_out\_list - e_s + e'_s(tid1),\ St.calculo\_list(e_o, index, tidsontid2, symbsontid2)\})$
if $St.symb = symb2$. e'_s is the new subterm edge and e_o is an edge to calculate. $tidsontid2$ and $symbsontid2$ are the tid and the symbol of a child of process with tid $tid2$.

The following two *O-transitions* indicate how the addition of an outgoing subterm edge causes orientation requests to be sent via this edge with respect to all orientation edges to calculate saved in *calculo_list*.

(61) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.calculo\_list(calculo(e_o(St.tid, St.symb, tid1, symb1))),\ St.S\_out\_list\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid, St.symb, tid1, symb1), e'_o(tid, tidsontid1, symbsontid1), index)[tid]\},$
$\{St.symb,\ St.tid,\ St.calculo\_list(calculo),\ St.S\_out\_list + e_s\})$
where e_o is an orientation edge between the process with tid *St.tid* and the process with tid $tid1$.

(62) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.calculo\_list(calculo(e_o(St.tid, St.symb, tid1, symb1), index,\ tidsontid1, symbsontid1)),\ St.S\_out\_list\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o(St.tid, St.symb, tid1, symb1), e'_o(tid, tidsontid1, symbsontid1), index)[tid]\},$
$\{St.symb, St.tid,\ St.calculo\_list(calculo),\ St.S\_out\_list + e_s\})$
where e_o is an orientation edge between the process with tid *St.tid* and the process with tid $tid1$. $tidsontid1$ is the tid of the $index^{th}$ child process of $tid1$ with symbol $symbsontid1$.

The following five *O-transitions* indicate how the addition of an incoming subterm edge causes up-propagation of orientation answers for all the saved requests having an answer.

(63) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.requesto1\_list(requesto1(e_o(St.tid, tid1), e'_o(tid, tid1),\ index)),\ St.O\_in\_list(e_o),\ St.S\_in\_list\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o, e'_o, index)[tid]\},\ \{St.symb,\ St.tid,\ St.requesto1\_list(requesto1),$
$St.O\_in\_list(e_o),\ St.S\_in\_list + e_s\})$

(64) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.requesto1\_list(requesto1(e_o(St.tid, tid1, symb1), e'_o(tid, tid1), index)),\ St.O\_in\_list(e_o),\ St.S\_in\_list\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o, e'_o, index)[tid]\},\ \{St.symb,\ St.tid,\ St.requesto1\_list(requesto1),$
$St.O\_in\_list(e_o),\ St.S\_in\_list + e_s\})$
if $symb1 \geq St.symb$ and $St.symb$ is a constant symbol.

(65) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.requesto2\_list(requesto2(e_o(St.tid,$
$tidsontid1), e'_o(tid, tid1), index),\ St.O\_in\_list(e_o),\ St.S\_in\_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e'_o, index, >)[tid]\},\ \{St.symb,\ St.tid,\ St.O\_in\_list(e_o),$
$St.requesto2\_list(requesto2),\ St.S\_out\_list + s\})$
where $tidsontid1$ is the tid of the $index^{th}$ child process of $tid1$

(66) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.requesto2\_list(requesto2(e_o(St.tid, tidsontid1,$
$symbsontid1), e'_o(tid, symb, tid1), index),\ St.O\_in\_list(e_o),\ St.S\_in\_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e'_o, index, >)[tid]\},\ \{St.symb,\ St.tid,\ St.O\_in\_list(e_o),$
$St.requesto2\_list(requesto2),\ St.S\_out\_list + s\})$
where $tidsontid1$ is the tid of the $index^{th}$ child process of $tid1$ with symbol
$symbsontid1$ and $symbsontid1 > St.symb$ and $St.symb$ is a constant symbol and $symb = symb1$.

(67) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.requesto2\_list(requesto2(e_o(St.tid, tidsontid1,$
$symbsontid1), e'_o(tid, symb, tid1), index),\ St.O\_eq\_list(e_o),\ St.S\_in\_list\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e'_o, index, =)[tid]\},\ \{St.symb,\ St.tid,\ St.O\_eq\_list(e_o),$
$St.requesto2\_list(requesto2),\ St.S\_out\_list + s\})$
where $tidsontid1$ is the tid of the $index^{th}$ child process of $tid1$ with symbol $symbsontid1$
and $symbsontid1 > St.symb$ and $St.symb$ is a constant symbol and $symb = symb1$.

(68) $(\{INITS(e_s(tid, index))\},\ \{St.symb,\ St.tid,\ St.O\_out\_list(e_o(St.tid, tid))$
$\xrightarrow{O}$
$(\{OUTO(e_o, e'_o(tid1, tid), index)[tid]\},\ \{St.symb,\ St.tid,\ St.O\_out\_list(e_o)\})$

The following three *O-transitions* show the processing of an *INITU*. These *INITU* messages implies the updating of *O_eq_list* and the up-propagation of Orientation answers.

(69) $(\{INITU(e_u(tid))\},\ \{St.O\_eq\_list\})$
$\xrightarrow{O}$
$(\{\},\{St.O\_eq\_list + e_o(tid)\})$
if $Constraint(e_u) = True$

(70) $(\{INITU(e_u(tid))\},\ \{St.S\_in\_list(e_s(tid1, index))\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(tid1, tid), St.tid, index, 2)[tid]\}, \{St.S\_in\_list(e_s)\})$
if $Constraint(e_u) = True$

(71) $(\{INITU(e_u(tid))\},\ \{St.tid,\ St.S\_in\_list(e_s(tid2, index)),$
$St.requesto1\_list(requesto1(tid1, tid2, symb1, symb2, tid, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o(St.tid, tidsontid1, >), e'_o(tid2, tid1), index, =)[tid2]\}, \{St.tid,$
$St.S\_in\_list(e_s),\ St.requesto1\_list(requesto1)\})$
where the process with tid $tidsontid1$ is the $index^{th}$ child of the process with tid $tid1$.

The following five *O-transitions* show the processing of a $CALCULO$ or $CALCULO1$ message. This messages causes requests to be sent.

(72) $(\{CALCULO(calculo(e_o(tid1, symb1)))\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s(tid, index))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid, tid1), e'_o(tid, tid1), index)[tid]\},\ \{St.symb, St.tid,$
$St.S\_out\_list(e_s)\})$

(73) $(\{CALCULO(calculo(e_o(tid1, symb1)))\}, \{St.symb,\ St.tid\})$
$\xrightarrow{O}$
$(\{CALCULO1(calculo1(e_o(St.tid, St, symb)))[tid1]\}, \{St.symb,\ St.tid\})$
if $St.symb = symb1$

(74) $(\{CALCULO(calculo(e_o(St.tid, tid1, symb1), e_s(tid1, tidsontid1, symbsontid1)))\},\ \{St.symb,$
$St.tid,\ St.S\_out\_list(e_s(tid, index))\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o, e'_o(tid, tidsontid1), index)[tid]\},\ \{St.symb, St.tid, St.S\_out\_list(e_s)\})$
where $tidsontid1$ is the tid of the $index^{th}$ child of the process with tid $tid1$ and $symbsontid1$ is its symbol.

(75) $(\{CALCULO1(calculo(e_o(tid1, symb1)))\},\ \{St.symb,\ St.tid,\ St.S\_out\_list(e_s(tid, index))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e_o(St.tid, tid1), e'_o(tid, tid1), index)[tid]\},\ \{St.symb, St.tid, St.S\_out\_list(e_s)\})$

(76) $(\{CALCULO(calculo(e_o(St.tid, tid1, symb1), e_s(tid1, tidsontid1, symbsontid1)))\},\ \{St.symb,$
$St.tid,\ St.S\_out\_list(e_s(tid, index))\})$
$\xrightarrow{O}$
$(\{REQUESTO2(e_o, e'_o(tid, tidsontid1), index)[tid]\},\ \{St.symb, St.tid, St.S\_out\_list(e_s)\})$
where $tidsontid1$ is the tid of the $index^{th}$ child of the process with tid $tid1$ and $symbsontid1$ is its symbol.

The following five *O-transitions* describe the processing of an $OUTO$ message depending on the flag contained in the message. In particular, we can see which test is made for each flag and the up-propagation of information if an outgoing orientation edge is added.

(77) $(\{OUTO(e_o(tid, symb), listsons, 3)\},\ \{St.symb,\ St.O\_out\_list,\ St.S\_in\_list(e_s(tid1, index))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(tid1, tid), St.tid, index, 2)\}, \{St.symb,\ St.O\_out\_list + e_o,\ St.S\_in\_list(e_s)\})$
if there exists an outgoing subterm edge from the process receiving the $OUTO$ message to all the processes of $list\_sons$ and if the number of elements of $list\_sons$ is equal to the arity of $St.symb$.

(78) $(\{OUTO(e_o(St.tid, tid), tid1, index, 2))\},\ \{St.S\_out\_list(e_s(tid1, index)),$
$St.O\_out\_list,\ St.S\_in\_list(e'_s(tid2, index2))\})$
$\xrightarrow{O}$
$(\{INO(St.tid)[tid], OUTO(e_o(tid2, tid), St.tid, index, 2)[tid2]\}, \{St.S\_out\_list(e_s),\ St.O\_out\_list,$
$St.S\_in\_list(e'_s)\})$

(79) $(\{OUTO(e_o(tid), 1)\},\ \{St.O\_out\_list,\ St.S\_in\_list(e_s(tid1, index))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(tid1, tid), St.tid, index, 2)\}, \{St.O\_out\_list + e_o, St.S\_in\_list(e_s)\})$

The following three *O-transitions* describe the processing of an INO message. In particular, we can see the up-propagation of answers when an incoming orientation edge is added.

(80) $(\{INO(e_o(tid1))\},\ \{St.O\_in\_list\})$
$\xrightarrow{O}$
$(\{\}, \{St.O\_in\_list + e_o(tid1))\})$

(81) $(\{INO(e_o(tid1))\},\ \{St.tid,\ St.requesto1\_list(requesto1(e_o(tid1, tid2, symb1, symb2),$
$o'(St.tid, tid1, St.symb, St.tid), index)),\ St.S\_in\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o, e'_o), index)[tid2]\}, \{St.tid,$
$St.requesto1\_list(requesto1),\ St.S\_in\_list(e_s)\})$
where the orientation edge e'_o is used to calculate the orientation edge between the process with tid $tid1$ and the process with tid $tid2$.

(82) $(\{INO(e_o(tid1))\},\ \{St.tid,\ St.requesto2\_list(requesto2(e_o(tid1, tid2, symb1, symb2)), e'_o(St.tid,$ $tidsontid1,\ St.symb, symbsontid1), index),\ St.S\_in\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e'_o, index, >)[tid2]\}, \{St.tid,$
$St.requesto2\_list(requesto2),\ St.S\_in\_list(e_s)\})$
where the orientation edge e'_o is used to calculate the orientation edge between the process with tid $tid1$ and the process with tid $tid2$.

The following four *O-transitions* describe the processing of a *REQUESTO1* message. If the process receiving the message can answer i.e. there exists an orientation edge or this orientation edge can be easily added, answers are propagated up.

(83) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1),$ $index))\}, \{St.tid,\ St.symb,\ St.O\_out\_list(e'_o),\ St.S\_in\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{OUTO(e_o, St.tid, index, 2)[tid2]\},\ \{St.tid,\ St.symb,\ St.O\_out\_list(e'_o),\ St.S\_in\_list(e_s)\})$

(84) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1),$ $index))\}, \{St.tid,\ St.symb,\ St.O\_in\_list(e'_o),\ St.S\_in\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(answero1(e_o, e'_o, index))[tid2]\},\ \{St.tid,\ St.symb,\ St.O\_in\_list(e'_o),$ $St.S\_in\_list(e_s)\})$

(85) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1),$ $index))\}, \{St.tid,\ St.symb,\ St.O\_in\_list,\ St.S\_in\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(answero1(e_o, e'_o, index))[tid2]\},\ \{St.tid,\ St.O\_in\_list + e'_o,$
$St.S\_in\_list(e_s)\})$
if there is no orientation edge between the process with tid $tid1$ and the process with tid $St.tid$ and $St.symb$ is a constant and $symb1$ is a function symbol or a constant and $symb1 > St.symb$.

(86) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1),$ $index))\}, \{St.tid,\ St.symb,\ St.O\_out\_list,\ St.S\_in\_list(e_s(tid2, index)),\ \})$
$\xrightarrow{O}$
$(\{OUTO(e_o, St.tid, index, 2)[tid2]\},\ \{St.tid,\ St.symb,\ St.O\_out\_list + e'_o,$
$St.S\_in\_list(e_s)\})$
if there is no orientation edge between the process with tid $St.tid$ and the process with tid $tid1$ and $symb1$ is a constant and $St.symb$ is a function symbol or a constant and $St.symb \geq symb1$.

The following two *O-transitions* express the case where, the process receiving the message can not answer. New orientations requests are then sent to make the information available.

(87) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1), index))\}, \{St.tid,\ St.symb,\ St.S\_in\_list(e'_s(tid2, index)),\ St.S\_out\_list(e_s(tid3, index3))\})$
$\xrightarrow{O}$
$(\{REQUESTO1(e'_o, e''_o(tid1, tid3, symb1, symb3), index3)[tid3]\},\ \{St.tid,\ St.symb,\ St.S\_in\_list(e'_s),\ St.S\_out\_list(e_s)\})$
if there is no orientation edge between the process with tid $tid1$ and the process with tid $St.tid$ and $St.symb$ and $symb1$ are function symbols and $St.symb < symb1$.

(88) $(\{REQUESTO1(requesto1(e_o(tid1, tid2, symb1, symb2), e'_o(St.tid, tid1, St.symb, symb1), index))\}, \{St.tid,\ St.symb,\ St.S\_in\_list(e'_s(tid2, index)),\ St.S\_out\_list(e_s(tid3, index3))\})$
$\xrightarrow{O}$
$(\{CALCULO(calculo(e'_o, e_s)[tid1]\},\ \{St.tid,\ St.symb,\ St.S\_in\_list(e'_s),\ St.S\_out\_list(e_s)\})$
if there is no orientation edge between the process with tid $tid1$ and the process with tid $St.tid$ and $St.symb$ and $symb1$ are function symbols and $St.symb \geq symb1$.

O-transitions are similar for a *REQUESTO2* messages and we do not express them.

The following three *O-transitions* shows the processing of an *ANSWERO1* message. In particular, this *ANSWERO1* message permits the process receiving it to orient an orientation edge between two processes having different symbols such that its direction is incoming.

(89) $(\{ANSWERO1(answero1(e_o(tid1, St.tid, symb1, St.symb), e'_o(tid2, tid1, symb1, symb2), index))\}, \{St.tid,\ St.symb,\ St.S\_out\_list(e_s(tid2, index)),\ St.O\_in\_list)\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(St.tid), 1)[tid]\},\ \{St.tid,\ St.symb,\ St.S\_out\_list(e_s),\ St.O\_in\_list + e_o\})$
if there is no orientation edge between the process with tid $tid1$ and the process with tid $St.tid$ if and the orientation edge e'_o to calculate is between the process with $tid1$ and the process with $St.tid$ and the number of messages *ANSWERO1* received for calculating the orientation of e_o is equal to the arity of the symbol $St.symb$.

(90) $(\{ANSWERO1(answero1(e_o(tid1, St.tid, symb1, St.symb), e'_o(tid2, tid1, symb1, symb2), index)), \{St.tid, St.symb,\ St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(tid3, index3)),\ St.requesto1\_list(requesto1(e_o,\ e''_o(tid1, tid3, symb1, symb3), index3))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o, e''_o, index3)[tid3]\}, \{St.S\_out\_list(e_s),\ S\_in\_list(e'_s),\ St.requesto1\_list(requesto1)\})$
The same conditions as in (89).

(91) $(\{ANSWERO1(answero1(e_o(tid1, St.tid, symb1, St.symb), e'_o(tid2, tid1, symb1, symb2), index)), \{St.tid,\ St.symb,\ St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e_s(tid3, index3)), St.requesto2\_list(requesto2(e_o, e''_o(tid4, tid3, symb4, symb3), index3))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e''_o, index3, >)[tid2']\}, \{St.S\_out\_list(e_s),\ S\_in\_list(e'_s), St.requesto2\_list(requesto2)\})$
The same conditions as in (89). In addition, the process with tid $tid1$ is the $index3^{th}$ child of the process with tid $tid4$.

The following six *O-transitions* deal with the processing of an *ANSWERO1* or *ANSWERO2* message, in particular, when these messages permit the process receiving them to orient an orientation edge between two processes having the same symbol such that its direction is incoming.

(92) $(\{ANSWERO2(answero2(e_o(St.tid, tid1), e'_o(tid2, tidsontid1), index, >))\},\ \{St.tid,\ St.symb, St.S\_out\_list(e_s(tid2, index)),\ St.O\_in\_list\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(St.tid), 1)[tid1]\}, \{St.tid,\ St.symb,\ St.S\_out\_list(e_s),\ St.O\_in\_list + e_o\})$
if $symb1 = St.symb$ and there is no orientation edge e_o between the process with tid $tid1$ p_1 and the process with tid $St.tid$ and the number of *ANSWERO1* received messages is equal to the arity of the symbol $St.symb$ and if the *ANSWERO2* received messages from the first k children of process with tid $St.tid$ contain= and the *ANSWERO2* received messages from its $k+1^{th}$ children contain +, where $k \in \{0..n-1\}$.The process with tid $tidsontid1$ is the $index^{th}$ child of the process with tid $tid1$.

(93) $(\{ANSWERO2(answero2(e_o(St.tid, tid1), e'_o(tid2, tidsontid1), index, >))\},\ \{St.tid,\ St.symb, St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(tid3, index3)),\ St.requesto1\_list(requesto1(e_o, e''_o(tid3, tid1),\ index3))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e_o, e''_o, index3)[tid3]\}, \{St.tid,\ St.symb,\ St.S\_out\_list(e_s),\ St.S\_in\_list(e'_s), St.requesto1\_list(requesto1),\ St.S\_in\_list(e_s)\})$
The same conditions as in (92).

(94) $(\{ANSWERO2(answero2(e_o(St.tid, tid1), e'_o(tid2, tidsontid1), index, >))\},\ \{St.tid,\ St.symb, St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(tid3, index3)),\ St.requesto2\_list(requesto2(e_o, e''_o(tid3, tid5), index3))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e_o, e''_o, index3, >)[tid3]\}, \{St.tid,\ St.symb,\ St.S\_out\_list(e_s),\ St.S\_in\_list(e'_s), St.requesto2\_list(requesto2),\ St.S\_out\_list(e_s)\})$
The same conditions as in (92). In addition, the process with tid $tid1$ is the $index3^{th}$ child of the process with tid $tid5$.

(95) $(\{ANSWERO1(answero1(e_o(St.tid, tid1, symb1), e'_o(tid1, tid2, symb2), index))\},\ \{St.tid,\ St.symb,\ St.S\_out\_list(e_s(tid2, index)),\ St.O\_in\_list\})$
$\xrightarrow{O}$
$(\{OUTO(e_o(St.tid), 1)[tid1]\}, \{St.tid,\ St.symb,\ St.S\_out\_list(e_s),\ St.O\_in\_list + e_o\})$
The same conditions as in (92).

(96) $(\{ANSWERO1(answero1(e_o(St.tid, tid1, symb1), e'_o(tid1, tid2, symb2), index))\},$
$\{St.tid,\ St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(tid3, index3)),$
$St.requesto1\_list(requesto1(e'_o, e''_o(tid1, tid3))),\ St.S\_out\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO1(e'_o, e''_o(tid3, tid1), index3)[tid3]\}, \{St.requesto1\_list(requesto1),$
$St.S\_out\_list(e_s),\ S\_in\_list(e'_s)\})$
The same conditions as in (92).

(97) $(\{ANSWERO1(answero1(e_o(St.tid, tid1, symb1), e'_o(tid1, tid2, symb2), index))\},$
$\{St.tid,\ St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(tid3, index3)),$
$St.requesto2\_list(requesto2(e'_o, e''_o(tid3, tid4))),\ St.S\_out\_list(e_s(tid2, index))\})$
$\xrightarrow{O}$
$(\{ANSWERO2(e'_o, e''_o(tid3, tid4), index3, >)[tid3]\}, \{St.requesto2\_list(requesto2),$
$St.S\_out\_list(e_s),\ S\_in\_list(e'_s)\})$
The same conditions as in (92). In addition, the process with tid $tid1$ is the $index3^{th}$ child process of $tid4$.

The two last *O-transitions* concern the processing of an *ANSWERO1* or *ANSWERO2* message. In particular, these messages tell the process receiving them to orient an orientation edge between two processes having the same symbol such that its direction is "equal".

(98) $(\{ANSWERO2(answero2(e_o(tid2, tid3), e'_o(St.tid, tid1, St.symb, symb1), index, =))\},$
$\{St.tid,\ St.S\_out\_list(e_s(tid2, index)),\ St.O\_eq\_list\})$
$\xrightarrow{O}$
$(\{\}, \{St.tid,\ St.S\_out\_list(e_s),\ St.O\_eq\_list + e'_o\})$
if $symb1 = St.symb$ and the process with tid $St.tid$ has received n messages *ANSWERO2* containing $=$ from its n children and n is the arity of $St.symb$.

(99) $(\{ANSWERO2(answero2(e_o(tid2, tid3), e'_o(St.tid, tid1, St.symb, symb1), index, =))\},$
$\{St.tid,\ St.S\_out\_list(e_s(tid2, index)),\ St.S\_in\_list(e'_s(index', tid'))\})$
$\xrightarrow{O}$
$(\{OUTO(e'_o(St.tid, tid1), St.tid, index, 2)[tid']\}, \{St.tid,\ St.S\_out\_list(e_s),\ St.S\_in\_list(e'_s)\})$
The same conditions as in (98).

The completeness results of this set of rules for our concurrent Orientation calculation are given in section 3.6.

3.4 Detection of the termination of the program

It is difficult to detect termination of an asynchronous concurrent program, because a process cannot detect that it will not receive any more messages. We give a termination detection algorithm different from the classical one [DFvG83]. Indeed, we do not need any broadcast and the termination is evaluated by a mailbox receiving messages.

The general definition of the termination of a concurrent program is given in definition 2.

Definition 2 *The program terminates when all processes are* idle *and all sent messages have been received.*

Our termination detection algorithm uses a mailbox, which can be handled by the *root process*. It is based on the following principles.

Sent and received messages are notified to the mailbox using a message called *NOTIFY*. In practice, a message can be specified in the mailbox as received and not yet as sent. The mailbox contains envelopes of sent or received messages. An envelope is composed of a flag *SENT* or *RECEIVED* for a sent or received message *mesg*, the source process of *mesg*, the destination process of *mesg* and an identification number, permitting us to distinguish two messages having the same origin, the same destination and the same contents. So, an envelope is one of the following forms: $(p_i, p_j, NB, SENT)$ for message NB sent by process p_i to p_j, and $(p_j, p_i, NB, RECEIVED)$ for message NB received by process p_i from p_j. A mailbox consists of two lists: *list_sent*, which contains envelopes of messages specified as sent but not received and *list_received*, which contains envelopes of messages specified as received but not sent. So we can consider the state of the mailbox or of the *root process* as the record of these two lists.

Processes cycle from idle to busy to idle. The busy state is entered when a message is received. Other messages are sent as a result of this message. In the busy state, other messages may be received and messages are sent as a result of them. When the process re-enters the idle state, it sends the mailbox one *NOTIFY* message for each message received in that cycle. The *NOTIFY* message contains one envelope for the received message, and an

envelope for each message sent as a result of this received message. We call this statement *statement 1*. This fact is crucial for the soundness of our termination detection algorithm.

The main question is now to detect termination. The condition of termination is that *list_sent* and *list_received* are empty.

When the mailbox receives a message $NOTIFY$ containing an envelope $(p_i, p_j, NB, SENT)$, the mate of this envelope $(p_j, p_i, NB, RECEIVED)$ is searched for in the list *list_received*. If the envelope $(p_j, p_i, NB, RECEIVED)$ does not belong to this list, $(p_i, p_j, NB, SENT)$ is added to the list *list_sent*, otherwise $(p_j, p_i, NB, RECEIVED)$ is deleted from *list_received*.

When the mailbox receives a message $NOTIFY$ containing an envelope $(p_i, p_j, NB, RECEIVED)$, its mate envelope $(p_j, p_i, NB, SENT)$ is removed from *list_sent*, if it belongs to this list otherwise $(p_i, p_j, NB, RECEIVED)$ is added to the list *list_received*.

Initially, *list_sent* contains all the envelopes of SENT messages of the *Initialization phase*. This ensures the soundness of the algorithm.

Definition 3 *A t-transition is an α-transition for the calculation of termination concerning the mailbox.*

The termination algorithm can be expressed using the following *t-transition*.

(100) $(\{NOTIFY((p_i, p, NB, RECEIVED), (p, p_j, NB', SENT), (p, p_k, NB'', SENT), \ldots)\},$
$\{St.list\_received((p, p_j, NB', RECEIVED)),\ St.list\_sent((p_i, p, NB, SENT))\})$
$\xrightarrow{t}$
$(\{\}, \{St.list\_received - (p, p_j, NB', RECEIVED),\ St.list\_sent + (p, p_k, NB'', SENT) -$
$(p, p_j, NB', SENT) - \ldots\})$
where St is the state of the mailbox p which receives the message $NOTIFY$ and p_i, p_j and p_k are processes.

We introduce lemma 1 that will permit us to prove the soundness of the termination.

Message passing between processes can be viewed as a tree such that each node is a son process and each edge represents the passing of a message between two processes. Edges are labelled with the message. The root of the constructed tree is a particular process: the *main process*, and it appears only once in the tree. If the program terminates, this tree is finite else it is infinite. On this tree, we define the measure μ. For a message *mesg* of the tree, $\mu(mesg)$ is the depth of the node, which represents the process P, which receives *mesg*. For example, for an initialization message *mesginit*, $\mu(mesginit) = 1$.

Lemma 1 *For all n, if* list_sent *and* list_received *contain only messages mesg such that $\mu(mesg) \geq n$, then all messages mesg such that $\mu(mesg) < n$ have already been processed by the mailbox.*

Proof:

To prove this lemma, we proceed by induction on n.

$n = 2$ **:**

If *list_sent* and *list_received* contain only messages *mesg* such that $\mu(mesg) \geq 2$, then *list_sent* and *list_received* have already contained all messages of initialization *mesginit* such that $\mu(mesginit) = 1$, because initially, *list_sent* and *list_received* contains all the envelopes of sent messages of the *Initialization phase*. So, all messages of initialization have been specified as sent and received in the mailbox. Also, because of *statement 1*, all messages *mesg* such that $\mu(mesg) = 2$ have been notified to the mailbox as sent. So, messages with μ-measure less than 2 have already been processed by the mailbox.

$n > 2$ **:**

The induction hypothesis is that: if *list_sent* and *list_received* contain only messages *mesg* such that $\mu(mesg) \geq n$, then all messages *mesg* such that $\mu(mesg) < n$ have already been processed by the mailbox.

We assume that if *list_sent* and *list_received* contain only messages *mesg* such that $\mu(mesg) \geq n+1$, and we will prove that all messages *mesg* such that $\mu(mesg) < n+1$ have already been processed by the mailbox.

If *list_sent* and *list_received* contain only messages *mesg* such that $\mu(mesg) \geq n+1$, then by the induction hypothesis, all messages *mesg* such that $\mu(mesg) < n$ have already been processed by the mailbox. network.

Furthermore, if *list_sent* and *list_received* contain only messages *mesg* such that $\mu(mesg) \geq n+1$, then *list_sent* and *list_received* have already processed all messages *mesg* such that $\mu(mesg) = n$. So all messages *mesg* sent such that $\mu(mesg) = n$ were in the mailbox as sent and received and because of *statement 1*, all sent messages with a μ-measure equal to $n+1$ have appeared in the mailbox. Now, all messages *mesg* such that $\mu(mesg) < n+1$ have already been processed by the mailbox. □

Theorem 1 *The program is terminated iff* list_sent *and* list_received *are empty.*

Proof:

$\boxed{\Rightarrow}$ It is trivial that if the program is terminated, then *list_sent* and *list_received* are empty, because all messages which have been sent, have been received.

$\boxed{\Leftarrow}$ The reverse is not trivial.

We proceed by contradiction. Suppose that *list_sent* and *list_received* are empty but program does not terminate. Then, we consider two cases:

- There is a message *mesg* in the network such that $\mu(mesg) = n$. By considering lemma 1, all messages such that $\mu(mesg) = n$ have already been processed. We meet then a contradiction.
- There is a busy process. At the end of the busy state of the process, the process sent a *NOTIFY* message to the mailbox. Suppose that this notification message contains a message *mesg* such that $\mu(mesg) = n$. By considering lemma 1, all messages such that $\mu(mesg) = n$ have already been processed. We meet then a contradiction.

Contradictions prove that if *list_sent* and *list_received* are empty, the program terminates.

□

3.5 Time Stamps

3.5.1 Why are time stamps needed?

In our approach, when an inference is performed, it is not necessary that other processes know about this inference immediately. We are allowed to use old information. That is why a concurrent algorithm is possible. However, we use time stamps to insure that an inference with a term is not performed using new information followed by another inference using an old version of that term. Time stamps prevent this from happening. Since operations like Unification and Orientation are distributed among subterms, we need the time stamps to compare when operations are performed.

Consider the following example.

Example 1 *Let* $E = \{g(f(a)) \approx b, f(a) \approx f(b), a \approx c\}$. *be the set of equations to complete. Consider the precedence* $g \prec f \prec c \prec b \prec a$. *The set of equations* E *can be represented by the* SOUR graph *of figure 7.*

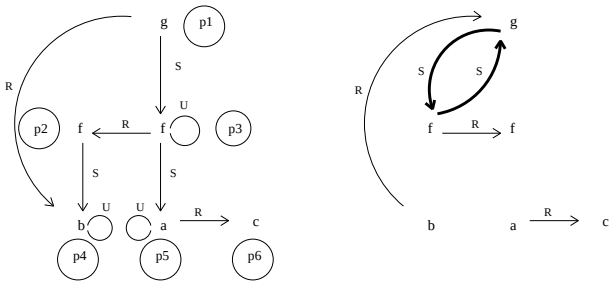

Figure 7: The SOUR Graph representing E and the result of the execution plan

Consider the following execution plan:

- *Process* p_3 *detects an SUR configuration* $conf1$ *and sends it to process* p_1.
- *Process* p_5 *detects an SUR configuration* $conf2$ *and sends it to process* p_3.
- *Process* p_3 *receives the SUR configuration* $conf2$ *and processes it.*
- *Process* p_3 *changes the direction of its rewrite edge because* $f(b) \succ f(c)$, *after a sequence of messages.*
- *Process* p_1 *receives a message from* p_3 *to tell it to change the direction of its rewrite edge because* $b \succ g(f(c))$.

- *Process p_4 has got an outgoing rewrite edge and detects an SUR configuration $conf3$ that it sends to p_2.*
- *Process p_1 receives and processes the SUR configuration $conf1$ so adds a subterm edge from process p_1 to process p_2.*
- *Process p_2 receives and processes the SUR configuration $conf3$ so adds a subterm edge from p_2 to process p_1.*
- *On the graph, we notice a cycle of subterm edges.*

To prevent this critical case, we use *Time Stamps*.

3.5.2 Time stamps definition and usage

Definition 4 *The time stamp of a process is a counter initialized to 0. We denote the current time stamp of a process p by TS_p.*

Each process p contains a time stamp TS_p and a table of time stamps that we denote TS_Table_p. The table has n entries, where n is the arity of the symbol of p. For process p', $TS\_Table_p[p']$ is the last time stamp $TS_{p'}$ of process p' known by p, initially 0. When p sends a message to one of its parent processes, it sends its current time stamp TS_p in the message.

Unification and orientation edges now contain a new kind of information.A unification edge between processes p and p' has two time stamps associated with it: the time stamps of the processes at the ends of the edge. These time stamps are stored in p and p'. The same thing is done for orientation edges.

Use of time stamps for the creation of configurations: Consider a semi-configuration containing a rewrite edge r from process p_3 to process p_4 and a unification edge between processes p_2 and p_3.

- This semi configuration is sent to p_2 only if the calculation of the direction of the rewrite edge r gives that r is from p_3 to p_4 and the calculation of Unification for the unification edge u between p_2 and p_3 gives a true unification constraint and the time stamp of p_3 used for calculating Unification is the same as the time stamp of p_3 used for calculating Orientation. The message $SEMICONF$ is sent to p_2 containing the time stamp of p_2 used to calculate Unification. Let t be this time stamp.
- When p_2 receives the message $SEMICONF$, it sends to process p_1 a message $CONFIG$ only if $TS_{p_2} = t$.
- If process p_1 receives a message $CONFIG$ containing an SUR configuration, it processes it only if the subterm edge in the message still exists and $TS\_Table_{p_1}[p_2] = t$ or $TS\_Table_{p_1}[p_2] < t$. Then, if $TS\_Table_{p_1}[p_2] < t$, TS_{p_1} is incremented and $TS\_Table_{p_1}$ is updated so that $TS\_Table_{p_1}[p_2] = t$.

- When process p_1 receives a message *CONFIG* containing an *RUR* or an *RUR-rhs* configuration, it processes it only if the incoming or the outgoing rewrite edge respectively still exists.

Use of time stamps for Unification : Suppose that we want to calculate the Unification constraint of unification edge u between process p_0 and process p_1, such that p_1 decides the calculation.

Suppose that process p_2 (k^{th} child process of p_1) receives a request *REQUESTU* asking if there is a unification edge with constraint true between processes p_2 and p_3 (k^{th} child process of p_0). Answers *ANSWERU* will be sent to p_1 (for conditions of this sending see section 3.2). Answer messages contain t_2 and t_3 the last time stamps of p_2 and p_3 used in the calculation of the unification constraint between p_2 and p_3.

- When p_1 receives an answer *ANSWERU* from p_2 containing time stamps t_2 and t_3, the answer message is processed only if the following conditions are respected:

  — There is a subterm edge going from p_1 to p_2 with index k.

  — $TS\_Table_{p_1}[p_2] = t_2$ or $TS\_Table_{p_1}[p_2] < t_2$. For this second case, $TS\_Table_{p_1}$ is updated, such that $TS\_Table_{p_1}[p_2] = t_2$ and the current time stamp of p_1 is incremented.

- If an answer message permits us to determine that the unification constraint of the unification edge u between p_0 and p_1 is true i.e. n answer messages have been received by p_1 where n is the arity of the symbol of p_1, then the message *INITU* corresponding to the fact that the unification constraint of u is true, is sent to p_0 containing t_3.

- When p_0 receives from p_1 a message *INITU* containing the time stamp t_3, this message is processed only if the following conditions are respected:

  — There is a subterm edge going from p_0 to p_3 with index k.

  — $TS\_Table_{p_0}[p_3] = t_3$ or $TS\_Table_{p_0}[p_3] < t_3$. For this second case, $TS\_Table_{p_0}$ is updated, such that $TS\_Table_{p_0}[p_3] = t_3$ and the current time stamp of p_0 is incremented.

Use of time stamps for Orientation: Use of time stamps for Orientation is similar to the use of time stamps for Unification.

Example 2

Consider the preceding example and the preceding execution plan. We now use time stamps and we obtain the following results.

- *Process p_3 detects an SUR configuration conf1 and sends it to process p_1. The corresponding message* CONFIG *contains the time stamp $TS_{p_3}(= 0)$. Let $t_3 = TS_{p_3}$.*

- *Process p_5 detects an SUR configuration conf2 and sends it to process p_3. The corresponding message* CONFIG *contains the time stamp $TS_{p_5}(=0)$. Let $t_5 = TS_{p_5}$.*
- *Process p_3 receives the SUR configuration conf2 and processes it, because $TS\_Table_{p_3}[p_5] = t_5(=0)$ (Process p_5 will never change its time stamp). A configuration is processed, so p_3 increments its time stamp, $TS_{p_3} = TS_{p_3} + 1(=1)$.*
- *Process p_3 changes the direction of its rewrite edge because $f(b) \succ f(c)$, using a message INO of p_1 with a time stamp equal to 0.*
- *Process p_1 receives a message from p_3 to tell it to change the direction of its rewrite edge because $b \succ g(f(c))$. This message contains the last time stamp of p_3 equal to 1. We have $TS\_Table_{p_1}[p_3](=0) < 1$, so the message is processed, $TS\_Table_{p_1}[p_3] = 1$, TS_{p_1} is incremented by one.*
- *Process p_4 has got an outgoing rewrite edge and detects an SUR configuration conf3 that it sends to p_2. This configuration is sent with time stamp $TS_{p_4}(=0)$.*
- *Process p_1 receives the SUR configuration conf1 containing the time stamp $t_3(=0)$. This configuration is forgotten because it is an old information $TS\_Table_{p_1}[p_3](=1) > t_3(=0)$.*
- *Process p_2 receives and processes the SUR configuration conf3 so adds a subterm edge from p_2 to process p_1. There is no more cycle of subterm edges.*

3.5.3 Semantics of a time stamped ground SOUR Graph

Definition 5 *We define the semantics of the term representing process p by $T(p, TS_p)$, where TS_p is the current time stamp of process p.*

We have: $T(p, TS_p) = f(T(p_1, t_1), \ldots, T(p_n, t_n))$ where:

- *f is the symbol of process p.*
- *p_i are processes such that there exists a subterm edge going from p to p_i, when the time stamp of p is TS_p with index i.*
- *$t_i = TS\_Table_p[p_i]$, the last time stamp of p_i known by p, $\forall i \in \{1, \ldots, n\}$.*

Remark 1 *If at a given time, process p has got an outgoing subterm edge to p_i such that the time stamp of p is TS_p and $TS\_Table_p[p_i] = t_i$ and the time stamp of p_i is TS_{p_i}, then $t_i \leq TS_{p_i}$, because t_i is the last time stamp of p_i known by p.*

Using time stamps, we have the following results.

Lemma 2 *For all time stamps t and t' such that $t' < t$, and for all processes p, we have: $T(p, t') \succeq T(p, t)$.*

Proof:

To prove this lemma, we proceed by induction on the depth of the processed configuration in the term.

Initial case:

Consider a process p. Let t be TS_p. The semantics of the term of p is: $T(p,t) = f(T(p_1,t_1)),\ldots,T(p_i,t_i)$,
$\ldots,T(p_n,t_n))$, where f is the symbol of p and $\forall k \in \{1,\ldots,n\}, TS\_Table_p[p_k] = t_k$.

Suppose that p receives an SUR configuration in a message $CONFIG$ and processes it. This configuration is described by: There is an outgoing subterm edge from process p to process p_i, there is an outgoing rewrite edge from process p_i to process p_j. We consider here an SUR configuration without unification edge, but it is similar with an SUR configuration with a unification edge.

The time stamp of p changes to $TS_p = t + 1$.

The semantics of the new term of p is: $T(p,TS_p) = f(T(p_1,t_1),\ldots,T(p_j,t_j),\ldots,$ $T(p_n,t_n))$, where $t_j = TS\_Table[p_j]$.

The configuration was processed, so the time stamp contained in the message $CONFIG$ is equal to or larger than $TS\_Table[p_i]$, furthermore $T(p,t) \succ T(p_i,t_i)$, using the subterm property and $T(p_i,t_i) \succ T(p_j,t_j)$, because of the rewrite edge.

$T(p_i,t_i) \succ T(p_j,t_j)$ implies that $T(p,TS_p) \succ T(p,t)$. As $t < TS_p$ and $T(p,TS_p) \succ T(p,t)$, the lemma is proved for the initial case.

General case:

Suppose that the time stamp of p changes now, because its TS_Table changes.

The reason of this change is that the semantics of a subterm of p changed due to the processing of an SUR configuration.

The induction hypothesis is that for all processes p' such that there exists a chain of subterm edges going from p to p' the lemma is verified.

Let p_1 be the subterm changing the semantic of p. We can apply the induction hypothesis to p_1 i.e. the term of process p_1 is replaced by a smaller term. So the term of p is getting smaller and this proves the lemma.

□

Definition 6 *The term of a process is persisting, if the time stamp of the process is persisting.*

Corollary 1 *For each process, its term becomes persisting.*

Proof: Using lemma 2 for a process p, in particular, $T(p,TS_p) \succ T(p,TS_p+1)$. As LPO is well-founded, the term of p is getting smaller and becomes persisting so the node becomes persisting. □

Theorem 2 *Our concurrent completion using ground SOUR graphs terminates.*

Proof: This theorem is a consequence of the precedent corollary. □

3.6 Soundness and Completeness Results

In this section, we give results for the soundness and completeness of our concurrent calculation of Unification and Orientation. Also, we give results for the soundness and the completeness for our concurrent Completion using SOUR graphs.

Let E be the equational theory the graph represents.

3.6.1 Soundness Results

This section contains the definitions and the proofs of the soundness of the concurrent unification and of our concurrent completion using SOUR graphs.

Definition 7 *A concurrent SOUR graph is a set of processes and their states.*

Definition 8 *A coucurrent sound SOUR graph is such that if there exists a unification edge with a True unification constraint or a rewrite edge between two terms s and t then $s =_E t$ and no subterm is added such that a cycle of subterm edges is created.*

Definition 9 *$G_0, \ldots G_n$ is a SOUR Graph derivation, where G_0 is the initial SOUR graph and each G_i is a concurrent SOUR graph forall $i \in \{0, \ldots, n\}$ and G_{i+1} is obtained from G_i by the processing of a configuration of G_i forall $i \in \{1, \ldots, n-1\}$.*

Definition 10 *Concurrent unification is* sound *if for all derivations $G_0,\ldots,G_n$, forall $i \in \{0, \ldots, n\}$ and for all unification edges of G_i between a process with term s and a process with term t with* True *unification constraint, we have $s =_E t$.*

Definition 11 *Our concurrent completion using SOUR graphs is* sound *if for all derivations $G_0,\ldots,G_n$, forall $i \in \{0, \ldots, n\}$, G_i is sound.*

Lemma 3 *If G is a sound SOUR graph, if one process of G processes a configuration, the obtained SOUR graph is sound.*

Proof: To prove the lemma, we prove each statements separately.

1. We first prove that, if a rewrite edge is added between process p with term s and the process with term t on the sound SOUR graph G, then $s =_E t$.

   A rewrite edge is added, when an *RUR* configuration or an *RUR-rhs* configuration is processed.

   If a configuration *RUR-rhs* is processed i.e. at a certain time, the schema of Figure 8 takes place, a rewrite edge is added between the process with term s

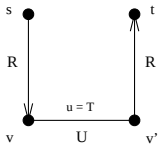

Figure 8: Processing of a configuration *RUR-rhs*

and the process with term t. We must prove that $s =_E t$. As G is sound, we have $s =_E v$, $v' =_E t$ and $v =_E v'$. By transitivity, we obtain $s =_E t$.

If a configuration *RUR* is processed i.e. at a certain time, the schema of figure 9 takes place, a rewrite edge is added between the process with term s and the process with term t (The orientation of this edge depends on whether $s \succ_{lpo} t$ or $t \succ_{lpo} s$). We must prove that $s =_E t$. As G is sound , we have $s =_E v$, $v' =_E t$ and $v =_E v'$. By transitivity, we obtain $s =_E t$.

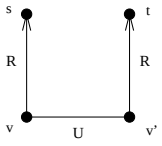

Figure 9: Processing of a configuration *RUR*

2. We now prove that if a subterm edge is added to p on the sound SOUR graph G and p's new term is s' and if there is a rewrite edge between p and the process with term t then $s' =_E t$.

   If a configuration *SUR* is processed i.e. at a certain time, the schema of Figure 10 takes place, a subterm edge is added between the process with term w' and the process with term w. We must prove that $s[w'[w]] =_E t$. As G is sound, we have $v' =_E w$, $s[w'[v]] =_E t$ and $v =_E v'$. By transitivity, we obtain $s[w'[w]] =_E t$.

3. If a subterm edge is added between the process p with term s and the process p' on the sound SOUR graph G and if there is a unification edge between process p with its new term s' and process with term t, then $s' =_E t$. This statement is proved in the same way as the previous statement.

4. We now prove that if a unification edge with a $True$ unification is added between two processes with terms s and t on the sound SOUR graph G, then $s =_E t$.

   Suppose there is a unification edge with a unification constraint $True$ between two processes representing terms $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$. So one of these processes has got information that there are unification edges with constraints $True$ between s_i and t_i for $i \in \{1, \ldots, n\}$ (see rules (28) to (31)). By soundness of

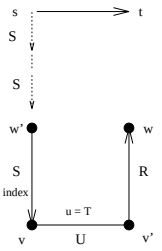

Figure 10: Processing of a configuration SUR

G, $s_i =_E t_i$ for $i \in \{1, \ldots, n\}$, so $f(s_1, \ldots, s_n) =_E f(t_1, \ldots, t_n)$ by the congruence property of $=_E$.

5. We now prove that no subterm is added on a sound SOUR graph G such that a cycle of subterm edges is created using Figure 11. Indeed, the processing of an SUR configuration is the only way to create a cycle of subterms edges. Here we consider the case where the SUR configuration does not contain any unification edge ,but the proof is similar when the SUR configuration contains a unification edge. Using this figure, we will prove that it is impossible to add a subterm edge from p_2 to p_n.
   In figure 11, TS_{p_n}, $TS_{p_{n-1}}$, ..., TS_{p_2} and TS_{p_1} are the current time stamps of processes p_n, p_{n-1}, ..., p_2 and p_1 in a concurrent SOUR graph G.
   Let $t_1 = TS\_Table_{p_2}[p_1], \ldots, t_{n_2} = TS\_Table_{p_{n-1}}[p_{n-2}], t_{n-1} = TS\_Table_{p_n}[p_{n-1}]$.

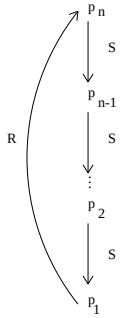

Figure 11:

In Figure 11, we have: $T(p_n, t_n) \succ_{lpo} T(p_{n-1}, t_{n-1}) \succ_{lpo} \ldots \succ_{lpo} T(p_2, t_{p_2}) \succ_{lpo} T(p_1, t_1)$ **(1)**.
We prove that $\forall i \in \{1, \ldots, n-1\}, T(p_{i+1}, TS_{p_{i+1}}) \succ_{lpo} T(p_i, TS_{p_i})$.

We have $T(p_{i+1}, TS_{p_{i+1}}) \succ_{lpo} T(p_i, t_i)$ by the subterm property. Furthermore, using lemma 2, as $TS_{p_i} \geq t_i$, $T(p_i, t_i) \succ_{lpo} T(p_i, TS_{p_i})$ and by transitivity, $T(p_{i+1}, TS_{p_{i+1}}) \succ_{lpo} T(p_i, TS_{p_i})$

An SUR configuration, containing a subterm edge from p_2 to p_1 and a rewrite edge from p_1 to p_n, is detected by process p_1 and sent to process p_2. When process p_2 receives this configuration, and if it processes it, by the subterm property, $T(p_2, TS_{p_2}) \succ_{lpo} T(p_1, t_1)$. In addition, because of the rewrite edge and if we freeze the system, we have: $\exists t, T(p_2, TS_{p_2}) \succ_{lpo} T(p_n, t)$ such that $t = TS\_Table_{p_1}[p_n]$ when the time stamp of p_1 is $TS\_Table_{p_2}[p_1]$ and the time stamp of p_2 is TS_{p_2}. If TS_{p_n} is the current time stamp of p_n when the SUR configuration is received, $TS_{p_n} \geq t$ and using lemma 2, we can deduce that: $T(p_n, t) \succeq_{lpo} T(p_n, TS_{p_n})$.

As $T(p_2, TS_{p_2}) \succ_{lpo} T(p_n, t)$ and $T(p_n, t) \succeq_{lpo} T(p_n, TS_{p_n})$, we have by transitivity : $T(p_2, TS_{p_2}) \succ_{lpo} T(p_n, TS_{p_n})$ **(2)**, which is in contradiction with **(1)**. So it is impossible to have a schema like in figure 11, so to create a cycle of subterm edges. This proves the lemma.

□

Theorem 3 *Our concurrent completion using SOUR Graphs is sound.*

Proof: By induction, the base case is trivial, and the induction step follows immediately from lemma 3. □

Corollary 2 *Concurrent Unification is sound.*

Proof: By induction, the base case is trivial, and the induction step follows immediately from lemma 3. □

3.6.2 Completeness Results

This section contains the definitions and the proofs of the completeness of the concurrent unification, of the concurrent orientation and of our concurrent completion using SOUR Graphs.

Definition 12 *Concurrent unification is* complete *if whenever s and t are persisting terms and s and t are unifiable, then at some time the unification edge between the processes representing terms s and t will have a $True$ unification constraint.*

Definition 13 *The concurrent computation of orientation is* complete *if whenever s and t are persisting terms and $s \succ_{lpo} t$ then, at some time, the orientation edge between processes representing terms s and t goes from s to t and does not change.*

Definition 14 *Our concurrent completion using SOUR Graphs is* complete, *if there are no configuration between persisting nodes.*

Theorem 4 *Concurrent unification is complete.*

Proof: To prove completeness, we proceed by structural induction on the pairs of terms of the SOUR graph.

Base case:

If we consider terms s and t as constants, the result is trivial.

Induction step:

Suppose terms $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ are persisting forms and are unifiable, so s_i and t_i are persisting forms and are unifiable for $i \in \{1, \ldots, n\}$. The induction hypothesis is that for all terms structurally less complex than $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, the statement for completeness becomes $True$. By the induction hypothesis, there are unification edges with unification constraints $True$ between s_i and t_i for $i \in \{1, \ldots, n\}$. By passing up information (see rules (28) to (31)), the constraint of the unification edge between s and t becomes $True$. It is clear, that information from children processes will not be ignored, because time stamps of children are persisting. this is because the TS_tables information of processes receiving information from their children is not in contradiction with the time stamps contained in messages passing up. □

Theorem 5 *The concurrent computation of orientation is complete.*

Proof: To prove completeness, we proceed by structural induction on the pairs of terms in the SOUR graph.

Base case:

If we consider terms s and t as constants, the result is trivial (see rules (64) and (85) for instance).

Induction step:

The induction hypothesis is that for all terms structurally less complex than $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, the statement for completeness is $True$.

- Suppose that terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ are persisting forms and $s \succ_{lpo} t$ such that $f > g$ and $\forall j \in \{1, \ldots, m\}, s \succ_{lpo} t_j$. Each t_j is a persisting form for $j \in \{1, \ldots, m\}$. By the induction hypothesis, at some time and for $j \in \{1, \ldots, m\}$, the orientation edge between processes representing terms s and t_j goes from s to t_j. By passing up information (see rules (64) and (85) for instance), the process representing term t will know all this information, and using rule (89), it will add a new orientation edge from the process representing terms s to the process representing term t.
- Suppose that terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ are persisting forms and $s \succ_{lpo} t$ such that $\exists i \in \{1, \ldots, n\}, s_i \succ_{lpo} t$. Each s_i is a persisting form. By the induction hypothesis, at some time, the orientation edge between the

processes representing terms s_i and t goes from s_i to t. By passing up information (see rules (63) and (78) for instance), the process representing term s will add a new orientation edge from the process representing term s to the process representing term t.

- Suppose that terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ are persisting forms and $s \succ_{lpo} t$ such that $f = g, n = m, \forall i \in \{1, \ldots, n\}, s \succ_{lpo} t_i$, and $(s_1, \ldots, s_n) \succ_{lex} (t_1, \ldots, t_n)$ such that $s_j \succ_{lpo} t_j$ and $\forall k < j, s_k = t_k$. So s_i and t_i are persisting forms for $i \in \{1, \ldots, n\}$. By the induction hypothesis, at some time, for $k \in \{1, \ldots, n\}$, the orientation edges between processes representing terms s and t_k go from s to t_k, the orientation edge between processes representing terms s_j and t_j goes from s_j to t_j, and the orientation edge between the processes representing terms s_i and t_i for $i \in \{1, \ldots, j-1\}$ is an "equal" orientation edge. By passing up information (see rules (64), (66) and (67) for instance), the process representing term s will know all this information and using rules (92) and (99), it will add a new orientation edge from the process representing term s to the process representing term t.

It is clear, that information from child processes will not be ignored, because time stamps of children are persisting. Indeed, the *TS_tables* information of processes receiving information from their children is not in contradiction with the time stamps containing in messages passing up.

□

Theorem 6 *Our concurrent completion using SOUR Graphs is complete.*

Proof: To prove completeness, we proceed by contradiction. Assume that node n_1 and n_2 are persisting, and that node n_1 has an SUR configuration to process. So node n_1 is not persisting. This prove the completeness.

□

4 Implementation and Experimental Results

We call *CWD* the concurrent implementation of completion using SOUR Graphs and all the concurrent operations that we describe. It implements the *D,P,U* and *O* transitions rules mentioned in this paper.

CWD is currently implemented in *C++* (about 15000 lines of commented C++) using *LEDA* (Library of Efficient Data types) and *PVM* (Parallel Virtual Machine). It is tested on a network of Unix and Solaris Sun4Stations (Sparc5, Sparc10, Sparc20), and Sgi Indy Workstations (processor R4400, 200 MHz, 64 Mb) and on a Power Challenge Array (PCA) (8 R8000 processors at 90 MHz and 160 Mb).

We try to make some optimizations on the implementation, we have expressed them in the paper and we summarize them in the following points.

- The number of messages has been limited. Indeed, communications take most of the running time of a distributed program.

  Identical requests messages (*REQUESTU*, *REQUESTO1*, and *REQUESTO2* messages), configurations (*CONFIG* messages) and *INFOU* messages are sent only once.

- Unification and Orientation are evaluated by a request-answer method.

- Unification is evaluated only when it is needed i.e. there exists a unification edge adjacent to an outgoing rewrite edge.

- Out-of-date messages are not processed.

- A message is redundant if its contents is already present in the sate of the process receiving it.

  For example, if a *INITS* message containing an incoming rewrite edge is received and if the process receiving this message has this rewrite edge in its state, there is no need to process the message. Consequences of this messages have already been done.

Experimental Results : Measurements have been made with many benchmark examples. *Table 1* shows a selection of results of measurements obtained on different platforms for four particular problems. *Table 1* summarizes, for each problem, the platform we use, the total number of messages sent between processes, the total number of *NOTIFY* messages (used for detecting termination), and the real time of execution given in seconds.

Example1 [GNP+93]: Counter5

Precedence : $f \succ g \succ c$.
Set of equations : $E = \{f(c) = g(c)\ ,\ f(g(c)) = g(f(c)),\ f(g(g(c)) = g(f(f(c))),\ f(g(g(g(c))) = g(f(f(f(c)))),\ f(g(g(g(g(c))))) = g(f(f(f(f(c))))))\}$
Number of child processes (the number of vertices of the initial graph representing the set of equations to complete) : 17
The reason that the counter example is interesting is that for some strategies of completion it runs exponentially, even if structure sharing is used.

Example2: expf6

Precedence : $aa \succ ab \succ ac \succ ad \succ ae \succ af \succ ag \succ f \succ b$.
Set of equations : $E = \{aa = f(ab, ab, ab, ab, ab, ab), ab = f(ac, ac, ac, ac, ac, ac), ac = f(ad, ad, ad, ad, ad, ad),\ ad = f(ae, ae, ae, ae, ae, ae),\ ae = f(af, af, af, af, af, af), af = f(ag, ag, ag, ag, ag, ag),\ f(aa, aa, aa, aa, aa, aa) = b\}$.
Number of child processes: 15
This example is interesting, because terms resulting rules are exponential. Because of structure sharing, we do not have this problem. On this example, *OTTER-3.0* runs out of

memory after 20 minutes on a Digital server with 2 CPUs and 256 Mb of Main Memory. *PaReDux* also gives no result for this example, because of the number of reclaimed cells.

Example3:

Precedence : $a > b > c > d > e > f > g > h > j > k > l$ Set of equations :
$E = \{a = b, a = c, f(a) = d, g(a, a) = h(b, c), k(a, c, d) = k(f(a), d, g(a, a)), j(f(b)) = j(g(a, b)), f(c) = g(f(a), g(b, c)), l(a, b) = e, l(g(f(a), f(b)), a) = e\}$.
Number of child processes: 20

Problem	Platform	Completion Messages	Termination Messages	Real time
Counter5	1 Sun4	582	627	13.00 s
Counter5	5 Sun4	470	514	4.00 s
Counter5	10 Sun4	648	696	3.62 s
Counter5	15 Sun4	616	663	3.23 s
Counter5	5 Sgi	500	545	4.77 s
Counter5	PCA	594	639	1.88 s
Expf6	1 Sun4	724	822	11.00 s
Expf6	5 Sun4	737	835	5.52 s
Expf6	10 Sun4	734	832	3.00 s
Expf6	15 Sun4	729	827	3.00 s
Expf6	5 Sgi	715	813	4.56 s
Expf6	PCA	722	820	2.32 s
Data	1 Sun4	1023	1099	10s
Data	5 Sun4	850	925	5s
Data	10 Sun4	937	1012	4s
Data	15 Sun4	893	968	3s
Data	5 Sgi	889	964	7s
Data	PCA	975	1051	7s

Table 1

Using *Table 1*, we can see that results are the best when we run our program on the PCA. This is because the PCA architecture is a distributed architecture with a shared memory component. Message passing is implemented through the shared memory. When we run our program on a network of SUN4 and SGI workstations, there are speeds up between 3 and 4. In fact, speeds up depend on the examples, on their sequentiality and on communication overhead. We consider these firstobtained results to be promising. Since our method is concurrent from its design, we think that the implementation can be improved. In particular, the number of termination messages can be reduced and processes can avoid idle time by performing background work.

5 Conclusion

We have presented a concurrent algorithm for completion using SOUR graphs. The initial set of equations is divided up into small pieces, stored in a graph. Initially each vertex represents one of the subterms in the initial set of equations. Edges represent relations between those pieces. Inferences are local graph transformations, which remove existing edges and create new edges constrained by constraints and renamings. In the concurrent implementation, vertices represent processes and edges represent communication links between processes.

For simplicity, the algorithm we have given in this paper is for ground completion. However, we have summarized in the preliminaries how the theory of SOUR graphs is lifted to the non-ground case. The modification of our algorithm to the non-ground case is simple. Unification messages now contain a satisfiable equational constraint instead of just an assertion that a unification problem is true. Orientation messages contain a satisfiable ordering constraint. Configuration messages pass around these constraints, plus renamings. And new edges that are added are labelled with a constraint and renaming, which is a combination of existing constraints and renamings plus new ones given by a unification message. This gives the processes more work to do. Whenever a node receives constraints, it must combine them, determine the satisfiability of the combination, and pass along the solved form.

Our presentation and algorithm has several desirable properties. As far as we know, it is the first fine-grained completion or theorem proving procedure, in the sense that each process represents a subterm of the equational system.

Another feature of most concurrent completion and theorem proving techniques is that they either require a global memory which all processes need to read from and write to, or else a master process which controls all the other processes. In contrast, our algorithm works in a completely local fashion. All processes operate independently, because the algorithm is based on local transformations of the SOUR graph algorithm. The work of a process is based on receiving a message from another process. However, a process does not need to wait, because there are lots of different things that are done by one process, and the message passing is asynchronous. Our algorithm has no need for any consistency checks, and two processes never do the same thing.

We feel that the property of having no global memory or global control is important. As mentioned in [BH94], contraction-based strategies are the most efficient strategies, but they are the most difficult to parallelize. Their parallelization almost always requires a global control or global memory, because anything can be a simplifier, and it is not known which simplifiers will simplify which equations. Our approach allows a contraction based strategy with all local operations.

Acknowledgements

This work is partially supported by the Esprit Basic Research working group 6028, CCL and by the "Centre Charles Hermite". We would like to thank Polina Strogova, Ilies Alouini, Dominique Fortin and the anonymous referees of the RTA commitee for their comments.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A system for distributed equational deduction. In Hsiang [Hsi95], pages 397–402.

[BGK95] R. Bündgen, M. Göbel, and W. Küchlin. Parallel ReDuX → PaReDuX. In Hsiang [Hsi95], pages 408–413.

[BH94] M. P. Bonacina and J. Hsiang. Parallelization of deduction strategies: an analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.

[BH95] M. P. Bonacina and J. Hsiang. Distributed deduction by clause-diffusion: Distributed contraction and the aquarius prover. *Journal of Symbolic Computation*, 19:245–267, March 1995.

[DFvG83] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters*, 16:217–219, 1983.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[GNP+93] J. Gallier, P. Narandran, D. Plaisted, S. Raatz, and W. Snyder. Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. *Journal of Association for Computing Machinery*, 40(1):1–16, 1993.

[Hsi95] J. Hsiang, editor. *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[KLS96] C. Kirchner, C. Lynch, and C. Scharff. A fine-grained concurrent completion procedure. In H. Ganzinger, editor, *Proceedings of RTA '96*, volume 1103 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, July 1996.

[Koz77] D. Kozen. *Complexity of Finitely Presented Algebras.* PhD thesis, Cornell University, 1977.

[LS90] R. Letz and J. Schumann. Partheo: A high-performance parallel theorem prover. In *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 446 of *Lecture Notes in Artificial Intelligence*, pages 40–56. Springer-Verlag, 1990.

[LS95] C. Lynch and P. Strogova. Sour graphs for efficient completion. Technical Report 95-R-343, CRIN, 1995.

[Lyn95] C. Lynch. Paramodulation without duplication. In D. Kozen, editor, *Proceedings 10th IEEE Symposium on Logic in Computer Science, San Diego (Ca., USA)*, pages 167–177, San Diego, June 1995. IEEE.

[PSK96] D. A. Plaisted and A. Sattler-Klein. Proof lengths for equational completion. *Information and Computation*, 125(2):154–170, 15 March 1996.

[Sny93] W. Snyder. A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. *Journal of Symbolic Computation*, 15:415–450, 1993.

[SS93] C. Suttner and J. Schumann. Parallel automated theorem proving. *Parallel Processing for Artificial Intelligence*, 1993.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399