A logic-functional approach to the execution of CCS specifications modulo behavioural equivalences

S. Gnesi, P. Inverardi, M. Nesi
IEI-CNR
via S. Maria n. 46, I-56100 Pisa

Abstract
This paper reports on a work that proposes a kernel for an execution environment for the operational semantics and the behavioural equivalences of CCS. The proposed execution environment distinguishes itself by being formal, by dealing with the behavioural equivalences as schemes of axioms, differently from other approaches based on automata, and by giving the possibility to define several strategies of verification in a modular and flexible way. The environment, obtained by techniques of logic-functional programming, treats basic CCS with bounded recursion. A particular strategy of verification is presented.

1. Introduction

Aim of the work reported in this paper is the implementation of an *environment for the verification of properties of concurrent systems*.

We consider the specification language CCS, without value-passing, regarded as set of operators to describe concurrent processes and set of equivalences to decide when two processes have an equivalent behaviour. The aim is to execute processes according to the operational semantics (from now on OPSEM [Mil80]) by interacting with the equivalences, where execution of processes means analysis of their behaviour and verification of their properties. Hence, an environment has to provide for the execution of OPSEM *modulo* the behavioural equivalences of CCS.

Our approach is to implement such environment by using *techniques of logic-functional programming*. In fact the rules of OPSEM can be seen as rules of a *(conditional) term rewriting system*, by which it is possible to infer the operational behaviour of each process; this immediately suggests the use of logic programming. On the other hand, behavioural equivalences, in their axiomatic formulation, are an *equational theory* thus suggesting the use of functional programming.

The environment is obtained in a *formal* way : the correctness and completeness of every transformation on OPSEM and on the equivalences are proved formally.

The paper first presents how to deal with the execution of OPSEM and equivalences for finite CCS and then shows how to build a particular proof strategy. Actually, the described approach has been, formally, extended to CCS with recursion ; the form of recursion we have considered is called *bounded* : given a term rec X.E, all the process variables (possibly) in E are only occurrences of the variable X and X is guarded. Such extension is based on [Mil86] where a correct and complete set of axioms is presented for, so called, finite-state behaviours, which in other words correspond to that subset of CCS processes that can be given a finite representation.

In this paper we will present results only on finite CCS without considering recursion, the strategy we finally present deals, anyhow, with recursive CCS processes and resembles, in essence, the way in which the extension of our environment to CCS with bounded ricorsion, following [Mil86], has been performed.

The execution of OPSEM is obtained by translating the rules of OPSEM into Horn clauses. A problem related to the termination of the process execution is solved by *metaprogramming techniques*, obtaining a logic program *executor*. Hence, we prove that the translation of OPSEM into executor is correct and complete wrt the action-tree semantics.

Then, we examine the equivalences. We study in particular the observational equivalence (OBS.EQ.) and consider it as *scheme of axioms*, [HM85] and [Mil84a]. This approach distinguishes itself from others in literature, where processes are modelled by finite state automata and the observational equivalence or congruence is decided by algorithms on automata.

By techniques of logic-functional programming OBS.EQ. is translated into Horn clauses, solving the problems of completion of the observational theory, cyclic reductions (caused by the commutative axiom for summation) and process representation.

Our solution *divides the observational theory OBS.EQ. in two subtheories*: the equivalences of the former subtheory are applied to the *process symbolic representation* in order to rewrite some CCS operators in terms of other CCS operators, while the equivalences of the latter subtheory are applied to the *action-tree* of the process, output of the former phase of reduction, thus abstracting from the subterms order.

We prove that the subtheories don't need completion (a canonical rewriting system is obtained by directing equivalences in an appropriate way [HO80]); equivalences are translated into Horn clauses via *flattening* (flattening allows SLD resolution to simulate narrowing and to ensure that every reducible subterm is reduced [BGM87]) and finally we prove the correctness and completeness of the translation of OBS.EQ.

At this point, in order to decide the congruence of two finite CCS processes it is possible to reduce both in normal form and then compare the normal forms. In the same way it is possible to execute a process: the existence of unique normal forms (according to OBS.EQ.) modulo subterms order, allows a *sequential* strategy in executing a process to be adopted. First the process is reduced according to OBS.EQ., then its normal form is executed according to OPSEM.

One of the main motivations of our approach is that of allowing "user" defined verification strategies to be implemented. As a practical example we present how it is possible to define a particular proof method for proving the equivalence of two CCS expressions, based on a strategy described in [Mil80], which, given two expressions E1 and E2, tries to rewrite E1 into E2 by alternating some steps of execution according to OPSEM with some steps of reduction according to OBS.EQ. or with some steps of (backward) replacement of subexpressions with a process variable bound by a rec operator.

This proof method may not terminate, but if it terminates, it has been shown correct, because every rewriting step preserves equivalence.

We apply this method to the *scheduling problem* proposed in [Mil80] in order to prove that the scheduler is a correct implementation of the requested specification.

The kernel environment presented in this paper, has been implemented in Quintus Prolog 2.0

on a Sun 3/50M machine under Unix 4.2 BSD. Three general reasons for preferring Quintus Prolog, and generally Prolog, can be given. The first one is the availibility of efficient implementations; the second one is a very natural translation algorithm of rewrite rules into Horn clauses and the availability of techniques, like flattening, allowing equivalences to be executed and the third reason is metaprogramming that allows the interaction between operational semantics and behavioural equivalences according to different user-defined proof strategies to be easily managed.

2. Basic definitions

In this section we report a set of basic definitions that will be extensively used throughout in the paper.

We deal with basic CCS (without value passing).

Let $\Delta = \{ \alpha, \beta, \gamma, \ldots \}$ be a fixed set of *names*, $\Delta^{-} = \{ \alpha^{-} \mid \alpha \in \Delta \}$ the set of *conames*, $\Lambda = \Delta \cup \Delta^{-}$ (ranged over by λ) the set of *observable actions*, τ the *unobservable action* and $\Lambda \cup \{\tau\}$ (ranged over by μ) the set of *atomic actions*. S is a *relabelling* of Λ which respects complement. Let *V* be a set of *process variables*, ranged over by X, Y, ... We define below (using a BNF-like style) the class of *CCS expressions E*, ranged over by E, E_1, ...; then we take *P* to be the class of *finite CCS processes* (without recursion and variables).

Syntax

$$E ::= X \mid \mu.E \mid E_1+E_2 \mid E_1 \mid E_2 \mid E[S] \mid E\backslash\alpha \mid NIL \mid rec\ X.E$$

Operational semantics

Prefix

$$\mu.E \xrightarrow{\mu} E$$

Summation

$$\frac{E_1 \xrightarrow{\mu} E}{E_1+E_2 \xrightarrow{\mu} E} \qquad \frac{E_2 \xrightarrow{\mu} E}{E_1+E_2 \xrightarrow{\mu} E}$$

Parallel composition

$$\frac{E1 \xrightarrow{\mu} E}{E_1 \mid E_2 \xrightarrow{\mu} E \mid E_2} \qquad \frac{E2 \xrightarrow{\mu} E}{E_1 \mid E_2 \xrightarrow{\mu} E_1 \mid E}$$

$$\frac{E_1 \xrightarrow{\lambda} E_1' \qquad E_2 \xrightarrow{\lambda^{-}} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'}$$

Relabelling

$$\frac{E \xrightarrow{\mu} E_1}{E[S] \xrightarrow{S(\mu)} E_1[S]}$$

Restriction

$$\frac{E \xrightarrow{\mu} E_1}{E\backslash\alpha \xrightarrow{\mu} E_1\backslash\alpha} \qquad \mu \notin \{\alpha, \alpha^{-}\}$$

$$E[rec\ X.E / X] \xrightarrow{\mu} E_1$$

Recursion $$\frac{}{\text{rec } X.E \xrightarrow{\mu} E_1}$$

OPSEM is given by a set of inference rules, which can be seen as "defining" a (conditional) term rewriting system. It is the *inference* component of CCS : all the behaviours af any process can be inferred by following the rules of OPSEM.

Among the different behavioural equivalences defined for CCS in literature, we refer to *observational equivalence* ([Mil80]) and *testing equivalence* ([DeN85]) and in this paper we present the first one. (A correct and complete proof system has been given for both the equivalences).

The laws for observational equivalence (OBS.EQ.), which is indeed a *congruence* = , are the following :

E1. $X + (Y + Z) = (X + Y) + Z$

E2. $X + Y = Y + X$

E3. $X + X = X$

E4. $X + NIL = X$

E5. $X + \tau.X = \tau.X$

E6. $\mu.\tau.X = \mu.X$

E7. $\mu.(X + \tau.Y) = \mu.(X + \tau.Y) + \mu.Y$

E8. $NIL[S] = NIL$

E9. $(X + Y)[S] = X[S] + Y[S]$

E10. $(\mu.X)[S] = S(\mu).X[S]$

E11. $NIL\backslash\alpha = NIL$

E12. $(X + Y)\backslash\alpha = X\backslash\alpha + Y\backslash\alpha$

E13. $$(\mu.X)\backslash\alpha = \begin{cases} \mu.(X\backslash\alpha) & \mu \notin \{\alpha, \overline{\alpha}\} \\ NIL & \text{otherwise} \end{cases}$$

E14. If $X = \sum \mu_i.X_i$ and $Y = \sum \nu_j.Y_j$ then
$$X \mid Y = \sum_i \mu_i.(X_i \mid Y) + \sum_j \nu_j.(X \mid Y_j) + \sum_{\mu_i = \overline{\nu_j}} \tau.(X_i \mid Y_j)$$

These axioms have been shown to be correct and complete for basic CCS without recursion ([Mil84a], [HM80]) and they represent the *equational* component of CCS : they form an equational theory, by which we can decide the observational equivalence of two finite processes. Often proving observational equivalence it happens that the same sequences of axioms are applied over and over again. Such sequences can be collected into theorems, which can be applied in one step. An example is the *expansion theorem* ([Mil80]).

Def.2.1

A *guarded sum* is a process of the form $\sum \mu_i.P_i$. Each $\mu_i.P_i$ is called *summand.*

Notation : if $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $P\backslash A$ denotes $P\backslash\alpha_1\backslash\alpha_2 \ldots \backslash\alpha_n$.

Expansion theorem

Let P denote $(P_1 \mid P_2 \mid \ldots \mid P_m)\backslash A$, where each P_i is a guarded sum. Then

$$P = \sum \{\, \mu_i.(\,(P_1 \mid \ldots \mid P'_i \mid \ldots \mid P_m)\backslash A\,) \mid \mu_i.P'_i \text{ is a summand of } P_i,\ \mu_i, \overline{\mu_i} \notin A \,\}$$
$$+ \sum \{\, \tau.(\,(P_1 \mid \ldots \mid P'_i \mid \ldots \mid P'_j \mid \ldots \mid P_m)\backslash A\,) \mid \mu_i.P'_i \text{ is a summand of } P_i ,$$

$\mu_i^-.P'_j$ is a summand of P_j, $i \neq j$ } .

3. Executing the operational semantics of finite CCS

3.1. The translation algorithm for OPSEM

The inference rules of OPSEM can be straightforwardly translated into Horn clauses by the following *translation algorithm.* The transition relation $P \xrightarrow{\mu} P'$ or equivalently $\rightarrow (P, \mu, P')$ with $\rightarrow \subseteq P \times \Lambda \cup \{\tau\} \times P$ is translated into the predicate trans (P, μ, P').
Each inference rule of OPSEM

$$\frac{P_1, \ldots, P_n}{C} \quad \text{Cond}$$

is translated into a Horn clause, whose head is the conclusion C and whose body is composed by the premises $P_1, \ldots, P_n$ and the conditions Cond of the rule.

Ex. 1
The rule of the restriction operator given in section 2 is translated into the following clause :

```
trans (restr (P, α), μ, restr (P', α)) :-  trans (P, μ, P') ,  μ \== α ,  μ \== compl (α) ,  name (α) .
```

Note that, some rules of OPSEM have conditions on the arguments of CCS operators. In the translation algorithm this would mean the introduction of other predicates checking for these conditions. Since tests like relabelling(S) for a term P[S] and name (α) for a term P\α are *syntactic*, in an execution environment provided of a *parser* (like in our case), these conditions can be thought to be checked before the execution of P[S] and P\α, thus deleting the corresponding tests from the clauses.

Moreover, the translation of OPSEM is able to verify the *dynamic* test on the observability of the action λ in the third rule of parallel composition, with no introduction of another predicate. Only the unobservable action τ has no complement and the translation of OPSEM into Horn clauses ensures that the observability test is implicitly satisfied.

This simple translation is not enough to obtain a logic program that when executed properly simulates the rewriting system associated with OPSEM.

Thus, let us enrich the program obtained translating OPSEM in such a way to be able to compute not only a single transition of a process, but a sequence of transitions (a behaviour) thus obtaining an *interpreter* for finite CCS processes.

On the other hand OPSEM can be seen as a (conditional) rewriting system, whose rules allow to rewrite every process until actions the process can do exist. The rewriting according to OPSEM stops when a process P has been reduced to a process P', which cannot do any action .

The logic program should be an implementation of the rewriting system associated with OPSEM, thus we expect that when executing a process P, via the logic program, resolution succeeds and the behaviour of P is returned. Since OPSEM provides no rules for the process termination in order to properly simulate rewriting we have to force, at metalevel, a successful termination of the logic program whenever P cannot execute more actions (i.e. no rewriting is possible).

Hence, the logic program *executor* executing the finite CCS OPSEM is the following :

```
:- dynamic trans /3 .
trans (prefix (Act, Pr), Act, Pr) .
trans (prefix (Act, Pr), Act1, Pr) :- Act == compl (compl (Act1)) .
trans (prefix (Act, Pr), Act1, Pr) :- Act1 == compl (compl (Act)) .
trans (sum (Pr, _), Act, Pr1) :- trans (Pr, Act, Pr1) .
trans (sum (_, Pr), Act, Pr1) :- trans (Pr, Act, Pr1) .
trans (par (Pr1, Pr2), Act, par (Pr3, Pr2)) :- trans (Pr1, Act, Pr3) .
trans (par (Pr1, Pr2), Act, par (Pr1, Pr3)) :- trans (Pr2, Act, Pr3) .
trans (par (Pr1, Pr2), tau, par (Pr3, Pr4)) :- trans (Pr1, Act, Pr3) , trans (Pr2, compl (Act), Pr4) .
trans (relab (Pr, [X | S]), Act, relab (Pr1, [X | S])) :- trans (Pr, Act1, Pr1) , apply ([X | S], Act1, Act) .
trans (restr (Pr, Act1), Act, restr (Pr1, Act1)) :- trans (Pr, Act, Pr1), Act \== Act1 , Act \== compl (Act1) .

:- dynamic apply /3 .
apply ([ ], Act, Act) .
apply ([Act / Act1 | _], Act1, Act) .
apply ([Act / Act1 | _], compl (Act1), compl (Act)) .
apply ([_ / Act2 | S], Act1, Act) :- Act1 \== Act2 , Act1 \== compl (Act2) , apply (S, Act1, Act) .

execute (Pr, [Act | Beh1]) :- solve (trans (Pr, Act, Pr1)) , execute (Pr1, Beh1) .
execute (Pr, [ ]) :- \+ solve (trans (Pr, _, _)) .

solve (true) :- ! .
solve ((Goal1, Goal2)) :- ! , solve (Goal1) , solve (Goal2) .
solve (Goal):- Goal \== true, functor (Goal, F, _), F \== '==', F \== '\==', F \==',' ,clause (Goal, Body), solve (Body).
solve (Goal) :- system (Goal) , ! , Goal .
system (Goal) :- functor (Goal, F, _) , F == '==' .
system (Goal) :- functor (Goal, F, _) , F == '\==' .
```

The predicate *trans* translates the rules of OPSEM : we note that for the prefix operator, which has only one inference rule in OPSEM, it is necessary to add two more clauses translating the semantics of the complement function, $\bar{\bar{\mu}}.P \xrightarrow{\mu} P$ and $\mu.P \xrightarrow{\bar{\bar{\mu}}} P$, thus obtaining three mutually exclusive clauses for the prefix operator.

The predicate *apply* translates the application of the relabelling function, where such function is implemented by an appropriate data structure, i.e. a list. We can verify ([Nes88]) that the above definition of apply is correct wrt the relabelling definition given in [Mil80].

The metapredicate *solve* is our metainterpreter (based on [Ste85] and [FGIM87]), where *clause* is a system predicate which is true if there exists a clause, whose head unifies with the goal Goal and whose body is Body, and *system* is a (not predefined) predicate which is true if Goal is a system predicate.

The metapredicate *execute* computes all the behaviours of a process catching the failures due to the normal forms (according to OPSEM) : a process Pr executes a sequence of actions [Act | Beh] if Pr evolves in Pr1 by executing Act and Pr1 executes a sequence of actions Beh1 or Pr terminates since it can execute no more action.

The Quintus Prolog operator \+ applies *SLD-NF resolution*, by which we catch failures taking place when Pr is a normal form according to OPSEM and it has no further transition. SLD-NF resolution gives no problem, because its use is controlled in such a way to preserve completeness.

In order to compute the behaviour Beh of a finite process P the goal :- execute (P, Beh) is solved in the program executor.

3.2. Correctness and completeness of the translation

In the previous section we have described the translation of the operational semantics of finite CCS into a logic program. Aim of this section is to provide a formal justification that such translation "preserves" semantics consistency between OPSEM and the corresponding logic program.

Every CCS process can be modelled by an *action-tree*, a tree representing the process behaviour, i.e. all the actions the process can execute and the order in which it can execute them.

Moreover, let P be a logic program and G a goal, an SLD-tree representing all the resolutions of G in P is associated to P and G. Hence, given the logic program in which OPSEM has been translated and a process P (whose behaviour is asked for) as a goal, we have, besides an action-tree, an SLD-tree associated to P, too.

Now we prove that the translation of finite CCS OPSEM into the logic program executor is correct and complete wrt the action-tree semantics. In other words, we show that there exists a correspondence between the action-tree of a process P and the SLD-NF-tree associated to the resolution of the initial goal :- execute (P, Beh) in the logic program executor (i.e. the resolution of executor $\cup$ { :- execute (P, Beh)}, [Llo84]).

To verify completeness we prove that for every path $\mu_1, \mu_2, ..., \mu_n$ in the action-tree of P there exists a successful path in the SLD-NF-tree with an answer substitution $\{[\mu_1, \mu_2, ..., \mu_n] /\mathrm{Beh}\}$; the opposite for correctness.

In the following theorems, we use the notion of *total path* in a tree, meaning a path starting from the root and reaching a leaf of the tree.

Def.3.2.1
Let executor be the given program, P a process and G the goal :- execute (P, Beh).
A *computed answer substitution* θ *for G* is the substitution obtained by restricting the composition $\theta_1....\theta_n$ to the variable Beh of G, where $\theta_1, ..., \theta_n$ is the sequence of mgu's used in an SLD-NF-refutation of G.

Theorem 3.2.1 (correctness)
Let executor be the given program, P a process and G the goal :- execute (P, Beh).
For every computed answer substitution $\theta = \{[\mu_1, ..., \mu_n] /\mathrm{Beh}\}$ for G, there exists a (total) path $\mu_1, ..., \mu_n$ in the action-tree of P.

Proof

Let $\theta = \{[\mu_1, ..., \mu_n] /Beh\}$ be the computed answer substitution of an SLD-NF-refutation of G. We have to show that the actions list Beh $\theta = [\mu_1, ..., \mu_n]$ is a (total) path in the action-tree of P. The result is proved by *induction on the length n of the actions list.*

i) $n = 0$.

$\theta = \{[\,] /Beh\}$ means that SLD-NF-resolution of the goal :- execute (P, Beh) implies the failure of the predicate solve (trans (P, Act, Pr1)) in the body of the 1st clause of execute and therefore its negation in the body of the 2nd clause of execute succeeds. This fact implies that the process P is a term in normal form according to OPSEM, P has no transitions and its action-tree is the only root $\Rightarrow$ every path is empty.

ii) The *inductive hypothesis* is the following : for every computed answer substitution $\theta = \{[\mu_1, ..., \mu_{n-1}] /Beh\}$ returning an actions list of length n-1 as behaviour of P, there exists a (total) path labelled $\mu_1, ..., \mu_{n-1}$ in the action-tree of P.

Let's prove this result also for SLD-NF-refutations computing actions lists of length n.

Let $\theta = \{[\mu_1, ..., \mu_n] /Beh\}$ be the computed answer substitution of an SLD-NF-refutation of G. Let's see how is this refutation.

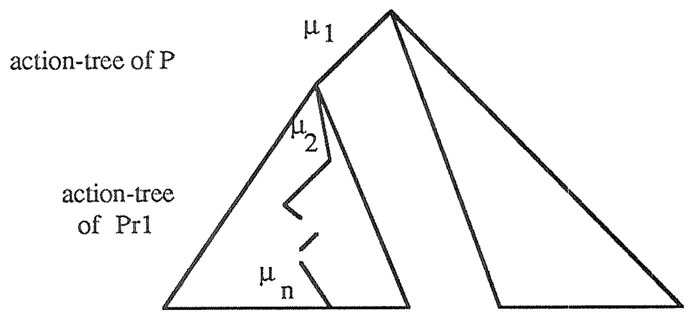
Given the goal :- execute (P, Beh) , it is unified with the head of the two clauses of execute. By the 1st clause we obtain Pr = P, Beh = [Act | Beh1] and the new goal is

:- solve (trans (P, Act, Pr1)) , execute (Pr1, Beh1) .

By the 2nd clause we obtain Pr = P, Beh = [ ] and the new goal is

:- \+ solve (trans (P, _, _)).

Resolution of solve (trans (P, Act, Pr1)) doesn't cause a finitely failed tree, therefore \+ solve (trans (P, _, _)) fails, while by the 1st clause resolution binds the variable Act to the action μ_1, istantiates ground the variable Pr1 and Beh = $[\mu_1 \mid Beh1]$. The current goal is G' = :- execute (Pr1, Beh1). For hypothesis a computed answer substitution for G is $\theta = \{[\mu_1, ..., \mu_n] /Beh\}$ and Beh = $[\mu_1 \mid Beh1]$, an SLD-NF-refutation of G' computes the answer substitution $\theta_1 = \{[\mu_2, ..., \mu_n] /Beh1\}$, binding the variable Beh1 of G' to an actions list of length n-1. For inductive hypothesis, in correspondence to θ_1 there exists a path labelled $\mu_2, ..., \mu_n$ in the action-tree of Pr1. But Pr1 is bound to P by the relation trans (P, μ_1, Pr1), so that the action-tree of P has the action-tree of Pr1 as subtree, linked to the root of the action-tree of P by an arc labelled μ_1.



It follows that it exists a (total) path labelled $\mu_1, ..., \mu_n$ in the action-tree of P.

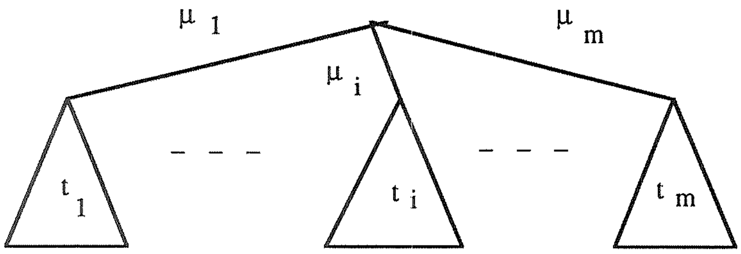
Theorem 3.2.2 (completeness)

Let executor be the given program, P a process and G the goal :- execute (P, Beh).

For every (total) path $\mu_1, ..., \mu_n$ in the action-tree of P there exists a computed answer substitution $\theta = \{[\mu_1, ..., \mu_n] / Beh\}$ for G.

Proof

Let t be the action-tree of P. We can write $t = \sum \mu_i.t_i$, where t_i is the action-tree of the process P_i such that the predicate trans(P, μ_i, P_i) is true. The result is proved by *induction on the length n of any path in the action-tree.*



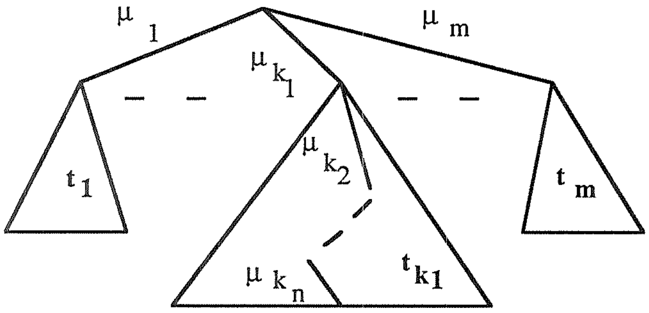
i) n = 1.

Let's consider a (total) path of length 1 of t, like $\mu'.t'$, where t' is the action-tree (consisting of the only root) corresponding to a process in normal form. There exists an SLD-NF-refutation computing the answer substitution $\theta = \{[\mu'] / Beh\}$ and it is exactly the refutation corresponding to the application of the 1st clause of execute, by which we obtain Act = μ', Beh = [μ' | Beh'] , and then the application of the 2nd clause of execute, where Pr1 is a process in normal form.We obtain Beh' = [ ] and therefore Beh = [μ'].

ii) The *inductive hypothesis* is the following : for every (total) path $\mu_1, ..., \mu_{n-1}$ of the action-tree of a process P of length n-1 there exists a computed answer substitution $\theta = \{[\mu_1, ..., \mu_{n-1}] / Beh\}$ for the goal

G = :- execute (P, Beh). Let's prove this result also for paths of length n.

We consider a (total) path of length n in the action-tree t of P labelled $\mu_{k_1}, ..., \mu_{k_n}$.



The subtree t_{k_1} is the action-tree of a process P_{k_1} such that trans (P, μ_{k_1}, P_{k_1}) is true. Therefore in the SLD-NF-tree corresponding to the goal G = :- execute (P, Beh) there exists a path, where the 1st clause of execute is applied obtaining the following bindings by resolving solve (trans (P, Act, Pr1)) :

Act = μ_{k_1} Pr1 = P_{k_1} Beh = [μ_{k_1} | Beh_{k_1}] (*)

and the new goal G' = :- execute (P_{k_1}, Beh_{k_1}) with P_{k_1} istantiated ground. For hypothesis the action-tree t_{k_1} of P_{k_1} contains a path $\mu_{k_2}, ..., \mu_{k_n}$ of length n-1 and for inductive hypothesis in correspondence to this path there exists a computed answer substitution θ_1 of an SLD-NF-refutation of G' such that $\theta_1 = \{[\mu_{k_2}, ..., \mu_{k_n}] / Beh_{k_1}\}$, so that we obtain Beh = [$\mu_{k_1}, \mu_{k_2}, ..., \mu_{k_n}$] in (*).

It follows that there exists an SLD-NF-refutation for G = :- execute (P, Beh) computing an answer substitution $\theta = \{[\mu_{k_1}, ..., \mu_{k_n}] / Beh\}$.

Let us add some comments on the correspondence between action-tree and SLD-tree.

1. Since we are considering finite CCS, it follows that the action-tree of any process P is a finite tree. That is also true for the SLD-tree corresponding to the resolution of executor ∪ { :- execute (P, Beh)} : its paths are either successful or failure, there are no infinite paths, because, according to the definition of the predicate *trans*, a process P *always* evolves into a process P' ≠ P. Therefore, it never happens that the same rules (transitions) are applied more than once for the same process.

2. The correspondence shown in the previous theorems is defined between action-trees and SLD-trees *modulo* failures, i.e. it is a correspondence between (total) paths in the action-tree and successful paths in the SLD-tree.

3. *Non-determinism* in the action-tree t of a process P is entitery preserved in the corresponding SLD-tree. Choice points in t correspond to occurrences of the operators + and | in P (in the action-tree | is expressed in terms of +). To these choice points there correspond nodes in the SLD-tree labelled by goal like :- solve (trans (sum (.., ..), .., ..)) or :- solve (trans (par (.., ..), .., ..)), whose resolution, by using clauses for summation and parallel composition, implies, in correspondence to the (partial) paths of t leaving from such choice points, successful SLD-paths.

4. Executing observational equivalences

Let us consider behavioural equivalences and let us examine what can be obtained by their execution. In this report we deal only with OBS.EQ., but the approach followed and the strategies developed can be immediately extended to other behavioural equivalences (notably testing equivalence).

Our approach in proving the congruence or equivalence of processes distinguishes itself from other approaches in literature by dealing with equivalences as *schemes of axioms*. In other approaches ([BS87], [KS83], [Ver86]) CCS processes are modelled by means of *finite state automata* and the behavioural congruence is decided by algorithms on automata.

In [KS83] and [BS87] the behavioural congruence of two processes P1, P2 is decided by reducing it to a *partitioning problem* on the states of the automata modelling P1 and P2. A similar algorithm is proposed in [Ver86], where the author considers a notion of reduction of an automata modulo behavioural equivalences (particularly OBS.EQ.), but the behavioural congruence of two processes P1, P2 is decided by using an algorithm based on the construction of a relation, called *equipollence*, according to which the states of the two automata for P1 and P2 are again partitioned in classes. P1 and P2 are congruent if at the end the initial states of the two automata belong to the same class.

In [Ver86] it is claimed that the implementation of several strategies of verification of process behaviour is easier by using algorithms on automata than by using equational theory. On the contrary our aim is to show that by means of an equational approach it is possible to define several strategies in a modular and flexible way.

Executing OBS.EQ. consists in computing a reduced form of a process according to these equivalences and deciding the observational congruence of two finite CCS processes. The congruence can be decided by reducing both processes according to OBS.EQ. and then by comparing the reduced forms. It is useful to point out that, while the inference rules of OPSEM are directly translated into Horn clauses (section 3.1), the translation of equivalences into rewrite rules and then into Horn clauses must be performed preserving the completeness of equational deduction.

4.1. From equivalences to rewrite rules

We refer to the theory of observational congruence of finite CCS given in section 2, where E1.- E4 are the algebraic laws of summation, E5.- E7. are the τ laws and E8.- E14 define relabelling, restriction and parallel composition in terms of the other operators NIL, prefix and summation.

In the following, we assume a restriction on the structure of the processes composed in parallel in E14. This is done in order to make the resulting executable logic program more intellegible. This restriction is in practice easily recovered in [Nes88]. Therefore, if X, Y $\in T_{\Sigma_1}$, where Σ_1 = {prefix, NIL} is a signature on the sort of atomic actions, we have terms X | Y where $X = \mu_1.X_1$, $Y = \nu_1.Y_1$, $\mu_1, \nu_1 \in \Lambda \cup \{\tau\}$ and $X_1, Y_1 \in T_{\Sigma_1}$.

The OBS.EQ. theory is equational-conditional and therefore an algorithm based on narrowing, called CNA ([Hus85]), can be applied to the rewriting system equivalent to the OBS.EQ. theory, not only for solving equations modulo OBS.EQ., but in particular for reducing a term to normal form according to OBS.EQ. On the other hand, flattening ([BGM87]) allows SLD resolution to simulate narrowing, if the rewriting system, which flattening is applied to, is canonical. The application of the *completion algorithm* ([HO80]) to OBS.EQ. fails because of the commutative property for summation. Instead of using extensions of the completion algorithm in order to treat permutative theories, we divide the OBS.EQ. theory in two subtheories : the subtheory $Eq_{Rewrite}$ formed by E8.- E14. and the subtheory $Eq_{sum+\tau} = Eq_{sum} + Eq_{\tau}$, where Eq_{sum} is formed by E1.- E4. and Eq_{τ} by the τ laws E5.- E7.

The equivalences in $Eq_{Rewrite}$ are applied to the symbolic term representation as rewriting rules directed from left to right in order to rewrite a CCS term t into an equivalent term t', with no occurrence of relabelling, restriction and parallel composition, i.e. t' $\in T_{\Sigma}$ where Σ = {sum, prefix, NIL} is a signature on the sort $\Lambda \cup \{\tau\}$.

On the contrary the equivalences in $Eq_{sum+\tau}$ can't be applied to the symbolic term representation, but a new representation easy to work out and abstracting from subterms order is needed.

Splitting the OBS.EQ. theory implies *splitting term reduction in two phases* : a 1st phase in which every term is reduced according to $Eq_{Rewrite}$ and a 2nd phase in which the term output of the 1st is reduced according to $Eq_{sum+\tau}$.
Note that defining τ - laws in a separate theory is a step towards modularity in our environment, in fact in order to introduce testing equivalence it is enough to define the corresponding τ - laws theory. We recall that testing equivalence axioms differ from observational exactly with respect to τ - laws.

4.2. The 1st phase of reduction : application of the equivalences $Eq_{Rewrite}$

4.2.1. Completeness of the 1st phase

The rewriting system obtained by directing the equivalences E8.- E14. of $Eq_{Rewrite}$ from left to right is *confluent* and *noetherian.* This fact can be verified by applying the completion algorithm to the subtheory $Eq_{Rewrite}$. However this system doesn't ensure that, given a term t, the normal form t' of t according to $Eq_{Rewrite}$ is such that $t' \in T_\Sigma$.

Ex. 3

Given the term $P = a.NIL \mid a^-.NIL$, by applying the rule obtained by directing E14. we have the following rewriting step : $P \rightarrow (a.(NIL \mid a^-.NIL) + a^-.(a.NIL \mid NIL)) + \tau.(NIL \mid NIL) = P'$.
P' is a normal form according to $Eq_{Rewrite}$, because neither E14. (the conditions on the structure of the terms in parallel are not satisfied) nor any other rule is applicable, but $P' \notin T_\Sigma$.

Given a term t containing occurrences of relabelling and restriction operators, the rules E8.- E10. for relabelling and E11.- E13. for restriction ensure that rewriting t according to $Eq_{Rewrite}$ results in a term with no occurrence of such operators. This result is false as regards the parallel composition, because E14. is applicable if X and Y are summations of prefixed processes and it isn't explicitly stated how a term $X \mid NIL$ or $NIL \mid X$ can be reduced. In order to ensure that the rewriting of a term t according to $Eq_{Rewrite}$ results in a term t' with no occurrence of parallel composition, and therefore $t' \in T_\Sigma$, it is necessary to insert some equivalences corresponding to the *terminal cases* of parallel composition, as for relabelling and restriction. Assuming the simplification hypothesis on the structure of the arguments of the parallel composition operator and considering that the commutative property is not available as a rewriting rule, the new added equivalences are :

E14.$_4$ $\mu.X \mid NIL = \mu.X$
E14.$_5$ $NIL \mid \mu.X = \mu.X$
E14.$_6$ $NIL \mid NIL = NIL$

Going back to Ex.3 and by applying the rules obtained by directing these last equivalences from left to right, P' is reduced to the normal form of P according to $Eq_{Rewrite}$
$(a.a^-.NIL + a^-.a.NIL) + \tau.NIL \in T_\Sigma$.

E13. and E14. can be written as follows :

E13.$_1$ $(\alpha.X)\backslash\alpha = NIL$
E13.$_2$ $(\alpha^-.X)\backslash\alpha = NIL$
E13.$_3$ $(\mu.X)\backslash\alpha = \mu.(X\backslash\alpha)$ if $\mu \neq \alpha,\ \mu \neq \alpha^-$
E14.$_1$ $\mu.X_1 \mid \mu^-.Y_1 = (\mu.(X_1 \mid \mu^-.Y_1) + \mu^-.(\mu.X_1 \mid Y_1)) + \tau.(X_1 \mid Y_1)$
E14.$_2$ $\mu^-.X_1 \mid \mu.Y_1 = (\mu^-.(X_1 \mid \mu.Y_1) + \mu.(\mu^-.X_1 \mid Y_1)) + \tau.(X_1 \mid Y_1)$
E14.$_3$ $\mu.X_1 \mid \nu.Y_1 = \mu.(X_1 \mid \nu.Y_1) + \nu.(\mu.X_1 \mid Y_1)$ if $\mu \neq \nu^-,\ \nu \neq \mu^-$

Hence, the equivalences defining relabelling, restriction and parallel composition in terms of NIL, prefix and summation are E8.- E12., E13.$_i$ $i = 1,2,3$ and E14.$_j$ $j = 1,..,6$, still called $Eq_{Rewrite}$. We have the following result :

Prop.4.2.1

The rewriting system $R_{Rewrite}$ obtained by directing the equivalences $Eq_{Rewrite}$ from left to rigth is canonical.

We can prove this proposition by applying the completion algorithm to $Eq_{Rewrite}$ and this result permits to assert that rewriting according to the rules of $R_{Rewrite}$ is *complete* wrt equational deduction in $Eq_{Rewrite}$.

4.2.2. Translation of $R_{Rewrite}$ into Horn clauses

Let us consider the translation into Horn clauses of the canonical rewriting system $R_{Rewrite}$ equivalent to the subtheory $Eq_{Rewrite}$.

4.2.2.1. The flattening procedure

In this section we shortly introduce flattening.

Given an equational theory E and an equivalent (canonical) rewriting system R, an E-unification algorithm based on narrowing allows to compute the complete set of E-unifiers of two terms t_1, t_2 ([HO80], [Hul80]). Since the narrowing relation includes the rewriting relation, the algorithm based on narrowing can be also used for rewriting a term into its normal form according to R. On the other hand, flattening allows SLD-resolution to simulate narrowing ([BGM87]), thus allowing a term to be reduced to normal form according to R by using Horn clauses.

Flattening is applied to a canonical rewriting system R describing an equational theory E, thus deriving a logic program R_{flat}, and to every equational goal $t_1 = t_2$. In [BGM87] it has been shown that SLD-resolution of the flattened version of $t_1 = t_2$ in $R_{flat} \cup \{x = x\}$ is correct and complete wrt the E-unification algorithm based on narrowing.

The version of the flattening procedure we use is that one in [BGM87] with a further test distinguishing terms formed only by data constructors, for which we don't introduce new variables. Below it is its procedural presentation.

```
flat ($\gamma \rightarrow \delta$) = if $\delta$ is a variable (also occurring in $\gamma$) or $\delta$ is a data constructors term
                    then $\gamma = \delta$
                    else $\gamma = z$ :- flatterm ($\delta$, z)                /* z is a new variable */
flatgoal (M = N) = flatterm (M, z), flatterm (N, z)
flatterm (f ($M_1$, ..., $M_n$), z) = flatterm ($M_{i_1}$, $x_1$), ..., flatterm ($M_{i_q}$, $x_q$), f ($z_1$, ..., $z_n$) = z
                                   where $M_{i_1}$, ..., $M_{i_q}$ are non-variable arguments of f
                                          and   $z_i = M_i$     if $M_i$ is a variable
                                                $z_i = x_k$     if $M_i$ is $M_{i_k}$ .
```

4.2.2.2. Translation of $R_{Rewrite}$

Let P be a term to rewrite and eq_rewr (X, Y) a binary predicate meaning "X rewrites as Y", by which we translate the rewriting rules. In order to obtain the normal form $F_n \in T_\Sigma$ of P (according to $R_{Rewrite}$), the rules of $R_{Rewrite}$ must be applied to every reducible subterm of P. Thus a flattening procedure must be applied to $R_{Rewrite}$ and P.

Flattening is applied to $R_{Rewrite}$ once and for all; below the *flat version of the rules* according to the Quintus Prolog syntax is reported. The term P to rewrite must, instead, be flattened *dynamically*, i.e. each time a process has to be reduced. We implement the subprocedure *flatterm* in the flattening procedure with a binary predicate *rewrite*, which is defined by cases on the outermost term operator; it works as flatterm and has the same arguments.
Goals are like :- rewrite (P, F_n) .

The logic program *first_phase* (reported below) implementing the 1st phase of the reduction of a finite CCS process is obtained by the definitions of eq_rewr, apply and rewrite.

```
eq_rewr (relab (nil, _ ), nil) .
eq_rewr (relab (sum (X, Y), S), Z)  :-  eq_rewr (relab (X, S), Z1) ,  eq_rewr (relab (Y, S), Z2) ,
                                        eq_rewr (sum (Z1, Z2), Z) .
eq_rewr (relab (prefix (Act, X), S), Z)  :-  apply (S, Act, Act1) ,  eq_rewr (relab (X, S), Z1) ,
                                             eq_rewr (prefix (Act1, Z1), Z) .
eq_rewr (restr (nil, _ ), nil) .
eq_rewr (restr (sum (X, Y), Act), Z)  :-  eq_rewr (restr (X, Act), Z1) ,  eq_rewr (restr (Y, Act), Z2) ,
                                          eq_rewr (sum (Z1, Z2), Z) .
eq_rewr (restr (prefix (Act, _ ), Act), nil) .
eq_rewr (restr (prefix (compl (Act), _ ), Act), nil) .
eq_rewr (restr (prefix (Act, X), Act1), Z)  :-  Act \== Act1 ,  Act \== compl (Act1) ,
                                                eq_rewr (restr (X, Act1), Z1) ,  eq_rewr (prefix (Act, Z1), Z) .
eq_rewr (par (prefix (Act, X), prefix (compl (Act), Y)), Z)  :-
            eq_rewr (par (X, prefix (compl (Act), Y)), Z1) ,  eq_rewr (par (prefix (Act, X), Y), Z2) ,
            eq_rewr (sum (prefix (Act, Z1), prefix (compl (Act), Z2)), Z3) ,
            eq_rewr (par (X, Y), Z4) ,  eq_rewr (sum (Z3, prefix (tau, Z4)), Z) .
eq_rewr (par (prefix (compl (Act), X), prefix (Act, Y)), Z)  :-
            eq_rewr (par (X, prefix (Act, Y)), Z1) ,  eq_rewr (par (prefix (compl (Act), X), Y), Z2) ,
            eq_rewr (sum (prefix (compl (Act), Z1), prefix (Act, Z2)), Z3) ,
            eq_rewr (par (X, Y), Z4) ,  eq_rewr (sum (Z3, prefix (tau, Z4)), Z) .
eq_rewr (par (prefix (Act1, X), prefix (Act2, Y)), Z)  :-
            Act1 \== compl (Act2) ,  Act2 \== compl (Act1) ,
            eq_rewr (par (X, prefix (Act2, Y)), Z1) ,  eq_rewr (par (prefix (Act1, X), Y), Z2) ,
            eq_rewr (sum (prefix (Act1, Z1), prefix (Act2, Z2)), Z) .
eq_rewr (par (nil, prefix (Act, X)), prefix (Act, X)) .
eq_rewr (par (prefix (Act, X), nil), prefix (Act, X)) .
eq_rewr (par (nil, nil), nil) .
eq_rewr (X, X) .
```

```
apply ([ ], Act, Act) .
apply ([Act / Act1 | _], Act1, Act) .
apply ([Act / Act1 | _], compl (Act1), compl (Act)) .
apply ([_ / Act2 | S], Act1, Act)  :-  Act1 \== Act2 ,  Act1 \== compl (Act2) ,  apply (S, Act1, Act) .

rewrite (nil, nil) .
rewrite (prefix (Act, Pr1), Pr)  :-  rewrite (Pr1, X) ,  eq_rewr (prefix (Act, X), Pr) .
rewrite (sum (Pr1, Pr2), Pr)  :-  rewrite (Pr1, X1) ,  rewrite (Pr2, X2) ,  eq_rewr (sum (X1, X2), Pr) .
rewrite (relab (Pr1, S), Pr)  :-  rewrite (Pr1, X1) ,  eq_rewr (relab (X1, S), Pr) .
rewrite (restr (Pr1, Act), Pr)  :-  rewrite (Pr1, X1) ,  eq_rewr (restr (X1, Act), Pr) .
rewrite (par (Pr1, Pr2), Pr)  :-  rewrite (Pr1, X1) ,  rewrite (Pr2, X2) ,  eq_rewr (par (X1, X2), Pr) .
```

Prop.4.2.2

Given the logic program first_phase, let P be a process and G a goal :- rewrite (P, Fn).
The process P', returned in Fn by the answer substitution corresponding to the leftmost path of the SLD-tree of first_phase ∪ { :- rewrite (P, Fn) }, is the normal form of P according to $Eq_{Rewrite}$ (*correctness*).
Vice versa, if the process P has normal form P' according to $Eq_{Rewrite}$, P' is returned in Fn by the answer substitution corresponding to the leftmost path of the SLD-tree of first_phase ∪ { :- rewrite (P, Fn) } (*completeness*).

Proof

We assume the standard Prolog computation rule (selecting the leftmost atom of goal) and depth-first strategy.
The transformation due to flattening is correct and complete ([BGM87]) and the predicate rewrite (P, F_n) is equivalent to flatgoal (P = F_n).
Because of flattening each time the rules of $R_{Rewrite}$ are applied to solve a predicate eq_rewr (op (X_1, X_2), X_k), the arguments X_1, X_2 are in reduced form and, since the clauses of eq_rewr are mutually exclusive (except with identity eq_rewr (X, X)), at most only one rule of $R_{Rewrite}$ is applicable (besides identity, always applicable).
Since identity is the last clause of the definition of eq_rewr, the SLD-tree built and visited according to the fixed strategy corresponding to the goal :- rewrite (P, F_n), has at most two branches leaving from each node (corresponding to a goal like :- eq_rewr (op (X_1, X_2), X_k) , ..., g_n), where the branch on the left corresponds to the application of the only applicable rule of $R_{Rewrite}$ (if it exists) and the one on the right corresponds to the application of identity (it always exists).
SLD-resolution and therefore rewriting terminates when :

i) the goal is like :- eq_rewr (P', F_n)

ii) the only applicable clause is identity.

It follows, by construction, that the process P', returned in F_n by the answer substitution corresponding to the leftmost path of the SLD-tree of first_phase ∪ { :- rewrite (P, F_n) }, is the normal form of P (*correctness*).
Analogously, by construction, it holds that for every process P with normal form P' (according to $Eq_{Rewrite}$), P' is returned in F_n by the answer substitution corresponding to the leftmost path of the SLD-tree of
first_phase ∪ { :- rewrite (P, F_n) } (*completeness*).

4.3. The 2nd phase of reduction : application of the equivalences $Eq_{sum+\tau}$

Let us consider the 2nd phase of term reduction, in which the equivalences $Eq_{sum+\tau}$ are applied to the process $P' \in T_{\Sigma}$ returned by the 1st phase. Because of the problems arising in applying $Eq_{sum+\tau}$ to the symbolic term representation, we adopt another kind of representation for a CCS term of T_{Σ}. We choose to use the *action-tree* of a process, because it is characterized by lack of parenthesis and order : a different order of summands of a term means a different order of the subtrees corresponding to these summands, but no particular meaning is given to subtrees order and therefore no problem of order arises in applying the rules obtained by directing $Eq_{sum+\tau}$. For applying a rule r we have to visit the action-tree and verify the existence of the subtrees corresponding to the subterms of the left side of r, indipendently from their position both in the action-tree (as regards subtrees) and in the left side of r (as regards subterms). If this verification succeeds, we can apply the reduction defined by r by deleting a subtree or a branch according to r. Hence, the equivalences $Eq_{sum+\tau}$ become *equivalences between action-trees*, called Eq_{tree}, allowing to reduce an action-tree t' to another equivalent t_{fn}.

The 2nd phase of reduction to normal form of a term P divides into the following 3 steps :

1. given the process P' output of the 1st phase of reduction, P' is represented by means of its action-tree t';
2. t' is reduced according to the equivalences Eq_{tree} obtaining a reduced action-tree t_{fn} ;
3. by the opposite transformation, the process P_{fn} , *a* normal form of P according to the full observational theory OBS.EQ., is derived from t_{fn}.

4.3.1. Implementation of step 1. of the 2nd phase

Given a process $P' = \sum \mu_i.P_i$, we implement its action-tree, having n subtrees, by means of a list of n records $\{\mu_i, L_i\}$ with 2 fields :

- the 1st field is the *action* μ_i prefixed to the term P_i;
- the 2nd field is the *list* L_i implementing the subtree t_i, action-tree of P_i .

$$\text{impl}\ (t) = [\{\mu_1, L_1\}, ..., \{\mu_i, L_i\}, ..., \{\mu_n, L_n\}]\ .$$

The *transformation* of a process $P' \in T_{\Sigma}$ *from its symbolic representation into its action-tree* is defined by an appropriate bynary predicate *tree* which, given a term in T_{Σ}, builds the list implementing the action-tree of the process.

4.3.2. Implementation of step 2. of the 2nd phase

Note that the application of the commutative property on an action-tree simply changes the order of its subtrees, while the application of the associative property leaves the action-tree unchanged. Hence, Eq_{tree} becomes:

$$Eq_{tree} = \{X + X = X\ ,\ X + \tau.X = \tau.X\ ,\ \mu.\tau.X = \mu.X\ ,\ \mu.(X + \tau.Y) = \mu.(X + \tau.Y) + \mu Y\}.$$

The *associative property* and the *existence of a neuter element for summation* are deleted because:

– the application of the associative property has no effect on an action-tree ;
– $X + NIL = X$ is implicitly applied during step 1. of the 2nd phase.

The *commutative property* isn't in Eq_{tree} (in the clauses translating the reduction according to Eq_{tree} there will be no clause for the commutative property), but its application is simulated by means of a test of equality modulo the summands order.

The equality of terms modulo the summands order introduces the notion of *sumcongruence.* It has been proved in [Mil84a] and [HM85] that every finite CCS term has a normal form according to the observational theory OBS.EQ., normal form unique modulo the summands order. Different normal forms of the same term are called *sumcongruent.*

Since we are using action-trees to represent processes, we define the notion of sumcongruence for action-trees : note that the *sumcongruence between action-trees is defined only in terms of the commutative property* because the application of the associative property has no effect on action-trees.

Def.4.3.1 (sumcongruence of action-trees)

Two action-trees t_1, t_2 are sumcongruent, $t_1 \approx_S t_2$, if it is possible to derive $t_1 = t_2$ by applying the commutative property, i.e. by performing appropriate exchanges of subtrees in t_1 and t_2 .

Sumcongruence, i.e. *equality of trees modulo their subtrees order*, is obtained by verifying that the subtrees of an action-tree are also subtrees of the other one and vice versa, independently from the order; this is done using a predicate *sumcongr.*

By applying the completion algorithm to Eq_{tree}, we can show that an equivalent rewriting system R is the following :

$R = \{X + X \rightarrow X , X + \tau.X \rightarrow \tau.X , \mu.\tau.X \rightarrow \tau.X , \mu.(X + \tau.Y) + \mu.Y \rightarrow \mu.(X + \tau.Y)\}.$

The clauses translating the rules of R are, thus, the following :

```
equiv ([X | L], [X | L1]) :- member_el (X1, L) , sumcongr ([X], [X1]) , delete_el (X1, L, L1) .
equiv (L, L1) :- member_list (Lx, L) , member_el ({tau, Lx1}, L) , sumcongr (Lx, Lx1) , delete_list (Lx, L, L1) .
equiv ([{Act, [{tau, L}]}], [{Act, L}]) .
equiv (L, L1) :- member_el ({Act, Lxy}, L) , member_el ({Act, Ly1}, L) , member_list (Lx, Lxy) ,
                 member_el ({tau, Ly}, Lxy) , sumcongr (Ly, Ly1) , delete_list (Lx, Lxy, L2) ,
                 delete_el ({tau, Ly}, L2, [ ]) , delete_el ({Act, Ly1}, L, L1) .
equiv (L, L) .

reduce ([ ], [ ]) .
reduce ([{Act, L}], X) :- reduce (L, L1) , equiv ([{Act, L1}], X) .
reduce ([X | L], L2) :- L\==[ ] , reduce ([X], [X1]) , reduce (L, L1) , equiv ([X1 | L1], L2) .
```

The identity clause equiv (L, L) is applied when there are no more reductions; *member* and *delete* are auxiliary predicates for elements and for lists. The predicate *reduce* implements flattening on the action-tree ensuring the application of reductions to every reducible subtree.

Obs.1

Except for the clause translating the rule $\mu.\tau.X \rightarrow \mu.X$, in the previous logic program we can't take advantage of the *power of unification between goal and clause head*. That is due to the representation which abstracts from subterms order, therefore in order to apply reductions we don't search for an exact pattern to unify, but we must examine the components of such pattern in the action-tree by means of tests in the clause body . That also explains why we don't apply flattening to the rules of R.

Obs.2

Flattening applied to the action-tree t', which has to be reduced, implies a **bottom-up visit** of the tree, i.e. *reductions are executed starting from the leaves up to the root and from left to rigtht.*

4.3.3. Implementation of step 3. of the 2nd phase

Step 3. consists of the opposite transformation : the action-tree t_{fn}, obtained by means of reductions at step **2.**, is transformed into the process it represents by a binary predicate *term*, whose first argument is a list, while the second one is a term of T_{Σ}.

Going back to the symbolic term representation with summation defined as binary operator, it is necessary to establish its associativity, i.e. parenthesis position. By following Prolog list definition, we assume the associativity to right and therefore we choose a determinate representation of normal forms modulo the associative property.

4.3.4. Normal form of a process

The three steps of the 2nd phase, previously examined, are packed together in the following clause that defines the predicate *fn* :

```
fn (P, Pfn) :-  tree (P, P_tree) ,  reduce (P_tree, Tmin) ,  term (Tmin, Pfn) .     (**)
```

Thus issuing the goal :- fn (P', Pfn) allows a reduced form of the process P', according to the 2nd phase of reduction, to be returned in Pfn.
We call *second_phase* all logic programs implementing the 2nd phase of process reduction.
As usual we assume an SLD-tree according to standard Prolog computation rule and depth-first strategy.

Prop. 4.3.1

Let $P' \in T_{\Sigma}$ be a process. The answer substitution related to the leftmost path of the SLD-tree corresponding to second_phase $\cup$ {(**)} $\cup$ { :- fn (P', Pfn)} returns in the variable Pfn *a* normal form of P' according to Eq_{tree}.

Proof

Resolution of the goal :- fn (P', Pfn) implies, for (**), the resolution of the predicates tree, reduce and term. Tree and term are deterministic, therefore the result of their resolution is unique (only one successful path). Reduce

implements flattening as rewrite (which is correct and complete, section 4.2.2) and applies reductions by means of the predicate equiv. Because of flattening the subtrees corresponding to the subterms arg_1, arg_2 of op (arg_1, arg_2), whose action-tree is currently reduced, are always in normal form.
The four rules of R are mutually exclusive, therefore on the action-tree corresponding to op (arg_1, arg_2) at most one is applicable, besides identity equiv (L, L) , last clause of equiv and always applicable. It follows that along the leftmost path in the SLD-tree of second_phase $\cup$ {(**)} $\cup$ { :- fn (P', Pfn)} the applicable reductions are applied, otherwise identity is applied.
When there exists no more rule to apply (the action-tree is in normal form), the leftmost path is closed by applying identity, thus returning in Tmin an action-tree in normal form according to Eq_{tree}; the other paths, where identity is applied when it is still possible to reduce, return action-trees not reduced in normal form.

Therefore, given a process P, a normal form Pfn is computed by solving the goal

:- normal_form (P, Pfn)

where *normal_form* is the predicate applying in sequence the two phases of reduction on P according to the following definition :

```
normal_form (P, Pfn) :- rewrite (P, P') , fn (P', Pfn) .
```

4.3.5. Observational congruence of processes

In order to decide the observational congruence of two processes P_1, P_2 we can apply reduction to both of them and, after having obtained the action-trees t_{fn1}, t_{fn2} in normal form according to Eq_{tree}, we can verify their equality modulo subtrees order, i.e. their sumcongruence, according to the following definition :

Def.4.3.2 (observational congruence of action-trees)
Two action-trees t_1, t_2 corresponding to the terms $P_1, P_2 \in T_\Sigma$ are *observationally congruent,*

$t_1 \approx_c t_2$, if there exist action-trees t_{fn1}, t_{fn2} such that :

i) $t_1 \xrightarrow[Eq\ tree]{*} t_{fn1}$ and $t_2 \xrightarrow[Eq\ tree]{*} t_{fn2}$

ii) t_{fn1} and t_{fn2} are in normal form according to Eq_{tree}

iii) $t_{fn1} \approx_s t_{fn2}$.

Hence, step **3.** of the 2nd phase of reduction is replaced by a test on sumcongruence of the two action-trees obtained. Step **3.** becomes :
3'. verification of sumcongruence of t_{fn1} and t_{fn2} .

Step **3'.** is translated into clauses by the predicate *osscongr*, which, given two processes P_1 and P_2 as inputs, returns "yes" if the processes are observationally congruent according to the following definition :

```
osscongr (P1, P2) :- rewrite (P1, X1) , rewrite (P2, X2) , tree (X1, T1) , tree (X2, T2) ,
                     reduce (T1, Tfn1) , reduce (T2, Tfn2) , sumcongr (Tfn1, Tfn2) .
```

The definitions of osscongr and normal_form can be put together in order to create a logic program *reduction* which, together with the other programs first_phase and second_phase, starts reduction by solving goals

```
 :- osscongr (P1, P2)
```

or

```
 :- normal_form (P, Pfn)
```

if we want to verify the observational congruence of two processes or compute a normal form of a process according to OBS.EQ.

5. Observational equivalences of finite CCS and recursion : a verification method for observational equivalence of processes

In this section we show how it is possible to use the so far defined theories to build a specific verification method. Such a method will, then, be applied to an interesting case, the scheduling problem.
The method is based on a proof technique introduced in [Mil80] and it allows recursion in CCS processes to be managed. To a certain extent the method is an example of the way a user can build his own strategy also in the direction of providing functionalities not supported by the kernel system. In this case it introduces the ability to manage recursive processes, without having provided any rule for recursion either in OPSEM and in OBS.EQ. As a matter of fact, the strategy we present realizes at user level something very similar, in the essence, to what has been done to extend the kernel theories of finite CCS to CCS with (bounded) recursion.

5.1. The verification method

Let E1, E2 be two expressions. The **verification method** verifies the observational congruence E1 = E2 trying to derive E2 from E1 : it applies, preserving the congruence at each step, steps of reduction modulo OBS.EQ. and steps of replacement of subexpressions with identifiers, by which it tries to manage recursion in order to prevent infinite rewritings.

The strategy adopted in [Mil80] is the following : E1 is rewritten by replacing each identifier X with the expression E bound to X by the rec operator (one step unfolding) and then E1 is reduced according to the 1st subtheory (notably using the expansion theorem, section 2) and the 2nd subtheory. After each step of reduction it is checked if the intermediate expression, into which E1 is rewritten, contains subexpressions bound to identifiers or already computed in previous steps. If it is the case, each of such subexpressions is replaced with the corresponding identifier (backward replacement). If E2 is derived, then E1 and E2 are observationally congruent; otherwise a new step of recursion is applied by replacing identifiers with bound expressions.

The verification method based on this strategy is *correct.*

5.2. Implementation of the verification method by Horn clauses

The verification method is realized by using OPSEM and finite CCS OBS.EQ. We rely on the following implementation assumptions:

– Every recursive expression rec X.E is equivalently defined by a declaration X = E and every occurrence of rec X.E in other expressions is replaced by X. Therefore the rec operator is replaced by the id operator. The binding between X and E is now established by the declaration X = E : it introduces a notion of *environment*, containing only couples <identifiers X, expressions E> denoting rec X.E.

– Because the implementation of the equivalences corresponding to parallel composition in the case of general structure of processes in parallel is heavy, our implementation replaces reduction according to the 1st subtheory, notably the application of the expansion theorem, by the execution of OPSEM, whose logic program is independent of process structure, thus performing an interaction between OPSEM and OBS.EQ. What we loose is the ability to treat *process termination*: OPSEM is incomplete wrt process termination (section 3.1), thus making incomplete our implementation. This incompleteness can be simply recovered implementing the expansion theorem.

The strategy we implement resembles Milner's strategy :
step 1. E1 is rewritten as a summation, Esum, of the expressions obtained by prefixing the first actions E1 can execute, to the expressions into which it evolves under such actions, thus simulating a step of its execution.
step 2. We verify if Esum contains subexpressions bound to an identifier in the environment. If there exist such subexpressions, each of them is replaced by the bound identifier (backward replacement) obtaining an expression Eint, which is bound to the identifier E1 in the environment.
step 3. Eint is reduced modulo the equivalences of the 2nd subtheory of finite CCS $\mathrm{Eq}_{\mathrm{tree}}$. If Eint and E2 are congruent, verification succeeds, otherwise the method proceeds by repeating the previous steps from Eint, until there exist executable actions without replacing an identifier already replaced.
step 4. Having obtained an expression Eint1 not congruent to E2 and from which further actions are not executable, a new step of recursion is applied by repeating the previous steps from Eint1.

The implemented strategy is correct.

Prop. (correctness)
Let E1, E2 be two expressions. If the steps 1.- 4. above (applied to E1 and E2) succeed, then E1 and E2 are observationally congruent.

In fact, each step of the method rewrites an expression E into an expression E' preserving congruence between E and E', whether E is rewritten by simulating a step of its execution, or some of its subexpressions are replaced by the identifiers bound in the environment or E is reduced modulo the equivalences of the 2nd subtheory.

A data structure (a list of records) is used for representing the environment, in which observational congruence has to be proved, while the related operations are implemented by appropriate predicates.

The clauses translating OPSEM are slightly modified in order to deal with the environment, by introducing it as input and as output parameter of the new predicate trans, thus obtaining a new logic program called *o.s._env*.

Note that, to preserve modularity we could also decide not to modify the clauses which translate OPSEM, operating at meta-level, that is writing a metaprogram which takes into account the environment and then calls the original predicate trans in the object theory. We have seen an example of this technique in sec. 3.1, and it is extensively used in [Ste85]. For simplicity of presentation we preferred to remain at object level; the final environment, anyhow, will work at metalevel.

Here we report only the clauses implementing the strategy :

```
oss_eq (E1, E2, Env)  :-  deriv (E1, I, Env) ,  verify (E1, I, Env, Eint, Env1) , equival (Eint, E2, E1, I, Env1) .
deriv (E, I, Env)  :-  bagof ({Act1, E1, Env1}, trans (E, Act1, E1, Env, Env1), I) .
deriv (E, I, Env)  :-  \+ trans (E, _, _, Env, _) ,  re_init (Env, Env1) ,  deriv (E, I, Env1) .
verify (E1, I, Env, Eint, Env1)  :-  costr_sum (I, Esum) ,  pattern (Esum, Eint, Env) , bind (E1, Eint, Env, Env1) .
equival (E1, E2, _, _, _)  :-  tree (E1, T1),  tree(E2, T2),  reduce (T1, T1min),  reduce (T2, T2min), sumcongr
(T1min, T2min) .
equival (_, E2, E1, I, Env) :- esec (E1, I, L, Env), verify (E1, L, Env, Eint, Env1), equival (Eint, E2, E1, L, Env1).
esec (E, [{Act1, [{Act2, I2, Env2} | L2], Env1} | L1], [{Act1, Lres, Env1} | Lres1], Env)  :-
                                        esec (E, [{Act2, I2, Env2} | L2], Lres, Env),  esec (E, L1, Lres1, Env) .
esec (E, [{Act1, E1, Env1} | L1], [{Act1, I1, Env1} | L2], Env)  :-  deriv (E1, I1, Env1), esec (E, L1, L2, Env) .
esec (_, [ ], [ ]) .
```

Let Dec be a sequence of declarations, E1 an identifier and E2 an expression : we want to verify the observational congruence of E1 and E2 in the environment defined by Dec. The goal is :

```
:-  input (Dec, E1, E2)
```

and it is solved by the following clause for *input*, that builds the initial environment defined by Dec and starts verification :

```
input (Dec, E1, E2)  :-  env (Dec, Env) ,  oss_eq (E1, E2, Env) .
```

Comments

step1 :

oss_eq (E1, E2, Env) is solved by computing transitions from E1 by means of the Quintus Prolog predicate *deriv* and E1 is rewritten as summation of the executable actions Act_i prefixed to the reachable expressions E_i.

step2 :

The predicate *verify* verifies if the expression obtained by rewriting E1 as summation of $Act_i.E_i$ can be rewritten by replacing some of its subexpressions by identifiers bound to them in Env.

step3 :

The predicate *equival* verifies if Eint is observational congruent to E2 by transforming both expressions into their action-trees and then reducing them modulo subtheory Eq_{tree}. In order to execute this reduction the program second_phase is used. If the expressions aren't congruent, the 1st clause of equival fails and the previous steps are repeated from Eint by using the 2nd clause.

step4 :

When there are no more transitions, the 1st clause of deriv fails and a new step of recursion is applied by using the 2nd clause, starting again to compute new transitions by means of deriv in the initial environment.

We call *verification* the program formed by the clauses implementing the strategy and the clauses for operations on environment.

5.3. An example: the scheduling problem

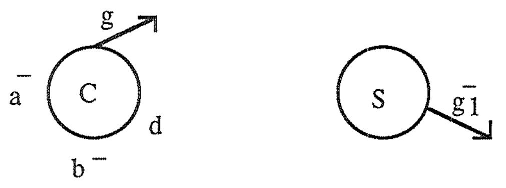
Let us consider an example of application of the verification method, the scheduling problem proposed in [Mil80].

Let P_1 and P_2 be two processes performing a certain task repeatedly. We want to design a scheduler to ensure that P_1 and P_2 perform the task in rotation, starting with P_1. We assume that P_1 and P_2 begin the execution of task in rotation, without constraining their performances to exclude each other in time, while each process cannot initiate its task before completing its previous execution.

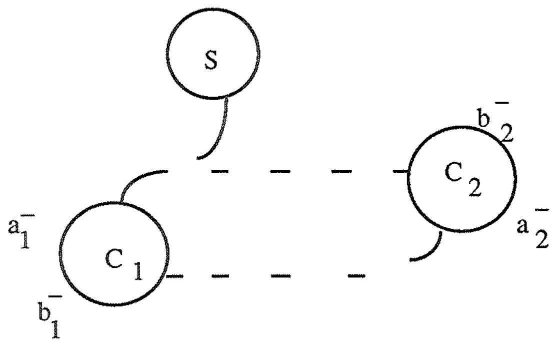
Let us suppose that P_i requests initiation by executing the action $a_i \in \Lambda$ and signals completion with the action $b_i \in \Lambda$, $i = 1, 2$. Let A be $\{a_1, a_2\}$ and B $\{b_1, b_2\}$. The scheduler Sch communicates with P_1 and P_2 by means of channels of sort a_i, b_i, therefore Sch has sort $A^- \cup B^- = \{a_1^-, a_2^-, b_1^-, b_2^-\}$.

The scheduler is obtained linking 2 elementary identical components (because we consider 2 processes), called *cyclers*, and a component *starter* starting the scheduler.

A generic cycler C and the starter S can be represented graphically as it follows :



Our *scheduler* Sch for processes P_1 and P_2 is represented as it follows :



where cyclers C_1 and C_2 are obtained by relabelling generic cycler C in an appropriate way :

$C_1 = C[a_1/a, b_1/b, g_1/g, g_2^-/d]$

$C_2 = C[a_2/a, b_2/b, g_2/g, g_1^-/d]$

and Sch is given by

$Sch = (S \mid C_1 \mid C_2)\backslash g_1\backslash g_2$

i.e. starter and cyclers are composed in parallel, communicate by using channels of sort g_1, g_2, which are then hidden by restriction (this last operation hides the actions g_1, g_1^-, g_2, g_2^- in the graph of Sch).

Let us describe the behaviour of the components of Sch :

– the starter S starts the scheduler by enabling the 1st cycler C_1 by the action g_1^- and then terminates. Therefore $S = g_1^-.NIL$.

– each cycler C works repeatedly :

1. it is enabled by the predecessor cycler by means of syncronization on the channel of sort g (at the beginning it is the starter to enable the successor cycler, but always using the same channel of sort g);

2. it receives initiation request from P by executing a^-;

3. it receives termination signal from P by executing b^- and then it enables the successor cycler by the action d or executes these two actions in the reverse order; then goto step 1.

Hence, a generic cycler C is defined as :

$C = g.a^-.(b^-.d.C + d.b^-.C)$.

We want to prove that P_1 and P_2 begin their task in rotation from P_1. We express this constraint by working on sequences of actions a, b, by which processes request initiation and signal completion to the scheduler. This constraint on rotation is satisfied if, given any sequence of actions in $(A \cup B)^*$, deleting all occurrences of b_1, b_2 (signaling task completion) the sequence becomes $(a_1.a_2)^*$, where a_1 before a_2 expresses rotation from P_1. In other words we can also say that, given Sch and *absorbing* all syncronizations of sort b, the result is observationally equivalent to the sequence $(a_1^-.a_2^-)^*$.

Let s be a non-empty sequence of Λ^*, s* is the behaviour given by $s^* = s.(s^*)$.

Let t be a term with sort L and $a \in L$, the action a is absorbed by obtaining the new term $(t \mid a^*)\backslash a$.

If we consider Sch and absorbe all actions b_1^-, b_2^- by syncronizations of sort b, all the observable actions are a_1^-, a_2^- in sequence. We prove that this sequence is observational equivalent to $(a_1^-.a_2^-)^*$, where the latter expresses the fact that P_1 initiates rotation. Hence, we prove :

$$(Sch \mid (b_1^* \mid b_2^*))\backslash b_1\backslash b_2 \;\approx\; (a_1^-.a_2^-)^* .$$

In this case we are interested in proving observational equivalence and not congruence, therefore we add the equivalence $\tau.X \approx X$, distinguishing observational equivalence from congruence, to the theory Eq_{tree}, obtaining a new theory *Oss_eq* and a new program *oss_eq* by adding the following clause to second_phase :

```
equiv ([{tau, L}], L)
```

where the list L implements the action-tree of the process X.

Let us introduce a new identifier Sch' and the declaration

$Sch' = (Sch \mid (b_1^* \mid b_2^*))\backslash b_1\backslash b_2$.

We prove that Sch' satisfies the definition equation of $(a_1^-a_2^-)^*$, i.e.

$Sch' \approx a_1^-.a_2^-.Sch'$ (*)

Before applying the verification method, following [Mil80], we rewrite Sch' by means of some properties of CCS operators :

$$\begin{aligned}Sch' &= (Sch \mid (b_1^* \mid b_2^*))\backslash b_1\backslash b_2\\ &= ((S \mid C_1 \mid C_2)\backslash g_1\backslash g_2 \mid (b_1^* \mid b_2^*))\backslash b_1\backslash b_2\\ &= ((S \mid C_1 \mid C_2) \mid (b_1^* \mid b_2^*)\backslash g_1\backslash g_2)\backslash b_1\backslash b_2\\ &= (S \mid (C_1 \mid b_1^*) \mid (C_2 \mid b_2^*))\backslash g_1\backslash g_2)\backslash b_1\backslash b_2\\ &= (S \mid (C_1 \mid b_1^*) \mid (C_2 \mid b_2^*))\backslash g_1\backslash g_2\backslash b_1\backslash b_2\\ &= (S \mid (C_1 \mid b_1^*) \mid (C_2 \mid b_2^*))\backslash b_1\backslash b_2\backslash g_1\backslash g_2\\ &= (S \mid (C_1 \mid b_1^*)\backslash b_1 \mid (C_2 \mid b_2^*)\backslash b_2)\backslash g_1\backslash g_2\end{aligned}$$

and by introducing two new identifiers C_i' with declarations $C_i' = (C_i \mid b_i^*)\backslash b_i$ $(i = 1, 2)$

(where the expression $(C_i \mid b_i^*)\backslash b_i$ represents the cycler C_i with b_i^- absorbed) we obtain

$Sch' = (S \mid C_1' \mid C_2')\backslash g_1\backslash g_2$.

Verification of (*) is developed in two steps : the 1st one proves that C_i' is observational equivalent to an expression containig C_i' itself, while the 2nd step, by considering the result of the 1st one, proves (*).

The 1st step :

We prove the observational equivalence $C_i' \approx g_i.a_i^-.g_{i+1}^-.C_i'$, where addition on subscripts is modulo 2. Let's prove it for i = 1 :

$C_1' \approx g_1.a_1^-.g_2^-.C_1'$. (**)

The proof starts with an environment with bindings for C_1', C_1 and for a new identifier X introduced in order to define the expression b_1^* according to the declaration $X = b_1.X$.

Therefore, the following goal is solved by using the programs verification, o.s._env and oss_eq :

:- input $([C_1' = (C_1 \mid X)\backslash b1), C_1 = g_1.a_1^-.(b_1^-.g_2^-. C_1 + g_2^-.b_1^-. C_1), X = b_1.X],\ C_1',\ g_1.a_1^-.g_2^-. C_1')$.

The verification method answers "yes" and we can show the intermediate expressions into which cycler C_1' is rewritten preserving the observational equivalence :

$C_1' = g_1. ((a_1^-.(b_1^-.g_2^-. C_1 + g_2^-.b_1^-. C_1) \mid X)\backslash b_1) = g_1.a_1^-.((b_1^-.g_2^-. C_1 + g_2^-.b_1^-. C_1) \mid X)\backslash b_1) =$
$g_1.a_1^-.(g_2^-.((b_1^-. C_1 \mid X)\backslash b_1) + \tau.((g_2^-. C_1 \mid X)\backslash b_1) = g_1.a_1^-.(g_2^-.\tau. C_1' + \tau.g_2^-. C_1')$.

The last expression is reduced by τ laws and the equivalence (**) is proved.

The 2nd step :

Let's prove the equivalence (*) in an environment that contains bindings for the identifiers Sch', S, C_1', C_2', where the last two are bound to the expressions obtained by (**). The following goal

:- input ([Sch' = (S | C_1' | C_2')\$g_1$\$g_2$, S = g_1¯.NIL, C_1' = g_1.a_1¯.g_2¯.C_1', C_2' = g_2.a_2¯.g_1¯.C_2'], Sch', a_1¯.a_2¯.Sch')

is solved by the verification method answering "yes" with the following intermediate expressions :

Sch' = τ.((NIL | a_1¯.g_2¯.C_1' | C_2')\ g_1\ g_2) = τ.a_1¯.((NIL | g_2¯.C_1' | C_2')\ g_1\ g_2) =

τ.a_1¯.τ.((NIL | C_1' | a_2¯.g_1¯.C_2')\ g_1\ g_2) = τ.a_1¯.τ.a_2¯.((NIL | C_1' | g_1¯.C_2')\ g_1\ g_2) = τ.a_1¯.τ.a_2¯.Sch'

The method applies τ laws to the last expression in order to delete both τ actions and prove (*).

7. Conclusions and extensions

In this paper we have presented our approach to the implementation of a kernel system to support execution and verification of CCS processes. The possibility of executing formal specification has received great attention in the last time. In fact, the execution of a formal specification can be seen as a first form of verification; when considering concurrent specification such possibility becomes more and more important being the inherent complexity of the specification greater. Furthermore no agreement nor experience on the kind of verification or validation a user would like to perform on his specification has been clearly established.

Thus, in proposing our system, we have followed two main criteria: on the one hand, we wanted to provide the possibility of executing either the operational semantics as well as behavioural equivalences of CCS processes, guaranteeing as much as possible the correctness of our implementation; on the other hand, we wanted to provide a flexible and open-ended system, in which tools but not policies to perform verification were provided. To this respect the equational approach to the execution of behavioural equivalences seemed to us the most promising one.

Our aim is, in fact, to provide the user with the possibility of defining and using his own verification strategies. The division of the observational theory for finite CCS into more subtheories (but the approach is intended to be extended to other theories like testing equivalence as well), besides the interaction among such subtheories and OPSEM, is a first step towards an environment *of theories* where a user should be able to use them according to *different meta-level-defined strategies*, instead of implementing them as an object-level program like in the program verification.

With respect to *facilities*, our kernel can be compared with already consolidated systems, like the Concurrency Workbench Prototype for CCS ([Par86]) and we see that all its facilities can be also obtained in our kernel, issuing appropriate goals.

Future work consists in :

- defining a *meta-environment* and a *meta-language* allowing the user to define and use his own verification strategies by applying different theories in a modular and flexible way;
- extending the kernel in order to treat other behavioural equivalences, like *testing equivalence*, for which it has been shown a correct and complete system of axioms and possible extension to general recursive CCS processes;
- providing a sophisticated *user interface* (eventually graphic) which is of prime importance for a successful utilization of the environment.

References

[BGM87] Bosco, P.G., Giovannetti, E., Moiso, C. Refined Strategies for Semantic Unification, Proc. TAPSOFT '87, LNCS 250, Vol.2, Springer-Verlag, (1987), pp.276-290.

[Bol86] Bolognesi, T. Verification of Equivalences between Finite Transition Systems, Theory and Applications, CNUCE/C.N.R.- Pisa, Internal Report nº C86-16, (December 1986).

[BS87] Bolognesi, T., Smolka, S. A. Fundamental Results for the Verification of Observational Equivalence : A Survey, Proc. IFIP TC6/WG 6.1, Zurich, North Holland, (May 1987).

[DeN85] De Nicola, R. Testing Equivalences and Fully Abstract Models for Communicating Processes, Ph.D. Thesis, University of Edinburgh, Internal Report CST-36-85, (1985).

[FGIM87] Fantechi, A., Gnesi, S., Inverardi, P., Montanari, U. An execution enviroment for the formal definition of Ada, Proc. ESEC '87, LNCS 289, (September 1987), pp.327-335.

[Gna87] Gnaedig, I. Knuth-Bendix procedure and non deterministic behavior - An example -, Proc. ICALP '87, Bulletin of the EATCS, No.32, (June 1987), pp.86-92.

[Hen86] Hennessy, M. Proving Systolic Systems Correct, ACM Transactions on Programming Languages and Systems, Vol.8, No.3, (July 1986), pp.344-387.

[HM85] Hennessy, M., Milner, R. Algebraic Laws for Nondeterminism and Concurrency, Journal of ACM, Vol.32, No.1, (1985), pp.137-161.

[HO80] Huet, G., Oppen, D.C. Equations and Rewrite Rules : A Survey, In "Formal Language Theory: Perspectives and Open Problems", Book R.V.(ed.), Academic Press, New York, (1980), pp.349-405.

[Hul80] Hullot, J.M. Canonical Forms and Unification, Proc. 5th CADE, LNCS 87, Springer-Verlag, (1980), pp.318-334.

[Hus85] Hussman, H. Unification in Conditional-Equational Theories, Proc. EUROCAL '85, LNCS 204, Vol.2, Springer-Verlag, (1985), pp.543-553.

[KS83] Kanellakis, P.C., Smolka, S.A. CCS Expressions, Finite State Processes and Three Problems of Equivalence, Dept. of Computer Science, Brown University, Providence, (February 1983).

[Llo84] Lloyd, J.W. Foundations of Logic Programming, Springer-Verlag, (1984).

[Mil80] Milner, R. A Calculus of Communicating Systems, LNCS 92, Springer-Verlag, (1980).

[Mil84a] Milner, R. Lectures on a Calculus for Communicating Systems, LNCS 197, Springer-Verlag, (1984), pp.197-220.

[Mil84b] Milner, R. A Complete Inference System for a Class of Regular Behaviours, Journal of Computer and System Sciences, Vol.28, No.3, (1984), pp.439-466.

[Mil86] Milner, R. A Complete Axiomatisation for Observational Congruence of Finite-State Behaviours, ECS - LFCS-86-8, (August 1986).

[Nes88] Nesi, M. Un approccio logico-funzionale all'esecuzione di linguaggi di specifica concorrenti (CCS) modulo equivalenze comportamentali, Tesi di Laurea, University of Pisa, (1988).

[Par86] Parrow, J. Concurrency Workbench Prototype : Operating Instructions, Internal Report, University of Edinburgh, (November 1986).

[Qui87] Quintus Prolog, User Guide and Reference Manual, Artificial Intelligence Limited, (1987).

[San82] Sannella, D.T. Semantics, Implementation and Pragmatics of Clear, A Program Specification Language, Ph.D. Thesis, Dept. of Computer Science, University of Edinburgh, (July 1982), pp.172-211.

[Ste85] Sterling, L. Expert System = Knowledge + Meta-Interpreter, Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS-84-17, (1985).

[Ver86] Vergamini, D. Verification by means of observational equivalence on automata, Rapports de Recherche, INRIA nº 501, (1986).