*Journal of Universal Computer Science, vol. 1, no. 6 (1995), 339-398*
*submitted: 20/10/94, accepted: 26/6/95, appeared: 28/6/95©Springer Pub. Co.*

# A Translation of the Pi-Calculus Into MONSTR

R. Banach
(Computer Science Dept., Manchester University, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk)

J. Balázs
(Computer Science Dept., P. J. Šafárik University, 041 54 Košice, Slovakia.
balazs@turing.upjs.sk)

G. Papadopoulos
(Computer Science Dept., University of Cyprus, Nicosia, P.O. Box 537, Cyprus.
george@jupiter.cca.ucy.cy)

**Abstract:** A translation of the π-calculus into the MONSTR graph rewriting language is described and proved correct. The translation illustrates the heavy cost in practice of faithfully implementing the communication primitive of the π-calculus and similar process calculi. It also illustrates the convenience of representing an evolving network of communicating agents directly within a graph manipulation formalism, both because the necessity to use delicate notions of bound variables and of scopes is avoided, and also because the standard model of graphs in set theory automatically yields a useful semantics for the process calculus. The correctness proof illustrates many features typically encountered in reasoning about graph rewriting systems, and particularly how serialisation techniques can be used to reorder an arbitrary execution into one having stated desirable properties.

**Key Words:** Concurrency, Pi-Calculus, Term Graph Rewriting, MONSTR, Process Networks, Simulation, Serialisability.

**Category:** D.1.3, D.3.1, F.3.2, F.4.2

## 1 INTRODUCTION

As [Aczel (1993)] has pointed out, the word "process" has very different connotations in different branches of computer science. For instance, those who study process algebra, those who work on operating systems, and those who construct systems for supporting "the business process", would hardly recognise each others' use of the word. The work in this paper may partly be seen as a comparison of notions of process from the first two of these, since in presenting a translation from the π-calculus to MONSTR, both areas may be brought into contact.

The π-calculus [Milner et al. (1992), Milner (1993a)] arose as a generalisation of CCS [Milner (1989)] to allow networks of processes to evolve dynamically. It is thus a process algebra language. MONSTR by contrast is a generalised term graph rewriting language that was used as the intermediate language for the Flagship machine. See [Banach et al. (1988), Banach and Watson (1989), Banach (1993a), Banach (1993b), Watson and Watson (1987), Watson et al. (1987), Watson et al. (1989)]. Since the machine needed a runtime system, whose implementation centred round MONSTR, the connection with operating systems emerges. (In fact MONSTR evolved as a restriction of a more general term graph rewriting language DACTL, [see Glauert et al. (1988a), Glauert et al. (1988b)], the restrictions being forced by implementation issues.)

MONSTR therefore rejoices in the virtue of having been implemented in anger for a real machine. In particular, the directed arcs of a MONSTR graph are intended to be directly modeled by pointers in a conventional store in the overwhelming majority of instances, (see [Banach (1993a)] for an exposition of exactly how). So translations of process algebra formalisms (or for that matter anything else) into MONSTR can give a reasonable idea of the practicality of the primitive notions inherent in these formalisms. In the present case we find that the atomicity and synchronisation properties inherent in the communication primitive of the $\pi$-calculus extract a heavy price in the translation. This aspect is common to all similar process algebra models such as CCS — one reason why we concentrate on the $\pi$-calculus in this paper is that the more flexible mechanisms for channel hiding and binding (compared with eg. CCS), pose no problems for a MONSTR implementation. Other features of the syntax, such as the identification of potential communications by complementary occurrences of the same channel name, free in some particular context, give rise to other sources of minor inconvenience when they interact with the rest of the syntax.

At the heart of these issues is the structure of the syntax of process algebra languages, which is patterned after the structure of the syntax of many conventional languages, and produces a strong desire to use syntax directed techniques in the theory of these systems. For stack based languages such as Pascal, this approach to the meaning of the language is particularly successful, as the denotational semantics of such languages bears out. Unfortunately, the structure of process networks is seldom closely related to the structure of the parse tree of the algebraic expression that defines them, which considerably weakens the case for exclusively pursuing syntax directed analyses. Graph theory is much more in sympathy with the structure of the typical process network, which makes a translation into a graph-based formalism even more attractive.

Ironically, presentations of process algebra, having once described the syntax and some operational semantics, are frequently awash with pictures of process networks — which are of course nothing but graphs of one kind or another. Prodigious manipulations of the syntax ensue; often demonstrating some fairly simple property of the network which could have been established on graph theoretic grounds by elementary means. In the translation presented below, many sources of intricacy residing in the standard syntactic presentation of the $\pi$-calculus, once properly understood, can be seen to correspond to elementary constructions in an appropriate category of term graphs, (though we hasten to add, we will not need to make any systematic use of categorical techniques in this paper).

Of course graph based languages also need syntax, but this is used merely as a handy notation for the standard semantic model of graphs, which is what we really have in mind all along. (One could of course contemplate non-standard models of languages for graph theory if one really longs for such exotica.) The emphasis is thus different than in process algebra: rather than starting with the syntax and then wondering what it means, we have the semantics *ab initio*.

The syntax of a graph based language tends to be rather flat — it usually does little more than list the nodes and edges of the graph in question. In the case of term graphs, some slight embellishment of this is possible because of the quasi-term structure of individual nodes which allows some nesting, but the underlying "just list 'em all" philosophy remains. The main consequence of this is that sophisticated notions of scope, or of binding, tend to be absent from such languages. This might be thought to be a great deficiency, but in fact it proves not to be so. All the jobs normally done by notions of scope inside the syntax are taken over by graph structure and by suitable notions of graph homomorphism. These are described at the meta-syntactic level and act directly on the semantic objects of interest. Of course for this to work, we need to know what the semantic objects of interest are — but we have already said that we have the semantics *ab initio* so this is not a problem. In the case of the π-calculus, in which the syntax has the familiar hierarchical flavour, distinct subprocess objects residing in remote peripheral areas of the parse tree may share private names despite their syntactic remoteness. Elaborate notions of scope and of binding are needed to manage the syntactic arm-twisting that forestalls the name clashes that are prone to occur due to the fact that the two subprocess objects may only express their relationship via their closest common ancestor in the tree. In a graph based language, this is unnecessary — one simply encodes the required relationship by suitable edges or arcs and that is it.

The present authors are not the only ones to notice that graphs have some utility in process algebras and similar systems. One may cite [Milner (1979), Degano and Montanari (1987), Milner (1993b), Corradini et al. (1994), Parrow (1994)] amongst others. However it is not clear that these other formalisms have the same closeness to direct implementation that MONSTR gains by virtue of its association with the Flagship machine.

The structure of the rest of this paper is as follows. In [Section 2] we give a description of MONSTR, while in [Section 3] we set out the version of π-calculus that we will use. [Section 4] describes the key features behind the translation strategy, and [Section 5] presents the details. [Section 6] establishes the basic properties of translated systems that are needed in proving the translation correct, and the correctness proof itself appears in [Section 7]. [Section 8] contains some discussion of aspects of the π-calculus not directly treated in the version of [Section 3]. [Section 9] concludes, and contains further discussion of the material herein, drawing analogies between the proof of soundness on the one hand, and serialisability theory or forcing techniques on the other.

## 2 MONSTR

MONSTR arose as a result of the attempt to reconcile the desire for an intermediate language with rewriting-based semantics, with the reality of a parallel machine where the primitive atomic actions were in principle of much smaller granularity than atomic rewrites of arbitrary size. The result was a term graph rewriting language MONSTR, for which the implementation problem did not make excessive demands on the architecture's semantics.

**Term Graph Rewriting**

The operational semantics of MONSTR deals with the transformation of term graphs. These are graphs in which the nodes are labelled with node symbols from an alphabet $\mathbf{S}$; each node $x$ having an arity $A(x) = \{1 \ldots n\}$, indicating that $x$ has a sequence of $n$ out-arcs. A node $x$ may be the target of an arbitrary number of in-arcs.

The nodes and arcs of MONSTR graphs are further decorated with certain markings which relate to reduction strategy. Specifically, if a node is marked with $*$, then it is active and can serve as the root of a redex. If it is marked with $\#^n$, then it is suspended waiting for $n$ “notifications” (see below), and then (usually) some of its out-arcs are notification arcs, i.e. are marked with the notification mark ^, which is whence the notifications will arrive. The only other possibilities are that nodes and arcs are unmarked (i.e. idle, written visibly as $\varepsilon$ where necesary).

Here is the formal definition. In definition 2.1, $N^*$ is the set of sequences over $N$, similarly for $\{\varepsilon, \wedge\}^*$; the domain of a sequence is the set of its indices; and the arity of a node $x$, $A(x)$, is defined in clause (3).

**Definition 2.1** A term graph (or just graph) $G$, is a quintuple $(N, \sigma, \alpha, \mu, \nu)$ where

(1) $N$ is a set of nodes,

(2) $\sigma$ is a map $N \to \mathbf{S}$, the symbol map,

(3) $\alpha$ is a map $N \to N^*$, giving the arcs of $x$, with for all $x$, $A(x) = \mathrm{dom}(\alpha(x))$,

(4) $\mu$ is a map $N \to \{\varepsilon, *, \#, \#\#, \#\#\#, \ldots \#^n\ (n \geq 1)\}$, the node marking map,

(5) $\nu$ is a map $N \to \{\varepsilon, \wedge\}^*$, the arc marking map, with for all $x$, $\mathrm{dom}(\nu(x)) = A(x)$.

(The nomenclature is meant to be alliterative: $\sigma$ for symbols, $\alpha$ for arcs, $\mu$ for markings, $\nu$ for notifications.) We refer to an arc of a graph by writing $(p_k, c)$ where $p$ is the parent and $c$ is its $k$'th child. Alternatively using $\alpha$, we write $c = \alpha(p)[k]$ where $-[-]$ is the lookup operator on sequences. [Fig. 1] below shows a term graph, in which each node is depicted by its symbol followed by its sequence of out-arcs in brackets, and only non-idle markings are shown.

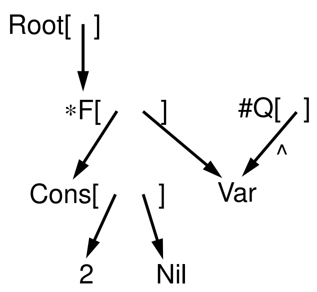


Fig. 1

For rewriting, we need a notion of pattern, and a sufficiently flexible notion of pattern matching. Accordingly a pattern satisfies definition 2.1 except that the signature of $\sigma$ is $N \to \mathbf{S} \cup \{\mathsf{Any}\}$ where Any is special node symbol not in $\mathbf{S}$, the intention being that Any-labelled nodes may match “anything”. For later convenience Any-labelled nodes are called implicit whereas other nodes are explicit. We restrict Any-labelled nodes to occur only at leaves of patterns so that

$$\sigma(x) = \mathsf{Any} \Rightarrow A(x) = \varnothing \quad \text{i.e. } \alpha(x) = \nu(x) = \varnothing$$

Evidently a graph is a kind of pattern, but not vice versa.

**Definition 2.2** A rule $D$ is a quadruple ($P$, *root*, *Red*, *Act*) where

(1) $P$ is a pattern, called the full pattern of the rule.

(2) *root* is an explicit node of $P$ called the root, and all implicit nodes of $P$ are accessible from the root. If $\sigma(root) = S$, then $D$ is a rule for $S$. The subpattern of $P$ accessible from (and including) *root* is called the left pattern $L$ of the rule, and nodes of $P$ not in $L$ are called contractum nodes. $L$ is unmarked, i.e. for all $x \in L$, $\mu(x) = \varepsilon$, and $\nu(x)[k] = \varepsilon$ for all $k \in A(x)$.

(3) *Red* is a set of pairs of nodes, (called redirections) such that $Red \subseteq L \times P$, and if $(x, y) \in Red$, then $x$ is explicit. *Red* is the graph of a function with distinctly labelled nodes in the domain, i.e. if $(x, y)$, $(u, v) \in Red$ then $x = u \Rightarrow y = v$ and $x \neq u \Rightarrow \sigma(x) \neq \sigma(u)$. For $(x, y) \in Red$, $x$ is called the LHS and $y$ the RHS of the redirection.

(4) *Act* is a set of nodes (called activations) of $P$ such that $Act \subseteq L$.

[Fig. 2] is a picture of a rule, with *root* indicated by the short stubby arrow, *Red* indicated by the dotted arrows, and *Act* indicated by adorning the relevant (single in this case) nodes of $L$ with a $*$ (these are unmarked according to definition 2.2.(2)).

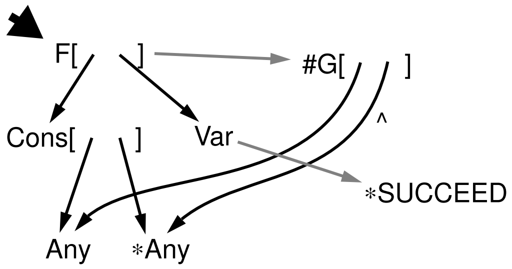


Fig. 2

In concrete syntax this becomes

```
F[Cons[a b] x:Var] => #G[a ^*b] , x := *SUCCEED
```

In this notation, clutter is saved by nesting node definitions where practicable. The *root* node is always the one listed first, and the nesting indicates that it has a Cons first child and a Var second child. The Cons child has two Any children, indicated by just mentioning the node identifiers **a** and **b** (as opposed to node labels which are always capitalised). The left pattern is everything that occurs to the left of the **=>**, and the material to the right describes everything else. Thus the contractum contains a once-suspended G node whose children are those of the left Cons node, (such shared references mean that in general node identifiers are needed as well as node labels). The fact that G's second child is marked with **^*** indicates firstly that the relevant arc is ^-marked, and secondly that that the node is in *Act* (being a *****-marked reference to a left pattern node — in general *any* *****-marked occurrence of a left pattern node on the right of a rule indicates that the node is in *Act*). The other contractum node is an active SUCCEED node. By convention the **=>** indicates that the root node is to be redirected to the node immediately following the **=>**, and the syntax **x := *SUCCEED** indicates that the Var node is to be redirected to the SUCCEED node, in agreement with [Fig. 2]. Note that only the Var node needs to be specified in full (i.e. using both a node identifier and a node label).

A rule system $\mathcal{R}$ is just a set of rules. In outline, given a rule system and some graph, an execution proceeds thus. First choose some active ($*$-marked) node of the graph; secondly examine the rule system to see which rules will match at that active node; if there are some, choose one of them and rewrite the graph using it; alternatively if there are none, perform notifications from the active node. Continue to repeat the whole process with the new graphs successively generated thereby as long as possible.

Here are the technical definitions, starting with matching or homomorphism.

**Definition 2.3** A matching of a pattern $P$ with root $r$ say, to a graph $G$ at a node $t \in G$, is a node map $h : P \to G$ such that

(1) $h(r) = t$

(2) If $x \in P$ is explicit then, $\sigma(x) = \sigma(h(x))$, $A(x) = A(h(x))$, and for all $k \in A(x)$, $h(\alpha(x)[k]) = \alpha(h(x))[k]$.

Omitting mention of roots, the same definition will suffice for matching arbitrary patterns to other patterns or, for matching graphs to other graphs. If $h : P \to G$ is a matching, then we say that $z \in G$ is explicitly matched if it is the $h$ image of an explicit node. Otherwise we say that it is implicitly matched.

Now the definitions pertinent to rewriting.

**Definition 2.4** Let $X$ be a graph, $t \in X$ a node of $X$ such that $\mu(t) = *$, and $\mathcal{R}$ a system. Let $Sel = \{D \mid$ there is a $D \in \mathcal{R}$ such that there is a matching $h : L \to X$ of the left pattern $L$ of the full pattern $P$ of the rule $D$ to $X$ at $t\}$. Rule selection is some (otherwise unspecified) process for choosing a member of *Sel* assuming it is non-empty. The chosen $D$ makes $t$ the root of the redex $h(L)$ and $D$ the selected rule that governs the rewrite.

Assuming we have $X$, $t$, $D = (P, root, Red, Act)$ and $h$ given as above, rewriting according to the rule proceeds via three phases (contractum building, redirection, activation), each of which can be viewed as a mini graph transformation. Naturally, our graph and rule given above provide a running example. There is clearly a redex rooted at F.

**Definition 2.5** Contractum building adds a copy of each contractum node of $P$ to $X$. Copies of arcs of $P$ from contractum nodes to their children are added in such a way that there is an extended matching $h'$ from the whole of $P$ to the graph being created, which agrees with $h$ on $L$. Node and arc markings for the new items are copied from $P$. Call the resulting graph $X'$ and let $i_{X,X'}$ be the natural injection.

In our running example, doing the above yields [Fig. 3]. We see that copies of exactly the contractum nodes and arcs, suitably marked, have been added, and that this enables the extended matching $h'$ of the whole of $P$ to be constructed.

**Definition 2.6** Redirection replaces each arc $(p_k, c)$ of $X'$, such that $c = h'(x)$ for some $(x, y) \in Red$, with $(p_k, h'(y))$. This can be done consistently since the LHSs of two distinct redirections cannot map to the same node of $X'$ since their node symbols are different by definition 2.2.(3). All such redirections are performed simultaneously. Let the resulting graph be called $X''$ and let $i_{X',X''}$ be the natural injection. Note that $i_{X',X''}$ is just an injective map on nodes rather than a matching as for $i_{X,X'}$. We define the map

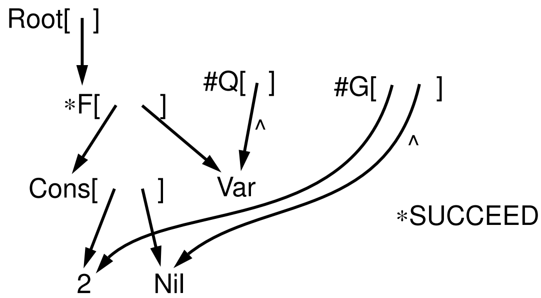


Fig. 3

$r_{X',X''}$ by $r_{X',X''}(c) = i_{X',X''}(c)$ unless $c = h'(x)$ for some $(x, y) \in Red$, in which case $r_{X',X''}(c) = i_{X',X''}(h'(y))$.

Performing the redirections on our example yields [Fig. 4].

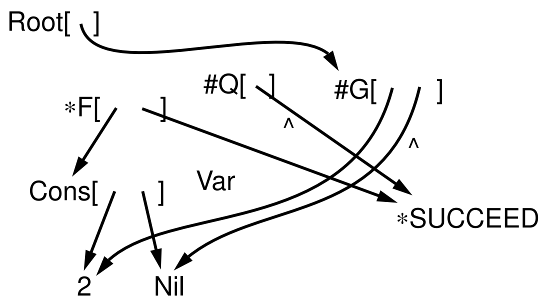


Fig. 4

**Definition 2.7** Activation merely alters some node markings. Roughly speaking, *root* is made idle and the nodes in *Act* are made active. More precisely $\mu(i_{X',X''}(h'(root)))$ is changed to $\varepsilon$, and for each $u \in Act$, provided $\mu(i_{X',X''}(h'(u))) = \varepsilon$ beforehand, the marking $\mu(i_{X',X''}(h'(u)))$ is changed to $*$. We call the resulting graph $Y$, and define $i_{X'',Y}$ as the natural injection.

Doing this for our running example yields [Fig. 5].

The graph $Y$ is taken to be the result of the rewrite, i.e. the result of a single atomic action in the rewriting model. Note that no node of the original graph $X$ is ever removed modulo the identifications of nodes among the various stages. This lack of garbage collection is an issue which will be remedied in due course.

By composing the various maps $i_{X,X'}$, $i_{X',X''}$ or $r_{X',X''}$, etc., we can track the history of a node through an execution of the system. We thus have $i_{X,Y}(x) = (i_{X'',Y} \circ i_{X',X''} \circ i_{X,X'})(x)$ as the node which is the copy in $Y$ of $x \in X$, and $r_{X,Y}(x) = (i_{X'',Y} \circ r_{X',X''} \circ i_{X,X'})(x)$ as the node of $Y$ that $x$ got redirected to. This notation is a little cumbersome, but consider the following. The phrase "adds a copy of each contractum node" is really a euphemism for disjoint union. If one knows in advance which such disjoint unions are needed, one can arrange that all the copies used are distinct, and thus implement disjoint union by

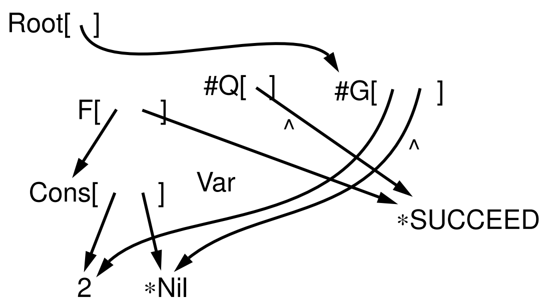


Fig. 5

ordinary union. In more general situations though this is not possible, and one has to take the demands of disjoint union more seriously (such cases arise in the detailed repercussions of the issues we discuss in [Section 8]). In such cases one uses some kind of tagging to make copies distinct, and in these cases the obvious natural injections are no longer identities. In this respect, the more involved notation is more portable. Furthermore, we will need to keep close track of nodes through an execution sequence in [Section 7] and our notation provides a firm foundation for this; also it is useful when we delve into the innards of a rewrite, as we do in lemma 6.10.

Suppose now that *Sel* is empty. Then as we said, instead of a rewrite, notification takes place. Again let $X$ be the graph, and $t \in X$ the chosen node of $X$ such that $\mu(t) = *$.

**Definition 2.8** Notification merely alters some node and arc markings. The node marking $\mu(t)$ is changed to $\varepsilon$. Further for all arcs $(p_k, t)$ in $X$ such that the arc making $\nu(p)[k]$ is ^, the marking $\nu(p)[k]$ is changed to $\varepsilon$, and if the node marking $\mu(p)$ is $\#^n$ (for $n \geq 1$), $\mu(p)$ is changed to $\#^{n-1}$, with $\#^0$ being understood as $*$. We call the resulting graph $Y$ and define $i_{X,Y}$ to be the natural injection.

The result of the notification is the graph $Y$ as before.

In [Fig. 5], assuming there are no rules for Nil or SUCCEED, there is scope for two notifications. When they have both been performed, [Fig. 6] results. What might happen subsequently depends on what rules, if any, there might be for Q and G.

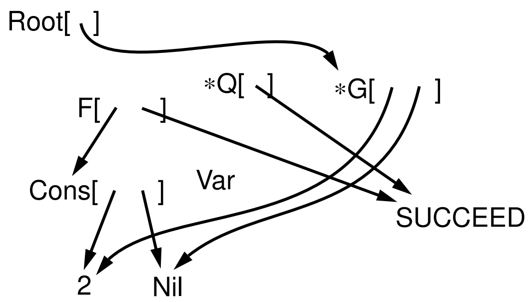


Fig. 6

The remaining technicalities we need in our rewriting model are disposed of in the following couple of definitions.

**Definition 2.9** An initial graph is one which consists of an isolated node of empty arity, with the active ($*$) node marking, and labelled by the symbol Initial.

**Definition 2.10** An execution $\mathcal{G}$ of a system $\mathcal{R}$ is a sequence of graphs $[G_0, G_1 \ldots]$ of maximum length such that $G_0$ is initial and for each $i \geq 0$ such that $i$+1 is an index of $\mathcal{G}$, $G_{i+1}$ results from $G_i$ either by rewriting (in case there is an applicable rule) or by notification (otherwise) at some arbitrarily selected active node $t_i$ of $G_i$. Graphs occuring in executions are called execution graphs.

The above presents a general framework in which term graph rewriting (with programmed control of strategy) may be developed. To be closer to executable machine semantics, MONSTR imposes a collection of restrictions as follows.

**The MONSTR Restrictions**

First the symbol alphabet **S** is partitioned into **F** $\cup$ **C** $\cup$ **V**, where **F** consists of *functions* which have rules but which cannot occur at subroot positions of patterns of rules; and **C** and **V**, consisting of *constructors* and *variables* (or *stateholders*) respectively, neither of which can occur at root positions of patterns of rules and therefore neither of which have rules; in addition constructors are not permitted to occur as the LHS of a redirection. Compared to the use of term graph rewriting as an implementation vehicle for functional languages, where only functions and constructors are needed, the presence of stateholders within MONSTR considerably increases the flexibility of the language for conveniently modelling imperative notions such as storage cells, synchronisation objects, and the logical variable. They will play a vital role in our translation of the $\pi$-calculus below by representing channels and encoding protocol states.

Next we insist that rules are of two kinds, normal rules and default rules. A default rule has a pattern which consists of an active function node and as many distinct implicit children as its arity dictates. Otherwise it is normal. Thus a default rule's pattern will always match at an active execution graph node labelled with the appropriate function symbol.

We insist that there is at least one default rule for every function symbol, and we require a normal rule to be selected in preference to a default rule whenever either will match. In the concrete syntax of MONSTR, we can enforce this rule selection policy using the nondeterministic rule separator **|** and the sequential rule separator `;`.

**Definition 2.11** MONSTR graphs, rules and rule systems must conform to the following list of restrictions.

(1) Symbols have fixed arities, i.e. the map sending a node $x$ to its arity $A(x)$ depends only on $\sigma(x)$, and thus $A(x) = A(\sigma(x))$ (where the second $A$ is a notation for symbol arity).

(2) Functions have fixed matching templates, i.e. for each $F \in$ **F** there is a subset $M(F) \subseteq A(F)$ such that for any normal rule for $F$ with root *root*, $k \in M(F)$ iff $\alpha(root)[k]$ is explicit.

(3) Functions may explicitly match a stateholder in at most one position, and must otherwise explicitly match only constructors, i.e. for each $F \in$ **F** there is a subset $\Sigma(F) \subseteq M(F) \subseteq A(F)$, at most a singleton, such that for any normal rule for $F$ with root *root*, if $k \in \Sigma(F)$, then $\sigma(\alpha(root)[k]) \in$ **C** $\cup$ **V**; else for explicit $\alpha(root)[l]$, such that $l \neq k$, $\sigma(\alpha(root)[l]) \in$ **C**.

(4) Left patterns are shallow, i.e. for each rule, any grandchild of the root is implicit.

(5) Any nodes may not be tested for pointer equivalence, i.e. for every rule, no implicit node may have more than one parent in the left pattern.

(6) Every node $x$ in every rule is balanced, i.e. $\mu(x) = \#^n$ for some $n \geq 1$ iff $n$ is the cardinality of $\{k \in A(x) \mid \nu(x)[k] = \wedge\}$.

(7) Every notification arc $(p_k, c)$ in every rule is state saturated or head activated, i.e. if $\nu(p)[k] = \wedge$, then if $\mu(c) = \varepsilon$ then either $c$ is explicit and $\sigma(c) \in \mathbf{V}$, or $c \in Act$.

(8) A redirection to an unactivated idle node is to a stateholder, i.e. for every rule, if $(x, y) \in Red$ with $\mu(y) = \varepsilon$ then either $y$ is explicit and $\sigma(y) \in \mathbf{V}$, or $y \in Act$.

(9) The root is always redirected, i.e. for every rule with root *root*, $(root, t) \in Red$ for some $t$.

(10) LHSs of redirections must not be activated unless they are also RHSs of redirections, i.e. for every rule, if $(y, z) \in Red$ and $y \in Act$, then $(x, y) \in Red$ for some $x$.

There isn't the space here to explain all the ramifications of these restrictions, or why they are a good idea (see [Banach (1993a)] for a thorough discussion). Essentially, the restrictions enable one to prove a number of run-time properties of arbitrary MONSTR systems, that are desirable from an implementation's point of view.

It is easy enough to check that our running example above, conforms to all of these restrictions, and that the rewrite we showed is in fact a MONSTR rewrite (up to garbage). We will deal with garbage shortly.

It turns out that the MONSTR systems that result from our translation have a relatively simple run-time structure, and using the general properties provable from the syntactic restrictions will not be necessary in the fairly involved correctness proof which is the main concern of this paper — all the facts we will need will be derived directly from the structure of the rule system. Accordingly, we point out one additional feature of MONSTR rewriting that is important in the general theory but that becomes superfluous in the specific systems we deal with.

The definition 2.3, of pattern matching, is insensitive to the markings on nodes and arcs, and aside from the fact that the root of a redex must be active, this carries through to the term graph rewriting model described above. For MONSTR rewriting, as well as the syntactic restrictions, we demand that the explicitly matched arguments of the root of a redex are idle; and in case an active node attempts to rewrite when this is not the case, the rewrite is suspended until such time as it becomes true. This is a run-time mechanism. Fortunately for us, we will be able to prove directly that in our translated systems, all the explicitly matched arguments in a rewrite are idle, and so we needn't concern ourselves with the details of this mechanism. To achieve this simplification though, one of the rules we will use later (in fact rule [9] of the communication protocol of [Section 5]) violates restriction 2.11.(8) as it performs a redirection to an unactivated implicit node. We will prove directly that in fact this Any node is only ever matched to idle stateholders in any execution of the system. As a consequence, our transgression does not affect any of the desirable run-time properties. Given the complexity of the correctness proof that we tackle below, we regard this avoidance of having to deal with suspensions, as reasonable under the circumstances. (In fact all such run-time suspensions can be eliminated. [Banach (1993b)] discusses in detail how this is done.)

It is time we addressed garbage collection, since the rewriting model described above, which never throws anything away, is rendered somewhat unsatisfactory thereby. The following definition of liveness is sound in the presence of the MONSTR restrictions (and run-time suspensions), in that garbage collection may be done eagerly after every rewrite or notification, or delayed, without changing the live part of any execution graph. (See [Banach (1993a)] for a full discussion.) In the sequel, we will be ambivalent about whether garbage is actually present in the graphs we consider. Obviously, when we do garbage collection, the maps $i_{X,X'}$ $r_{X',X''}$, etc. become partial, as some of the codomain elements disappear.

**Definition 2.12** Given a MONSTR graph $X$, garbage collection removes all non-live nodes and arcs from $X$, giving a subgraph Live($X$). A live node $x$ is one that can be proved so on the basis of the following rules of inference:

(1) If $\sigma(x)$ is a special symbol Root (a constructor), then $x$ is live.

(2) If $\mu(x) = *$, then $x$ is live.

(3) If $p$ is live and $(p_k, x)$ is an idle arc, then $x$ is live.

(4) If $c$ is live and $(x_k, c)$ is a notification arc, then $x$ is live.

A live arc is one for which both head and tail nodes are live; and non-live nodes and arcs are garbage.

Returning to our running example, the original graph in [Fig. 1] clearly contains no garbage. When we perform the rewrite getting [Fig. 5], a certain amount of garbage is generated. Removing this results in [Fig. 7].

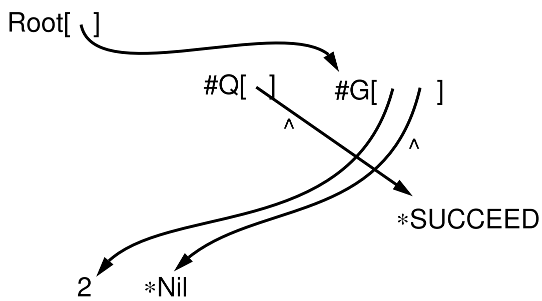


Fig. 7

In general, and despite the soundness result, Live($X$) does not satisfy all the conditions for being a MONSTR graph (since eg. a live node may have a garbage child node); however, this possibility will not occur in the systems we consider below. In more general cases, the possibility makes theoretical treatments of MONSTR easier when garbage is retained. Once more, the reader is refered to [Banach (1993a)] for a fuller discussion.

## 3 THE PI-CALCULUS

The $\pi$-calculus first appeared in [Milner et al. (1992)] and since that time has been seen in a number of minor variants. We will fix on a version of the monadic calculus, as presented in the first part of [Milner (1993a)], since it is in many ways the most economic version, and so leads to the most transparent translation. We regard as given a suitable alphabet **CN** of channel names, ranged over by $x$, $y$, $z$ etc. Here is the formal syntax.

**Definition 3.1** The $\pi$-calculus language of process expressions is given by the following syntax where $P$ is a process and the various $Q_i$ are subprocesses (thus corresponding to the same nonterminal in a formal BNF). We will take it for granted that parentheses may be used in the usual way.

$$
\begin{aligned}
P = \ & \pi_1.Q_1 + \pi_2.Q_2 + \ldots + \pi_n.Q_n \\
| \ & Q_1 \mid Q_2 \mid \ldots \mid Q_n \\
| \ & \nu x Q \\
| \ & !Q \\
| \ & \mathbf{0}
\end{aligned}
$$

Speaking informally, the first case is guarded summation where each $\pi_i$ is of the form $x(y)$ or $\overline{x}z$. Here the parentheses and the overbar are constant symbols within the syntax, and as below $x$, $y$, $z$ are in **CN**. The input expression $x(y).Q$, means that some channel name $q$ say, is to be read over the channel $x$ which plays the role of a communication link, and then $q$ is bound to all free occurences of $y$ in $Q$; $y$ is a bound name in $x(y).Q$ and $Q$ is its scope. Conversely the output expression $\overline{x}z.Q$ means that the channel name $z$ (the data) is to be written to the channel $x$ which acts as the communication link. A process which is a sum can evolve into exactly one of the alternatives, the others being discarded. The second case is parallel composition; all the $Q_i$'s are parallel processes and evolve concurrently. The third case is restriction; in $\nu x Q$, where the $\nu$ is a constant of the syntax, the channel name $x$ is bound, and refers to a channel that is private to $\nu x Q$, so $\nu x$ is a binder and its scope is $Q$. The fourth case is replication, in which the ! in $!Q$ is another constant of the syntax. $!Q$ is intended to be (syntactically) equivalent to the parallel composition of as many copies of $Q$ as one might wish for, i.e. $!Q \equiv Q \mid Q \mid Q \mid \ldots \mid !Q$. However we will take a different approach to replication as described below. Finally the process $\mathbf{0}$ does nothing.

Process algebra definitions are normally supplemented by demanding that summation and parallel composition are monoidal operators with $\mathbf{0}$ as unit. We will simplify our subsequent task a little by not demanding the "$\mathbf{0}$ as unit" part, so that any top level $\mathbf{0}$'s that get exposed during the evolution of a process expression (see below) just end up lying around as inactive parallel subprocesses. (We could overcome this at the price of extra complexity.)

Further, since the subprocesses in our sums are always prefixed, we do not demand that (prefixed) summation is associative, else prefix might in some sense become left distributive, a possibility that is usually regarded as not *a priori* desirable. (In this respect, we must regard a process expression such as $((\pi_1.Q_1 + \pi_2.Q_2) + \pi_3.Q_3)$ as merely a meta-level shorthand for a flattened ternary summation $(\pi_1.Q_1 + \pi_2.Q_2 + \pi_3.Q_3)$ rather than as a true two level summation according to the formal syntax.) Of the monoidal laws, we are thus left with associativity of parallel composition, and commutativity of both summation and parallel composition. More formally:

**Definition 3.2** The language of π-calculus expressions is required to conform to the congruences generated by the following equations

$$(Q_1 \mid Q_2) \mid Q_3 \equiv Q_1 \mid (Q_2 \mid Q_3)$$
$$Q_1 \mid Q_2 \equiv Q_2 \mid Q_1$$
$$\pi_1.Q_1 + \pi_2.Q_2 \equiv \pi_2.Q_2 + \pi_1.Q_1$$

The free channel names of π-calculus expressions in particular, will play an important part in the translation of [Section 5]. We give below the free and bound channel names for each of the syntactic constructs. (For the first case, we just give the binary variant to avoid clutter.)

**Definition 3.2** The free and bound channel names of a π-calculus expression are given by recursion by the rules:

$$\mathrm{Free}(x(y).Q_1 + \bar{z}w.Q_2) = \{x, z, w\} \cup (\mathrm{Free}(Q_1) - \{y\}) \cup \mathrm{Free}(Q_2)$$
$$\mathrm{Free}(Q_1 \mid Q_2 \mid \ldots \mid Q_n) = \mathrm{Free}(Q_1) \cup \mathrm{Free}(Q_2) \cup \ldots \cup \mathrm{Free}(Q_n)$$
$$\mathrm{Free}(\nu x Q) = \mathrm{Free}(Q) - \{x\}$$
$$\mathrm{Free}(!Q) = \mathrm{Free}(Q)$$
$$\mathrm{Free}(\mathbf{0}) = \varnothing$$

$$\mathrm{Bound}(x(y).Q_1 + \bar{z}w.Q_2) = \{y\} \cup \mathrm{Bound}(Q_1) \cup \mathrm{Bound}(Q_2)$$
$$\mathrm{Bound}(Q_1 \mid Q_2 \mid \ldots \mid Q_n) = \mathrm{Bound}(Q_1) \cup \mathrm{Bound}(Q_2) \cup \ldots \cup \mathrm{Bound}(Q_n)$$
$$\mathrm{Bound}(\nu x Q) = \mathrm{Bound}(Q) \cup \{x\}$$
$$\mathrm{Bound}(!Q) = \mathrm{Bound}(Q)$$
$$\mathrm{Bound}(\mathbf{0}) = \varnothing$$

The above makes clear that the $x(y)$ in $x(y).Q$ and the $\nu x$ in $\nu x Q$, are binders. We will need to regard the alpha-convertibility of bound variables as fundamental below, so we have the next definition.

**Definition 3.3** The language of π-calculus expressions is required to conform to the congruence generated by alpha-convertibility

$$\Phi(x(y).Q) \equiv \Phi(x(y').Q\{y'/y\})$$
$$\Phi(\nu x Q) \equiv \Phi(\nu x' Q\{x'/x\})$$

where in the above, $\Phi$ is a π-calculus expression containing eg. $\nu x Q$ as a subexpression, $x'$ and $y'$ are some other names not free in $Q$, and eg. $Q\{x'/x\}$ refers to $Q$ with all free occurrences of $x$ substituted by $x'$. In general, when we exploit alpha-convertibility, we will typically assume $x'$ and $y'$ are fresh names not appearing anywhere else in the whole expression, rather than just not appearing free in $Q$.

We turn now to the dynamics of the π-calculus. For most of the time (until [Section 8] in fact), we will restrict our attention to the behaviour of closed systems.

**Definition 3.4** A closed π-calculus system evolves using the replication rewrite rule

$$!P \to_{\mathrm{R}} P \mid !P$$

and the communication rewrite rule

$$(\ldots + x(y).P + \ldots) \mid (\ldots + \bar{x}z.Q + \ldots) \to_{\mathrm{C}} P\{z/y\} \mid Q$$

where in the RHS of the latter, $P\{z/y\}$ again refers to the substituted version of $P$.

The replication rule shows that rather than regarding replication as a syntactic congruence which is the usual approach in the π-calculus, we will regard the spawning of copies of a replicated subexpression as being done via an explicit rule within the dynamics of a π-calculus system. This is because the translation will also manufacture copies of (the translation of) a replicated subexpression by explicitly rewriting, and consequently the correctness proof will become more managable if we can pick out points in the dynamics of the original π-calculus expression at which replication was needed.

It is to be understood that both of the dynamic rules are applicable only "near the top level" of a π-calculus expression, which brings out an analogy with the Chemical Abstract Machine [Berry and Boudol (1990)]. The top level proviso may be stated in precise terms as follows.

**Definition 3.5** The contexts within which the rules of π-calculus dynamics are applicable, are given by the additional rules

$$\frac{P \rightarrow_Y Q}{P \mid R \rightarrow_Y Q \mid R}$$

and

$$\frac{P \rightarrow_Y Q}{\nu x P \rightarrow_Y \nu x Q}$$

where above, $\rightarrow_Y$ stands for either of $\rightarrow_{\mathrm{R}}$ and $\rightarrow_{\mathrm{C}}$ (we will use this notation below where convenient).

Thus dynamic behaviour can only take place under parallel compositions and $\nu x$ binders. As a result, a communication must be done either entirely inside, or entirely outside the scope of a $\nu x$ binder. The reason for this is as follows. Consider the input and output subprocesses of a potential communication, $x(y).P$ and $\overline{x}z.Q$. If $z$ is free in the whole expression, then the communication can go ahead, since then if $x$ is bound in a $\nu x$ binder, both processes will be in the scope of the $\nu x$, otherwise it doesn't matter. But if $z$ is bound in a $\nu z$ binder, then if the $x(y).P$ were to occur outside the scope of the $\nu z$, doing the communication would create via substitution an occurrence of $z$ outside its scope, and and such a $z$ would be a different name according to the conventions regarding bound variables. Therefore we canot allow bound names to escape their scopes in this manner and must forbid such communications. However this has the undesirable consequence of forbidding potential communications wherein the sender and recipient have a communication link in common, but the sender is prevented from sending his data because it would escape the scope that that happens to contain that data at the given moment.

The original description of the π-calculus in [Milner et al. (1992)] provided mechanisms to overcome this. We will take a simpler approach that allows us to simply enlarge the scope of a $\nu z$ binder sufficiently, so that any communication link $x$ over which $z$ might be transmitted using a prefix $\overline{x}z$, has all corresponding input prefixes $x(y)$ which are visible near the top level, within the scope of the $\nu z$ binder. More precisely we have the following.

**Definition 3.6** The syntactic reduction $\equiv\!\!>$ is defined as follows.

$$(\nu z Q_1 \mid Q_2 \mid \dots \mid Q_n) \equiv\!\!> \nu z'(Q_1\{z'/z\} \mid Q_2 \mid \dots \mid Q_n)$$

Above, the $\nu z'$ and $\{z'/z\}$ refer to an alpha-conversion of the binder $\nu z$ performed to avoid potential capture of free variables in the enlarged scope. We write $\Rrightarrow^{+}$ and $\Rrightarrow^{*}$ for the transitive and reflexive transistive closure of $\Rrightarrow$. The contexts in which $\Rrightarrow$ is allowed to apply are given once more by the rules in definition 3.5.

**Lemma 3.7** The relation $\Rrightarrow^{*}$ is a simulation; i.e. if $P \Rrightarrow^{*} P'$ and $P \rightarrow_Y Q$, then there is a $Q'$ with $Q \Rrightarrow^{*} Q'$ such that $P' \rightarrow_Y Q'$.

*Proof.* Immediate from definitions 3.5 and 3.6. ☺

The above result allows us to enlarge the scopes of $\nu z$ binders until they permit all prospective communications to take place. We will regard the extended potential for communications that arises in this way as part of the operational semantics of a $\pi$-calculus expression.

**Definition 3.8** Let $E = \Phi(\bar{x}z.P, x(y).Q)$ be a $\pi$-calculus expression such that $\bar{x}z.P$ and $x(y).Q$ are at top level, i.e. the syntactic constructs above $\bar{x}z.P$ and $x(y).Q$ in the parse tree of $\Phi$ consist of parallel compositions and restrictions. Then $E$ is standard with respect to $z$, iff any $\nu z$ binder (for the specific channel name $z$) in $\Phi$, contains either both $\bar{x}z.P$ and $x(y).Q$, or neither of them, in its scope. $E$ is in standard form iff it is standard with respect to all output prefix data channel names occurring at top level.

For practical purposes therefore, we will enhance the dynamics of $\pi$-calculus expressions to include conversion to standard form, by applying $\Rrightarrow^{*}$ after each $\rightarrow_Y$ step until all output prefix data channel names at top level eg. $\bar{x}z$ which occur within a $\nu z$ binder, have all corresponding input prefixes eg. $x(y)$ included in the scope of the binder.

The following result will be useful later.

**Lemma 3.9** Let $\Phi(\nu xP, Q)$ be a $\pi$-calculus expression (containing $\nu xP$ and $Q$ as subexpressions), and let

$$\Phi(\nu xP, Q) \Rrightarrow^{*} \Psi(\nu x'\Delta(P\{x'/x\}, Q))$$

i.e. the $\nu x$ binder has been lifted till its scope has captured $Q$. Then the free and bound names of $Q$ in both LHS and RHS of the above relation are the same, and such a name is free (resp. bound) in $\Phi(\nu xP, Q)$ iff it is free (resp. bound) in $\Psi(\nu x'\Delta(P\{x'/x\}, Q))$.

*Proof.* This is because the bound name in the $\nu x$ binder will have been alpha-converted to $x'$ precisely to ensure this. ☺

**Remark 3.10** We point out that lemmas 3.7 and 3.9 remain true if we also include the clauses

$$(\pi_1.\nu zQ_1 + \pi_2.Q_2 + \ldots + \pi_n.Q_n) \Rrightarrow \nu z'(\pi_1.Q_1\{z'/z\} + \pi_2.Q_2 + \ldots + \pi_n.Q_n)$$
$$\nu x\nu yQ \Rrightarrow \nu y\nu xQ$$

in definition 3.6, but we will not need this fact subsequently.

We cannot however extend the lifting of $\nu z$ binders arbitrarily. Eg. we note that $\nu z$ binders cannot be lifted past replications. It is clear why: a replication rewrite of $!Q$ creates a copy of any $\nu z$-bound scope within $Q$, and any such copy will refer to its own distinct bound name (regardless of whether this bound name is called $z$, or alpha-converted). If the $\nu z$ binder were floated above the !, then the references to $z$ within these scopes would become references to a common name, free in all copies of $Q$, quite the opposite of what is intended.

Here is a small example of a π-calculus system and one possible evolution. It will provide a running example for the translation later on.

$$\begin{aligned} & x(u).u(t).\mathbf{0} \mid (\bar{x}v.\bar{v}s.\mathbf{0} + \bar{x}v.\mathbf{0}) \mid x(y).\mathbf{0} \\ & \quad \rightarrow_C \quad v(t).\mathbf{0} \mid \bar{v}s.\mathbf{0} \mid x(y).\mathbf{0} \\ & \quad \rightarrow_C \quad \mathbf{0} \mid \mathbf{0} \mid x(y).\mathbf{0} \end{aligned}$$

Readers may check that the same system may also evolve to $v(t).\mathbf{0} \mid \mathbf{0} \mid x(y).\mathbf{0}$ or to $x(u).u(t).\mathbf{0} \mid \bar{v}s.\mathbf{0} \mid \mathbf{0}$ or to $x(u).u(t).\mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$.

We emphasise once more that we are dealing with closed π-calculus systems for the moment. This gives us a more easily comprehensible goal for translation. The original formulation of the π-calculus in [Milner et al. (1992)] was presented via a more finegrained transition system suited to the description of open systems (ones with external as well as internal communications), and featuring phenomena such as the opening, closing, and extrusion of the scopes of restriction operators. These latter permit the extension of our syntactic reduction $\Rrightarrow$ above to a congruence, and of our simulation $\Rrightarrow^*$ to a true bisimulation, and legitimise our use of $\Rrightarrow^*$ to enhance the communication capabilities of a π-calculus expression. Once we have the translation of closed systems under control, we will see that it is not hard to understand these more subtle mechanisms using the concepts that arise in the development of the translation. We will discuss this more fully in [Section 8], after we have presented the translation and proved it correct.

## 4 AN OVERVIEW OF THE TRANSLATION

The general idea of the translation is that π-calculus processes in an evolving system, are represented by active function nodes in a MONSTR execution graph, since in the MONSTR world such nodes represent independent loci of control in a computation. Therefore π-calculus processes which are potentially able to communicate by virtue of not being ancestrally guarded, correspond to active function nodes. Channels are represented by stateholder nodes, and all processes with an interest in a given channel share (i.e. have an out-arc to) the stateholder representing that channel. Two facts make this an appropriate representation strategy. Firstly, MONSTR nodes have fixed arities, so modelling the sharing of a channel by out-arcs from the channel to the community of processes that share it, would be awkward in view of the fact that this community changes as the system evolves; on the other hand, there is no such restriction on the in-arcs of a node. Secondly, the notion of arc redirection, having been designed as the natural directed-graph generalisation of substitution in the term world, is ideally suited to model the substitution operation that takes place when a pair of processes communicate.

Suppose then a pair of active processes P and Q share a channel chan, (we assume that the symbols P and Q encode the potential behaviours of the two processes), and P wishes to send a channel chan_out along chan, and Q wishes in turn to receive a channel along chan to bind to its channel chan_in. We represent this action as the term graph transformation in [Fig. 8].

Note that this achieves the substitution of chan_in by chan_out via the redirection chan_in := chan_out, the channel of communication chan playing an almost incidental role. P′ and Q′ represent the subsequent potential behaviours of P and Q. (We have assumed for the sake of argument that both P′ and Q′ retain an interest in both channels, though this needn't be the case.)

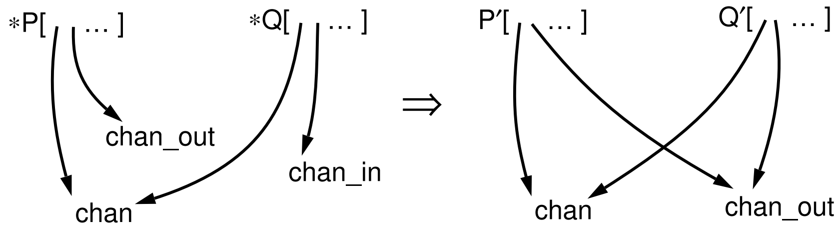


Fig. 8

One thing prevents us from turning this insight directly into a MONSTR rewrite rule, and that is that MONSTR forbids multi-rooted LHSs of rules — our left hand configuration above is double-rooted. The reason for this is purely to do with the efficiency of pattern matching of single-rooted LHSs of rules; they are operationally much easier to test for a successful match than multi-rooted ones. (Nevertheless, given that the formal notion of matching is that of graph homomorphism, there is no obstacle to multi-rooted-LHS rules as far as the abstract semantics of graph rewriting is concerned.) This has a number of consequences. Neither P nor Q can be assumed to know about the other in any rule that initiates communication. The best that they can do is to propose a communication via chan, and hope that a suitable partner process offers to cooperate.

At this point π-calculus semantics enters the fray. Offers of communication by individual processes must be rescindable, otherwise deadlock could occur if a cycle of processes were involved in making offers to others without any of them being reciprocated. Furthermore, the actions that constitute the playing out of the communication protocol for any representation of a π-calculus system must be equivalent to some serial shedule of atomic communication events in the original system.

The easiest way to ensure this is to impose a global synchronisation on the execution. A global semaphore, shared by all processes, is introduced, and processes accede to a mutex discipline in order to enter some offers of communication on some channel, or to rescind or cooperate with an offer already made. Such a protocol can easily be shown to have the correct serialisability properties. Expressing such a protocol in MONSTR is however quite expensive in terms of the number of rewrite rules needed. Further, the concurrency permitted by such a protocol is easily seen to be rather small, which is perhaps rather against the spirit of a formalism specifically designed to express concurrency. Instead, we prefer a much simpler, much more concurrent protocol, synchronised on a per-channel basis. It does however suffer from a busy waiting overhead, because each process proposing an offer to communicate over some channel is responsible for (nondeterministically) rescinding its own offer, since in the end, it may be the only process with an interest in that given channel.

Here is an outline of our preferred protocol. The states of a channel are represented by the three stateholder symbols Empty, Busy_Unlocked, and Busy_Locked. Empty means no reader has recorded an offer on this channel. Busy_Unlocked means that a reader has registered an offer on this channel. At this stage, either the offer may be rescinded by the original reader and the state of the channel reverts to Empty, or a writer completes a rendezvous with the reader, installing its data, and changing the state to

Busy_Locked. In the latter case it becomes the reader's responsibility to extract the data and bind it to the input channel, and to reset the channel state to Empty.

The behaviour of a process $P = (\ldots + chan(chan\_in).Q + \ldots)$ is represented by a collection of rules, one for each summand. The rule for the summand displayed, records a decision, made nondeterministically, for $P$ to attempt to communicate via *chan*, and when successful to evolve to $Q$. The node for P rewrites to P_Q and spawns a helper process Help_r[chan chan_in] to manage the protocol. If the attempt is unsuccessful, P_Q backtracks to P once more.

## 5 THE TRANSLATION IN DETAIL

The translation of a π-calculus expression proceeds in a bottom up fashion. First of all we label all nodes of the parse tree of the expression with new (process) names; we do this by introducing a pair of squiggly brackets round each possible subexpression, and labelling each pair with the new name. For example $Q_1 \mid Q_2 \mid \ldots \mid Q_n$ becomes $\{Q_1 \mid Q_2 \mid \ldots \mid Q_n\}_P$ and $\nu xQ$ becomes $\{\nu xQ\}_P$. For later convenience, we permit identical or alpha-convertible subexpressions to be identically labelled provided that all their own corresponding identical or alpha-convertible subexpressions are also identically labelled, but we do not insist on this. Generally at the meta level we will use $P$ for the name of the subexpression and $Q$, $Q_i$ etc. for the names of the immediate subcomponents. The names are available to serve both as meta-names for the subexpressions themselves and as names for the MONSTR function symbols that encode the behaviour of the relevant subprocess.

Secondly, we will translate each process subexpression $P$ to a pair. The second component of each pair is a set of MONSTR rules for the function symbol **P** that encodes the behaviour of $P$. The first component of the pair is a mapping $Args_P : \mathrm{Free}(P) \rightarrow \boldsymbol{args}_P$ from the free channel names of $P$ as given in [Section 3], to the arguments $\boldsymbol{args}_P$ of the corresponding MONSTR function symbol. Strictly speaking, this is a map from free channel names to positive integers (argument positions); but in the context of a rule, an $\boldsymbol{args}_P$ sequence will always be a sequence of node identifiers. Equalities between the channel node identifiers occuring in the codomains of *Args* maps for symbols on the left and right sides of rules, shows how channels migrate through the execution of a MONSTR representation of a π-calculus system.

As in all rule systems, from the rewriting viewpoint, all occurrences of node identifiers in MONSTR rules are bound; they are templates for nodes of execution graphs that are either located during pattern matching, or instantiated during the contractum building phase of the rewrite. As such their identity is fluid in that they can be renamed (alpha-converted) to avoid node identifier clashes. Therefore when we speak of $\boldsymbol{args}_P$ etc. below, we assume it contains a sequence of distinct node identifiers, all different from any node identifiers that might arise from the translation of channel identifiers occurring visibly in a π-calculus subexpression, eg. $x$ in $\nu xQ$; such explicit channel identifiers are translated "by font change". A consequence of the latter is that we assume that all bound variables occurring in the π-calculus expression that we are translating have been renamed apart from each other and apart from any free names in the expression. This forestalls the need to actually invoke alpha-conversion in the translation. Note that since all occurrences of node identifiers in rules are bound, the relationship between occurrences of node identifiers in $\boldsymbol{args}_P$ and $\boldsymbol{args}_Q$ lists in a rule needs to be consistent only on a per-rule basis. Note also that the bound names of the π-calculus do not occur as such in the translation. Only their free instances get translated. This is in line with our comments in the introduction on the absence of scope and binding mechanisms in (the syntax of) graph rewriting. We will make suitable remarks as we go.

Lastly, if the $\pi$-calculus expression is to be treated as a module to be combined with others at some future stage, the output of the translation is the set of rules generated, together with the $Args_P$ map for the top level function symbol. If the expression is to stand for a self contained system, the output is the set of generated rules together with a rule for **`Initial`** which instantiates the free channels of the top level function.

**The Translation Body**

Here is the translation, easiest cases first. We recall that all channels have been suitably renamed apart. The separate cases below generate rules for various MONSTR function symbols, but do not give much of a clue as to what rule selection strategy is to be employed. It is a property of the rules we generate here that all normal (i.e. non-default) rules have non-overlapping patterns. The appropriate rule selection strategy is therefore: "Given an active function node, attempt to match a normal rule; if no normal rule matches, match a default rule". We therefore omit the rule separators **|** and **`;`** from the text of the translation.

$\{\mathbf{0}\}_P$ : Free($P$) = $\varnothing$ ; $Args_P = \varnothing$ ; Rules($P$) $\equiv$ { **`P => *Root`** }

$\{\nu x Q\}_P$ : Free($P$) = Free($Q$) – $\{x\}$ ;

If $x \notin$ Free($Q$) then Set $Args_P = Args_Q$ and $\boldsymbol{args}_P = \boldsymbol{args}_Q$ ;

Rules($P$) $\equiv$ { **`P[`*`args`*$_P$`] => *Q[`*`args`*$_Q$`]`** }

else ($x \in$ Free($Q$)) then assume for simplicity that **`x`** occurs last in $\boldsymbol{args}_Q$ ;

Set $Args_P = Args_Q\big|_{\mathrm{Free}(P)}$ and $\boldsymbol{args}_P$ =

*all_except_the_last_of*($\boldsymbol{args}_Q$) ;

Rules($P$) $\equiv$ { **`P[`*`args`*$_P$`] => *Q[`*`args`*$_P$` x:Empty]`** }

Note that there is no trace of $x$ in **`P[...]`**. Only when **`P`** evolves to **`Q`** is **`x`** created as a fresh **`Empty`** stateholder.

$\{!Q\}_P$ : Free($P$) = Free($Q$) ; Set $Args_P = Args_Q$ and $\boldsymbol{args}_P = \boldsymbol{args}_Q$ ;

Rules($P$) $\equiv$ { **`P[`*`args`*$_P$`] => *Q[`*`args`*$_Q$`],*P[`*`args`*$_P$`]`** }

Note that since $\boldsymbol{args}_P = \boldsymbol{args}_Q$, all the correct channels are shared by **`P`** and **`Q`** on the RHS.

$\{Q_1 \mid Q_2 \mid \dots \mid Q_n\}_P$ : Free($P$) = $\bigcup_{i \in [1\dots n]}$ Free($Q_i$) ;

Let $blend_P$ be a function that merges a set of sequences into a single sequence without repetitions (so providing an implementation of set union). Set

$\boldsymbol{args}_P = blend_P(\boldsymbol{args}_{Q_1}, \boldsymbol{args}_{Q_2}, \dots, \boldsymbol{args}_{Q_n})$ ;

$\forall i \bullet \forall d \in$ Free($Q_i$) $\bullet$ $Args_P(d) = Args_{Q_i}(d)$ ;

Rules($P$) $\equiv$ { **`P[`*`args`*$_P$`] =>`**

**`*Q`$_1$`[`*`args`*$_{Q_1}$`],*Q`$_2$`[`*`args`*$_{Q_2}$`],...,*Q`$_n$`[`*`args`*$_{Q_n}$`]`** }

$\{\Sigma_{i \in [1\ldots n]}\, x_i(u_i).Q_i^R + \Sigma_{j \in [1\ldots m]}\, \bar{z}_j w_j.Q_j^W\}_P$ :

Free($P$) = $\{x_1,\ldots, x_n, z_1,\ldots, z_m, w_1,\ldots, w_m\}$

$\cup \bigcup_{i \in [1\ldots n]} (\mathrm{Free}(Q_i^R) - \{u_i\}) \cup \bigcup_{j \in [1\ldots m]} \mathrm{Free}(Q_j^W)$ ;

Assume a $blend_P$ function as above. Set

$\boldsymbol{args}_P = blend_P\,([x_1,\ldots, x_n, z_1,\ldots, z_m, w_1,\ldots, w_m],$
$\boldsymbol{args}_{Q_1^R}, \ldots, \boldsymbol{args}_{Q_n^R}, \boldsymbol{args}_{Q_1^W}, \ldots, \boldsymbol{args}_{Q_m^W})$ ;

and

$\forall i \bullet \forall d \in \mathrm{Free}(Q_i^R) - \{u_i\} \bullet Args_P(d) = Args_{Q_i^R}(d)$ ;

$\forall j \bullet \forall d \in \mathrm{Free}(Q_j^W) \bullet Args_P(d) = Args_{Q_j^W}(d)$ ;

Rules($P$) is given by firstly $\bigcup_{i \in [1\ldots n]}$ of

**P[$\boldsymbol{args}_P$] => #P_Q$_i^R$ [^**Help_r[x$_i$ u$_i$] $\boldsymbol{args}_P$**
**u$_i$:Empty]**

**P_Q$_i^R$ [Yes $\boldsymbol{args}_P$ u$_i$] => *Q$_i^R$ [$\boldsymbol{args}_{Q_i^R}$]**

(note that if $u_i \in \mathrm{Free}(Q_i^R)$ then $\mathbf{u}_i$ occurs in $\boldsymbol{args}_{Q_i^R}$)

**P_Q$_i^R$ [No $\boldsymbol{args}_P$ u$_i$] => *P[$\boldsymbol{args}_P$]**

**P_Q$_i^R$ [a $\boldsymbol{args}_P$ u$_i$] => *P_Q$_i^R$ [a $\boldsymbol{args}_P$ u$_i$]**

and secondly $\bigcup_{j \in [1\ldots m]}$ of

**P[$\boldsymbol{args}_P$] => #P_Q$_j^W$ [^**Help_w[z$_j$ w$_j$] $\boldsymbol{args}_P$]**

**P_Q$_j^W$ [Yes $\boldsymbol{args}_P$] => *Q$_j^W$ [$\boldsymbol{args}_{Q_j^W}$]**

**P_Q$_j^W$ [No $\boldsymbol{args}_P$] => *P[$\boldsymbol{args}_P$]**

**P_Q$_j^W$ [a $\boldsymbol{args}_P$] => *P_Q$_j^W$ [a $\boldsymbol{args}_P$]**

The above constitutes the body of the translation. This must be supplemented by the communication protocol rules described below. To make a component module of a larger system, on the assumption that $T$ is the top level symbol labelling the outermost construct of the original π-calculus expression $E$, the accumulated set of rules (with the selection strategy mentioned above) is combined with the map $Args_T : \mathrm{Free}(T) \to \boldsymbol{args}_T$ to form the output of translation called $\mathrm{Tr}^-(E)$. The latter holds the information on how arguments of **T** correspond to the free channels of the module, which is needed for interfacing to other modules. To form a stand alone system, we needn't retain $Args_T$, but we need to form an initial rule. If $\boldsymbol{args}_T$ has $k$ entries, this rule is

**Initial => *T[u$_1$:Empty, . . . ,u$_k$:Empty]**

which simply instantiates the top level free channels and sets the system in motion. To prevent confusion, we call this version of the output of translation Tr($E$).

Like many translations, the one above is prone to some inefficiencies. Some of the translation steps do not do very much. Nevertheless, it is simple enough to be reasonably transparent for pedagogical purposes. The reader who has grasped the structure of the translation above would have no difficulty in altering it so that it translated a more meaty chunk of syntax such as

$$\{\nu x_1,\ldots, x_n(\Sigma\, \pi_{i_1}.Q_{i_1} \mid \Sigma\, \pi_{i_2}.Q_{i_2} \mid \ldots \mid \Sigma\, \pi_{i_m}.Q_{i_m})\}_P$$

all in one go, saving on both rules and rewrites.

**The Communication Protocol**

To effect a communication, the helper functions Help_r[x u] and Help_w[z w] must actually make contact and transfer data. A collection of rules is needed to handle various aspects of the protocol. Unlike the body rules above which were mainly default rules as all they had to do was to manage the plumbing, the rules below do a fair amount of pattern matching. The default rules for the symbols in question are forced by the definition of MONSTR and are mainly superfluous. Again the normal rules have non-overlapping patterns (with the exception of **Help_r_test_chan** which has two overlapping rules that implement a nondeterministic busy wait), and so the strategy for rule selection in the complete system is once again "(nondeterministically) select a normal rule if one will match, otherwise a default rule". The rules are numbered for future reference.

Rules for read helper.

**Help_r** initiates an offer; no offers in progress. [1]

```
Help_r[chan:Empty chan_in]
 => *Help_r_test_chan[chan':Busy_Unlocked chan_in],
     chan := chan'
```

**Help_r** default rule; backs off. [2]

```
Help_r[chan chan_in]
 => *No
```

Rules for write helper.

**Help_w** sees a channel containing an offer; starts a rendezvous. Below, this rule will be known as the communication commit rule (or just commit rule); rewrites using this rule will be called commit rewrites. [3]

```
Help_w[chan:Busy_Unlocked chan_out]
 => *Yes , chan := chan':Busy_Locked[chan_out]
```

**Help_w** default rule. [4]

```
Help_w[chan chan_out]
 => *No
```

Rules for **Help_r_test_chan**.

**Help_r_test_chan** waits a bit longer. [5]

```
Help_r_test_chan[chan:Busy_Unlocked chan_in]
 => *Help_r_test_chan[chan chan_in]
```

**Help_r_test_chan** revokes its own offer. [6]

```
Help_r_test_chan[chan:Busy_Unlocked chan_in]
 => *No , chan := chan':Empty
```

**Help_r_test_chan** detects a rendezvous. [7]

```
Help_r_test_chan[chan:Busy_Locked[data] chan_in]
 => *Help_r_assign_data[chan data chan_in]
```

**Help_r_test_chan** default rule. [8]

```
Help_r_test_chan[chan chan_in]
 => *Help_r_test_chan[chan chan_in]
```

Rules for **Help_r_assign_data**.

**Help_r_assign_data** assigns data and prepares to unlock **chan**. [9]

```
Help_r_assign_data[chan data chan_in:Empty]
 => *Help_r_unlock[chan] , chan_in := data
```

**Help_r_assign_data** default rule. [10]

```
Help_r_assign_data[chan data chan_in]
 => *Help_r_assign_data[chan data chan_in]
```

Rules for **Help_r_unlock**.

**Help_r_unlock** unlocks chan and resets protocol. [11]

```
Help_r_unlock[chan:Busy_Locked[data]]
 => *Yes , chan := chan':Empty
```

**Help_r_unlock** default rule. [12]

```
Help_r_unlock[chan]
 => *Help_r_unlock[chan]
```

It is clear from the above structure that the protocol offered is by no means the only one that will do the job. It just seems to us to be the simplest one that makes the points that we wish to make. From the relatively straightforward way in which the protocol rules interface to the body rules, it is obvious that a more hard-nosed protocol could be substituted for ours, if for example one wished to avoid the penalty of busy waiting, (as exemplified by rule [5] for **Help_r_test_chan**). However each such protocol poses its own challenge where correctness is concerned. See below.

**An Example**

Let us see what our translation scheme does to the small $\pi$-calculus example we discussed in [Section 3]. Here it is again in fully bracketed form.

$$\{ \ \{x(u).\{u(t).\{\mathbf{0}\}_Z\}_U\}_{X1} \mid \{\bar{x}v.\{\bar{v}s.\{\mathbf{0}\}_Z\}_V + \bar{x}v.\{\mathbf{0}\}_Z\}_S \mid \{x(y).\{\mathbf{0}\}_Z\}_{X4} \ \}_T$$

When translated, as well as the protocol rules, the following rules would be generated. For variety, we write them out with rule separators, but emphasise that these merely embody the rule selection strategy mentioned above.

```
Z => *Root                                                    ;

U[u] => #U_Z[^**Help_r[u t] u t:Empty]                        ;

U_Z[Yes u t] => *Z                                            |
U_Z[No u t] => *U[u]                                          ;
U_Z[a u t] => *U_Z[a u t]                                     ;

X1[x] => #X1_U[^**Help_r[x u] x u:Empty]                      ;

X1_U[Yes x u] => *U[u]                                        |
X1_U[No x u] => *X1[x]                                        ;
X1_U[a x u] => *X1_U[a x u]                                   ;

X4[x] => #X4_Z[^**Help_r[x y] x y:Empty]                      ;

X4_Z[Yes x y] => *Z                                           |
X4_Z[No x y] => *X4[x]                                        ;
X4_Z[a x y] => *X4_Z[a x y]                                   ;

V[v s] => #V_Z[^**Help_w[v s] v s]                            ;

V_Z[Yes v s] => *Z                                            |
V_Z[No v s] => *V[v s]                                        ;
V_Z[a v s] => *V_Z[a v s]                                     ;

S[x v s] => #S_V[^**Help_w[x v] x v s]                        |
S[x v s] => #S_Z[^**Help_w[x v] x v s]                        ;

S_V[Yes x v s] => *V[v s]                                     |
S_V[No x v s] => *S[x v s]                                    ;
S_V[a x v s] => *S_V[a x v s]                                 ;

S_Z[Yes x v s] => *Z                                          |
S_Z[No x v s] => *S[x v s]                                    ;
S_Z[a x v s] => *S_Z[a x v s]                                 ;

T[x v s] => *X1[x] , *S[x v s] , *X4[x]                       ;

Initial => *T[x:Empty v:Empty s:Empty]                        ;
```

In [Fig. 9] below we give a picture of the execution graph just after the system has been set in motion; and in [Fig. 10] we show the execution graph just after the first successful data transfer by process S which has synchronised with process X1.

## 6 PROPERTIES OF TRANSLATED SYSTEMS

In this section we state a number of definitions and establish a number of mostly easy lemmas, which enable us to speak more incisively about the structure of execution graphs of translated systems, and about the transitions between them effected by the rules we have proposed. We assume henceforth that we are dealing with complete systems, i.e. given a $\pi$-calculus expression $E$, the translation of $E$ is Tr($E$), which contains a suitable rule for Initial.

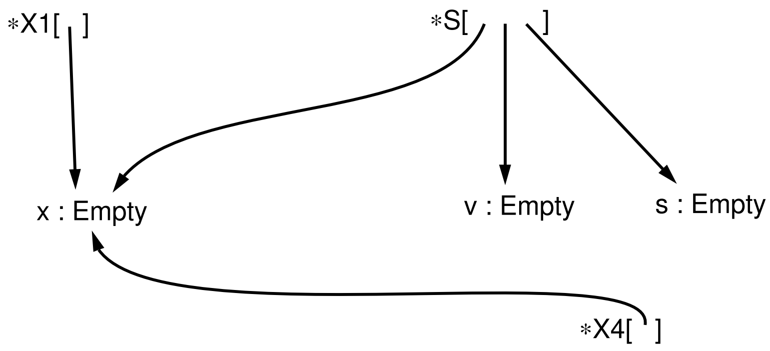


Fig. 9

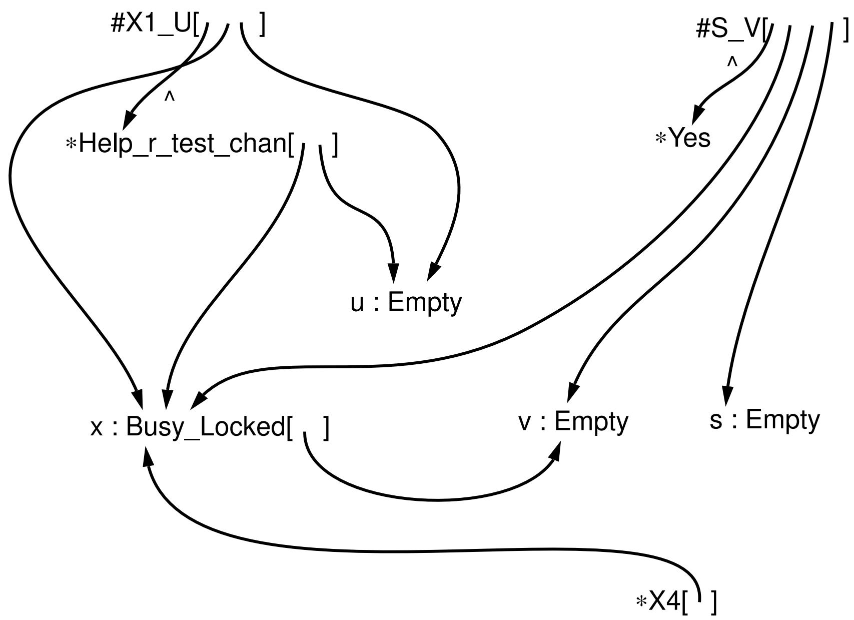


Fig. 10

**Definition 6.1** A function symbol $F$ arising from the translation of a construct $C$ will be called a

- ∗ proposer function symbol if $C$ is $\{\Sigma_i\, \pi_i.Q_i\}_P$ and $F$ is $\mathsf{P}$,
- ∗ proposer-intermediate function symbol if $\mathsf{P}$ is a proposer symbol and $F$ is $\mathsf{P\_Q}_i$ for some summand $Q_i$ of $\{\Sigma_i\, \pi_i.Q_i\}_P$,

* auxiliary function symbol if $C$ is $\{\mathbf{0}\}_P$, $\{\nu x Q\}_P$, or $\{Q_1 \mid Q_2 \mid \ldots \mid Q_n\}_P$ and $F$ is P; and more specifically, P is called a zero symbol, ν symbol, or composition symbol respectively,
* replication function symbol if $C$ is $\{!Q\}_P$ and $F$ is P,
* initial function symbol if $F$ is Initial,
* protocol function symbol if $F$ is one of the functions defined in the communication protocol section of the translation, i.e. Help_r, Help_w, Help_r_test_chan etc.

**Definition 6.2** The stateholder symbols Empty, Busy_Unlocked, and Busy_Locked will be called channel symbols.

**Lemma 6.3** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Then the live part of every execution graph of Tr($E$) has the following properties.

(1) Each non-protocol function node and each Root constructor has no parent.

(2) Each function node is active unless it is a proposer-intermediate node in which case it is either active or suspended.

(3) Each protocol function node has a unique parent which is a suspended proposer-intermediate node. Each active Yes or No constructor node has a unique parent which is a suspended proposer-intermediate node, and each idle Yes or No constructor node has a unique parent which is an active proposer-intermediate node.

(4) Each function node has only channel nodes as children, except for proposer-intermediate nodes, which in addition have as a child either a protocol function node, or a Yes, or a No constructor.

(5) Each protocol function node has only channel nodes as children, either two or three of them according to arity.

(6) Each channel node is an idle Empty, Busy_Unlocked, or Busy_Locked[data], where data is another channel node.

*Proof*. We proceed by induction on the structure of executions. The initial graph obviously satisfies the properties. For the induction step suppose all execution graphs up to $G_i$ in some execution have the properties. Then $G_{i+1}$ is given by either a notification or a rewrite.

For a notification, it is easy to see that all the properties are preserved as the only possible notifications in an execution are from Root constructors which by induction hypothesis (1) for $G_i$ notify nobody, and from Yes and No constructors which by induction hypothesis (3) for $G_i$ notify their unique suspended proposer-intermediate node parent. (All stateholders are always idle by induction hypothesis (6) for $G_i$ so never notify.)

For a rewrite there are a large number of cases to check, six for each rule type in the translation. Fortunately all of them are easy and we briefly examine one example rule and leave the diligent or skeptical reader to check as many others as he wishes. Consider a rule of the form

```
Help_r_test_chan[chan:Busy_Locked[data] chan_in]
 => *Help_r_assign_data[chan data chan_in]
```

By hypothesis (6) for $G_i$, **data** and **chan_in** are (matched in $G_i$ to) idle channel nodes. Therefore the creation of a ∗Help_r_assign_data[chan data chan_in] node as specified in the rule, creates a new protocol function node satisfying (4). The other conditions are equally easy. ☺

**Definition 6.4** Let $D$ be a MONSTR rule, and $p$ a node of the left pattern. We say that $p$ is rewritten to $q$ if

* $D$ specifies a redirection ($p$, $q$), or
* $p$ is the root of the left pattern of $D$ and $q$ is a contractum node.

We also say that $p$ is rewritten to $q$ to refer to the fact that there is a graph $X$, a match $h$ of the left pattern of some rule $D$ to $X$, and a rewrite governed by $D$ with result graph $Y$ and either

* $D$ specifies a redirection ($u$, $w$), and $h(u) = p$ in $X$ is redirected to $r_{X,Y}(h(u)) = q$ in $Y$ where $r_{X,Y}$ is the redirection function from [Section 2], or
* $u$ is the root of the left pattern of $D$, $w$ is a contractum node, $h(u) = p$ is the root of the redex in $X$, and $i_{X',Y}(h'(w)) = q$ is $w$'s copy in $Y$ where $i_{X',Y}$ (the composition $i_{X'',Y} \circ i_{X',X''}$), is the injection function from [Section 2].

The use of the same phraseology to refer to both syntactic and semantic phenomena as legitimised in definition 6.4 avoids excruciating circumlocutions in the discussion below, without losing the reader's conviction that we are telling the truth. Further, we will say that $p$ is rewritten to $q$ when (in the semantic sense) the reflexive transitive closure of the above phenomena is intended, i.e. there is a sequence of zero or more rewrites of $X$ to $X1$ to $X2$ … to $Y$ such that $p$ in $X$ is (in the preceding sense) rewritten to $p1$ in $X1$, which is rewritten to $p2$ in $X2$, … , which is rewritten to $q$ in $Y$. Where necessary, we will allow rewrites which do not pattern match any of the $p$ … , and also notifications, to interrupt the rewriting sequence [$X$ … $Y$].

**Lemma 6.5** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Let P be an initial, auxiliary or replication function symbol of Tr($E$) and let p:P[…] be a node of an execution graph $X$. Then p can be rewritten to a collection of nodes which are active proposer function nodes, active replication nodes, and idle Root constructors.

*Proof.* Since P is a function symbol of Tr($E$), it arises from a labelled subexpression of $E$, $\{\dots\}_P$ say. Consider the parse tree of $\{\dots\}_P$. Each node of the tree with its children corresponds to a syntactic construct of the π-calculus, and each leaf corresponds to a zero; each of these having corresponding rules (call them $\{\}_P$-rules) in Tr($E$). Consider an execution graph $Y$ formed from $X$ by: (a), allowing p to be rewritten using $\{\}_P$-rules corresponding to Initial, | , ν , and **0** as long as there are redexes for such rules; (b) performing all notifications of active Root constructors; (c), allowing a finite number of uses of $\{\}_P$-rules corresponding to replication and applying (a) and (b) to any non-replication nodes generated thereby. Since the parse tree is finite, a finite amount of work is involved. Then $Y$ has the property claimed. ☺

**Definition 6.6** The protocol function symbols Help_r_test_chan, Help_r_assign_data, Help_r_unlock will be called protagonist symbols. All other function symbols will be called non-protagonist symbols. A channel node $c$ is said to be in the chan position of a protocol function node $p$ in a graph $X$ if there is an arc $(p_1, c)$ in $X$, i.e. $c$ occurs in the position matched to **`chan`** in rules for $\sigma(p)$. In this case $p$ is called a chan position parent of $c$.

**Lemma 6.7** Let $E$ be a $\pi$-calculus expression, and Tr($E$) its translation. Then

(I) Every channel node in every execution graph of Tr($E$) is in one of the following states.

($\alpha$) Empty, and all its chan position parents are non-protagonists.

($\beta$) Busy_Unlocked, with exactly one Help_r_test_chan chan position protagonist parent and zero or more other chan position non-protagonist parents,

($\gamma$1) Busy_Locked[data] with exactly one Help_r_test_chan chan position protagonist parent and zero or more other chan position non-protagonist parents,

($\gamma$2) Busy_Locked[data] with exactly one Help_r_assign_data chan position protagonist parent and zero or more other chan position non-protagonist parents,

($\gamma$3) Busy_Locked[data] with exactly one Help_r_unlock chan position protagonist parent and zero or more other chan position non-protagonist parents.

(II) The state changing transitions for a channel node $c$ in chan position in an execution graph are the following (where the reference numbers of the rules used are noted).

($\alpha$) $\rightarrow$ ($\beta$) when $c$ is rewritten by a Help_r chan position parent, [1]

($\beta$) $\rightarrow$ ($\alpha$) when $c$ is rewritten by a Help_r_test_chan chan position parent, [6]

($\beta$) $\rightarrow$ ($\gamma$1) when $c$ is rewritten by a Help_w chan position parent, [3]

($\gamma$1) $\rightarrow$ ($\gamma$2) when $c$ is matched by a Help_r_test_chan chan position parent, [7]

($\gamma$2) $\rightarrow$ ($\gamma$3) when $c$ is matched by a Help_r_assign_data chan position parent, [9]

($\gamma$3) $\rightarrow$ ($\alpha$) when $c$ is rewritten by a Help_r_unlock chan position parent. [11]

(III) The non state changing transitions for a channel node $c$ in chan position in an execution graph are the following (again including rule numbers).

($\alpha$) $\rightarrow$ ($\alpha$) when $c$ is matched by a Help_w chan position parent, [4]

($\beta$) $\rightarrow$ ($\beta$) when $c$ is matched by either a Help_r, or a Help_r_test_chan chan position parent, [2, 5]

($*$) $\rightarrow$ ($*$) when $c$ is matched by either a Help_r or a Help_w chan position parent, where ($*$) is any of ($\gamma$1), ($\gamma$2), ($\gamma$3), [2, 4]

($*$) $\rightarrow$ ($*$) when $c$ is matched by any non protocol chan position function parent, where ($*$) is any of ($\alpha$), ($\beta$), ($\gamma$1), ($\gamma$2), ($\gamma$3).

*Proof.* Again by induction on the structure of executions. In fact we need to strengthen the induction hypothesis by adding a number of clauses. Rather than present them all at once, we will introduce them only as needed in discussing features of the proof.

The base case is trivial, as is the inductive step for notifications. For rewrites, we need to merely check that the rewrite rules which match in chan position indeed implement the required behaviour. This has two aspects. Firstly that the normal protocol rewrite rules effect the transitions stated; and secondly that any transitions in principle permitted by the rules but unstated above, do not in fact take place. The latter transitions are the ones determined by the default rules for Help_r_test_chan, Help_r_assign_data, and Help_r_unlock, as an inspection of the rules used in parts (II) and (III) shows.

To show that these rules are never used, it is sufficient to strengthen the induction hypothesis to assert that

(IV) (a) the chan position child of every Help_r_test_chan function node is a Busy_Unlocked channel node or a Busy_Locked[data] channel node,

(b) that the third (**`chan_in`**) child of every Help_r_assign_data function node is an Empty channel node,

(c) the chan position child of every Help_r_unlock function node is a Busy_Locked[data] channel node,

since then Help_r_test_chan, Help_r_assign_data, and Help_r_unlock will always be able to match a normal rule.

To prove (IV).(b), the only part that doesn't follow from a trivial inspection of the rules, we need to strengthen the induction hypothesis yet further to assert that

(V) (a) a read proposal initiated by a proposer node rewriting to a proposer-intermediate node, instantiates the input node as a new Empty channel node whose only parents are the suspended proposer-intermediate node and the corresponding read helper,

(b) a read helper that matches an Empty chan position channel, passes its Empty input node to Help_r_test_chan (in second position), whereupon the Empty input node's only live parents are the suspended proposer-intermediate node and the Help_r_test_chan node,

(c) a Help_r_test_chan that detects a rendezvous, passes its Empty input node to Help_r_assign_data (in third position), whereupon the Empty input node's only live parents are the suspended proposer-intermediate node and the Help_r_assign_data node.

It is easy to see that (V).(a) $\Rightarrow$ (V).(b) $\Rightarrow$ (V).(c) $\Rightarrow$ (IV).(b).

Checking the induction step for rewrites involves showing that the various rules used, preserve the properties of states claimed in part (I), and implement the various transitions described in the remaining parts. Essentially there are two sorts of deductions.

Firstly, if a rule rewrites its chan position node $c$ then it first matches it explicitly, and in such cases it is a feature of the protocol rewrite rules that all redirections of nodes in chan position by protocol functions are also to explicit nodes of the rule, which makes the behaviour immediately evident.

Otherwise the rule just matches $c$ without redirection. This possibility has two cases, the Help_r_test_chan function which implements ($\gamma$1) $\rightarrow$ ($\gamma$2) by inspection, and the Help_r_assign_data function which implements ($\gamma$2) $\rightarrow$ ($\gamma$3). For the latter we need a final strengthening of the induction hypothesis to assert that

(VI) the chan position child of every Help_r_assign_data function node is a Busy_ Locked[data] channel node,

which is immediate given the form of rule [7] for Help_r_ test_chan.

With the full induction hypothesis laid bare, the induction step for rewrites is mostly trivial, consisting of a large number of rather elementary cases. All that is required is a simple inspection of the information available in the rules, modulo the properties of the set of chan position parents of channel nodes in various states; all subtler properties needed are captured in the various clauses above. We omit the tedious details. ☺

As a corollary to lemma 6.7, we have shown that the communication protocol does not deadlock assuming that every active function node will rewrite eventually, a property that we will formalise as weak fairness in [Section 7]. Part (II) of lemma 6.7 is summarised in the transition diagram of [Fig. 11] below.

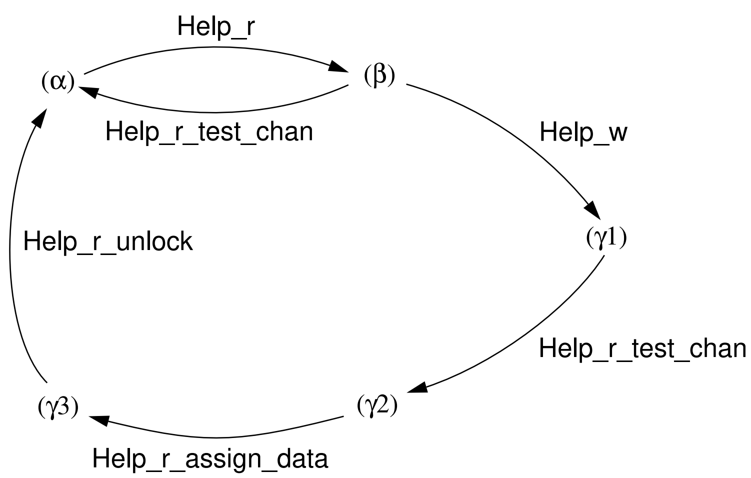


Fig. 11

The final topic we need to consider in this section concerns some subcommutativity lemmas which allow us to permute rewriting steps under suitable circumstances. In the following, for technical simplicity, we assume that we rewrite without removing garbage, so that the injection and redirection functions $i$ and $r$ are total.

**Lemma 6.8** In a MONSTR execution sequence, two adjacent notification steps may be interchanged.

**Lemma 6.9** In a MONSTR execution sequence, a notification step adjacent to a rewrite may be interchanged with it.

Given that notifications merely make the notifying constructor or stateholder idle, and alter the non-idle markings on some other nodes and arcs, we regard the above two results as sufficiently obvious to allow us to both omit their proofs and even be a bit vague in their statement. We are rather more careful about the next lemma, which incidentally provides a guide round which lemmas 6.8 and 6.9 may be rephrased more accurately.

**Lemma 6.10** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Let $G$ be an execution graph of Tr($E$). For $i = 1, 2$, let $D_i = (P_i, root_i, Red_i, Act_i)$ be two rules of Tr($E$) with left subpatterns $L_i$, and let $g_i : L_i \to G$ be two redexes. Suppose that $g_1(root_1) \neq g_2(root_2)$, and that if $s_i$ is an explicit stateholder of $L_i$, then $g_1(s_1) \neq g_2(s_2)$. Let $H_i$ be the graph obtained by rewriting $g_i : L_i \to G$ and let $r_{G,H_i}$ be the corresponding redirection function. For either choice of $i$, let $j$ denote the alternative choice. Then

(1) $D_j$ has a redex $h_i : L_j \to H_i$ in $H_i$ given by

$$h_i = r_{G,H_i} \circ g_j$$

(2) Let $K_i$ be the graph obtained by rewriting $h_i$ and let $r_{H_i,K_i}$ be the respective redirection functions. Then there is an isomorphism $\theta : K_1 \to K_2$ and for all $x \in G$,

$$\theta(r_{H_1,K_1} \circ r_{G,H_1}(x)) = r_{H_2,K_2} \circ r_{G,H_2}(x)$$

(3) Restricted to Live($K_1$), θ provides an isomorphism

$$\theta : \text{Live}(K_1) \to \text{Live}(K_2)$$

*Proof.* Assume that both $L_1$ and $L_2$ explicitly match a stateholder; otherwise we just have a simplified form of what follows. So $g_1(root_1)$, $g_2(root_2)$, $g_1(s_1)$, $g_2(s_2)$ are all distinct nodes of $G$. Consequently, the arcs of $G$ partition into five classes: those with heads at $g_1(root_1)$, at $g_2(root_2)$, $g_1(s_1)$, $g_2(s_2)$, and fifthly those with head at some other node.

Suppose $g_1$ rewrites first. Contractum building, forming $G_1'$ does not alter any arc from $g_2(root_2)$ so $i_{G,G_1'} \circ g_2 : L_2 \to G_1'$ is a redex for $D_2$. Likewise redirection, forming $G_1''$ does not alter (the copy in $G_1'$ of) the arc $(g_2(root_2)_{m2}, g_2(s_2))$ between the root and single explicitly matched stateholder child of the redex $i_{G,G_1'} \circ g_2$. Other arcs of this redex may be affected by the redirection but since they are all implicitly matched, this does no harm and so $r_{G_1',G_1''} \circ i_{G,G_1'} \circ g_2 : L_2 \to G_1''$ is a redex for $D_2$. Finally, the activation phase, forming $H_1$ merely makes the root of $g_1$ idle, so

$$r_{G_1'',H_1} \circ r_{G_1',G_1''} \circ i_{G,G_1'} \circ g_2 = r_{G,H_1} \circ g_2 \ : L_2 \to H_1$$

is a redex for $D_2$ and by symmetry we get (1).

To get (2) we note that the contractum building phases of both rewrites independently add copies of contractum nodes of $P_i$ to the execution graph. Since we are refraining from garbage collection, we immediately infer the existence of a node bijection $\theta : K_1 \to K_2$. This obviously extends to a bijection on tails of arcs, and to show that θ extends to a graph isomorphism, we need to examine the effects of the various redirections on the heads of arcs. This is because the remaining phases, the activation phases, are easily seen to be independent of each other and of the rest of the rewriting phases.

As before, arcs partition into those with heads at (a): $g_1(root_1)$, (b): $g_2(root_2)$, (c): $g_1(s_1)$, (d): $g_2(s_2)$, and (e): none of the preceding. We extend θ to the various sets of arcs as follows.

Obviously if a node is not redirected, one can make it the head of an additional collection of arcs by adding these arcs in any order, so θ extends to arcs in (e) immediately. From the form of Tr($E$) rules and from lemma 6.3.(1) and 6.3.(3), nodes in (a) and (b) either have no parent or have a unique one, so do not acquire any new parents via contractum building, and independently get redirected to contractum nodes. So θ easily extends to arcs in (a) and (b). This leaves the arcs in (c) and (d).

Now by inspection of the rules in Tr($E$), all non-root redirections are of channel nodes to fresh (contractum) channel nodes, apart from in the normal rule [9] for Help_r_assign_data. So if neither $D_1$ nor $D_2$ is rule [9], then we can interchange the the order of rewriting with impunity as contractum building and redirection are independent for the two rewrites. If both $D_1$ and $D_2$ are this rule, then since $s_1$ and $s_2$ are both the matched input nodes of this rule, by clause (V).(c) in the proof of lemma 6.7, neither of them is accessible from the other redex, so neither **`data`** argument matches $s_1$ or $s_2$ and the rewrites may again be swapped.

If say $D_1$ is rule [9] and $D_2$ is not, if $D_1$'s **`data`** argument does not match $s_2$, then we can obviously swap the rewrites as before. If $D_1$'s **`data`** argument does match $s_2$, then if $D_1$ is done first, we find that the arcs in (c) get redirected to $s_2$, (from the form of $D_1$ it is clear that no new arcs are added to (c) during contractum building); and then they and all arcs in (d) (both arcs existing in $H_1$ and any new ones added during contractum building) get redirected to $z_2$, the redirection target of $s_2$. Otherwise the $D_2$ rewrite is done first, and redirects the arcs in (d) to $z_2$ (these are arcs existing in $H_2$, any new ones added during contractum building, and including any arc(s) of the $D_1$ redex that matched the **`data`** argument of Help_r_assign_data). The $D_1$ redex, now $r_{G,H_2} \circ g_1$, redirects the arcs in (c) to $z_2$ directly. A symmetric argument works if $D_2$ is rule [9], so the results of either rewriting sequence are equivalent and $\theta : K_1 \to K_2$ extends to a graph isomorphism as claimed.

To conclude that θ restricts to an isomorphism on live parts, we note that θ preserves graph structure and node symbols, but particularly that it also preserves node and arc markings. Therefore the rules of inference in definition [2.12.(1) – (4)] are invariant under θ, and a proof of liveness of a node or arc in $K_1$ will map by induction into a proof of liveness of a node or arc in $K_2$, and vice versa. Likewise for garbage. This gives us (3). ☺

In the above, we have indicated that we can interchange the order of rewriting given some mild conditions. Essentially the same result holds true for quite arbitrary MONSTR systems, though we have to take note of non-trivial activations and dynamic suspensions. However in general, we do not have the wealth of detailed information about the structure of execution graphs that we exploited to make the preceding proof fairly straightforward, and the demonstration in the general case is rather more arduous.

# 7 CORRECTNESS OF THE TRANSLATION

What can we say about the correctness of the translation? When source and target languages are as far apart as in this case, one has to be careful about what one means by correctness. For instance in the π-calculus itself, there is no possibility that a single communication may thrash without making progress, whereas the busy waiting feature of the MONSTR communication protocol leaves Tr($E$) systems open to the possibility of readers and writers perpetually failing to make appropriate contact, and the system as a whole stalling thereby, despite there being a copious quantity of rewrites being performed. Under such circumstances it is reasonable to guard claims of correctness by suitable fairness assumptions. Even with fairness though, an execution of the MON-

STR system will contain quite an amount of fruitless work, as processes make attempts to communicate which are futile for one reason or another; eg. there may be attempts to communicate while offers or rendezvouses are in progress on the same channel, or when there are no available writers on the channel in question.

Our claim for correctness amounts to two facts. The first states that "anything that a $\pi$-calculus expression $E$ can do, can be simulated by its translation Tr($E$)". It is a completeness statement proved by establishing the existence of a "standard simulation" of any $\pi$-calculus expression and appears as theorem 7.9. The second states that "anything that the translation Tr($E$) of a $\pi$-calculus expression $E$ can do, corresponds in a certain sense to (at least a prefix of) a trace of replication and comunication steps of the original $\pi$-calculus expression $E$". This soundness result is guarded by a fairness assumption, and involves a fairly intricate manipulation of an arbitrary fair execution until it becomes a standard simulation. It appears as theorem 7.16. The details of the manipulation constitute one of the main technical contributions of this paper.

**Definition 7.1** Let $E$ be a $\pi$-calculus expression. A top-level parallel subexpression (TLPSE) of $E$ is a subexpression of the form $\{\mathbf{0}\}_P$, $\{!Q\}_P$, or $\{\Sigma_i\, \pi_i.Q_i\}_P$, such that there is no subexpression of one of these forms that properly contains it.

So a general $\pi$-calculus expression is built up out of TLPSEs by using $\nu$ and parallel composition, as one would expect.

**Definition 7.2** Let $E$ be a $\pi$-calculus expression, and Tr($E$) its translation. An execution graph $G$ of Tr($E$) represents $E$ if

(1) There is a bijection $\rho$ between TLPSEs of $E$ and live nodes of $G$ such that

- $\rho(\{\mathbf{0}\}_P)$ is an idle Root constructor node,
- $\rho(\{!Q\}_P)$ is an active replication function node $*\mathsf{P}[\ldots]$,
- $\rho(\{\Sigma_i\, \pi_i.Q_i\}_P)$ is an active proposer function node $*\mathsf{P}[\ldots]$.

(2) $\rho$ extends to a bijection between free channel names of the TLPSEs of $E$ and idle Empty channel nodes of $G$ such that

- if channel $x$ is free in the TLPSEs $E_1 \ldots E_k$ then there is a normal arc from each of the $\rho(E_i)$ to $\rho(x)$.

Note that in a representation of an expression $E$, the bound names of top-level $\nu$'s (those not occurring inside any TLPSE), are "unwrapped" and appear explicitly in the graph. For our example $\pi$-calculus expression, [Fig. 9] above represents the original expression, before any communications have taken place.

**Definition 7.3** Let $E$ be a $\pi$-calculus expression, and Tr($E$) its translation. We denote by $\rightarrow_{\mathrm{A}(E)}$ the relation on execution graphs of Tr($E$) given by rewriting from an active initial or auxiliary function node, or by notifying from an active Root constructor. We denote by $\rightarrow_{\mathrm{R}(E)}$ the relation on execution graphs of Tr($E$) given by rewriting from an active replication function node. We denote by $\rightarrow_{\mathrm{C}(E)}$ the relation on execution graphs of Tr($E$) given by rewriting from an active proposer, or proposer-intermediate, or protocol function node, or by notifying from an active Yes or No constructor. We write eg. $\rightarrow_{\mathrm{A}(E)}{}^+$ or $\rightarrow_{\mathrm{A}(E)}{}^*$ for the transitive, resp. reflexive transitive, closure of these.

**Lemma 7.4** Let $E$ be a $\pi$-calculus expression, and Tr($E$) its translation. Then there is an execution graph $G$ of Tr($E$) such that

$$*\mathsf{Initial} \rightarrow_{\mathrm{A}(E)}{}^{+} G$$

and $G$ represents $E$.

*Proof.* Essentially this is a byproduct of lemma 6.5. We rewrite $*\mathsf{Initial}$ using rules for $\mathsf{Initial}$, $|$ , $\nu$, and perform notifications by $\mathsf{Root}$ constructors as long as we can do so yielding $G$. An induction on the structure of the derivation of $G$ using the structure of the "rewrites to" relation of definition 6.4 ensures that the correct function nodes are generated, and the properties of the $Args_{\mathrm{P}}$ and $blend_{\mathrm{P}}$ functions of the translation ensure that suitable channel nodes are linked to the correct function nodes. ☺

Most of the remaining results in this section must be understood as holding up to isomorphism, or up to isomorphism of live subgraphs (as appropriate), as in lemma 6.10. We abuse language somewhat by not mentioning the relevant mappings.

**Lemma 7.5** Let $E$ be a $\pi$-calculus expression, and Tr($E$) its translation. Let execution graph $G$ of Tr($E$) represent $E$. Let $E \Rrightarrow^* F$ be a reduction to standard form of $E$. Then $G$ represents $F$, provided TLSEs of $E$ and $F$ are consistently tagged and such tags are consistently translated into function symbols.

*Proof.* Consider the defining clause of $\Rrightarrow$ in tagged form.

$$\{\ \{\nu z\{Q_1\}_{Q_1}\}_N \mid Q_2 \mid \dots \mid Q_n\ \}_P \Rrightarrow \{\ \nu z'\{\{Q_1\}_{Q_1} \mid Q_2 \mid \dots \mid Q_n\ \}_{P'}\ \}_{N'}$$

Here we have tagged the subexpresion $Q_1$ consistently on both sides (leading to the same function symbol $\mathsf{Q}_1$) and have introduced new tags for the restriction and parallel combinators on the RHS. Consider the rules generated by the translations of the LHS and RHS. On both sides, the rules for $\mathsf{Q}_1$, translating $Q_1$, would be identical since the free and bound names of $Q_1$ are the same on both sides. The same applies to the other $Q_i$ by lemma 3.9. On the LHS, $\mathsf{P}$ would spawn nodes for $\mathsf{Q}_2$ to $\mathsf{Q}_n$ and $\mathsf{N}$, the last of which would create a fresh $\mathsf{Empty}$ channel node for $z$ and then spawn $\mathsf{Q}_1$. On the RHS, $\mathsf{N}'$ would create a fresh $\mathsf{Empty}$ channel node for $z'$ and then spawn $\mathsf{P}'$, which would spawn nodes for $\mathsf{Q}_1$ to $\mathsf{Q}_n$. Clearly in an execution, the nodes $\mathsf{P}$, $\mathsf{N}$, (resp. $\mathsf{P}'$, $\mathsf{N}'$), would be garbaged, so the live subgraphs containing $\mathsf{Q}_1$ to $\mathsf{Q}_n$ and the channel nodes that they refer to would be isomorphic. The same applies if the expressions shown on the LHS and RHS were merely subexpressions at top level of a larger expression, and also if the $Q_i$ merely contained restrictions and and parallel compositions of TLSEs. The rest of the proof is an induction on the length of the derivation $E \Rrightarrow^* F$. ☺

We note that if we had included the first clause of remark 3.10 in our definition of reduction to standard form, then $\nu z$-binders in the interior of a summand would have been able to float above such summands making the bound channel names in question free in those summands. This would have destroyed the isomorphism (up to garbage) of lemma 7.5 since the representatives of such summands would now have had extra arguments to these channel nodes. However these would just have been dummy arguments, carried around dormant until they were actually needed, so would not have altered the behaviour of the translation.

**Lemma 7.6** Let $E_1$ be a π-calculus expression, and Tr($E_1$) its translation. Let $E_1 \rightarrow_R E_2$ be a replication rewrite of $E_1$. Then there are execution graphs $G_1$ and $G_2$ such that $G_1$ represents $E_1$ and $G_2$ represents $E_2$ and

$$\begin{tikzcd} G_1 \arrow[r, "\mathrm{R}(E_1)"] & H_1 \arrow[r, "\mathrm{A}(E_1)^*"] & G_2 \\ *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_1)^*"'] & *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_1)^*"'] & *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_2)^*"'] \end{tikzcd}$$

*Proof.* It is clear that there is an execution graph $G_1$ that represents $E_1$ by lemma 7.4. Consider the expression $E_1$ and its replicated subexpression $\{!Q\}_P$. The latter corresponds to a replication node P of $G_1$. Assuming "$_Q$" labels the subexpression $Q$, rewriting $*$P[…] yields a graph $H_1$ in which $*$P[…] is replaced by $*$P[…], $*$Q[…]. Now a replication rewrite of $E_1 = \ldots \{!Q\}_P \ldots$ yields $E_2 = \ldots \{\{Q\}_Q \mid \{!Q\}_P\}_R \ldots$. Assuming that $R$ is fresh, that the other subexpressions of $E_2$ are labelled as in $E_1$, and assuming that the new instance of $Q$ and its subexpressions are labelled identically to the instance inside $\{!Q\}_P$, it is easy to see by using lemma 6.5 to reduce $*$Q[…] to proposer nodes, idle Root nodes and replication nodes, that a representation $G_2$ of $E_2$ results. ☺

**Lemma 7.7** Let $E_1$ be a π-calculus expression, and Tr($E_1$) its translation. Let $E_1 \rightarrow_C E_2$ be a communication rewrite of $E_1$. Then there are execution graphs $G_1$ and $G_2$ such that $G_1$ represents $E_1$ and $G_2$ represents $E_2$ and

$$\begin{tikzcd} G_1 \arrow[r, "\mathrm{C}(E_1)^+"] & H_1 \arrow[r, "\mathrm{A}(E_1)^*"] & G_2 \\ *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_1)^*"'] & *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_1)^*"'] & *\mathsf{Initial} \arrow[u, "\mathrm{A}(E_2)^*"'] \end{tikzcd}$$

*Proof.* By lemma 7.4 we obtain a $G_1$ that represents $E_1$. Let the communication step $E_1 \rightarrow_C E_2$ involve the summands $\{(\ldots + \pi_r.Q' + \ldots)\}_{P'}$ and $\{(\ldots + \pi_w.Q'' + \ldots)\}_{P''}$ where $\pi_w$ is a write prefix, and $\pi_r$ is a suitable read prefix, giving

$$\begin{aligned} &(\ \ldots \mid \{(\ldots + \pi_r.Q' + \ldots)\}_{P'} \mid \{(\ldots + \pi_w.Q'' + \ldots)\}_{P''} \mid \ldots\ ) \\ &\qquad \rightarrow_C (\ \ldots \mid Q'[\chi] \mid Q'' \mid \ldots\ ) \end{aligned}$$

where $Q'[\chi]$ is $Q'$ with the appropriate channel substitution applied. The TLPSEs $P'$ and $P''$ correspond to proposer nodes P′ and P″ in $G_1$. Running the communication protocol for these nodes for the choices P′ => P′_Q′ and P″ => P″_Q″ we obtain $H_1$; in which P′ and P″ have been replaced by Q′ and Q″, the input channel node of Q′ has been instantiated and redirected, and perhaps some other channels of P′ and P″ have been garbaged.

The subgraph of $H_1$ which omits Q′ and Q″ (and their out-arcs, and any garbage that this omission generates), is a representation of the subexpression of $E_2$ which omits the summands $Q'[\chi]$ and $Q''$, provided that the other subexpressions of $E_2$ are labelled consistently with their $E_1$ counterparts. $H_1$ itself fails to represent $E_2$ unless Q′ and Q″ are proposer or replication nodes, but only for this reason. Applying lemma 6.5 to $H_1$ to reduce Q′ and Q″ to proposer nodes, idle Roots and replicator nodes using auxiliary rules and notifications only, we obtain $G_2$ which does represent $E_2$, and which can obviously be obtained from $*\mathsf{Initial}$ using $\rightarrow_{A(E_2)}{}^*$ only, provided as before, we label the subexpressions of $E_1$ and $E_2$ consistently. ☺

**Theorem 7.8** Let $E \equiv E_0$ be a π-calculus expression, and $\mathrm{Tr}(E)$ its translation. Let $\rightarrow_{A(E)}$, $\rightarrow_{R(E)}$ and $\rightarrow_{C(E)}$ be the corresponding rewrite relations. Then for any trace of communication and replication steps from $E$, eg.

$$E \equiv E_0 \rightarrow_{C} E_1 \rightarrow_{R} E_2 \rightarrow \ldots$$

there is a rewriting sequence

$$\begin{tikzcd}
G_0 \arrow[r, "{C(E)}"', "+"] & H_0 \arrow[r, "{A(E)}"', "*"] & G_1 \arrow[r, "{R(E)}"'] & H_1 \arrow[r, "{A(E)}"', "*"] & G_2 \arrow[r] & \ldots \\
*\mathsf{Initial} \arrow[u, "*", "{A(E_0)}"'] & & *\mathsf{Initial} \arrow[u, "*", "{A(E_1)}"'] & & *\mathsf{Initial} \arrow[u, "*", "{A(E_2)}"'] &
\end{tikzcd}$$

such that for each $i$, $G_i$ represents $E_i$.

*Proof.* We assume that we label all the subexpresions of the $E_i$ consistently as we did in lemmas 7.6, 7.7. Noting that $\rightarrow_{C(E_i)} \subseteq \rightarrow_{C(E)}$, that $\rightarrow_{R(E_i)} \subseteq \rightarrow_{R(E)}$, and that $\rightarrow_{A(E_i)} \subseteq \rightarrow_{A(E)}$, if for no other reason than that $E_i$ is an execution graph of $\mathrm{Tr}(E_0)$, we can use induction on the structure of the trace from $E$, using lemmas 7.6, 7.7 for the induction steps. ☺

**Theorem 7.9** Let $E \equiv E_0$ be a π-calculus expression, and $\mathrm{Tr}(E)$ its translation. Let $\rightarrow_{A(E)}$, $\rightarrow_{R(E)}$ and $\rightarrow_{C(E)}$ be the corresponding rewrite relations. Then for any trace of communications and replications from $E$ enhanced by reduction to a standard form at each step, eg.

$$E \equiv E_0 \Rrightarrow^* F_0 \rightarrow_{C} E_1 \Rrightarrow^* F_1 \rightarrow_{R} E_2 \Rrightarrow^* F_2 \rightarrow \ldots$$

there is a rewriting sequence

$$\begin{tikzcd}
G_0 \arrow[r, "{C(E)}"', "+"] & H_0 \arrow[r, "{A(E)}"', "*"] & G_1 \arrow[r, "{R(E)}"'] & H_1 \arrow[r, "{A(E)}"', "*"] & G_2 \arrow[r] & \ldots \\
*\mathsf{Initial} \arrow[u, "*", "{A(E_0)}"'] & & *\mathsf{Initial} \arrow[u, "*", "{A(E_1)}"'] & & *\mathsf{Initial} \arrow[u, "*", "{A(E_2)}"'] & \\
*\mathsf{Initial} \arrow[uu, bend left, "{A(F_0)}", "*"'] & & *\mathsf{Initial} \arrow[uu, bend left, "{A(F_1)}", "*"'] & & *\mathsf{Initial} \arrow[uu, bend left, "{A(F_2)}", "*"'] &
\end{tikzcd}$$

such that for each $i$, $G_i$ represents both $E_i$ and $F_i$.

*Proof.* This is a straightforward enhancement of the induction of theorem 7.8. By lemma 7.5 we note that up to garbage, the same execution graph $G_i$ will do duty for both $E_i$ and $F_i$, so once we note that a replication can be done in $F_i$ iff it can be done in $E_i$ (and thus can be simulated in $G_i$), we just need to check that $G_i$ is capable of performing all the communications of $F_i$ which were unavailable to $E_i$. That this holds, follows once we note that in our translation, sending and receiving are independent activities apart from their need to synchronise. Thus the sender's success in transmitting a ν-bound channel does not depend on knowing that the receiver is in the scope of the ν. Equally, the receiver's success is independent of whether or not the received channel is ν-bound, and if so of what the ν's scope is. ☺

The later remarks in the above proof bring out with some force the fact that our graph based formalism, by directly expressing connectivity in a communication network via connectivity in the graph, handles easily issues that demand some technical pain in the π-calculus.

**Definition 7.10** Let $E$ be a π-calculus expression, and $\mathrm{Tr}(E)$ its translation. An execution such as the one described in theorem 7.9 is called a standard execution of $\mathrm{Tr}(E)$.

Theorem 7.9 says that the translation is complete in providing a standard execution of any possible trace of $E$. It is clear from the properties of standard executions that they are in fact weak simulations of the traces of $E$.

Soundness is rather harder. Given an arbitrary execution $\mathcal{H}$ of a system $\mathrm{Tr}(E)$, we want to show that there is a trace $\mathcal{T}$ of communications and replications enhanced by reduction to standard form from $E$, such that $\mathcal{H}$ corresponds to $\mathcal{T}$ in some acceptable way. Given the standard executions furnished by theorem 7.9 for $\mathrm{Tr}(E)$ systems, we regard it as sufficient to manufacture from $\mathcal{H}$ a standard execution $\mathcal{G}$ that is equivalent to $\mathcal{H}$ in a convincing sense. This manufacturing process has to accomplish a number of things. The actions corresponding to a successful run of the communications protocol have to be clustered together (the reference points for this are the commit rewrites of the protocol, the rewrites governed by the Help_w normal rule); waste rewrites corresponding to unsuccessful essays of the protocol must be eliminated; and rewrites using auxiliary rules must be suitably clustered to ensure, for each successful commit point or replication point (say the $i$'th), before the initiation of the communication or replication, that $G_i$ really does represent $E_i$.

For future notational convenience, if $x$ is a node of an execution graph $G_i$ of an execution $\mathcal{G}$, we presuperscript it, writing ${}^{(i)}x$, to distinguish it from other nodes in other execution graphs of $\mathcal{G}$. So by definition, ${}^{(i)}x \in G_i$. If $j \geq i$ we write ${}^{(j)}x$ for $r_{G_i,G_j}(x) = r_{G_i,G_j}({}^{(i)}x)$ where as in previous sections, $r_{G_i,G_j}$ is the function which maps ${}^{(i)}x$ in $G_i$ to its redirection target in $G_j$. Also when more than one execution is being discussed, we additionally presubscript nodes with notation to indicate which execution they belong to, eg. ${}^{(i)}{}_{\mathrm{G}}x \in G_i \in \mathcal{G}$.

**Definition 7.11** Let $\mathcal{R}$ be a rule system and $\mathcal{X} = [X_0, X_1, X_2, \ldots]$ and $\mathcal{Y} = [Y_0, Y_1, Y_2, \ldots]$ be two executions of $\mathcal{R}$ where both $X_0$ and $Y_0$ are the initial graph. We define the relation $\Xi$ between nodes of execution graphs of $\mathcal{X}$ and $\mathcal{Y}$ as follows.

(1) ${}^{(0)}{}_{X_0}\mathsf{init} \;\Xi\; {}^{(0)}{}_{Y_0}\mathsf{init}$, where ${}^{(0)}{}_{X_0}\mathsf{init}$ and ${}^{(0)}{}_{Y_0}\mathsf{init}$ are the initial nodes of $X_0$ and $Y_0$,

(2) If ${}^{(i)}x \;\Xi\; {}^{(j)}y$, $i \leq i'$, $j \leq j'$, then ${}^{(i')}x \;\Xi\; {}^{(j')}y$,

(3) If ${}^{(i)}x \;\Xi\; {}^{(j)}y$, and ${}^{(i)}x$ and ${}^{(j)}y$ are roots of redexes of the same rule $D$ of $\mathcal{R}$, and $X_{i+1}$ and $Y_{j+1}$ are the results of rewriting these redexes, then if ${}^{(i+1)}x' \in X_{i+1}$ and ${}^{(j+1)}y' \in Y_{j+1}$ are copies of the same contractum node $p$ of the full pattern of the rule $D$ introduced during these rewrites, then ${}^{(i+1)}x' \;\Xi\; {}^{(j+1)}y'$.

By clause (1) of the above, any two executions of $\mathcal{R}$ are $\Xi$-related to some extent. This relationship may easily be a not very useful one if the two executions swiftly diverge from one another, the Ξ-related parts becoming consigned to garbage. However, the more closely the two executions follow one another disregarding inessential detail, the larger the proportion of execution node instances in the first execution, that will be in a useful non-garbage way, Ξ-related to counterparts in the second execution.

We can extend Ξ to arcs, connectivity, and other graph theoretic concepts as required. For example.

**Definition 7.12** With the provisions of definition 7.11 understood, let $({}^{(i)}p_k, {}^{(i)}c)$ be an arc of $X_i$ and $({}^{(j)}q_k, {}^{(j)}d)$ be an arc of $Y_j$. If ${}^{(i)}p \ \Xi \ {}^{(j)}q$, and ${}^{(i)}c \ \Xi \ {}^{(j)}d$ then we say $({}^{(i)}p_k, {}^{(i)}c) \ \Xi \ ({}^{(j)}q_k, {}^{(j)}d)$.

**Lemma 7.13** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Let $\mathcal{G} = [G_0, G_1, \ldots]$ and $\mathcal{H} = [H_0, H_1, \ldots]$ be two executions of Tr($E$) such that

(a) $G_i = H_i$ for $i \in [\ 0 \ldots m, m{+}2 \ldots\ ]$.

(b) $G_m$ contains two redexes rooted at ${}^{(m)}{}_{\mathrm{G}}r_1$ and ${}^{(m)}{}_{\mathrm{G}}r_2$ which satisfy the hypotheses of lemma 6.10. Understood as nodes in $H_m$, these roots are written ${}^{(m)}{}_{\mathrm{H}}r_1$ and ${}^{(m)}{}_{\mathrm{H}}r_2$.

(c) In $\mathcal{G}$, $G_m \to G_{m+1}$ rewrites the redex rooted at ${}^{(m)}{}_{\mathrm{G}}r_1$ and $G_{m+1} \to G_{m+2} = H_{m+2}$ rewrites the redex rooted at ${}^{(m+1)}{}_{\mathrm{G}}r_2$. While in $\mathcal{H}$, $H_m \to H_{m+1}$ rewrites the redex rooted at ${}^{(m)}{}_{\mathrm{H}}r_2$ and $H_{m+1} \to H_{m+2} = G_{m+2}$ rewrites the redex rooted at ${}^{(m+1)}{}_{\mathrm{H}}r_1$.

Then

(1) For every node ${}^{(i)}x \in G_i \in \mathcal{G}$, ${}^{(i)}x \ \Xi \ {}^{(j)}y$ for some ${}^{(j)}y \in H_j \in \mathcal{H}$, and if ${}^{(i)}x$ is live then ${}^{(j)}y$ is live. And conversely.

(2) For every arc $({}^{(i)}p_k, {}^{(i)}c) \in G_i \in \mathcal{G}$, $({}^{(i)}p_k, {}^{(i)}c) \ \Xi \ ({}^{(j)}q_k, {}^{(j)}d)$ for some arc $({}^{(j)}q_k, {}^{(j)}d) \in H_j \in \mathcal{H}$. And conversely.

*Proof*. For $i \in [\ 0 \ldots m, m{+}2 \ldots\ ]$ we can obviously set Ξ to (the closure under the recursive clause of definition 7.11 of) the identity relation on nodes ${}^{(i)}x \in G_i = H_i$ and correspondingly for arcs. For $i = m{+}1$, for non-contractum nodes arising from ${}^{(m)}{}_{\mathrm{G}}x = {}^{(m)}{}_{\mathrm{H}}x$, we set ${}^{(m+1)}{}_{\mathrm{G}}x \ \Xi \ {}^{(m+1)}{}_{\mathrm{H}}x$. For contractum nodes, if $p$ is a contractum node of the rule for the $r_1$ rewrite, we set ${}^{(m+1)}{}_{\mathrm{G}}p \ \Xi \ {}^{(m+2)}{}_{\mathrm{H}}p$ in an obvious notation for the introduced copies, and likewise ${}^{(m+2)}{}_{\mathrm{G}}q \ \Xi \ {}^{(m+1)}{}_{\mathrm{H}}q$ for a contractum node $q$ of the rule for the $r_2$ rewrite. The results for arcs follow readily, and the converses are immediate. ☺

**Corollary 7.14** The results of Lemma 7.13 hold when one or both of $G_m \to G_{m+1}$ and $G_{m+1} \to G_{m+2}$ are notifications.

**Definition 7.15** Let $\mathcal{G}$ be an execution of a rule system $\mathcal{R}$. Let ${}^{(i)}x$ be an active node of $G_i$ in $\mathcal{G}$. Suppose for some $j \geq i$ the next execution step is either a notification by the constructor/stateholder ${}^{(j)}x$ or a rewrite of a redex whose root is ${}^{(j)}x$. If for each $i$ and for each active ${}^{(i)}x$ in $G_i$ there is such a $j$, then we say the execution is weakly fair.

Note that standard executions as per theorem 7.9 are not necessarily weakly fair unless the trace of $E$ satisfies additional “reasonableness” criteria.

Now for the main theorem.

**Theorem 7.16** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Let $\mathcal{H} = [H_0, H_1, \ldots]$ be a weakly fair execution of Tr($E$). Then there is a trace $\mathcal{T}$ of communications and replications from $E$, enhanced by reductions to standard form, such that for a standard execution $\mathcal{G} = [G_0, G_1, \ldots]$ of Tr($E$) corresponding via theorem 7.9 to $\mathcal{T}$, for some prefix $[G_0, G_1, \ldots, G_N]$ of $\mathcal{G}$, with $0 \le N \le \infty$, if $i \le N$

(1) For every node ${}^{(i)}x \in G_i \in \mathcal{G}$, ${}^{(i)}x \mathrel{\Xi} {}^{(j)}y$ for some ${}^{(j)}y \in H_j \in \mathcal{H}$, and if ${}^{(i)}x$ is live then ${}^{(j)}y$ is live.

(2) For every arc $({}^{(i)}p_k, {}^{(i)}c) \in G_i \in \mathcal{G}$, $({}^{(i)}p_k, {}^{(i)}c) \mathrel{\Xi} ({}^{(j)}q_k, {}^{(j)}d)$ for some arc $({}^{(j)}q_k, {}^{(j)}d) \in H_j \in \mathcal{H}$.

(3) If $H_j \to H_{j+1} \in \mathcal{H}$ is a rewrite of a redex rooted at ${}^{(j)}y \in H_j$ and governed by either a rule for a replication symbol, or a communication commit rule, there is a corresponding rewrite $G_i \to G_{i+1} \in \mathcal{G}$ of a redex rooted at a node ${}^{(i)}x \in G_i$, which is governed by the same rule, and ${}^{(i)}x \mathrel{\Xi} {}^{(j)}y$. Further, if $H_j \to H_{j+1}$ and $H_{j'} \to H_{j'+1}$ are two distinct such rewrites, their corresponding $G_i \to G_{i+1}$ and $G_{i'} \to G_{i'+1}$ are also distinct, and all such rewrites occur in the same order in $\mathcal{H}$ and $\mathcal{G}$.

Thus on the one hand every part of the standard execution $\mathcal{G}$ can be located in $\mathcal{H}$; on the other, every replication or communication step of $\mathcal{H}$ can be found in $\mathcal{G}$ also. On this basis $\mathcal{G}$, which faithfully depicts $\mathcal{T}$, shows that the essence of any execution of Tr($E$) corresponds to a trace of $E$, giving soundness.

*Proof*. The proof proceeds through a number of phases, gradually transforming $\mathcal{H}$ into the required $\mathcal{G}$, while retaining the properties (1) – (3). Most of the phases are fairly similar so we treat the first in detail, and the others more curtly.

The first few phases eliminate "waste work" of various kinds from the execution $\mathcal{H}$.

Let $\mathcal{X}^{\,O.0} = [X_0^{\,O.0}, X_1^{\,O.0}, \ldots]$ be a working name for the execution $\mathcal{H}$.

**PHASE I** — Elimination of failed write attempts. These arise from the following sequence of events. (Here and below, the communication protocol rules used in each event are indicated by their reference numbers.)

(a) A proposer node ${}^{(a)}p$ rewrites to a proposer-intermediate node ${}^{(a+1)}p$ with a Help_w child ${}^{(a+1)}h$.

(b) The Help_w child ${}^{(b)}h$ matches a non-Busy_Unlocked channel node ${}^{(b)}c$ and rewrites to a No constructor ${}^{(b+1)}h$ using its default rule. [4]

(c) The No constructor ${}^{(c)}h$ notifies its suspended parent ${}^{(c)}p$.

(d) The parent ${}^{(d)}p$ matches the No constructor ${}^{(d)}h$ and reverts to a proposer node ${}^{(d+1)}p$.

Weak fairness assures us that once such a sequence of events starts within $\mathcal{X}^{\,O.0}$ it runs to completion. So as indices of graphs of $\mathcal{X}^{\,O.0}$, a < b < c < d. Suppose further that no a′ < a is the first element of such a subsequence of $\mathcal{X}^{\,\mathrm{O.0}}$, so we are dealing with the first failed write attempt. Let $\mathcal{X}^{\,\mathrm{O.1}}$ be the sequence of graphs obtained from $\mathcal{X}^{\,\mathrm{O.0}}$ by deleting the rewrites/notifications mentioned in (a) – (d). Thus $X_0{}^{\mathrm{O.1}} = X_0{}^{\mathrm{O.0}}$; $X_1{}^{\mathrm{O.1}} = X_1{}^{\mathrm{O.0}}$ ; … ; $X_{\mathrm{a}}{}^{\mathrm{O.1}} = X_{\mathrm{a}}{}^{\mathrm{O.0}}$; $X_{\mathrm{a+1}}{}^{\mathrm{O.1}}$ is similar to $X_{\mathrm{a+2}}{}^{\mathrm{O.0}}$ in that the latter has a Help_w node, and the symbol labelling its parent ${}^{(\mathrm{a+2})}{}_0p \in X_{\mathrm{a+2}}{}^{\mathrm{O.0}}$ is a proposer-intermediate symbol, while the symbol labelling its counterpart ${}^{(\mathrm{a+1})}{}_1p \in X_{\mathrm{a+1}}{}^{\mathrm{O.1}}$ is the original proposer symbol; similarly for $X_{\mathrm{a+2}}{}^{\mathrm{O.1}}$ and $X_{\mathrm{a+3}}{}^{\mathrm{O.0}}$ etc.; $X_{\mathrm{b}}{}^{\mathrm{O.1}}$ is similar to $X_{\mathrm{b+2}}{}^{\mathrm{O.0}}$ and so on; $X_{\mathrm{c-1}}{}^{\mathrm{O.1}}$ is similar to $X_{\mathrm{c+2}}{}^{\mathrm{O.0}}$ etc.; $X_{\mathrm{d-2}}{}^{\mathrm{O.1}} = X_{\mathrm{d+2}}{}^{\mathrm{O.0}}$ up to garbage (since we have reached the point where the failed write attempt has aborted) and so on.

We claim that $\mathcal{X}^{\,\mathrm{O.1}}$ is an execution of Tr($E$). This is easy to see since no rewrite of $\mathcal{X}^{\,\mathrm{O.1}}$ matches a node whose symbol has been changed compared to $\mathcal{X}^{\,\mathrm{O.0}}$ (the only such node being $p$ mentioned above in (a) – (d), which being a proposer/proposer-intermediate node, has no parents by lemma 6.3.(1)). Therefore all transitions $X_i{}^{\mathrm{O.1}} \to X_{i+1}{}^{\mathrm{O.1}}$ of $\mathcal{X}^{\,\mathrm{O.1}}$ are legal execution sequence steps: the notifications obviously so, and the rewrites also legally so since no change of rule selection is necessitated by the change of symbol of $p$.

We can now establish the conclusions (1) – (3) of the theorem for $\mathcal{X}^{\,\mathrm{O.1}}$. To get (1), we see that for each node ${}^{(i)}{}_1x \in X_i{}^{\mathrm{O.1}}$ we have

$$ {}^{(i)}{}_1x \;\Xi\; {}^{(i+\delta)}{}_0x $$

where ${}^{(i+\delta)}{}_0x \in X_{i+\delta}{}^{\mathrm{O.0}}$ and where

$$ \begin{aligned} \delta \;&= 0 \quad \text{if } 0 \le i \le \mathrm{a}, \\ &= 1 \quad \text{if } \mathrm{a}+1 \le i \le \mathrm{b}-1, \\ &= 2 \quad \text{if } \mathrm{b} \le i \le \mathrm{c}-2, \\ &= 3 \quad \text{if } \mathrm{c}-1 \le i \le \mathrm{d}-3, \\ &= 4 \quad \text{if } \mathrm{d}-2 \le i. \end{aligned} $$

Clearly if ${}^{(i)}{}_1x$ is live then so is ${}^{(i+\delta)}{}_0x$ given the relatively slight changes made to the execution.

In a similar vein, for (2) we can see that arcs behave well, i.e.

$$ ({}^{(i)}{}_1p_k,\, {}^{(i)}{}_1c) \;\Xi\; ({}^{(i+\delta)}{}_0p_k,\, {}^{(i+\delta)}{}_0c) $$

for all arcs except those emerging from the affected $p$ node. For those we can see that

$$ ({}^{(i)}{}_1p_k,\, {}^{(i)}{}_1c) \;\Xi\; ({}^{(\mathrm{d+1})}{}_0p_k,\, {}^{(\mathrm{d+1})}{}_0c) $$

where $\mathrm{a}+1 \le i \le \mathrm{d}-3$. And (3) becomes clear once we notice that we have not affected any of the successful communication or replication rewrites.

Thus we have eliminated the first failed write attempt (if there was indeed one at all) from $\mathcal{X}^{\,\mathrm{O.0}}$ giving $\mathcal{X}^{\,\mathrm{O.1}}$. Likewise we can eliminate the first failed write attempt from $\mathcal{X}^{\,\mathrm{O.1}}$ giving $\mathcal{X}^{\,\mathrm{O.2}}$ etc. We get a sequence of executions $\mathcal{X}^{\,\mathrm{O}.i}$ which it is easy to see have a non-decreasing invariant prefix and such that for all relevant $i$, $\mathcal{X}^{\,\mathrm{O}.i}$ is related to $\mathcal{X}^{\,\mathrm{O}.i+1}$ by conditions (1) – (3).

If $\mathcal{X}^{\,\mathrm{O.0}}$ is finite then this process stops after a finite number of steps. Call the final execution generated $\mathcal{X}^{\,\mathrm{I.0}}$. If $\mathcal{X}^{\,\mathrm{O.0}}$ is infinite then there are two possibilities. Either the non-decreasing invariant prefix is never eventually constant; in which case the $\mathcal{X}^{\,\mathrm{O}.i}$

converge to an infinite execution. Call it $\mathcal{X}^{\text{I.0}}$ as before. (Note that $\mathcal{X}^{\text{I.0}}$ may not be a weakly fair execution. This would arise if some particular proposer node consistently failed to succeed in communicating. In such a case the graphs in $\mathcal{X}^{\text{I.0}}$ would eventually all contain an active node (the said proposer) that was never the root of a rewrite.) Otherwise the the invariant prefix stops increasing after some point say $i_0$. In this case all active nodes of execution graphs beyond the prefix of $\mathcal{X}^{\text{O}.j}$ for $j \geq i_0$ are involved with failing write attempts. (Such behaviour would be forced if say the expression $E$ contained only writers at the top level, eg. $E = \bar{x}z.\mathbf{0}$.) In this case call the final stable prefix $\mathcal{X}^{\text{I.0}}$. Note that strictly speaking it is not an execution since its final graph will contain active nodes. Nevertheless we will overlook this below. Finally, if there were no failed write attempts at all in $\mathcal{X}^{\text{O.0}}$, we set $\mathcal{X}^{\text{I.0}} = \mathcal{X}^{\text{O.0}}$.

**PHASE II** — Elimination of clashing read attempts. These arise from the following sequence of events.

(a) A proposer node ${}^{(a)}p$ rewrites to a proposer-intermediate node ${}^{(a+1)}p$ with a Help_r child ${}^{(a+1)}h$, and instantiates the input channel node ${}^{(a+1)}u$.

(b) The Help_r child ${}^{(b)}h$ matches a non-Empty channel node ${}^{(b)}c$ and rewrites to a No constructor ${}^{(b+1)}h$ using its default rule. [2]

(c) The No constructor ${}^{(c)}h$ notifies its suspended parent ${}^{(c)}p$.

(d) The parent ${}^{(d)}p$ matches the No constructor ${}^{(d)}h$ and reverts to a proposer node ${}^{(d+1)}p$.

Since Phase I did not interfere with clashing read attempts, once such a sequence of events starts, it will run to completion by the weak fairness of $\mathcal{X}^{\text{O.0}}$. So we eliminate the first such sequence from $\mathcal{X}^{\text{I.0}}$ giving $\mathcal{X}^{\text{I.1}}$. The technical details are as for Phase I. Again we generate a sequence of executions $\mathcal{X}^{\text{I.0}}$, $\mathcal{X}^{\text{I.1}}$, $\mathcal{X}^{\text{I.2}}$ etc. with non-decreasing invariant prefixes. Once more there are three cases depending on whether $\mathcal{X}^{\text{I.0}}$ was finite, and if not, whether the non-decreasing invariant prefix increased indefinitely or not. In all cases we call the resulting execution $\mathcal{X}^{\text{II.0}}$. As previously $\mathcal{X}^{\text{II.0}}$ need not be weakly fair.

**PHASE III** — Elimination of failed read attempts. These arise from the following sequence of events.

(a) A proposer node ${}^{(a)}p$ rewrites to a proposer-intermediate node ${}^{(a+1)}p$ with a Help_r child ${}^{(a+1)}h$, and instantiates the input channel node ${}^{(a+1)}u$.

(b) The Help_r child ${}^{(b)}h$ matches an Empty channel node ${}^{(b)}c$, and using its normal rule, rewrites to a Help_r_test_chan function ${}^{(b+1)}h$, rewriting the channel ${}^{(b)}c$ to a Busy_Unlocked channel ${}^{(b+1)}c$. [1]

($c_1$) The Help_r_test_chan function ${}^{(c_1)}h$ matches the Busy_Unlocked channel ${}^{(c_1)}c$ and rewrites to a Help_r_test_chan function ${}^{(c_1+1)}h$. [5]

($c_2$) The Help_r_test_chan function ${}^{(c_2)}h$ matches the Busy_Unlocked channel ${}^{(c_2)}c$ and rewrites to a Help_r_test_chan function ${}^{(c_2+1)}h$. [5]

… … … … … … …

($c_m$) The Help_r_test_chan function ${}^{(c_m)}h$ matches the Busy_Unlocked channel ${}^{(c_m)}c$ and rewrites to a Help_r_test_chan function ${}^{(c_m+1)}h$. [5]

(d) The Help_r_test_chan function ${}^{(d)}h$ matches the Busy_Unlocked channel ${}^{(d)}c$ and rewrites to a No constructor ${}^{(d+1)}h$, rewriting the Busy_Unlocked channel ${}^{(d)}c$ to an Empty channel node ${}^{(d+1)}c$. [6]

(e) The No constructor ${}^{(e)}h$ notifies its suspended parent ${}^{(e)}p$.

(f) The parent ${}^{(f)}p$ matches the No constructor ${}^{(f)}h$ and reverts to a proposer node ${}^{(f+1)}p$.

Again once such a sequence of events starts, it will run to completion by the weak fairness of $\mathcal{X}^{\,\mathrm{O.0}}$, although this time, it is possible that $m = \infty$ and the events (d) – (f) never take place. Apart from the fact that more events need to be dealt with in eliminating such a sequence, the details are sufficiently similar that we can omit them. One point to note is that unlike the previous phases, elimination of a sequence (a) – (f) changes the state of the channel node $c$ from Busy_Unlocked to Empty between stages (b) and (d) inclusive. Since channel nodes are shared, any rewrite explicitly matching $c$ in this period would find a different symbol and so would need to use a different rule. However, we can deduce by lemmas 6.3 and 6.7, that any such rewrite must belong to a clashing read attempt, and these have already been eliminated above. So the change of state goes unobserved, and the elimination is safe.

So we generate a sequence of executions as previously, $\mathcal{X}^{\,\mathrm{II.0}}$, $\mathcal{X}^{\,\mathrm{II.1}}$, $\mathcal{X}^{\,\mathrm{II.2}}$ etc. with non-decreasing invariant prefixes. Once more there are three cases depending on whether $\mathcal{X}^{\,\mathrm{II.0}}$ was finite, and if not, whether the non-decreasing invariant prefix increased indefinitely or not. In all cases we call the resulting execution $\mathcal{X}^{\,\mathrm{III.0}}$.

**PHASE IV** — Elimination of useless work from successful read attempts. These are to be found within sequences of events as follows; where $m > 0$.

(a) A proposer node ${}^{(a)}p$ rewrites to a proposer-intermediate node ${}^{(a+1)}p$ with a Help_r child ${}^{(a+1)}h$, and instantiates the input channel node ${}^{(a+1)}u$.

(b) The Help_r child ${}^{(b)}h$ matches an Empty channel node ${}^{(b)}c$, and using its normal rule, rewrites to a Help_r_test_chan function ${}^{(b+1)}h$, rewriting the channel ${}^{(b)}c$ to a Busy_Unlocked channel ${}^{(b+1)}c$. [1]

($c_1$) The Help_r_test_chan function ${}^{(c_1)}h$ matches the Busy_Unlocked channel ${}^{(c_1)}c$ and rewrites to a Help_r_test_chan function ${}^{(c_1+1)}h$. [5]

($c_2$) The Help_r_test_chan function ${}^{(c_2)}h$ matches the Busy_Unlocked channel ${}^{(c_2)}c$ and rewrites to a Help_r_test_chan function ${}^{(c_2+1)}h$. [5]

… … … … … … …

($c_m$) The Help_r_test_chan function $^{(c_m)}h$ matches the Busy_Unlocked channel $^{(c_m)}c$ and rewrites to a Help_r_test_chan function $^{(c_m+1)}h$. [5]

(d) The Help_r_test_chan function $^{(d)}h$ matches the Busy_Locked[data] channel $^{(d)}c$ and rewrites to a Help_r_assign_data function $^{(d+1)}h$. [7]

(e …) The remaining steps of the communication protocol complete successfully.

In this case all we wish to do is to eliminate the steps ($c_1$) – ($c_m$) without affecting the rest of the communication. (This time we can assert that $m$ is finite, regarding all infinite $m$ cases as partially complete failed read attempts.) The elimination can be done without complication, especially when we note that all the eliminated steps are actually null rewrites modulo garbage: none of them changes the live graph at all. The same strategy as before now applies. Once more we generate a sequence of executions, $\mathcal{X}^{\,\mathrm{III.0}}$, $\mathcal{X}^{\,\mathrm{III.1}}$, $\mathcal{X}^{\,\mathrm{III.2}}$ etc. We call the resulting execution $\mathcal{X}^{\,\mathrm{IV.0}}$.

At this point, we have eliminated all spurious activity from the execution. What remains, is to standardise $\mathcal{X}^{\,\mathrm{IV.0}}$ by reordering the rewrites in a sensible way. This consists of two subtasks. The first is to cluster the rewrites corresponding to a successful run of the communication protocol at the commit points. The second is to ensure that the rewrites of auxiliary and replication functions occur at suitable places so that we can identify execution graphs that actually represent the expressions $E_i$ of a trace from $E$. All this must be done in a way that preserves the order of communications and replications so as not to fall foul of causality considerations that would prevent eg. a rewrite ($\beta$) from being permuted to a place earlier in the execution than the rewrite ($\alpha$) which created ($\beta$)'s redex root as a contractum node. We start with the communications.

**PHASE V** — Compression of successful communication sequences. We will standardise on the following sequence of events for a successful communication.

(a) A proposer node $^{(a)}pw$ rewrites to a proposer-intermediate node $^{(a+1)}pw$ with a Help_w child $^{(a+1)}hw$.

(b) A proposer node $^{(b)}pr$ rewrites to a proposer-intermediate node $^{(b+1)}pr$ with a Help_r child $^{(b+1)}hr$, and instantiates the input channel node $^{(b+1)}u$.

(c) The Help_r child $^{(c)}hr$ matches an Empty channel node $^{(c)}c$, and using its normal rule, rewrites to a Help_r_test_chan function $^{(c+1)}hr$, rewriting the channel $^{(b)}c$ to a Busy_Unlocked channel $^{(c+1)}c$. [1]

(d) The Help_w child $^{(d)}hw$ matches the Busy_Unlocked channel node $^{(d)}c$ and rewrites to a Yes constructor $^{(d+1)}hw$, rewriting the channel to a Busy_Locked[-data] channel $^{(d+1)}c$. [3]

(e) The Help_r_test_chan function $^{(e)}hr$ matches the Busy_Locked[data] channel $^{(e)}c$ and rewrites to a Help_r_assign_data function $^{(e+1)}hr$. [7]

(f) The Help_r_assign_data function $^{(f)}hr$ matches the Empty input channel $^{(f)}u$, and rewrites to a Help_r_unlock function $^{(f+1)}hr$, rewriting the input channel to the data channel $^{(f+1)}d$. [9]

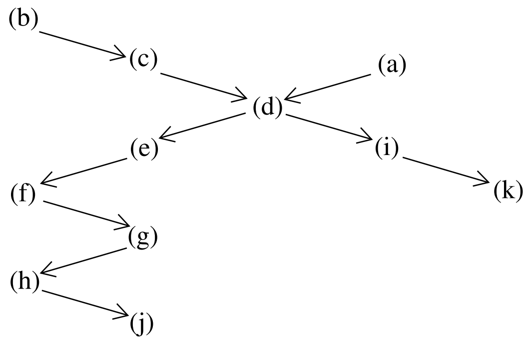


Fig. 12

(g) The Help_r_unlock function ${}^{(g)}hr$, matches the Busy_Locked[data] channel ${}^{(g)}c$ and rewrites to a Yes constructor ${}^{(g+1)}hr$, rewriting the channel to an Empty channel ${}^{(g+1)}c$. [11]

(h) The Yes constructor ${}^{(h)}hr$ notifies its suspended proposer-intermediate node parent ${}^{(h)}pr$.

(i) The Yes constructor ${}^{(j)}hw$ notifies its suspended proposer-intermediate node parent ${}^{(j)}pw$.

(j) The active proposer-intermediate parent ${}^{(i)}pr$ matches the Yes constructor ${}^{(i)}hr$ and rewrites successfully.

(k) The active proposer-intermediate parent ${}^{(k)}pw$ matches the Yes constructor ${}^{(k)}hw$ and rewrites successfully.

The above is one possible ordering compatible with causality. The general situation is illustrated in [Fig. 12] where an arrow indicates that the higher event must causally precede the lower event. That this is indeed the case is easily shown on the basis of lemmas 6.3 and 6.7 and the form of the protocol rules. In fact [Fig. 12], with time flowing down the page, is an elementary event structure for a successful communication according to our protocol [Nielsen et al. (1981), Winskel (1986), Winskel (1988)].

For obvious reasons, the commit events (d) are regarded as pinpointing the position of a communication within an execution. Thus even if the event sequences for two communications overlap, they are still regarded as taking place in the order of their commit events.

**Remark 7.16.1** We recall the fact (also pertinent to Phases III and IV above), that between events (c) and (g) inclusive of a communication sequence, no function nodes other than those involved in the communication sequence itself explicitly match the channel $c$. This is because by lemmas 6.3 and 6.7 such nodes must be participating in a clashing read or failing write attempt, and these have already been eliminated above.

To a communication with events (a) – (k) we apply the following transformation steps.

(1) Interchange event (c) with its succeeding events repeatedly until it becomes the event immediately preceding event (d).

(2) Interchange event (b) with its successors until it immediately precedes event (c).

(3) Interchange event (a) with its successors until it immediately precedes event (b).

(4) Interchange event (e) with its predecessors until it immediately succeeds event (d).

(5) Interchange event (f) with its predecessors until it immediately succeeds event (e).

(6) Interchange event (g) with its predecessors until it immediately succeeds event (f).

(7) Interchange event (h) with its predecessors until it immediately succeeds event (g).

(8) Interchange event (i) with its predecessors until it immediately succeeds event (h).

(9) Interchange event (j) with its predecessors until it immediately succeeds event (i).

(10) Interchange event (k) with its predecessors until it immediately succeeds event (j).

We must be sure that doing the above to a succesful communication sequence transforms an execution of Tr($E$) into another execution of Tr($E$). For brevity we pretend that all intervening execution steps that we have to consider are rewrites, the case of notifications being simpler by corollary 7.14. To justify step (1) we argue as follows.

If the event following (c) is (d), then we are done and step (1) yields an execution of Tr($E$). Otherwise the redex rewritten in rewrite (c+1) $\rightarrow$ (c+2) already existed in execution graph (c), since the only redex that rewrite (c) $\rightarrow$ (c+1) creates is the redex for the corresponding event (d). Consequently, both redexes exist in execution graph (c) and by remark 7.16.1, satisfy the hypotheses of lemma 6.10. Therefore by lemma 6.10 we can do the rewrites in the other order. This yields a new execution which by lemma 7.13 has properties (1) and (2) of the present theorem; while property (3) is obviously preserved since we do not move event (d).

The justifications for the other steps are similar and are omitted.

As in previous phases, we start with the first succesful communication sequence, and the interchanges performed during the compression generate a number of new execution sequences which we resist the temptation of trying to catalogue. Upon completing the compression of the first sequence we proceed to the second. And so on. We will name the end product of this activity $\mathcal{X}^{\mathrm{V.0}}$.

**PHASE VI** — Compression of auxiliary rewriting sequences. These arise through a sequence of events such as the following.

(a) An active function node ${}^{(a)}f$ rewrites and creates $k$ active composition, $\nu$, or $\mathsf{Root}$ nodes ${}^{(a+1)}n_1$, ${}^{(a+1)}n_1$, … , ${}^{(a+1)}n_k$ as copies of its contractum nodes.

(b$_1$) One of the active nodes ${}^{(b_1)}n_1$, ${}^{(b_1)}n_1$, … , ${}^{(b_1)}n_k$, ${}^{(b_1)}n_i$ say, rewrites (if it is a composition or ν node), or notifies (if it is a Root node).

(b$_2$) One of the remaining active nodes of $\{{}^{(b_2)}n_1, {}^{(b_2)}n_1, \dots, {}^{(b_2)}n_k\} - \{{}^{(b_2)}n_i\}$ rewrites or notifies.

… … … … … … …

(b$_k$) The last active node ${}^{(b_k)}n_m$ rewrites or notifies.

In such a sequence we call (a) the auxiliary-parent event and the (b$_i$) the auxiliary-child events. There is a direct correspondence between such auxiliary-parent / auxiliary-children configurations, and fragments of the parse tree of the original π-calculus expression $E$, because the auxiliary rules used, are generated directly from the parse tree in the translation. Rather as in Phase V, such an auxiliary-parent / auxiliary-children configuration, *aka* fragment of parse tree, can be regarded as a mini elementary event structure for the collection of auxiliary rewrites generated, with the parent causally preceding the children.

To such a sequence we can apply the following transformation.

(1) Interchange event (b$_1$) with its predecessors until it immediately succeeds event (a).

(2) Interchange event (b$_2$) with its predecessors until it immediately succeeds event (b$_1$).

… … … … … … …

($k$) Interchange event (b$_k$) with its predecessors until it immediately succeeds event (b$_{k-1}$).

Such a series of interchanges is easily justified by noting that no auxiliary node ever pattern matches to rewrite since its rule is a default rule, so arguments like those used in Phase V apply even more readily.

We apply the transformation above to the first or initial rewrite of $\mathcal{X}^{\text{ V.0}}$ if applicable (i.e. if the initial rewrite, interpreted as an auxiliary-parent event, generates any composition, ν, or Root nodes), yielding $\mathcal{X}^{\text{ V.1}}$. We then apply the transformation to the second rewrite of $\mathcal{X}^{\text{ V.1}}$ if applicable yielding $\mathcal{X}^{\text{ V.2}}$. We then apply the transformation to the third rewrite of $\mathcal{X}^{\text{ V.2}}$ if applicable yielding $\mathcal{X}^{\text{ V.3}}$. And so on. The end product is an execution $\mathcal{X}^{\text{ VI.0}}$. It obviously satisfies the properties (1) and (2) of the theorem, also (3) since no replication rewrite (or communication commit rewrite) is moved in Phase VI. An important property of $\mathcal{X}^{\text{ VI.0}}$ is that as a result of the order in which the transformations are applied, the only events that can occur between an auxiliary-parent event (a) and any of its auxiliary-child events (b$_i$), are sibling events of (b$_i$), their own auxiliary-children etc. In fact following any auxiliary-parent event (a), there is a segment of the execution which is a sequence of auxiliary rewrites corresponding exactly to the rewriting of the root of the auxiliary-parent event redex to a collection of proposer nodes, replication nodes, and idle Root nodes according to lemma 6.5, with no intervening other events. This sequence corresponds to a preorder listing of the sub parse tree rooted at the vertex corresponding to (a), and truncated at summation and replication vertexes.

The order to which "preorder" refers, is constructed by, at each level of the sub parse tree, ordering the child vertexes of a vertex in the same order as the corresponding auxiliary-child events appear in the original execution (this order is obviously preserved by Phase VI). This truncated parse tree in turn corresponds to a larger elementary event structure obtained by gluing together the mini event structures mentioned above. All of this is easy enough to see by induction.

One important consequence of this transformation is that the last two events (j) and (k) of a successful communication, which can both in principle rewrite their roots to auxiliary function nodes, can now both be immediately followed by rewrite sequences that turn these functions into idle Roots, and active proposer or replication functions, i.e. the events (j) and (k) have become separated. This prompts the last Phase.

**PHASE VII** — Reattachment of communication events (k). We just repeat step (10) of Phase V.

(1) Interchange event (k) with its predecessors until it immediately succeeds event (j).

As before we apply this transformation to each of the successful communications in turn, reattaching their (k) events. This yields the execution $\mathcal{X}^{\text{VII.0}}$.

Execution $\mathcal{X}^{\text{VII.0}}$ which we rename as $\mathcal{G}$, is the execution sought in the theorem. By construction, it satisfies the properties (1) – (3) as required. Further we claim it is (a possibly proper prefix of) a standard execution of Tr($E$) corresponding to a trace of communications and replications from $E$, enhanced by reductions to standard form. The proper prefix property arises since Phases I – III may individually or together dispose of an infinite suffix of $\mathcal{H}$. For the rest we argue as follows.

Let $G_\alpha \to G_{\alpha+1}$ be an execution step of $\mathcal{G}$ which is either a replication rewrite, or a rewrite corresponding to an event (a) of a successful communication. We call the graphs $G_\alpha$ of such steps witness graphs, including also the last graph of $\mathcal{G}$ as a witness graph in case $\mathcal{G}$ is finite. We claim that the sequence of witness graphs represent the expressions in a trace of communications and replications from $E$, enhanced by reductions to standard form.

We proceed by induction. First the base case. Let $G_{(\alpha 0)}$ be the first witness graph. We claim it represents $E \equiv E_0$. For consider the parse tree of $E_0$. It will depict how the TLPSEs of $E_0$ are combined to form $E_0$. It is clear that:

(1) The initial rewrite and ensuing sequence of auxiliary rewrites (and notifications by Roots) mirrors a preorder listing of this parse tree of the TLPSEs of $E_0$.

(2) The collection of idle Roots, active replication and proposer nodes of $G_{(\alpha 0)}$ generated, is in a bijective correspondence $\rho_0 : E_0 \to G_{(\alpha 0)}$ with the TLPSEs of $E_0$, as required by definition 7.2.(1).

(3) $\rho_0$ extends to an appropriate bijection between free channel names of TLPSEs of $E_0$ and Empty channel nodes of $G_{(\alpha 0)}$, again as required by definition 7.2.(2).

The above can be verified in detail by a subinduction on the structure of the derivation of $G_{(\alpha 0)}$ from $*$Initial; using the structure of the "rewrites to" relation of definition 6.4 to ensure that the correct function nodes are generated, and the properties of the $\mathit{Args}_P$ and $\mathit{blend}_P$ functions of the translation to ensure that channel nodes are linked to the correct function nodes. Now let $E_0 \Rrightarrow^* F_0$ be a reduction to standard form, performed in case any of the subsequent communications of $\mathcal{G}$ require top level ν's to have larger scopes than they possess in $E_0$. By lemma 7.5 we know that $G_{(\alpha 0)}$ represents $F_0$ too, via a map $\rho'_0 : F_0 \to G_{(\alpha 0)}$.

For the induction step, assume $G_{(\alpha i)}$ represents $F_i$ through the bijection $\rho'_i : F_i \to G_{(\alpha i)}$. There are two cases. Either the next rewrite $G_{(\alpha i)} \to G_{(\alpha i)+1}$ is a replication rewrite of a replication node P of $G_{(\alpha i)}$ or not. If so, then there is a replication TLPSE $\{!\ldots\}_P$ of $E_i$ corresponding to P via $\rho_i$. Let $E_{i+1}$ be the π-calculus expression obtained by replicating $\{!\ldots\}_P$. In general the replicated subexpression will not be a TLPSE of $E_{i+1}$ but will be a combination of TLPSEs of $E_{i+1}$ using | and ν. The execution steps deriving $G_{(\alpha(i+1))}$ from $G_{(\alpha i)+1}$ mirror a preorder listing of the appropriate sub parse tree. So the next representation $\rho_{i+1} : E_{i+1} \to G_{(\alpha(i+1))}$ can be constructed. This is followed by a reduction to standard form $E_{i+1} \Rrightarrow^* F_{i+1}$ for the usual reason. Using lemma 7.5 again, we find the representation $\rho'_{i+1} : F_{i+1} \to G_{(\alpha(i+1))}$. In detail, this can be established using subinductions on the structure of the derivation $G_{(\alpha i)+1} \to^* G_{(\alpha(i+1))}$, and of the reduction $E_{i+1} \Rrightarrow^* F_{i+1}$.

Otherwise the next rewrite $G_{(\alpha i)} \to G_{(\alpha i)+1}$ is event (a) of a successful communication between proposer nodes P and Q of $G_{(\alpha i)}$, using a channel node x say as transmission link. Then there are two TLPSEs $\{+\ldots+\}_P$ and $\{+\ldots+\}_Q$ of $F_i$ corresponding to P and Q via $\rho'_i$, and they are able to communicate via channel name $x$ because of the properties of $F_i$ and $\rho'_i$; specifically the data channel of the communication does not escape any scope it might be contained in because $F_i$ is in standard form. Let $E_{i+1}$ be the π-calculus expression obtained by doing the communication in $F_i$. In general $\{+\ldots+\}_P$ and $\{+\ldots+\}_Q$ will be replaced by two subexpressions which are not themselves TLPSEs of $E_{i+1}$, but combinations using | and ν of TLPSEs of $E_{i+1}$. The execution steps deriving $G_{(\alpha(i+1))}$ from $G_{(\alpha i)+1}$ mirror preorder listings of the two relevant sub parse trees. The execution will be such that the two sub parse tree roots are mirrored first, followed by the two remainders of the preorder listings, one after the other. This is a consequence of the detailed operation of Phases VI and VII. As before, the next representation $\rho_{i+1}$ : $E_{i+1} \to G_{(\alpha(i+1))}$ can now be constructed. This is followed by a reduction to standard form $E_{i+1} \Rrightarrow^* F_{i+1}$ for the usual reason. Using lemma 7.5, we again find the representation $\rho'_{i+1} : F_{i+1} \to G_{(\alpha(i+1))}$. As before, all of the above can be checked in detail using a subinduction on the structure of the derivation $G_{(\alpha i)+1} \to^* G_{(\alpha(i+1))}$, and of the reduction $E_{i+1} \Rrightarrow^* F_{i+1}$.

Evidently $\mathcal{G}$ is a standard execution with the advertised properties, which corresponds via theorem 7.9 to a trace $\mathcal{T}$ of $E$ as required. We are done. ☺

We can modify the theorem in two significant ways. If we drop the fairness assumption, then something like the original conclusion still holds true. The main problem is that for any event in the execution, its logical successor may be absent, which substantially messes up the technical details of the various early phases. Nevertheless, by discarding partially completed event sequences, an analogue of a prefix of a standard execution may be constructed. The main way in which it might fail to be standard, would be if auxiliary rewriting sequences failed to run to completion, leaving auxiliary nodes which did not rewrite to idle Roots, proposers and replicators, blocking the construction of a representation, particulary when reductions to standard form were involved. One could invent a modified notion of representation, or one could impose a more pernickety notion of weak fairness that only applied to auxiliary symbols, in order to cope with this.

In the opposite direction, if we strengthen the fairness assumption by assuming that no proper suffix of an infinite $\mathcal{H}$ consists entirely of useless work, we can drop the caveat about suffixes in the theorem. Viz.

**Corollary 7.17** Let $E$ be a π-calculus expression, and Tr($E$) its translation. Let $\mathcal{H} = [H_0, H_1, \ldots]$ be a weakly fair execution of Tr($E$). Suppose that if $\mathcal{H}$ is infinite, it contains an infinite number of commit rewrites. Then there is a trace $\mathcal{T}$ of communications and replications from $E$, enhanced by reductions to standard form, such that for a standard execution $\mathcal{G} = [G_0, G_1, \ldots]$ of Tr($E$) corresponding via theorem 7.9 to $\mathcal{T}$

(1) For every node ${}^{(i)}x \in G_i \in \mathcal{G}$, ${}^{(i)}x \;\Xi\; {}^{(j)}y$ for some ${}^{(j)}y \in H_j \in \mathcal{H}$, and if ${}^{(i)}x$ is live then ${}^{(j)}y$ is live.

(2) For every arc $({}^{(i)}p_k, {}^{(i)}c) \in G_i \in \mathcal{G}$, $({}^{(i)}p_k, {}^{(i)}c) \;\Xi\; ({}^{(j)}q_k, {}^{(j)}d)$ for some arc $({}^{(j)}q_k, {}^{(j)}d) \in H_j \in \mathcal{H}$.

(3) If $H_j \to H_{j+1} \in \mathcal{H}$ is a rewrite of a redex rooted at ${}^{(j)}y \in H_j$ and governed by either a rule for a replication symbol, or a communication commit rule, there is a corresponding rewrite $G_i \to G_{i+1} \in \mathcal{G}$ of a redex rooted at a node ${}^{(i)}x \in G_i$, which is governed by the same rule, and ${}^{(i)}x \;\Xi\; {}^{(j)}y$. Further, if $H_j \to H_{j+1}$ and $H_{j'} \to H_{j'+1}$ are two distinct such rewrites, their corresponding $G_i \to G_{i+1}$ and $G_{i'} \to G_{i'+1}$ are also distinct, and all such rewrites occur in the same order in $\mathcal{H}$ and $\mathcal{G}$.

Now that we have the preceding results, we remark that we can easily adapt them to the more conventional view of the π-calculus in which replication is viewed as a syntactic congruence. All we need to do is to forget the replication steps in the trace $\mathcal{T}$, concentrating on the witness graphs that represent starting configurations of successful communications. We do not repeat the relevant theorems.

## 8 OTHER ASPECTS OF THE PI-CALCULUS

Theorem 7.16 is the main objective of this paper. In pursuing it, we omitted mention of a number of aspects of the π-calculus present in the original description [Milner et al. (1992)]. In this section, we return to some of these.

### Other Syntactic Features

The original description features zero and parallel composition as before, but summation is *unprefixed*. In a translation such as ours, we can deal with such more general nondeterministic sums in much the same way as we did above. Namely, the symbol representing the sum, nondeterministically rewrites to a symbol representing one of the possibilities, which then attempts to engage in a transition corresponding to that possibility. Enough information must be retained so that backtracking can take place, and the system of choices may need to go (and if necessary to backtrack) several levels deep. The reason for this is that whereas with the prefixed sums of [Section 3], a choice followed by a successful communication commits the system irrevocably, with unprefixed sums, a choice may not yield a possibility capable of committing the system immediately, eg. the chosen summand may be itself an unprefixed sum. Worse, because the prefixes in the original description are not restricted to just the communication primitives, not all of them are committing. All of this offers much scope for optimisations in a real implementation. Let us look through the prefixes now.

The prefixes divide into the committing ones which are $\tau.Q$, $\bar{x}y.Q$, $x(y).Q$, and the non-committing ones namely $[x = y].Q$ and $\nu xQ$. We discuss these in a convenient order.

The case of $\tau.Q$ is relatively easy. Since in the $\pi$-calculus one has the transition

$$\tau.Q \xrightarrow{\tau} Q$$

one could introduce rules in the translation for symbols that "did nothing for one step" easily enough. In the treatment of correctness, in an augmented theorem 7.16, the rewrites for such rules would simply be left where they were.

The next easiest case is the noncommitting $[x = y].Q$. In standard MONSTR, there is no "pointer equality test". However many systems find such a test extremely useful, so in practice many MONSTR systems admit the additional pair of rules

```
PointersEqual[x x] => *Yes ;
PointersEqual[x y] => *No
```

the first being considered a non-default rule. An analysis of the architectural demands of such rules reveals that they are not excessive so their inclusion is permissible, though invevitably they clutter the case analysis of proofs about MONSTR systems, which explains why they are often omitted.

The natural way to incorporate the syntactic construct $\{[x = y].Q\}_P$ into the translation is thus to have symbol P call PointersEqual[x y], and depending on the result, to either proceed to behave as Q if x = y, or to backtrack to P (or an ancestor of P) if not. Here is one sense in which the equality test is noncommitting; if the test fails, it is still perfectly possible that some other process might subsequently make $x$ and $y$ equal, allowing the test to succeed. Furthermore the $\pi$-calculus transition rule

$$\frac{Q \xrightarrow{\alpha} Q'}{[x = x].Q \xrightarrow{\alpha} Q'}$$

gives us cause for concern, since it appears to demand the synchronisation of the equality test with the succeeding action $\alpha$. This is another sense in which the equality test is noncommitting; any committment is contingent on the success of $\alpha$. The synchronisation poses no problem in a standard execution, but in an arbitrary execution the two subactions may be separated in time. However, an augmented theorem 7.16 may exploit the semantics of substitution and redirection, which both hold that once $b$ has been substituted for / redirected to $a$, the action cannot be subsequently undone. Thus the rewrites for a successful equality test can be postponed until they occur just before the rewrites for the subsequent $\alpha$ action. In this manner correctness may be extended to include the equality test.

Interestingly, the same does not hold for a hypothetical inequality test, $[x \neq y].Q$, since in the translation of the transition rule

$$\frac{Q \xrightarrow{\alpha} Q'}{[x \neq y].Q \xrightarrow{\alpha} Q'}$$

an arbitrary execution may perform the successful inequality test, and the critical action of $\alpha$, at distant points of the execution. A priori it is not permissible to move either action to be close enough to the other, so the translated system could exhibit behaviours incommensurate with the original $\pi$-calculus expression. Worse, the architectural exigiencies of an inequality test are much more severe than those of an equality test, and thus inequality tests are definitely excluded from the remit of MONSTR. Cf. also remarks on a hypothetical inequality test in [Milner et al. (1991)].

We now briefly mention the prefix form $\nu x Q$. This would be implemented very much as in [Section 3], by rules that instantiate the bound channel. As for the equality test, the actions for a $\nu x$-prefixed process are not committing eg.

$$\frac{Q \xrightarrow{\alpha} Q'}{\nu x Q \xrightarrow{\alpha} Q'}$$

Therefore, as with the input communication primitive, enough information must be retained to enable backtracking to occur; and when it does occur, the instantiated bound channel is garbaged.

The other prefix forms, $\bar{x}y.Q$ and $x(y).Q$, refer to the communication abilities of a $\pi$-calculus expression $E$. Up to a point we have dealt with these in our translation, insofar as we have translated internal communications faithfully. However, the main reason that the original formulation of the $\pi$-calculus is interesting, is the fact that it deals also with external communications, i.e. how a $\pi$-calculus expression $E$ communicates with its environment. Once we have understood how environments can be modelled in the context of our translation, we will be able to discuss the analogues of scope extrusion and intrusion within our framework.

Consider the question of environments. If $Env$ is (a $\pi$-calculus expression representing) an environment for $E$, then one places $E$ in the environment $Env$ by forming

$$(E \mid Env).$$

If $\mathrm{Tr}(E)$ and $\mathrm{Tr}(Env)$ are the corresponding translations, then assuming that subexpression labels have been renamed apart in $E$ and $Env$ to forestall unfortunate name clashes, the analogue of the immersion can be viewed three ways.

One can first consider the ruleset $\mathrm{Tr}(\{E \mid Env\}_T)$ directly, where $T$ is a new top level symbol.

Secondly one can regard it as having arisen from $\mathrm{Tr}(E)$ and $\mathrm{Tr}(Env)$ by discarding the two initial rules, and introducing a new rule for $\mathsf{Initial}$ which causes $*\mathsf{Initial}$ to rewrite to a fresh function $\mathsf{T}$, this in turn rewriting as would any other rule for a composition symbol, to active $\mathsf{T}_1$ and $\mathsf{T}_2$ nodes (with suitable channel correspondences), these being the top level symbols for $\mathrm{Tr}(E)$ and $\mathrm{Tr}(Env)$.

Thirdly one can view the preceding ad-hoc procedure from a more formal perspective, regarding it as a specific example of the modular composition of translated systems alluded to in [Section 5]. If $\mathrm{Tr}^{-}(E)$ and $\mathrm{Tr}^{-}(Env)$ are the modular translations of $E$ and $Env$ without initial rules but with $Args$ functions specifically mentioned, then we form

$$\mathrm{Tr}(\{\ [\mathrm{Tr}^{-}(E)]_{T_1} \mid [\mathrm{Tr}^{-}(Env)]_{T_2}\ \}_T)$$

where

$$\{ [\ ]_{T_1} \,|\, [\ ]_{T_2} \}_T$$

is a specific labelled π-calculus context, into the holes of which (the square brackets), we are expected to "place" $\mathrm{Tr}^{-}(–)$ rulesets, in order to subsequently translate the entire expression to a MONSTR rule system. Depending on exactly how we view the composition of systems as taking place, this "placing" has a number of different interpretations. In the present case, we would just add the rule for $\mathsf{T}$ (and then either the rule for $*\mathsf{Initial}$, or the $Args_T : \mathrm{Free}(T) \to \boldsymbol{args_T}$ function, as required).

Let us now examine this process for an arbitrary context, say

$$\{ C([\ ]_{T_1}, [\ ]_{T_2}, \ldots, [\ ]_{T_n}) \}_T$$

where $C$ is a π-calculus context expression, i.e. an expression of π-calculus syntax which is "syntactically non-ground" in that $n$ leaves of its parse tree are π-calculus expression non-terminals. These non-terminals correspond to the $n$ $[\ ]_{T_i}$ holes into which some $\mathrm{Tr}^{-}(E_i)$ translations of systems are to be "placed". For it to work as it should, we need some restrictions on the behaviour of names and symbols in the different components.

(1) There are no clashes of subexpression labels, either between labels coming from the various $E_i$, or between labels coming from one of these and labels in the enclosing context $C$, *except* that the symbols labelling the top-level constructs of the $E_i$ match the symbols labelling the holes into which their $\mathrm{Tr}^{-}(E_i)$ translations are placed.

(2) The $Args_{T_i}$ functions for the top-level constructs $T_i$ output by the translations of the subcomponents in $\mathrm{Tr}^{-}(E_i)$, must match the $Args_{T_i}$ functions assumed for these subcomponents by the translation of the context. More specifically, the two versions of the $Args_{T_i}$ functions must agree in their domains i.e. the sets of free channel names involved, and in the argument positions of the various $\mathsf{T}_i$ symbols that these free channel names get mapped to. A byproduct of this is that capture of subcomponents' free names by binders in the interior of the context is permitted.

Elaborating a little on the second point, inspection of the translation in [Section 5] shows that the various recursive constructs of a π-calculus expression act as transformers of $Args_P$ functions. The same must be true for "syntactically non-ground" compositions of recursive constructs, i.e. contexts. This leads us to view the translation of an arbitrary context from two complementary perspectives.

Firstly the translation of the context may be "delayed" until the translations of the subcomponents to be inserted into the holes are available, (specifically till the top-level $Args_{T_i}$ functions are available), and then the bottom-up translation may be completed. In this case, the "placing" of the $\mathrm{Tr}^{-}(–)$ translations we mentioned above, is simply set union of the $\mathrm{Tr}^{-}(E_i)$ translations with what is generated by the remainder of the translation.

Secondly, the translation may be done eagerly, before the translations of the subcomponents are available, by abstracting away all information obtained from the $Args_{T_i}$ functions within the body of the context's translation. In this case, we obtain the concept of a "standalone" translation of a context $C$ with holes, which as for a closed system, again has two components. The first component is an $Args_T$ function transformer, which takes a collection of $Args_{T_i}$ functions (to be eventually supplied by the $\mathrm{Tr}^{-}(E_i)$ translations for its holes), and maps them into the $Args_T$ function of the top-level symbol $\mathsf{T}$ of the context. Abstracting away from lower level detail, this function constructs the union of the free channel names mentioned in the domains of the $Args_{T_i}$ functions, removes any names that are captured by binders in $C$, and maps what remains (together with any channel names that are mentioned in $C$ and occur free at the top level) to the argument positions of $\mathsf{T}$. The second component is a partially instantiated set of MONSTR rules for the syntactic structure of $T$. The uninstantiated features of these rules (eg. their arity) will depend upon which and how many free channels are input from the $\mathrm{Tr}^{-}(E_i)$ subsystems, which of them are captured by binders in the context, and which of them need to be output in the top-level of the result. Again such a partially instantiated set of rules can be viewed as a function, taking as input a collection of $Args_{T_i}$ functions, and mapping them into the fully instantiated set of rules. In this case, the "placing" of the $\mathrm{Tr}^{-}(E_i)$ translations corresponds to the composition of a top-level context module $\mathrm{Tr}(C)$ with an appropriate number of input systems $\mathrm{Tr}^{-}(E_i)$. This is accomplished by the application of the context's $Args_T$ transformer and rule generation function to the $Args_{T_i}$ functions of the input systems, and including the rules of the input systems. This yields an $Args_T$ function for the top-level symbol of the new system, and a fully instantiated set of rules for the system as a whole.

However, in this more general setting, we can contemplate composing a top-level context with an appropriate number of input contexts (with holes). The composition of an $Args_T$ function transformer with a suitable collection of input $Args_{T_i}$ function transformers is another $Args_T$ function transformer. Similarly the composition of a function from a collection of $Args_{T_i}$ functions to a fully instantiated ruleset, with a suitable collection of input $Args_{T_i}$ function transformers, is another function from $Args_{T_i}$ functions to a ruleset. Performing these compositions, and including the input functions from the $Args_{T_i}$ to rulesets in the latter, yields the resulting module

$$\mathrm{Tr}^{-}(\{\ C([\mathrm{Tr}^{-}(C_1([\ldots]))]_{T_1}, [\mathrm{Tr}^{-}(C_2([\ldots]))]_{T_2}, \ldots, [\mathrm{Tr}^{-}(C_n([\ldots]))]_{T_n})\ \}_T)$$

Pursuing this idea further would lead us to a graph rewriting based fibration semantics for the π-calculus (cf. [Coquand et al. (1989), Asperti and Martini (1992)]). However we will not follow this up here.

We note that a closed system, regarded as a context with no holes, translates to a constant $Args_T$ function transformer (which just yields the top level $Args_T$ function), and a constant function yielding a fully instantiated set of rules, which is as we would expect.

Now that we understand environments, we can make sense of such transitions as

$$\bar{x}y.Q \xrightarrow{\bar{x}y} Q$$

within the translation. Let us consider the more general form

$$E \xrightarrow{\bar{x}y} F$$

which says that (it can be proved within the deduction system of π-calculus that) $E$ is capable of evolving to $F$ by virtue of engaging in a communication in which it ouputs $y$ on $x$ when placed in an environment capable of inputting on $x$. The natural analogue of this in the MONSTR world is to say that $\mathrm{Tr}^{-}(E)$ is capable of an equivalent communication in a modular composition

$$\mathrm{Tr}(\{\ [\mathrm{Tr}^{-}(E)]_{T_1} \mid [\mathrm{Tr}^{-}(Env)]_{T_2}\ \}_T)$$

in which $Env$ has channel name $x$ free at the top level and is capable of inputing on it.

An alternative way of saying this is to consider a graph $G$ which represents $E$ (by rewriting using $\mathrm{Tr}(E)$ as in lemma 7.4 for example). The representation $\rho_E : E \to G$ identifies nodes of $G$ which represent the free channels, and channel $x$ in particular. If we have a graph $G_{env}$ which represents a suitable environment $Env$, then the representation $\rho_{Env} : Env \to G_{env}$ picks out a node representing $x$ in $G_{env}$. If $B$ is a graph consisting exactly of as many $\mathsf{Empty}$ channel nodes as $E$ and $Env$ have free channel names in common, there are bijections $e : B \to E$ and $env : B \to Env$ identifying nodes in $B$ with the relevant free channel names in $E$ and $Env$ respectively. The compositions

$$\rho_E \circ e : B \to G \quad \text{and} \quad \rho_{Env} \circ env : B \to G_{env}$$

are arrows in the category of MONSTR graphs and graph homomorphisms. We can form the pushout $G \boxplus_B G_{env}$ of these two arrows, which just joins the graphs $G$ and $G_{env}$ by identifying nodes representing the common free channel names like $x$. Pushouts in general are the appropriate analogue in the graph world of modular composition, and it is not hard to see that $G \boxplus_B G_{env}$ represents $(E \mid Env)$.

The significance for the MONSTR world of transitions like

$$E \xrightarrow{\bar{x}y} F$$

should now be clear. Essentially such a piece of notation refers to the possible behaviour of $E$ in a suitable environment, but with the environment abstracted out. Accordingly, we can do the same with the translation and write

$$G \xrightarrow{\bar{x}y} H$$

to mean that $G$, which represents $E$, is capable of offering a write of channel $y$ over channel $x$, and upon successful completion, to perform some auxilliary rewrites in order to get into shape to represent $F$ via graph $H$. We justify this by an appeal to lemma 7.7 with the role of the environment abstracted away. (Note that it is preferable to have $G$ and $H$ present in such a notation rather than $\mathrm{Tr}(E)$ and $\mathrm{Tr}(F)$ since the former are the semantic objects which are engaged in any actual communications.)

We can take a similar approach to the input transition of the $\pi$-calculus involving the prefix $x(y).Q$.

The main thing from the original description that remains undiscussed is the business of scope extrusion and intrusion. There are no such notions in the system of [Section 3]. In that system, if a private channel $y$ in one part of the $\pi$-calculus expression is intended to be communicated to some other part of the expression, the binding $\nu y$ must have a scope large enough to enclose both parts, for which reason we introduced the syntactic reduction $\equiv\!\!\rangle^*$. This means that if one subsystem wishes to send to another subsystem a channel $y$ that it prefers to consider as private, the binding $\nu y$ must occur in the context into which both subsystems are to be placed. Technically, the the data channel $y$ becomes free in the sending subsystem, and thus there is the risk of unintended name capture unless the channel names in the subsystems and enclosing context are chosen wisely.

Scope extrusion and intrusion delays the necessity of wise name choice, by allowing $y$ to remain bound (and thus alpha-convertible), with scope initially within the sending subsystem. The open rule

$$\frac{Q \xrightarrow{\bar{x}y} Q'}{\nu y Q \xrightarrow{\bar{x}(u)} Q'\{u/y\}}$$

effectively cuts open the scope of the $\nu y$ binder by transforming the output action $\bar{x}y$ into $\bar{x}(u)$ (where $u$ is a fresh name), which indicates that the data channel name $u$ is bound and is not to be captured by other free names. The wise choice of channel names (implemented now by alpha-converting the bound name $u$ as needed), is thereby postponed until the communication takes place via the close rule

$$\frac{P \xrightarrow{\bar{x}(u)} P' \qquad Q \xrightarrow{x(u)} Q'}{P \mid Q \xrightarrow{\tau} \nu u(P' \mid Q')}$$

which demands that the bound channels in sender and receiver must match (thus forestalling inappropriate name clashes), and binds the result of the communication within a fresh binder whose scope is now large enough to enclose both subsystems, as in the previous discussion.

To achieve this effect in the framework of our translation we need do nothing special. A graph representing the sender subexpression can be generated as usual, but because the data channel $u$ say, is $\nu$-bound, $u$ does not appear in the domain of the $Args_{T_i}$ function of the sender. Because the transmission link channel, $x$ say, must be free in both sender and receiver, $x$ does appear in the domain of the $Args_{T_i}$ functions of both sender and receiver. Now when we perform the pushout to join the sender's and receiver's representing graphs, the two nodes corresponding to $x$ will be identified, though $u$ will remain private to the sender's subgraph. As we said in theorem 7.9, which data channel gets sent in a successful communication is purely the business of the sender and does not depend on the data channel's scope, and the receiver is equally indifferent to this. Thus upon successful completion of the communication, the resulting graph would correspond to a $\pi$-calculus expression in which the $\nu u$-binder had drifted into the context via a $\equiv\!\rangle^*$ reduction. This is as we would expect.

Scope extrusion and intrusion are thus superfluous in the translation. This confirms our earlier remarks, that graph connectivity is able to easily accomplish what is achieved (and perhaps a little awkwardly at that) by subtle notions of scope manipulation in the syntax of the $\pi$-calculus. In the end, the whole of the theory of the $\pi$-calculus can be translated into the world of the MONSTR representatives on the basis of the results in [Section 7], albeit that the precise technical details become more intricate.

**Equalities**

Much of the theory of the $\pi$-calculus and similar systems is concerned with the formulation of equality theories over the expressions of the calculus, formed by considering various bisimulations [see Milner et al. (1992)]. Given that we can translate the $\pi$-calculus faithfully into MONSTR, all such theories can be reformulated as properties of the corresponding class of MONSTR graphs as we indicated above. Rather than do this though, we make some comments about equalities that arise naturally in the context of MONSTR rewriting. In general they will be weaker notions than those which arise through the $\pi$-calculus bisimulations.

The natural notion of equality in the context of graph rewriting, is that of graph isomorphism (of live subgraphs). Because we are interested in the dynamics of systems, we must ensure that the underlying notion of homomorphism includes equality of the node and arc markings. Isomorphism is quite a weak notion compared to most of the ones studied for the $\pi$-calculus, for example it distinguishes between different numbers of copies of replicated processes. But it has some good points. Perhaps its main virtue is that it clearly distinguishes concurrency from interleaving. When $x$, $y$, $u$, $v$ are all different, the two expressions $\overline{x}u \mid y(v)$ and $\overline{x}u.y(v) + y(v).\overline{x}u$, are strongly ground equivalent in [Milner et al. (1992)], but in the MONSTR translation, the representative of $\overline{x}u \mid y(v)$ has two function nodes whereas the representative of $\overline{x}u.y(v) + y(v).\overline{x}u$ has only one (of course this is a consequence of how we chose to do the translation). Opinions differ on whether these expressions ought to be considered equal or not. That they are ground bisimilar rests on the fact that bisimilarity depends on sequential observation. This in turn can be laid at the feet of the inherently sequential rewriting model used to express the transition relation. And that is as true of the MONSTR translation as it is of the $\pi$-calculus.

If one goes beyond the sequential rewriting model to a more concurrent one, in which more than one redex may be rewritten simultaneously provided they don't interfere, then a more concurrent environment eg. $x(a) \mid \overline{y}b$ (as opposed to $x(a).\overline{y}b + \overline{y}b.x(a)$) could distinguish between the two expressions. Thus $\overline{x}u \mid y(v)$ might rewrite to $\mathbf{0} \mid \mathbf{0}$ in one step, while $\overline{x}u.y(v) + y(v).\overline{x}u$ would always require two steps to rewrite to $\mathbf{0}$. To explore this in more detail though, would take us far beyond the scope of this paper.

Restricting attention once more to the syntax of [Section 3], another good thing about our translation (up to graph isomorphism) is that it respects the various equivalences of $\pi$-calculus systems that we introduced in [Section 3].

Let us mention the semigroup rules for $+$ and $\mid$. For the former, we have commutativity since summation of alternatives is translated into set union of rules for distinct function symbols. (And insofar as in [Section 3] we informally permit ourselves to consider summations such as $((\pi_1.Q_1 + \pi_2.Q_2) + \pi_3.Q_3)$, the implied associativity thereof is just a feature of this set union of rules.) For the latter we have commutativity since the parallel composition of a set of subprocesses is translated into the set union of the distinct contractum nodes that represent them in the RHS of a rule for $\mid$. Associativity for $\mid$ arises since the derivation of the representing graph of a compound parallel composition differs from the derivation of the representing graph of its flattened version, by one or more auxiliary rewrites (rather as in lemma 7.5). It is clear that in standard executions, once the garbage is removed, the two resulting representing graphs come out isomorphic, (or even possibly equal if a convenient implementation of a premeditated suite of contractum building operations is adopted).

Another equivalence demanded in [Section 3] is alpha-convertibility. It is clear that this is respected by the translation (up to graph isomorphism), since each bound channel name get translated to an Empty contractum node of some rule, whose instantiations will all be equivalent under graph isomorphism. Finally, our syntactic reduction $\Rrightarrow^*$ has been dealt with in lemma 7.5.

A slightly stronger notion of equality for MONSTR systems arises through the "innocent" renaming of symbols. This is worth considering given that the tags that label subexpressions in the translation and ultimately correspond to function symbols in the translated system are arbitrary.

**Definition 8.1** Let $\mathcal{R}$ and $\mathcal{S}$ be two MONSTR rule systems. A substitution $\theta : \mathbf{S} \to \mathbf{S}$ on symbols is a system homomorphism iff for all $D = (P, root, Red, Act) \in \mathcal{R}$, $\theta D$, interpreted pointwise, is isomorphic to a rule of $\mathcal{S}$. It is a renaming of $\mathcal{R}$ iff $\theta\mathcal{R} = \mathcal{S}$, where $\theta\mathcal{R}$ is interpreted pointwise.

Thus a renaming of a rule system takes rules in $\mathcal{R}$ to rules in $\mathcal{S}$ which "do the same thing" modulo the renaming. Note that renamings automatically respect the constant Initial, and ought to respect the constant Root if garbage collection is to be unchanged.

**Definition 8.2** Let $\mathcal{R}$ be a MONSTR rule system and $\mathcal{X} = [X_0, X_1, X_2, \ldots]$ be an execution of $\mathcal{R}$. Likewise for $\mathcal{S}$ and $\mathcal{Y} = [Y_0, Y_1, Y_2, \ldots]$. Let $\theta\mathcal{R} = \mathcal{S}$ be a renaming. Then we write

$$^{(i)}x \in X_i \;\Xi_\theta\; ^{(j)}y \in Y_j \quad \text{iff} \quad ^{(i)}\theta(x) \;\Xi\; ^{(j)}y$$

where $^{(i)}\theta(x)$ is the node $^{(i)}x$ with its symbol substituted according to $\theta$ i.e. $\sigma(^{(i)}\theta(x)) = \theta(\sigma(^{(i)}x))$.

**Theorem 8.3** Let $\mathcal{R}$ be a MONSTR system and let $\theta\mathcal{R} = \mathcal{S}$ be a renaming.

(1) For every execution $\mathcal{X} = [X_0, X_1, X_2, \ldots]$ of $\mathcal{R}$ there is an execution of $\mathcal{Y} = [Y_0, Y_1, Y_2, \ldots]$ of $\mathcal{S}$ such that for all $i$, and all $^{(i)}x \in X_i$,

$$^{(i)}x \;\Xi_\theta\; ^{(j)}y$$

for a suitable $^{(j)}y \in Y_j$.

(2) For any execution $\mathcal{Y} = [Y_0, Y_1, Y_2, \ldots]$ of $\mathcal{S}$, there is an execution $\mathcal{X} = [X_0, X_1, X_2, \ldots]$ of $\mathcal{R}$ such that for all $i$, and all $^{(j)}y \in Y_j$

$$^{(i)}x \;\Xi_\theta\; ^{(j)}y$$

for a suitable $^{(i)}x \in X_i$.

We regard the above as self-evident and do not bother proving it.

By altering the tags that label subexpressions during translation we end up with a system that is renamed compared to if we had not done so. Theorem 8.3 confirms that we do not alter the intrinsic behaviour thereby. Furthermore, there is one place where we have evaded the necessity of considering renamed systems already. That is when early in [Section 5], we allowed identical subexpressions of a $\pi$-calculus expression to be identically labelled. This was exploited in lemma 7.6 when a replicand $P$ was identically tagged to its originating $!P$. Had we been forced to tag the new copy with a fresh label, we would have been forced to consider lemma 7.6 only up to a renaming, an undesirable complication in [Section 7].

Renaming thus provides a natural notion of bisimulation when node symbols are considered as arbitrary "user-supplied" names (as they are for the auxiliary symbols in the translation), rather than universal constants (as applies to symbols such as Initial, Root, and the symbols of the communication protocol). Further, it is clear that renaming is still a weaker bisimulation than the strong ground equivalence of [Milner et al. (1992)].

Being well defined in conventional set theory, both graph isomorphism and renaming provide adequate semantics for the π-calculus via the translation of [Section 5]. Unlike most semantics for process algebras, they are not directly manufactured from the syntactic components that constitute the original expression.

We note finally that [Milner et al. (1992)]'s strong equivalence, which asserts bisimilarity under all substitutions, corresponds in the MONSTR world to a rather unusual relation on graphs given by externally imposed redirections, of channel nodes to other channel nodes. Such a notion does not make any sense for arbitrary MONSTR graphs but can be made to do so for those arising from Tr($E$) systems.

# 9 CONCLUSIONS

The translation of the π-calculus into MONSTR brings out a number of useful points. Firstly it necessitates the clear understanding of the issues of free and bound names and of scoping, as they arise in very different styles of computational system. A good part of the material above can be interpreted as an essay about this. Specifically, we have seen that in a graph based language, where connections between parts of the computational structure are explicitly represented using arcs or edges, the ideas of bound variables and scope as used in the syntax of traditional languages become largely superfluous; the structure of the graph and the notion of graph matching provide an information channel that supercedes the use of the parse tree for this purpose.

Secondly, by targetting our translation to a language designed with a very concrete implementation in mind, the reasonableness in practice of the primitives of the source language may be judged. This is a particularly valuable objective if one is interested in bringing together notions of "process" in use in disparate areas of computer science, as we, by dint of remarks in the introduction, are, at least implicitly. In the case of the π-calculus, we have seen that the amount of synchronisation implicit in the communication rule can make substantial demands of implementations. In MONSTR, in which the capabilities of a single rewrite are rather closely geared to what is cheaply implementable in a single atomic action of a concurrent distributed system, we have seen that it is realistic to have a quantity of state change in such an action, equivalent to the update of the root function and of one other non-root node. Unfortunately, to implement the true dynamic synchronised point to point communication of the π-calculus, in a system featuring maximal concurrency, we need to be able to update at least one further piece of state within a single action. There is nothing to prevent us from doing so within the syntax of graph rewriting (it is easy enough in DACTL, MONSTR's parent), but there are good operational reasons why it is prohibited in MONSTR. Given this state of affairs, to stay within MONSTR, one has to either go for a protocol featuring some degree of wastefulness, or for a much more heavily serialised implementation such as afforded by a global semaphore. In this paper we have chosen the former course.

In the real world, "agents" are normally connected to a set of communication channels of which they are well aware. Generally, the agents are active and the channels are passive. Even if the agents are hazy about which other agents are connected to their channels, little synchronisation hinges on the interactions with channels due to the latters' passivity. The $\pi$-calculus communication primitive thus comes across as rather more high level and abstract than might be expected of a basic communication primitive; this particularly so since the collection of channels that an agent may use is a function of the agent's context and the dynamics of the system. In the real world, the closest that we might get to a situation where mutually ignorant agents communicate over a shared medium, is the internet. No one knows precisely who is connected to the internet at any given moment. However, even in this situation, the internet is not used to effect serialised and synchronised point to point communications between an arbitrary mutually ignorant sending/receiving pair of agents. On the contrary side, one cannot argue with the syntactic and algebraic simplicity of the communication rule of the $\pi$-calculus, to which its existence is largely attributable. Of course similar remarks apply to many other process calculi, but we have been specific in this paper.

Thirdly, the treatment of correctness deserves comment. Essentially, a suitable weak bisimulation has been set up but the techniques to construct it owe more to rewriting theory and to serialisability theory than to the usual finitistic techniques frequently found in process algebra. A fairly comprehensive and self-contained treatment of the correctness issue has been given, and many of its aspects are to be found in correctness arguments for any MONSTR program. In fact this paper contains the first such MONSTR correctness argument to be written out in reasonable detail, which gives it independent interest.

So in one sense this paper may be viewed as a concrete exercise in MONSTR program verification, the program being the output of translation Tr($E$). In another sense, because of the generic nature of the proof, it is also an exercise in compiler correctness, the compiler being the meta-level translation process. The correctness argument is visibly non-trivial, and not formal in the usual sense of the word, but it certainly sets the agenda for what such a formal proof would have to address. A fully formal proof would be a sizeable undertaking, but in reality, given suitable theories for a number of what are well understood but properly higher order concepts such as "graph", "execution" etc., [Section 6] and [Section7] involved nothing other than what could be straightforwardly expressed, in a logic in which the universal quantifiers occur bounded over some well understood set, and the existential quantifiers refered to objects that were explicitly constructed. So a formal proof would not be completely out of the question.

In a third sense, one can see the soundness proof in particular, as an exercise in serialisability theory, another case of a distant area of computer science concerned with notions of process, namely concurrency control theory from the database world, having an impact on a problem in process algebra. In this regard, the recent work on atomicity of [Lynch et al. (1994)] (see also references therein), bears comparison with the contents of this paper. Certainly the complexity of the serialisability proofs there is rather reminiscent of what appears in the present paper. Pursuing the analogy for a moment, we can view the communications of a $\pi$-calculus expression as high level transactions (from the viewpoint of the MONSTR system). Nevertheless, unlike normal database systems, individual rewrites themselves have many features of transactions too, in that serialisation is not just a matter of choosing a suitable order for them. The sheduling strategy for rewrites is determined by MONSTR rule selection semantics, and any serialisation performed, must be done within the constraints allowed by this. Up to a point, this makes life harder in our case.

Viewed from yet a different perspective, one can see the serialisability proof of theorem 7.16 (and other serialisability proofs), as a particularly easy example of a forcing or priority or finite injury argument (to use recursion-theoretic jargon), in that the object of interest, the execution $\mathcal{G}$, is constructed as the limit of a number of other executions, all featuring a decreasing proportion of undesirable characteristics. Two things contribute to the easy nature of the argument, the first being the explictly constructible nature of the transformation process, and the second is the vital observation that the process can be neatly split into phases, the earliest of which serve to simplify matters considerably for their successors. A "one pass" version of the theorem would be perfectly feasable, but the technical details would be considerably more intricate, as the reader is invited to imagine.

The translation itself was inspired by other translations into term graph rewriting systems. In particular by those in [Banach and Papadopoulos (1993), Banach and Papadopoulos (1995)], which are concerned with concurrent logic languages. Also [Glauert (1992)] does related work on mapping a process calculus into a term graph rewriting system, however with the crucial ommission of the guarded summation construct. It is precisely that which forces us to adopt a communication protocol and its synchronisation problems, due to the amount of atomic state change implicit in the general case of a single comunication of the $\pi$-calculus. It is also that which is the source of most of the fun and games in [Section 6] and [Section 7].

## 10 References


[Aczel (1993)] Aczel P.H.G., Processes and Final Universes. Seminar, Dept. of Computer Science, Manchester University, (1993).

[Asperti and Martini (1992)] Asperti A., Martini S., Categorical Models of Polymorphism. Information and Computation **99**, (1992), 1-79.

[Banach and Papadopoulos (1993)] Banach R., Papadopoulos G., Parallel Term Graph Rewriting and Concurrent Logic Programs. *in*: Proc. WPDP-93, Bulgarian Acad. of Sci., Boyanov (ed.), (1993), 303-322. (North Holland, to appear.)

[Banach and Papadopoulos (1995)] Banach R., Papadopoulos G., Linear Logic Behaviour of Term Graph Rewriting Programs. *in*: Proc. A.C.M. SAC-95, (1995), 157-163.

[Banach et al. (1988)] Banach R., Sargeant J., Watson I., Watson P., Woods V., The Flagship Project. *in*: Proc. UK-IT-88, (Alvey Technical Conference), 242-245, Information Engineering Directorate, Department of Trade and Industry, IEE Publications, (1988).

[Banach and Watson (1989)] Banach R., Watson P., Dealing with State in Flagship: the MONSTR Computational Model. *in*: Proc. CONPAR-88, Jesshope, Reinhartz (eds.), 595-604, B.C.S. Workshop Series, Cambridge University Press, (1989).

[Banach (1993a)] Banach R., MONSTR I — Fundamental Issues and the Design of MONSTR. *Submitted to* New Generation Computing, (1993).

[Banach (1993b)] Banach R., MONSTR II — Suspending MONSTR Semantics. *Submitted to* New Generation Computing, (1993).

[Banach (1993c)] Banach R., MONSTR: Term Graph Rewriting for Parallel Machines. *in*: Term Graph Rewriting: Theory and Practice, Sleep, Plasmeijer, van Eekelen (eds.), 243-252, John Wiley, (1993).

[Berry and Boudol (1990)] Berry G., Boudol G., The Chemical Abstract Machine. *in*: 17th Annual Symposium on Principles of Programming Languages, A.C.M., (1990). *Also in*: Theoretical Computer Science **96**, (1992), 217-248.

[Coquand et al. (1989)] Coquand T., Gunter C., Winskel G., Domain Theoretic Models of Polymorphism. Information and Computation **81**, (1989), 123-167.

[Corradini et al. (1994)] Corradini A., Montanari U., Rossi F., An Abstract Machine for Concurrent Modular Systems: CHARM. Theoretical Computer Science **122**, (1994), 165-200.

[Degano and Montanari (1987)] Degano P., Montanari U., A Model of Distributed Systems Based on Graph Rewriting. J.A.C.M. **34**, (1987), 411-449.

[Glauert (1992)] Glauert J.R.W., Asynchronous Mobile Processes and Graph Rewriting. *in*: Proc. PARLE-92, Etiemble, Syre (eds.), LNCS **605** 63-78, Springer, (1992).

[Glauert et al. (1988a)] Glauert J.R.W., Kennaway J.R., Sleep M.R., Somner G.W., Final Specification of DACTL. Internal Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, U.K, (1988).

[Glauert et al. (1988b)] Glauert J.R.W., Hammond K., Kennaway J.R., Papdopoulos G.A., Sleep M.R., DACTL: Some Introductory Papers. School of Information Systems, University of East Anglia, Norwich, U.K, (1988).

[Lynch et al. (1994)] Lynch N., Merritt M., Weihl W., Fekete A., Atomic Transactions. Morgan Kaufmann, (1994).

[Milner (1979)] Milner R., Flow Graphs and Flow Algebras. J.A.C.M. **26**, (1979), 794-818.

[Milner (1989)] Milner R., Communication and Concurrency. Prentice-Hall, (1989).

[Milner (1993a)] Milner R., The Polyadic Pi-Calculus: A Tutorial. *in*: Logic and Algebra of Specification, Bauer, Brauer, Schwichtenberg (eds.), 203-246, Springer, (1993).

[Milner (1993b)] Milner R., An Action Structure for Synchronous Pi-Calculus. *in*: Proc. FCT-93, Esik (ed.), LNCS **710** 87-105, Springer, (1993).

[Milner et al. (1991)] Milner R., Parrow J., Walker D., Modal Logics for Mobile Processes. *in*: Proc. CONCUR-91, Baeten, Groote (eds.), LNCS **527** 45-60, Springer, (1991).

[Milner et al. (1992)] Milner R., Parrow J., Walker D., A Calculus of Mobile Processes – I / II. Inf. and Comp. **100**, (1992), 1-40, 41-77.

[Nielsen et al. (1981)] Nielsen M., Plotkin G., Winskel G., Petri Nets, Event Structures and Domains, Part I. Theoretical Computer Science **13**, (1981), 85-108.

[Parrow (1994)] Parrow J., Interaction Diagrams. *in*: A Decade of Concurrency, de Bakker, de Roever, Rozenberg (eds.), LNCS **803** 477-508, Springer, (1994). *and* Manuscript, SICS, Kista, Sweden.

[Watson and Watson (1987)] Watson P., Watson I., Evaluating Functional Programs on the Flagship Machine. *in*: Proc. FLCA-87, Kahn (ed.), LNCS **274** 80-97, Springer, (1987).

[Watson et al. (1987)] Watson I., Woods V., Watson P., Banach R., Greenberg M., Sargeant J., Flagship: A Parallel Architecture for Declarative Programming. *in*: Proc. 15th Annual International Symposium on Computer Architecture, Hawaii, ACM, (1987).

[Watson et al. (1989)] Watson I., Sargeant J., Watson P., Woods V., The Flagship Parallel Machine. *in*: Proc. CONPAR-88, Jesshope, Reinhartz (eds.), 125-133, BCS Workshop Series, Cambridge University Press, (1989).

[Winskel (1986)] Winskel G., Event Structures. *in*: Petri Nets, An Advanced Course, LNCS **255**, 325-392, (1986).

[Winskel (1988)] Winskel G., An Introduction to Event Structures. *in*: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. de Bakker, de Roever, Rozenberg (eds.), LNCS **354**, 364-397, (1988).