IMPLEMENTING INTERACTION NETS IN MONSTR

Richard Banach

Department of Computer Science
University of Manchester
Manchester M13 9PL
U.K.

banach@cs.man.ac.uk

George A. Papadopoulos

Department of Computer Science
University of Cyprus
P.O.B. 537, CY-1678, Nicosia
CYPRUS

george@turing.cs.ucy.ac.cy

ABSTRACT

Two superficially similar graph rewriting formalisms, Interaction Nets and MONSTR, are studied. Interaction Nets come from multiplicative Linear Logic and feature undirected graph edges, while MONSTR arose from the desire to implement generalised Term Graph Rewriting efficiently on a distributed architecture and utilises directed graph arcs. Both formalisms feature rules with small left hand sides consisting of two main graph nodes. A translation of Interaction Nets into MONSTR is described, thus providing an implementation route for the former based on the latter and particularly suited to distributed implementations.

Keywords: *Term Graph Rewriting Systems; MONSTR; Interaction Nets; Distributed Systems.*

INTRODUCTION

There are many different kinds of graph that have been studied over the years, and inevitably, people have invented a rather large number of ways of rewriting them, yielding a vast number of different models of computation. In this paper we study the relationship between two models that bear a superficial resemblance, but that were inspired by very different motivations: Interaction Nets and MONSTR.

Interaction Nets ([10,11]) evolved from the multiplicative fragment of Linear Logic ([8]). The basic idea is that a multiplicative proof object consists of inference steps. The object is represented by a graph, in which the individual inference steps, combining a number of hypotheses to form a conclusion, are represented by agent nodes for which the adjacent edges represent the hypotheses and conclusions. The special nature of a conclusion singles it out, making it "principal". The dynamics of proof objects is enshrined in the notion of elimination of cuts, whereby the two conclusions meeting in a cut are eliminated by transforming the proof object in the vicinity of the cut. In the world of the representing graphs, two agents joined by a connection which is principal for both of them is the analogue of the cut, and its elimination is a rewrite rule for such graphs. Thus arises the Interaction Net model of graph rewriting.

MONSTR ([1,3]) originated from the desire to implement the generalised Term Graph Rewriting language DACTL ([9]) on a distributed parallel machine, the Flagship machine ([13]). The demands of (even an imperfectly adhered to notion of) serialisability for DACTL executions, necessitated curtailing the expressive power of DACTL rules rather drastically. Because it was vital for the Flagship machine that the computational model encompassed a reasonable notion of state despite the predominantly declarative programming models that it was primarily intended for, the MONSTR computational model as it eventually emerged, permitted each rule to include at most one unit of updatable state per rewrite, apart from the root of the rewrite itself, giving two key nodes on the left hand side of each rule. The similarity to Interaction Nets is clear.

The rest of the paper is organised as follows. The next section describes MONSTR and some of its more important properties. The following section does the same for Interaction Nets, following the treatment of Banach ([2]). The two models are brought together in the fourth section which describes a translation of Interaction Nets into MONSTR.

MONSTR

Unlike most typical graph rewriting formalisms (see e.g. [9,12]), MONSTR was designed with the repercussions of efficient distributed implementation uppermost in mind. This meant tuning the expressiveness of the basic atomic actions of the model to the capabilities of a typical distributed architecture, so as not to overtax the synchronisation properties of the latter unduly — something which would lead to a dramatic loss of performance as a result of having to implement a lot of distributed locking.

MONSTR Rewrites

The fundamental objects of MONSTR are *term graphs*. A term graph, is a directed graph where the nodes are labelled with symbols, assumed of fixed arity, and each node has a sequence of out-arcs to its child nodes. The nodes and arcs of term graphs are marked to control rewriting strategy as we will see below. The term graph that represents the instantaneous state

"Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."
© 1997 ACM 0-89791-850-9 97 0002 3.50

of the computation is modified by the application of some rule. Let us look at a rule in action, to see what happens during a rewrite.

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

First the LHS (the part before =>) is matched. F is the root node and has two children, the Cons node, and the Var node. The Cons node has two unlabelled children; such undefined nodes may match anything. Note that the pattern is shallow; this is fundamental to MONSTR as large patterns demand large scale locking to ensure atomicity.

Once a match is located, which must be at an active (*-marked) node of the graph, the nodes on the RHS are built into the redex area. Thus a once-suspended (#-marked) G node is constructed, with arcs to the existing LHS nodes referred to by a and b (so these nodes become shared even if they weren't previously). Also the arc to b is a notification arc (^-marked). The other new node is the active SUCCEED node.

The notation => indicates that the root is to be redirected to the node immediately following the => i.e. G. Also the Var node is to be redirected to SUCCEED by the notation s:=SUCCEED. During redirection, all in-arcs to the respective redirection subjects (i.e. F and Var) are replaced by in-arcs to the respective targets (i.e. G and SUCCEED). Redirection is the fundamental notion of update in Term Graph Rewriting, being a graph-oriented version of substitution.

The final tasks of a MONSTR rewrite are to make the root inactive (idle, written visibly as ε when necessary); and to activate specified LHS nodes (which causes them to be marked active if otherwise unmarked). In the concrete syntax, this is accomplished by mentioning the relevant nodes on the RHS of the rule, with a * marking e.g. b above. We illustrate the action of the rule described above in Figure 1.

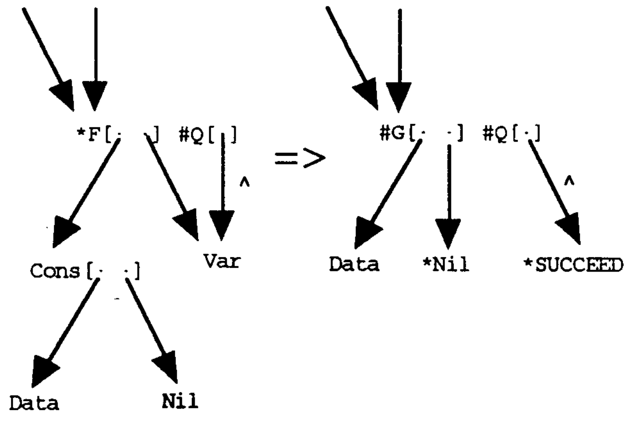

FIGURE 1

In the diagram, note how the in-arcs of F now point to G after redirection, and those of Var point to SUCCEED. We are assuming in the rewrite illustrated, that the LHS nodes F and Cons, had no further in-arcs, and thus became inaccessible and were garbage collected.

The above assumed that there was a rule which matched, and that the explicitly matched arguments of the root of the redex (i.e. those arguments whose symbol needs to be inspected for pattern matching to succeed, Cons and Var in our example), are idle. If any of the explicitly matched arguments of the root are not idle then *suspension* occurs, in which the root of the redex becomes suspended on as many of its explicitly matched arguments as happen to be non-idle; i.e. the root node acquires that many suspension markings, and each of the relevant out-arcs becomes a notification arc (i.e. ^-marked).

If no rule can match regardless of the markings, then notification occurs, in which the root becomes idle, and for all its notification in-arcs, the ^ marking is removed, and the number of suspensions (#'s) in the parent node's marking is decremented (with #0 = *). In this manner subcomputations can signal their completion to their parents (and suspended rewrites can thereby be reawakened).

MONSTR Syntactic Restriction and Runtime Properties

To make the above ideas into a computational model suited to distributed machines, a number of restrictions are imposed on the syntactic structure of systems so that some useful runtime properties can be rigorously demonstrated. We point out the main ones now rather informally, referring the reader to [1,3] for a more detailed study of why these are appropriate and what their consequences are:

- All nodes respect the arities of their symbols. (Within rules; and by means of a simple induction, within all execution graphs.)
- The alphabet of symbols is divided into *functions*, *constructors* and *stateholders*. Functions label root nodes of LHSs of rules (but not subroot nodes), and function symbols must always have at least one *default rule* which has no explicitly matched arguments, enabling such a rule to always rewrite at runtime, regardless of its arguments. Constructors and stateholders can label subroot nodes of LHSs of rules (but not the root nodes). Functions and stateholders (but not constructors) can label LHS nodes of redirections, and all redirections must specify an explicit function or stateholder as LHS node, (one of which must be the root). Thus no attempt is ever made to redirect a constructor at runtime.
- The pattern matching requirements of each redex, depend solely on the symbol at the root (and so can be delegated to simple hardware). More specifically, each function symbol has a fixed matching template, one level deep, which specifies which of the root's children need to have their symbols inspected to match a non-default rule for the function. Furthermore, a single fixed position within this template can be designated for matching stateholders; the other positions may only match constructors (thus, justifying the acronym MONSTR which stands for Maximum of One Non-root STateholder per Rewrite). And no pointer equality testing is permitted except for the matched constructors, (and for some special builtins of which we will have no need in this paper).
- All nodes in rules are *balanced;* i.e. have exactly as many suspension markings as they have notification out-arcs. (And by a simple induction, all nodes in all execution graphs are balanced too.)
- All nodes in rules are either *state saturated* (i.e. if they have one or more notification in-arcs and are idle, then they must be stateholders), or designated for *activation*.
- Any redirection whose RHS node is idle and not activated, must be a stateholder. (As a consequence of this and the previous point, all nodes in all execution graphs are state saturated.)

- In all rules, the LHS node of a redirection should not be activated unless it is also the RHS of some other redirection. (In practice this enables the convenient representation of rewriting by packet store manipulations, (and particularly the representation of most redirections by packet overwriting).)

By convention rewriting always starts with a single active node labelled `INITIAL`; and MONSTR provides a rule selection policy which permits non-default rules to be selected before default rules when either would match (; is the sequential rule selector in concrete syntax). Note that we have said nothing very specific about garbage collection. For the rest of this paper it will be sufficient to assume that this works in a "sensible way"; see [1,3] for a more precise discussion.

INTERACTION NETS

Interaction Nets were invented for describing finegrained computations graphically. Their theory builds on prior work in multiplicative Linear Logic that gives the Interaction Net model particularly transparent properties regarding confluence, and to a lesser extent normalisation. We will use the formulation of Banach ([2]) as it will prove more convenient for the translation that we will subsequently give.

Interaction Nets can be viewed as bipartite graphs where the two node kinds are *agent* nodes and *port* nodes. Each agent bears a *symbol,* which determines the number of *port edges* incident on it, and the attributes of those edges. These port edge attributes are: the port edge's *name*; whether it is *principal* or *auxiliary*; and the port edge's *type*. The types come in complementary pairs ($\alpha+$, $\alpha-$), for α drawn from a suitable type alphab t. Exactly one of an agent's ports is principal, and the rest are auxiliary. Finally, we have the all-important port invariant which states that at most two port edges may be incident on a port node, and that they must be of complementary types, say $\alpha+$ and $\alpha-$. Figure 2 illustrates the situation and also introduces the notion of *port connection,* which we will use as required below. Note that we indicate principal port edges using an arrowhead, while auxiliary port edges are unadorned. Also we will suppress some of the detail to avoid clutter in future. We will say that a port connection consisting of two principal port edges is a principal port connection.

An Interaction Net rewrite rule has on the left hand side, two agents joined by a principal port connection, and with all their auxiliary ports free, (i.e. not connected to other port edges). The right hand side is an arbitrary Interaction Net with the same external interface as the left hand side, which is to say that part of the rule's data is a bijective mapping between the free port edges of the LHS and RHS nets, which preserves the types. The only exceptions to the bijective law are *short circuits,* where two free port nodes of the LHS with complementary types are allowed to be identified in the RHS. Figure 3 shows a picture of a rule. The numbers on the interface port edges define the aforementioned bijection between left and right hand sides. The blobs labelled ($2:\beta+$, $9:\beta-$) and ($3:\gamma-$, $4:\gamma+$) are the short circuits in an obvious notation. And any fresh port nodes introduced in the RHS, i.e. port nodes not belonging to the interface, will be called *internal ports* in future (e.g. port node 10 in Figure 3).

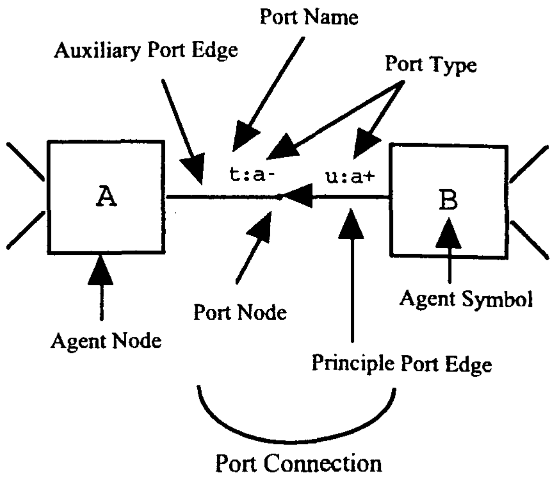

FIGURE 2

The operational semantics of such a rule starts by finding a matching of the LHS of the rule in the net being rewritten. Then the matched subnet is removed, and replaced with a copy of the RHS of the rule. It is easy to see that the type preserving bijective law with short circuits of type-matched pairs, means that the port invariant is preserved by rewrites. Figures 4a and 4b show a rewrite according to the rule introduced previously before and after the rule has been applied.

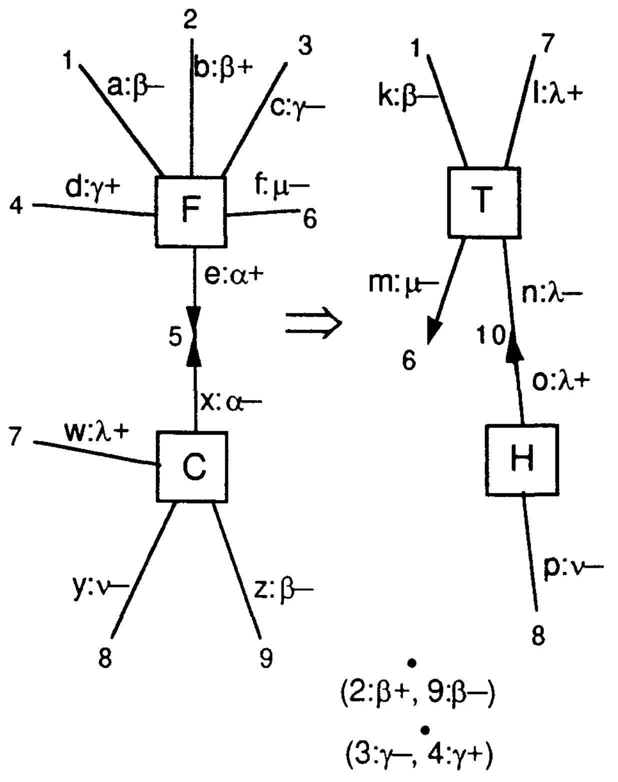

FIGURE 3

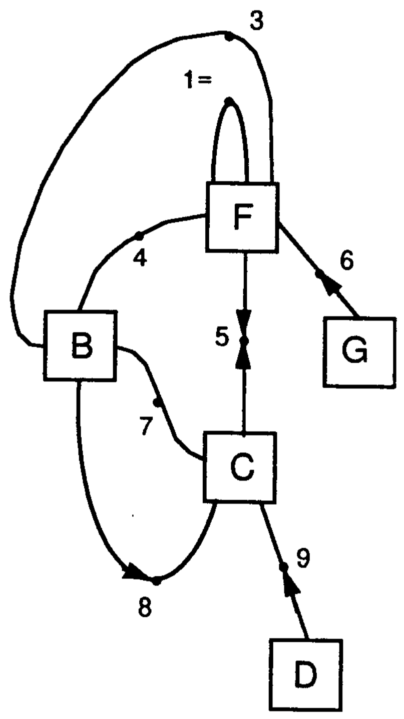

FIGURE 4a

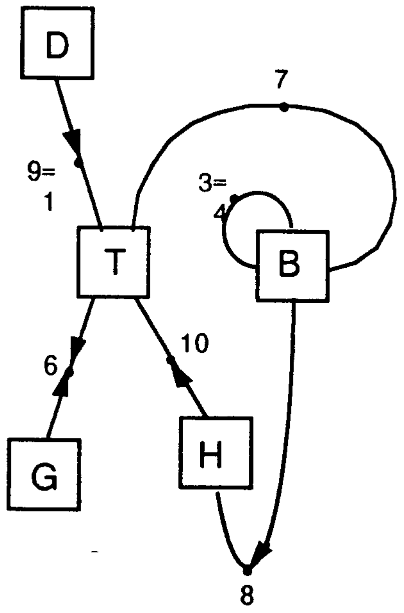

FIGURE 4b

The principal/auxiliary distinction on ports, and the fact that rules must feature a principal connection, leads to the study of *deadlock prevention* for Interaction Nets. A deadlock is where there is a cycle of agents, each of whose principal port edges connects with an auxiliary port edge of the next agent. Obviously in such a situation, none of the agents involved can rewrite...ever. A rich theory can be developed to ensure that such situations cannot arise, but this will not be needed in this paper. See the cited references for details.

We end this brief exposition of Interaction Nets with some further observations. Since each agent has only one principal port, it can interact with at most one other agent, the one connected to the said principal port, and then only if that agent's corresponding port is itself principal. This means that apart from auxiliary port nodes that they might have in common, any two distinct redexes in an arbitrary net are non-overlapping, and provided for each possible pair of agents on the LHS there is exactly one rule, Interaction Net rewriting is Church-Rosser. If moreover, the RHS's of rules are smaller than their LHS's (as is the case e.g. for the Interaction Net version of LLM cut elimination), then Interaction Net rewriting is also terminating.

The form of rules, and the one principal port restriction for agents, combine to imply the following structure for the life history of a port node: It is created during some rewrite. Perhaps the two port edges incident on it are auxiliary. While an incident port edge is auxiliary, it may be replaced by another port edge, or short circuited, as its owning agent interacts along some different port connection. Once an auxiliary port edge is replaced by a principal port edge however, the port edge is commited. It can no longer be replaced, except if the other incident edge also becomes principal, and the whole connection becomes the redex of a rewrite, at which point the connection, and the pair of connected agents are garbaged. For any given port, the whole of the above is not compulsory, and the port may undergo only a subsequence of the indicated transformations, but the structure of that subsequence must always fit within the indicated pattern. This fact can be usefully exploited by implementations, as we will see below.

MAPPING INTERACTION NETS TO MONSTR

The most striking thing about Interaction Nets from an implementation point of view is that the port edges are unoriented. (We disregard the arrowheads of the principal port edges for this purpose.) Usually, implementing an unoriented edge in a concrete data structure requires a pair of oppositely oriented pointers. With this in mind, if concurrent update of the data structure by many rewriting processes is envisaged, then unoriented edges can spell disaster performancewise, since one has to avoid race conditions arising from two agents competing to update the same edge from opposite ends. This can involve all the overheads of locking and perhaps of deadlock avoidance. By contrast, MONSTR, with a close eye kept on implementation matters, features only directed arcs, avoiding the problems indicated above.

The similar shape of left hand sides in the two models is striking. The obvious thing that we would like to do, is to relate the two agents connected by a principal and undirected port connection in the LHS of a net rule, to the function and stateholder connected by a directed arc in a corresponding MONSTR rule — fortunately it is possible to do this if one exploits the orientedness of the Interaction Net type system to provide an orientation for the principal port connection. Indeed it is possible to go further. Noticing that the agents of an Interaction Net computation get inspected and replaced exactly once each (since each agent participates in exactly one interaction), enables us to represent some agents by constuctors rather than the more general stateholders.

Somewhat arbitrarily, we choose to encode agents with principal ports of positive type by MONSTR function nodes, and those with principal ports of negative type by MONSTR constructor nodes. Further, we encode port nodes (which are synchronisation points) by stateholders labelled with the symbol `Port`. To start with, all port edges are represented by arcs from the agent nodes to port nodes, and all port connections are represented by a pair of in-arcs of a `Port` node — in a representation of an Interaction Net that is rather obvious. For a principal connection, we have to turn the inward pointing pair of arcs into a single arc; furthermore doing this in a manner which respects the independence of the two port edges. Fortunately, the typical life history of a port edge sketched in the previous section helps, as both edges are following similar trains of activity.

At the point that a function node is created, it is created active. It matches its child. If this is a constructor representing an agent, a MONSTR rewrite representing an Interaction Net rewrite takes place. If not, and the function sees only a `Port` node, it suspends waiting to be notified of a change of state. Conversely, the constructor representing the negative type agent is created along with an additional `Assign` function whose job is to redirect the constructor's principal port node to the constructor itself, thereby making the constructor visible to any waiting function node, should there indeed be one there now or at some point in the future. In fact it is clear that with the desired behaviour of the `Assign` function, the original arc from the constructor to the `Port` node becomes superfluous, and can be dispensed with. It is easy to see that the protocol works as required, and in particular that it allows auxiliary port edge representatives to be replaced at will. In particular, the replacement of agents and of their port edges works by garbage collection: in a rewrite, the nodes representing the LHS of the rule are created and connected to the relevant `Port` nodes; meanwhile, the LHS agent nodes loose all live references and thus become garbaged. (Obviously, if a principal port connection is being created all at once within the RHS of some rule rather than dynamically, then the protocol can be optimised away.)

We now give the translation formally. First we write down a generic Interaction Net rule $\mathtt{DI}_N$.

LHS: Agents:

F, with auxiliary ports $\mathtt{f1}:\alpha1^\circ \ldots \mathtt{fi}:\alpha i^\circ \ldots \mathtt{fn}:\alpha n^\circ$

C, with auxiliary ports $\mathtt{c1}:\beta1^\circ \ldots \mathtt{ci}:\beta i^\circ \ldots \mathtt{cm}:\beta m^\circ$

where ° is either + or – in each case

RHS: Agents:

E1…Ei…Er, with principal ports $\mathtt{Q1}:\Delta1^\circ \ldots \mathtt{Qi}:\Delta i^\circ \ldots \mathtt{Qr}:\Delta r^\circ$
and with auxiliary ports

$\mathtt{q1}_1:\delta1_1^\circ \ldots \mathtt{q1}_{j1}:\delta1_{j1}^\circ \ldots \mathtt{q1}_{t1}:\delta1_{t1}^\circ$ (of E1)

$\mathtt{qi}_1:\delta i_1^\circ \ldots \mathtt{qi}_{ji}:\delta i_{ji}^\circ \ldots \mathtt{qi}_{ti}:\delta i_{ti}^\circ$ (of Ei)

$\mathtt{qr}_1:\delta r_1^\circ \ldots \mathtt{qr}_{jr}:\delta r_{jr}^\circ \ldots \mathtt{qr}_{tr}:\delta r_{tr}^\circ$ (of Er)

Internal Ports: p1…pi…ps

Short Circuits: $(\mathtt{x1}:\gamma1-,\ \mathtt{y1}:\gamma1+) \ldots$
$(\mathtt{xi}:\gamma i-,\ \mathtt{yi}:\gamma i+) \ldots$
$(\mathtt{xu}:\gamma u-,\ \mathtt{yu}:\gamma u+)$

where in the above there is an onto mapping

$$\theta:\{\mathtt{Q1}\ldots\mathtt{Qr}\} \cup \{\mathtt{Q1}_1\ldots\mathtt{Qr}_{tr}\} \cup \{\mathtt{x1}\ldots\mathtt{yu}\}$$
$$\rightarrow \{\mathtt{f1}\ldots\mathtt{fn}\} \cup \{\mathtt{c1}\ldots\mathtt{cm}\} \cup \{\mathtt{p1}\ldots\mathtt{ps}\}$$

where θ^{-1} is 1-1 on $\{\mathtt{f1}\ldots\mathtt{fn}\} \cup \{\mathtt{c1}\ldots\mathtt{cm}\}$, and each $\theta^{-1}(\mathtt{pi})$ is of cardinality 2. (This just expresses the port invariant for the RHS.)

In the MONSTR translation, *italic Courier* items will correspond to symbols or parts of symbols mapped from the components of the above generic rule, while `upright Courier` items will stand for constants of the translation. In general, we use font change to identify pieces that correspond in the Interaction Nets and MONSTR rules. The MONSTR rule D_M that translates the above rule is:

*F[C[c1 ... ci ... cm] f1 ... fi ... fn] ⇒ *OK,*
p1:Port ,..., pi:Port ,..., ps:Port ,
e1:μ1 E1[Q1 $q1_1 \ldots q1_{j1} \ldots q1_{t1}$ *] ,...,*
ei:μi Ei[Qi $qi_1 \ldots qi_{ji} \ldots qi_{ti}$ *] ,...,*
er:μr Er[Qr $qr_1 \ldots qr_{jr} \ldots qr_{tr}$ *] ,*

where if Qi : Δi+ (i.e. Ei is an agent of pos type principal port,
and thus *Ei* is a function symbol),
then μi = * ,
else if Qi : Δi - (i.e. Ei is an agent of neg type principal port,
and thus *Ei* is a constructor symbol),
then μi = ε , *Qi* is absent from the arguments of *Ei*,
and we also have **Assign[Qi ei]*
fi
**Assign[x1 y1],…,*Assign[xi yi],…,Assign[xu yu]*

where in the above, the map θ is interpreted as syntactic identity, i.e. if θ(Q4) = c9 say, then *Q4* <u>is identical to</u> *c9*, giving the connectedness of the corresponding term graph according to the syntactic conventions of MONSTR.

In addition we need the following suite of rules.

```
Assign[ v:Port a ] => *OK , v := *a ;
Assign[ v a ] => #Assign[ ^*v ] ;

F[ p:Port f1 ... fi ... fn ] => #F[ ^p f1 ... fi ... fn ] ;
F[ p f1 ... fi ... fn ] => #F[ ^*p f1 ... fi ... fn ] ;
```

Here is the translation of the specific example we had previously.

```
F[ C[ c7 c8 c9 ] f1 f2 f3 f4 f6 ] => *OK ,
     p10:Port ,
     e1:T[ f1 c7 p10 ] , *Assign[ f6 e1 ] ,
     e2:*H[ p10 c8 ] ,
     *Assign[ c9 f2 ] , *Assign[ f3 f4 ] ;
```

To further illustrate the mapping procedure, we now show how an Interaction Nets version of the unavoidable `Append` is translated using the framework just developed. (We recall that the convention we are adopting in this paper is that the principal ports of the "functions" have a positive sign and those of the "constructors" have a negative one; the opposite would also of course be valid provided it would be used consistently and indeed this is the one adopted in Lafont's papers ([10,11])).

```
type atom, list

symbol Cons:list-;atom+,list+
       Nil:list-
       Append:list+;list+,list-

Cons[x,Append(v,t)] >< Append[v,Cons(x,t)]
Nil >< Append[v,v]
```

It should be now easier for the reader to understand how the following MONSTR rule system is derived.

```
Append[Cons[x u] w v] => *OK,
                         a:Port,
                         e1:Cons[x a],
                        e2:*Append[u a v],
                        *Assign[w e1];
Append[Nil w v] => *OK,
                   *Assign[w v];
Append[p:Port w v] => #Append[^p w v];
Append[p w v] => #Append[^*p w v];
```

A typical MONSTR query involving the above program is shown below. (We recall that `INITIAL` denotes the first piece of graph to be attempted for reduction in a MONSTR/Dactl program.)

```
INITIAL => p,
           *Append[l1 p l2],
           l1:Cons[1 Cons[2 Nil]],
           l2:Cons[3 Cons[4 Nil]],
           p:Port;
```

Note, that the result of appending the two lists will appear in the second argument of `Append`.

CONCLUSIONS AND RELATED WORK

In this paper we have studied the relationship between two graph rewriting formalisms, namely Lafont's Interaction Nets and Banach's MONSTR and we have presented a concrete translation from the former to the latter. The two formalisms have evolved from rather different perspectives and for different reasons — Interaction Nets as a version of Linear Logic based on normalised Proof Nets and rather abstract and MONSTR as a down stripped verion of a compiler target language with emphasis on easiness of implementation on distributed machines. Thus, being able to provide a concrete mapping framework from the former onto the latter has a number of advantages and interesting possibilities: (1) it provides an "implementation apparatus" of Interaction Nets via the compiler target language MONSTR - bear here in mind that Interaction Nets exhibit a high degree of parallelism which can be fully exploited by MONSTR; (2) it illustrates how Interaction Nets based rule systems can be transformed into "ordinary" graph rewriting rule based code for execution using traditional rewriting formalisms (like the MONSTR computational model) and associated architectures; (3) it offers ways to reason about some of MONSTR's operations and features by lifting their interpretation to the level of the computational model that is being mapped onto MONSTR (as in the case of multiple non-root overwrites and the rather unusal stateholder object); (4) it allows MONSTR to be used as a point of reference and comparison between different computational models by examining their common points and differences at the MONSTR level.

This paper complements work by the authors (and others) to justify the characterization of Term Graph Rewriting in general but, more to the point, MONSTR in particular, as a "generalised computational model" able to accommodate the needs of, often divergent in behaviour, computational models; needs that range from those associated primarily with reasoning and specification to those more related to implementation issues. In [5] it is shown how concurrent logic languages can be implemented in MONSTR. In [4] MONSTR is used to reason about pi-calculus and in [7] it is shown how MONSTR can be used as an implementation and specification framework for concurrent object-oriented languagues. Finally, [6] studies the possibility of using MONSTR to implement concurrent languages based on Linear Logic but also discusses the definition of a "linear" MONSTR sublanguage.

Currently, we are examining ways to map onto MONSTR *untyped* Interaction Nets but also the reverse, i.e. the mapping of the MONSTR rewriting formalism onto Interaction Nets.

REFERENCES

[1] Banach R., "MONSTR: Term Graph Rewriting for Parallel Machines", in [12], pp. 243-252.

[2] Banach R., "The Algebraic Theory of Interaction Nets", Department of Computer Science, University of Manchester, Technical Report MUCS-95-7-2, http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-95-7-2.html,1995.

[3] Banach R., "MONSTR I — Fundamental Issues and the Design of MONSTR", *Journal of Universal Computer Science*, **2** (4), 1996, pp. 164-216, http://www.iicm.tu-graz.ac.at/jucs.

[4] Banach R., Balazs J., Papadopoulos G. A., "A Translation of the Pi-Calculus into MONSTR", *Journal of Universal Computer Science*, **1** (6), 1995, pp. 339-398, http://www.iicm.tu-graz.ac.at/jucs.

[5] Banach R., Papadopoulos G. A., "Parallel Term Graph Rewriting and Concurrent Logic Programs", *WPDP'93*, Sofia, Bulgaria, May 4-7, 1993, pp. 303-322.

[6] Banach R., Papadopoulos G. A., "Linear Behaviour of Term Graph Rewriting Programs", *ACM SAC'95*, Nashville, TN, USA, Feb. 26-28, 1995, ACM Press, pp. 157-163.

[7] Banach R., Papadopoulos G. A., "Term Graph Rewriting as a Specification and Implementation Framework for Concurrent Object Oriented Programming Languages", *MPPM'95*, Berlin, Germany, Oct. 9-12, 1995, IEEE Press, pp. 151-158.

[8] Girard J-Y., "Linear Logic", *Theoretical Computer Science* **50**, 1987, pp. 1-102.

[9] Glauert J. R. W., Kennaway J. R., Sleep M. R., "DACTL: An Experimental Graph Rewriting Language", *GraGra*, Bremen, Germany, March 5-9, 1990, LNCS **532**, Springer Verlag, 1991, pp. 378-395.

[10] Lafont Y., "Interaction Nets", in 17*th ACM POPL*, San Francisco, Ca., Jan. 17-19, 1990, ACM Press, 95-108.

[11] Lafont Y., "The Paradigm of Interaction", Working Paper, file://lmd.univ-mrs.fr/pub/lafont/paradigm1.ps.Z, 1991.

[12] Sleep M. R., Plasmeijer M. J., van Eekelen M. C. J. D. (eds.), *Term Graph Rewriting: Theory and Practice*, John Wiley, New York, 1993.

[13] Watson I., Woods V., Watson P., Banach R., Greenberg M., Sargeant J., "Flagship: A Parallel Architecture for Declarative Programming", 15*th ISCA*, Hawaii, May 30 - June 2, 1988, ACM Press, pp. 124-130.