MONSTR I — Fundamental Issues and the Design of MONSTR

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

Abstract: This is the first in a series of papers dealing with the implementation of an extended term graph rewriting model of computation (described by the DACTL language) on a distributed store architecture. In this paper we set out the high level model, and under some simple restrictions, prove an abstract packet store implementation correct modulo garbage. The abstract packet store model is compared to a more realistic and finegrained packet store model, more closely related to the properties of a genuine distributed store architecture, and the differences are used to inspire the definition of the MONSTR sublanguage of DACTL, intended for direct execution on the machine. Various alternative operational semantics for MONSTR are proposed to reflect more closely the finegrained packet store model, and the prospects for establishing correctness are discussed. The detailed treatment of the alternative models, in the context of suitable sublanguages of MONSTR where appropriate, are subjects for subsequent papers.

Key Words: Intermediate Languages, Term Graph Rewriting, MONSTR, Semantic Models.

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

1 INTRODUCTION

This is the first of a series of papers about MONSTR, in which we will study the problem of implementing an extended term graph rewriting model of computation described by the language DACTL [Glauert et al. (1988a)], [Glauert et al. (1988b)], [Glauert et al. (1990)] on a distributed store architecture, from a fairly theoretical vantage point.

The issue arose in the late 80's when the Flagship Project intended to use DACTL as a general purpose intermediate language for a distributed store multiprocessor, the Flagship Machine, [Watson and Watson (1987)], [Watson et al. (1987)], [Banach et al. (1988)], [Watson et al. (1989)]. This proved too ambitious an objective, and a sublanguage of DACTL, MONSTR, was designed to make the intermediate language problem more tractable. As well as being useful and appropriate as an intermediate language, MONSTR subsequently proved to be a flexible and expressive model of computation in its own right. For a variety of applications in addition to its use in the Flagship Machine, see eg. [Banach and Papadopoulos (1993)], [Banach and Papadopoulos (1995a)], [Banach and Papadopoulos (1995b)], [Banach et al. (1995)], [Banach and Papadopoulos (1996a)], [Banach and Papadopoulos (1996b)].

Given the retrenchment to the MONSTR sublanguage, the original question then became that of the correctness of the MONSTR implementation. The question is non-trivial since the semantics of the language and of the machine differ substantially. Nevertheless, it turns out that "reasonable programs" do not throw up problems when run on the machine, and this therefore indicates that the semantic differences may in many cases be circumvented. The present series of papers is intended to explore this in

detail. Though some hints as to what may be expected have appeared before, eg. in [Banach and Watson (1989)], [Banach (1993)], the present series of papers is the thorough study cited in these two references.

The main areas in which the models of computation differ are: depth of pattern matching, granularity of atomic primitives, and concurrency. The language model favours big serial atomic actions, the machine model favours small concurrent atomic actions. Thus as well as being an essay in language semantics *per se*, the study of MONSTR includes the examination of issues of serialisability and atomicity in a somewhat unusual setting (that of language semantics). For a thorough presentation of serialisability and atomicity theory from a modern perspective and in its usual setting, see [Lynch et al. (1994)].

The main purpose of this first paper, is to set out the DACTL model that we will use, to prove that a packet store implementation of it is correct under suitable circumstances, to describe our version of the machine model in packet store terms, and then to see how the characteristics of the machine model are used as a guide in the design of the MONSTR sublanguage of DACTL. This is done in the sections which follow. Thus [Section 2] introduces the model informally, using simple examples. [Section 3] then gives an illustrated formal definition of the model. On the basis of some simple syntactic assumptions [Section 4] derives fundamental properties of DACTL systems that will be crucial later in the paper, and throughout the subsequent series. [Section 5] tackles garbage, a topic that has some rather strange repercussions in the DACTL framework. [Section 6] deals with the representability in a packet store model, of the graph rewriting model previously set out, using packet overwriting to model arc redirection.

The next few sections examine the impact of a distributed implementation on the packet store model. Thus [Section 7] deals with packet mobility in a distributed packet store; [Section 8] deals with packet copying and related issues; [Section 9] deals with redex size; while [Section 10] looks at atomicity, and sketches informally the low level execution model that results from the preceding considerations. At last [Section 11] pulls all of these insights together into the formal syntactic definition of the MONSTR sublanguage, and displays a couple of examples in full detail. [Section 12] looks ahead at some of the correctness issues that the preceding design throws up.

The historical roots of the present undertaking can be traced back to the keen interest in graph reduction architectures characteristic of the mid 80's. Many models broadly comparable to the Flagship Machine model were being explored and implemented at that time. We cite [Fasel and Keller (1986)], [FPLCA], [Woods (1986)] as containing representative collections of papers from that period. See also [Treleaven et al. (1982)]. The present study differs from these in two important respects. Firstly the present work is a self-contained theoretical enterprise (though obviously strongly influenced by considerations of practice), while most of the cited work has a strongly implementation-oriented flavour. Secondly, apart from the inspiration of the original intended applications, MONSTR is not application specific. This has both costs and benefits. The costs include an unavoidable increase in the complexity of the technical details that have to be kept under control. For example, the notion of packet, originating in [Darlington and Reeve (1981)], takes on a life of its own when coupled with considerations specific to MONSTR such as balancedness, state saturatedness [Section 4], and the firewall principle [Section 7]. The benefits include much wider applicability, as witnessed by the more recent citations from the 90's.

2 AN INTRODUCTION TO EXTENDED TERM GRAPH REWRITING AND DACTL

Term graph rewriting [Barendregt et al. (1987)], and then DACTL, grew out of the desire to place many of the low-level operational features found in implementations of declarative languages onto a more formal basis, especially in the context of implementations on parallel machines. This invevitably means dealing with graphs rather than terms, since implementations inevitably feature sharing of subexpressions for the sake of efficiency. So term rewriting becomes superseded by graph rewriting. The desire to keep as close as possible to reality, means that the nodes of the graph look as much like parts of a term as is reasonable. So term graphs as they were called, have nodes each of which is labelled with a symbol and has a sequence of out-arcs to other nodes in the graph (the node's children).

The dynamic feature of term rewriting, substitution, is handled by the elegant notion of arc redirection, whereby all the in-arcs pointing to a node may be redirected to point at some other node. Of course for this to make sense, the other node has to be in the graph already, and so when substitution by a completely new subgraph is desired, redirection has to be preceded by formally incorporating the new material into the existing graph: contractum building.

Redirection turns out to be a versatile concept, and can be applied to any node of the graph, not just roots of redexes. This generalises the term-graph rewriting model allowing many ideas from imperative programming to be smoothly expressed in the same formalism as is used for the declarative side. All in all we end up with a flexible methodology for describing computation, and this explains why DACTL turned out to be a natural choice as intermediate language for the Flagship machine.

Let us now examine a couple of examples. Since our concern in this series of papers is with equivalence of semantic models, we eschew discussion of the concrete syntax of DACTL in the main, and draw term graphs directly, effectively dealing with an abstract syntax. Nevertheless some conventions from DACTL's concrete syntax creep in to our pictures, eg. the caption of [Fig. 1.(a)], and in the examples in [Section 11].

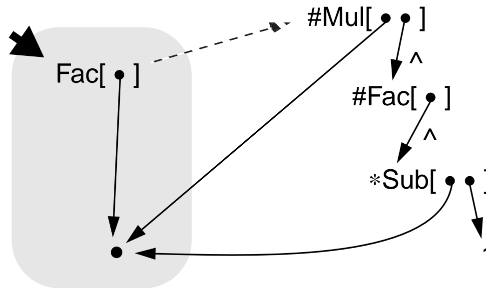

Fig. 1.(a) Rule for Fac[n] => #Mul[n ^#Fac[^∗Sub[n 1]]]

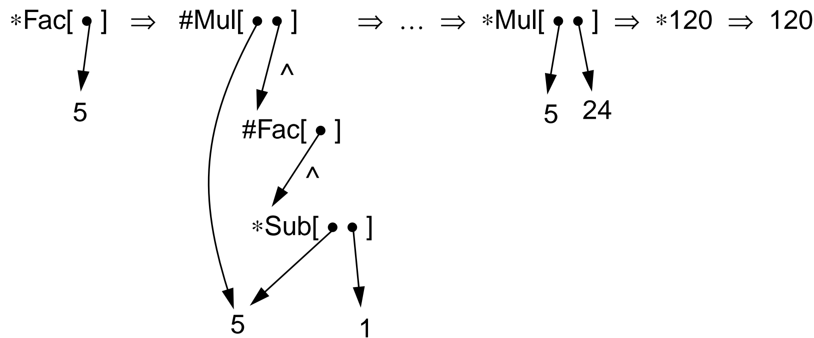

Fig. 1.(b) An evaluation of Fac[5]

[Fig. 1.(a)] illustrates the non base case rule for the factorial function. The shaded part is the left hand side, which is pattern matched to the graph being evaluated. Note that it is the part accessible from the node pointed to by the fat arrow. (After this section, we won't bother doing the shading in pictures of rules.) The non-shaded part is the contractum, a copy of which is glued to the redex when a match is found. The faint arrow represents the single redirection of this rule, which stipulates that in the redirection phase, in-arcs of Fac are to be redirected to the new Mul node.

The markings (#, ^, ∗), are DACTL's mechanism for controlling evaluation strategy. Briefly, a node marked with a number of #'s is suspended, and must wait for that number of "notifications" to arrive from those of its children to which it is connected along out-arcs marked with ^, the notification marking. Each time such a notification arrives (notifications are the other kind of atomic action besides rewrites), the ^ on the arc in question is deleted, and the number of #'s on the parent is decremented. When the latter reaches zero, the parent is marked with ∗, the active marking, and the parent is said to be active. An unmarked node or arc is said to be idle, (unmarked arcs are also called normal).

DACTL stipulates that only active nodes may be roots of redexes, and that selection of which active node to reduce next is nondeterministic. When an active node is selected for rewriting one of two things happens. If there is a rule, the root of whose LHS will match at the given active node, the active marking is removed from the root of the redex and rule execution proceeds as indicated above. If there is no such rule, the active marking is removed as in the other case, and notifications are sent up along all ^-marked in-arcs of the node. Normally the parents of these nodes at the tails of such in-arcs are waiting for these notifications by being #-marked. (In fact we will demand this later on in this paper.)

Note that the metaphor of sleeping while waiting for a sub-computation to complete, before being woken and continuing, is fairly universal in computing, being found in implementations of declarative languages, in the procedure call/return mechanism of im-

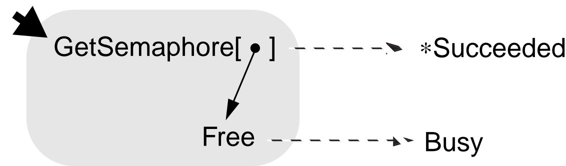

Fig. 2.(a) Rule for obtaining a free semaphore.

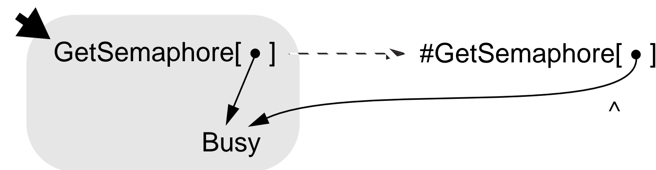

Fig. 2.(b) Rule for waiting for a busy semaphore.

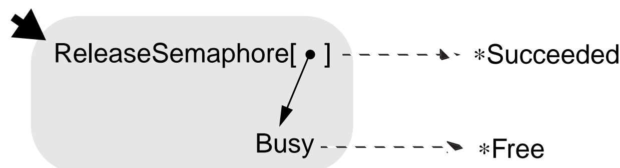

Fig. 2.(c) Rule for releasing a semaphore.

perative languages, and at higher levels of abstraction too. This at least partly explains DACTL's suitability for a wide variety of applications.

[Fig. 1.(b)] illustrates some of the stages of the evaluation of Fac[5], where for clarity, garbage is collected "as we go". In reality DACTL does not say anything at all about garbage, in the sense that no node in any computation step is ever destroyed, no matter how useless it has become. We will address the issue of garbage more comprehensively in [Section 5]. Aside from this perhaps surprising feature, DACTL provides mechanisms for rule selection, inbuilt operators for common data-types such as integers and booleans, a module discipline, interfacing between modules and to the outside world, and so on. We will not however pursue these matters except as they directly affect the definition of the MONSTR language below.

[Fig. 2] shows some basic rules for semaphore handling. Among other things this illustrates the utility of non-root redirection, particularly in imperative programming. [Fig. 2.(a)] shows how a free semaphore is claimed, and how the success of the operation is

communicated to the waiting parent. (It is assumed that symbol Succeeded has no rules.) In [Fig. 2.(b)], if the semaphore is Busy, the suspension mechanism using # and ^ markings achieves the required semantics. In [Fig. 2.(c)], the active Free symbol (also assumed to have no rules), will when it rewrites, notify any suspended GetSemaphores, which will then have an opportunity to claim the resource required.

3 A FORMAL DEFINITION OF DACTL REWRITING

Although the intuitions of the previous section are very useful in thinking about DACTL rewriting, a formal definition is ultimately necessary, and we provide one here. The version of DACTL we present is tuned to our future requirements, and differs in minor detail from the current language definition, but this is not of importance.

We assume we are given an alphabet **S** = {S, T, …} of node symbols. When we wish to refer to specific symbols we will write them like S, T. But when we speak about symbols in general in the meta-language we will use italics thus S, T.

Definition 3.1 A term graph (or just graph) G, is a quintuple $(N, \sigma, \alpha, \mu, \nu)$ where

(1) N is a set of nodes.

(2) σ is a map $N \to$ **S**, which labels each node.

(3) α is a map $N \to N^*$, which maps each node to its sequence of children.

(4) μ is a map $N \to \{\varepsilon, *, \#, \#\#, \#\#\#, \dots \#^n\ (n \geq 1)\}$, which maps each node to its node marking (idle, active, once, twice … n times suspended).

(5) ν is a map $N \to \{\varepsilon, \hat{}\ \}^*$, which maps each node to the sequence of arc markings on the arcs to its children (each either the normal or notification marking).

Clearly we must have for all $x \in N$, $\mathrm{dom}(\alpha(x)) = \mathrm{dom}(\nu(x))$, where the domain of a sequence is the set of its indices.

We write $A(x)$, the arity of a node x, for $\mathrm{dom}(\alpha(x)) = \mathrm{dom}(\nu(x))$. Note that $A(x)$ is a set of consecutive natural numbers starting at 1, or empty. When dealing with more than one graph (or pattern — see below), we subscript the objects defined in (1) – (3) above with the name of the graph in question to avoid ambiguity. Also we allow ourselves to write $x \in G$ instead of $x \in N(G)$ or $x \in N_G$ etc. Each child node c of some node p determines an arc of the graph, and we will refer to arcs using the notation (p_k, c) to indicate that c is the k'th child of p; i.e. that $c = \alpha(p)[k]$ for some $k \in A(p)$. The maps μ, ν are referred to as the markings and are mainly concerned with encoding execution strategies, while N, σ, α are referred to as the graph structure and provide the main information content of the graph.

For ease of use, the names are meant to be reasonably alliterative: σ for symbols, α for arcs, μ for markings, ν for notifications.

[Fig. 3] 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. Obviously term graphs are directed graphs. We use standard digraph terminology below where necessary without further comment; eg. path, semipath, and accessibility of one node from another. (Recall a semipath ignores the orientation of arcs.)

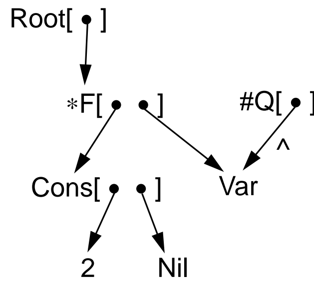

Fig. 3 A term graph.

For rewriting, we need a notion of pattern, and a sufficiently flexible notion of pattern matching. For this reason we introduce a symbol Any not in **S**.

Definition 3.2 A pattern is defined as in definition 3.1 except that the signature of σ is $N \rightarrow \mathbf{S} \cup \{\mathsf{Any}\}$, and Any-labelled nodes must satisfy

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

Any-labelled nodes are called implicit whereas other nodes are explicit. (Thus every graph is a pattern but not vice-versa.)

Patterns have a natural notion of homomorphism.

Definition 3.3 Let P, Q be patterns (and let P have a root r). A node map $h : P \rightarrow Q$ is a homomorphism to Q at $t \in Q$ iff ($h(r) = t$ and) for all explicit nodes $x \in P$

(1) $\sigma(x) = \sigma(h(x))$, i.e. h is label-preserving.

(2) $A(x) = A(h(x))$, i.e. h is arity-preserving.

(3) For all $k \in A(x)$, $h(\alpha(x)[k]) = \alpha(h(x))[k]$, i.e. h is order-preserving.

Suppose in addition the following hold:

(4) $\mu(x) = \mu(h(x))$, i.e. h is node-marking-preserving.

(5) For all $k \in A(x)$, $h(\nu(x)[k]) = \nu(h(x))[k]$, i.e. h is arc-marking-preserving.

In such a case we say that h preserves markings. (To emphasise the converse, we can call ordinary homomorphisms, graph structure homomorphisms.)

Omitting mention of roots, definition 3.3 serves just as well for graphs and unrooted paterns as it does for rooted patterns. Homomorphisms are also called matchings.

Definition 3.4 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 of nodes and arcs 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 *Red* satisfies the invariants (RED-1), (RED-2) and (RED-3) below:

(RED-1) *Red* is the graph (in the set theoretic sense) of a partial function.

(RED-2) $(a, b) \in Red \Rightarrow a$ is an explicit node of L.

(RED-3) $\{(a, b), (a', b')\} \subseteq Red$ and $a \neq a' \Rightarrow \sigma(a) \neq \sigma(a')$.

For $(a, b) \in Red$, a is called the LHS and b the RHS of the redirection.

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

[Fig. 4] 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 3.4.(2)). Note that we have labelled the implicit nodes with the Any symbol here in contrast to [Fig. 1].

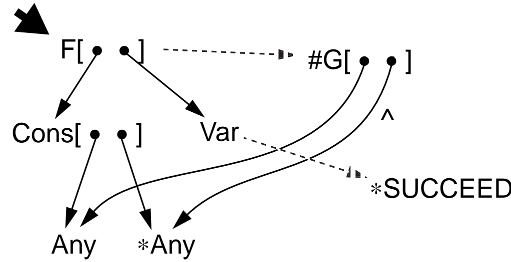

Fig. 4 A rule.

A DACTL system is (for us) just a set of rules. Rewriting proceeds via three stages: root selection, rule selection and rule execution.

Let G be a graph that is to be rewritten, and assume some system $\mathcal{R}$ understood.

Definition 3.5 Root selection is the nondeterministic choice of one of the active nodes of G to act as the root of the redex for rewriting. Call it t, the chosen root.

Definition 3.6 Let t be the chosen root in G. Let $Sel = \{D \mid$ there is a $D \in \mathcal{R}$ such that there is a matching $g : L \to G$ of the left pattern L of the full pattern P of the rule D to G at $t\}$. Rule selection is the nondeterministic choice of a member of *Sel* assuming it is non-empty. The chosen D makes t the root of the redex $g(L)$ and D the selected rule that governs the rewrite.

Evidently there is a matching of the left subpattern of the rule in [Fig. 4] to the graph in [Fig. 3], with a redex rooted at F, and this rule and graph will provide us with a running example.

Assuming we have a selected rule and redex, rule execution (or rewriting according to the rule governing the rewrite) proceeds in three phases: contractum building, redirection, and activation. Assume G, t, $D = (P, root, Red, Act)$ and g given as above.

Contractum building adds a copy of each contractum node of P to G. Node markings for such nodes are taken from P. Copies of arcs of P from contractum nodes to their children are added in such a way that there is a graph structure homomorphism (called the extended matching) $g' : P \to G'$ from the whole of P to the graph being created, which agrees with g on L. Arc markings are again taken from P.

Doing this for our running example yields [Fig. 5]. We see that copies of exactly the contractum nodes and arcs, suitably marked, have been added, and that this enables the extended matching g' of the whole of P to be constructed.

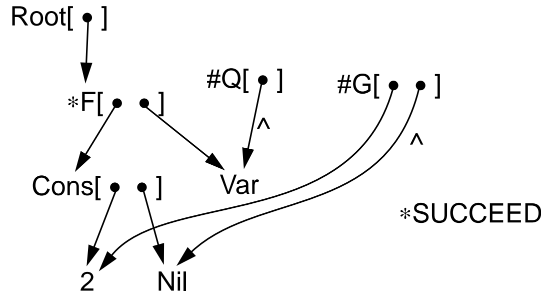

Fig. 5 Contractum Building.

More formally we have the following.

Definition 3.7 Assume the preceding notation. Let the graph G' be given by

(1) $N_{G'} = N_G \uplus (N_P - N_L)$ where $\uplus$ is disjoint union.

(2) $\sigma_{G'}(x) = \sigma_G(x)$ if $x \in G$,
$\sigma_{G'}(n) = \sigma_P(n)$ if $n \in P - L$.

(3) $\alpha_{G'}(x)[k] = \alpha_G(x)[k]$ if $x \in G$, for $k \in A(x)$,
$\alpha_{G'}(n)[k] = \alpha_P(n)[k]$ if both n and $\alpha_P(n)[k] \in P - L$, for $k \in A(n)$,
$g(\alpha_G(n)[k])$ if $n \in P - L$ and $\alpha_P(n)[k] \in L$, for $k \in A(n)$.

(4) $\mu_{G'}(x) = \mu_G(x)$ if $x \in G$,
$\mu_{G'}(n) = \mu_P(n)$ if $n \in P - L$.

(5) $\nu_{G'}(x)[k] = \nu_G(x)[k]$ if $x \in G$, for $k \in A(x)$,
$\nu_{G'}(n)[k] = \nu_P(n)[k]$ if $n \in P - L$, for $k \in A(n)$.

Redirection takes each arc (p_k, c) such that $c = g'(a)$ for some $(a, b) \in Red$ and replaces it with $(p_k, g'(b))$. This can be done consistently since the LHSs of two distinct redirections cannot map to the same node of G since (RED-3) means that their node symbols

are different and so the nodes cannot be identified in a matching. All such redirections are performed simultaneously.

Performing the redirections on our example yields [Fig. 6].

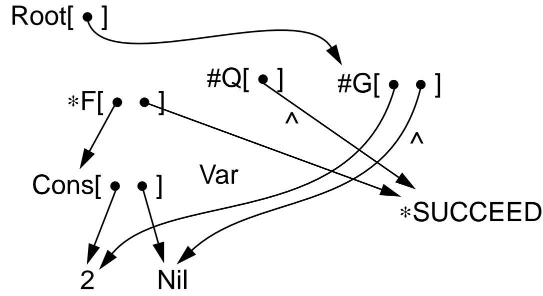

Fig. 6 Redirection.

More formally we have the following.

Definition 3.8 Assume the preceding notation. Let the graph G'' be given by

(1) $N_{G''} = N_{G'}$.

(2) $\sigma_{G''} = \sigma_{G'}$.

(3) $\alpha_{G''}(x)[k] = g'(b)$ if $(a, b) \in Red$ and $g'(a) = \alpha_{G'}(x)[k]$, for $k \in A(x)$,
$\alpha_{G'}(x)[k]$ for $k \in A(x)$, otherwise.

(4) $\mu_{G''} = \mu_{G'}$.

(5) $\nu_{G''} = \nu_{G'}$.

On G'' the map g' induces a map $g'' : P \to G''$ which is now just a node map, rather than a homomorphism.

Activation makes active the g''-images of idle nodes in Act, and also makes the g''-image of t idle, unless it was one of them itself. Doing this for our running example yields [Fig. 7]. More formally we have the following.

Definition 3.9 Assume the preceding notation. Let the graph H be given by

(1) $N_H = N_{G''}$.

(2) $\sigma_H = \sigma_{G''}$.

(3) $\alpha_H = \alpha_{G''}$.

(4) $\mu_H(x) =$ **If** $a \in Act$ and $g''(a) = x$ and $\mu_{G''}(x) = \varepsilon$ **Then** $*$
Else If $x = t$ and $t \notin g''(Act)$ **Then** ε
Else $\mu_{G''}(x)$.

(5) $\nu_H = \nu_{G''}$.

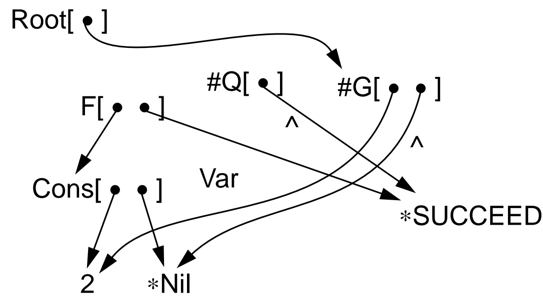

Fig. 7 Activation.

Note how root quiescence is effectively done last in this definition, in contrast to the informal account of [Section 2] where it was mentioned first. The net effect is the same of course, but definition 3.9 will prove more convenient later. On H, g'' induces a map $h : P \to H$ which is of course another node map.

Definition 3.10 The result of the rewrite of the redex $g : L \to G$ according to the rule $D = (P, root, Red, Act)$ is the graph H produced by applying definitions 3.7 – 3.9.

Given a chosen root, the above applies provided *Sel* is non-empty. If *Sel* is empty, then instead of a rewrite, notification takes place. Notification causes the chosen root to be quiesced as in a rewrite, all notification in-arcs to the chosen root to lose their notification marking, and parent nodes of such in arcs to have any non-zero suspension marking to be decremented.

In [Fig. 7], assuming there are no rules for Nil or SUCCEED, there is scope for two notifications. When they have both been performed, [Fig. 8] results.

More formally we have the following.

Definition 3.11 Let t be the chosen root in a graph G such that *Sel* is empty. Let the graph H be given by

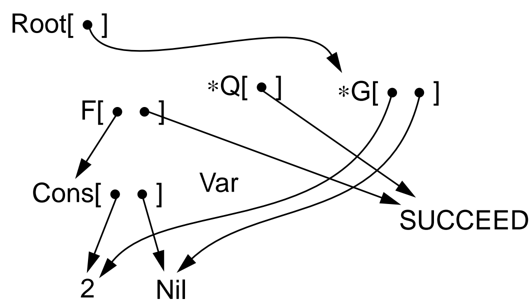

Fig. 8 Two notifications.

(1) $N_H = N_G$.

(2) $\sigma_H = \sigma_G$.

(3) $\alpha_H = \alpha_G$.

(4) $\mu_H(x) =$ **If** $\alpha_G(x)[k] = t$ and $\mu_G(x) = \#^n$ (with $n \geq 1$) and
$\nu_G(x)[k] = \wedge$ **Then** $\#^{n-1}$ (where $\#^0 = *$)
Else If $x = t$ **Then** ε
Else $\mu_G(x)$.

(5) $\nu_H(x)[k] =$ **If** $\alpha_G(x)[k] = t$ and $\nu_G(x)[k] = \wedge$ **Then** ε
Else $\nu_G(x)[k]$.

The result of the notification is the graph H.

Definition 3.12 Rewrites (done when *Sel* is non-empty) and notifications (done when *Sel* is empty) are called execution steps.

Definition 3.13 An initial graph is one which consists of an isolated node with empty arity, with the active node marking, and labeled by the symbol Initial.

Definition 3.14 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 by some execution step at some arbitrarily selected active node t_i of G_i. Graphs occuring in executions are called execution graphs.

Heavy use of induction over executions will be made in the remainder of this series of papers. It is not a great overestimate to say that it is the only technique that works in general for establishing properties of systems.

Remark. Our definitions so far concealed a little subtlety concerning disjoint unions. In constructive definitions of disjoint union, the members of such a union are tagged so that one can discern their origin. Our definition 3.7 omitted to do this. This is not normally a source of difficulty unless one is interested in “all possible” disjoint unions that can be built from some base sets. However a proper definition would take this into account. In this case a node x in G and its representative in G' after contractum building, are no longer the same thing, and there is a natural injection $i_{G,G'} : G \to G'$ that takes x to its representative in G'. Similarly one can introduce the notations $i_{G',G''} : G' \to G''$ and $i_{G'',H} : G'' \to H$ for the obvious injections which happen to be identities. Also of interest is the map $r_{G',G''} : G' \to G''$ which maps nodes to their targets under redirection in the second phase, and agrees with $i_{G',G''}$ on nodes that are unaffected by redirection (so $r_{G',G''}(x) = i_{G',G''}(x)$ unless $x = h'(a)$ for some $(a, b) \in$ *Red*, in which case $r_{G',G''}(x) = i_{G',G''}(h'(b))$).

By composing the various maps $i_{G,G'}$, $i_{G',G''}$ or $r_{G',G''}$, etc., we can track the history of a node through a rewrite. Thus $i_{G,H}(x) = (i_{G'',H} \circ i_{G',G''} \circ i_{G,G'})(x)$ is the node which is the copy in H of $x \in G$, and $r_{G,H}(x) = (i_{G'',H} \circ r_{G',G''} \circ i_{G,G'})(x)$ is the node of H that x got redirected to. In future we will often need to keep a close track of nodes through the phases of a rewrite, particularly when relevant properties of nodes change from one phase of a rewrite to another, so the above notation is a useful alternative to the g, g', g'', h maps already introduced in pinpointing bits of the rewriting process.

For uniformity, we introduce the same notation $i_{G,H} = r_{G,H}$ for the obvious injection on nodes in a notification step. Therefore, composing a sequence of $i_{G,H}$ maps or of $r_{G,H}$ maps, allows us to track the history of a node through an execution of the system. The former tracks a node's identity, and the latter tracks what a node "becomes" via redirection. Generically, any such composition will be called $i_{X,Y}$ or $r_{X,Y}$ where X and Y are the first and last graphs in the sequence. An arc (p_k, c) is evidently tracked by $(i_{X,Y}(p)_k, r_{X,Y}(c))$. A final property of these notations is that they are portable to situations in which one wishes to define operations on graphs "universally", i.e. up to isomorphism.

4 FUNDAMENTAL PROPERTIES — BALANCEDNESS AND STATE SATURATEDNESS

In this section we show that DACTL rewriting preserves two simple invariants which will prove crucial later on.

Definition 4.1 A node x in a graph G is balanced iff for $n \geq 1$,

$$\mu(x) = \#^n \iff |\{k \mid \nu(x)[k] = \wedge\}| = n$$

Thus a node is balanced if it is waiting for exactly the number of notifications it might ever receive. We say that a pattern or graph is balanced iff every node is balanced.

Theorem 4.2 Suppose in a DACTL system $\mathcal{R}$, in every rule $D = (P, \ldots)$, P is balanced. Then every execution graph of $\mathcal{R}$ is balanced.

Proof. By induction on executions. An initial graph is balanced. Furthermore, notifications preserve balancedness, since for each notification marking removed from an arc, a suspension marking is removed from the parent node. We check that the phases of a rewrite do not affect balancedness. Contractum building preserves balancedness, as all new nodes added to the graph are balanced by the assumption on the full patterns of rules. Redirection only affects the heads of some arcs, so preserves balancedness. Finally, activation and root quiescence only affect the node markings on non-suspended nodes, thus preserving balancedness. So we have the result by induction over executions. ☺

Let $\mathbf{V}$ be a subset of the symbols, called stateholders (or variables). For the moment, we impose no extra conditions on $\mathbf{V}$, but later we will do so.

Definition 4.3 An arc (p_k, c) of a graph G is state saturated iff

$$\nu(p)[k] = \wedge \text{ and } \mu(c) = \varepsilon \implies \sigma(c) \in \mathbf{V}$$

A node of a graph is state saturated iff all of its in-arcs are state saturated. Thus a node is state saturated if it cannot be both idle, and the target of a notification arc, without being labelled by a stateholder. Likewise, a graph or pattern is state saturated if all of its nodes and arcs are.

Theorem 4.4 Suppose in a DACTL system $\mathcal{R}$, for every rule $D = (P, root, Red, Act)$ we have:

(1) For all $x \in P$, x is state saturated, or $x \in Act$.

(2) $(root, b) \in Red$ for some $b \in P$,

(3) $(a, b) \in Red$ and $\mu(b) = \varepsilon \implies \sigma(b) \in \mathbf{V}$ or $b \in Act$.

Then every execution graph of R is state saturated.

Proof. By induction over executions. An initial graph is state saturated. A notification step clearly preserves state saturatedness, since for the only node which becomes idle all notification in-arcs become idle. We argue that rewrites preserve state saturatedness as follows.

Let G_i be rewritten to G_{i+1}, using a rule $D = (P, root, Red, Act)$, and a redex $g_i : L \to G_i$. Assume the usual notation for the pieces of a rewrite (eg. maps g_i, g_i', g_i'', and $g_{i+1} : P \to G_{i+1}$). Consider contractum building. It is easy to check that all new nodes introduced in G'_i are state saturated by (1) since $Act \subseteq L$. Obviously the nodes of $G_i' - g_i'(P)$ are state saturated since they continue to have (g_i' copies of) just the same arcs they had in G_i, and G_i is state saturated by the induction hypothesis. This leaves the nodes of $g_i'(L)$. Nodes in $g_i'(L) - g_i'(Act)$ are state saturated because any new arcs they acquired are state saturated by (1). This leaves a set of nodes $g_i'(\chi) \subseteq g_i'(Act) \subseteq g_i'(L) \subseteq G_i'$ which fail to be state saturated as they acquired a non-zero number of notification in-arcs during contractum building, but were idle, not **V**-labelled, and without notification in-arcs (and thus state saturated by default) in G_i. Therefore G_i' may fail to be state saturated, but just for this reason.

Now consider redirection and activation. All arcs of G_{i+1} are copies, or redirected copies of arcs of G_i'. We check that all arcs of G_i' end up state saturated in G_{i+1} which is sufficient. Leaving aside the phenomenon of root quiescence for the moment, there are three cases. Case (a): in-arcs of nodes in $g_i'(\chi) \subseteq g_i'(Act)$ which are not redirected. These are unchanged by redirection, and have their child nodes activated during activation, restoring state saturatedness to case (a) nodes. Case (b): in-arcs of nodes y which are redirected. Let (x_k, y) be a redirected arc, with $g_i'(a) = y$ and $(a, b) \in Red$. If $\sigma(b) \in$ **V** or $\mu(g_i'(b)) \neq \varepsilon$, then (the g_i'' copy of) $(x_k, g_i'(b))$ is state saturated. In case not, we know $b \in Act$ by (3) above. Thus the activation phase, making G_{i+1}, will make $\mu(g_{i+1}(b)) = *$ by definition 3.9.(4). This restores state saturatedness to all case (b) nodes. Case (c): in-arcs of all other nodes. These are state saturated in G_i' and do not suffer redirection. They remain state saturated throughout the redirection and activation phases.

Finally we return to the root, to deduce that root quiescence cannot destroy state saturatedness. If $g_i''(root)$ has any in-arcs, it must have been the target of a redirection and thus either $\mu(g_{i+1}(root)) = *$ or $\sigma(g_{i+1}(root)) \in$ **V** by (3). Alternatively if $g_i''(root)$ wasn't the target of any redirection, it hasn't any in-arcs by (2), and so is state saturated trivially. ☺

It is not hard to check that all our example rules so far have been both balanced and state saturated, provided we have {Busy, Var} $\subseteq$ **V**. (Later we will also want Free $\in$ **V**, for other reasons.)

Together, balancedness and state saturatedness serve to exercise control over dependencies of related subcomputations. Suspension chains must be mediated by suitably marked nodes and arcs, and must end, either in pending subcomputations, or in nodes labelled specially for the purpose by **V**. This will prove to be important later on.

5 GARBAGE AND STANDARD REDEXES

By now the reader will be keenly aware that no node is ever destroyed during an execution step. Therefore the graphs in an execution tend to accumulate more and more nodes which have long since ceased to be useful. These are garbage nodes, and any genuine implementation will need to collect such garbage if it is to be capable of anything other than trivial computations. Our treatment of garbage in this section is thus geared towards implementation models as discussed later in this paper.

In reality many different notions of liveness and garbage may be imposed on the abstract rewriting model described in [Section 3]. Here is the one we will use.

Definition 5.1 Let G be a graph, and x a node of G. Then x is live iff it can be proved so on the basis of the following rules of inference:

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

Definition 5.1 is nothing more than a proof system. (We will use an obvious sequent notation where appropriate below.) Thus by clauses (1) and (2), Root-labelled and active nodes form base cases of (axioms for) proofs of liveness; and liveness is propagated down normal arcs and up notification arcs (which makes clauses (3) and (4) into analogues of modus ponens). In the case of notification arcs, this squares well with the idea that nodes which are suspended waiting for some notification, should not be regarded as redundant.

Definition 5.2 Let G be a graph. The set of live nodes of G is denoted Live(G), and N_G – Live(G) is denoted Gar(G), the garbage set of G. An arc (p_k, c) of G is live iff both p and c are live; otherwise it is garbage. Note that the inference rules in definition 5.1.(3) and 5.1.(4) give "local" means of proving the liveness of any given live arc (p_k, c) even if both parent and child can be proved live without mentioning (p_k, c), as a byproduct of some connectivity properties of the graph G.

Definition 5.3 The live subgraph of a graph G, LSG(G), consists of the live nodes and live arcs of G.

Note that the live subgraph need not be a graph in the sense that it satisfies all the invariants implied by definition 3.1, since a live node may have a garbage notification outarc to a garbage child node. Live nodes may also have garbage normal in-arcs from garbage parent nodes, though this does not threaten the invariants of definition 3.1.

The most important thing about garbage is its persistence. Once a node of an execution graph is proclaimed garbage, no execution step should cause it to be capable of being proved live ever again. In this respect, our definition is lacking.

Counterexample 5.4 [Fig. 9] shows a case in which a garbage node is made live by rewriting. The redex is the whole of the illustrated G_i. The node A is thus garbage. The action of the rule is merely to create the Root node as another parent of A, and to redirect both F and G to Root. By parts (1) – (3) of definition 5.1, A becomes live in G_{i+1}. Thus definition 5.1 is not directly useful without additional conditions.

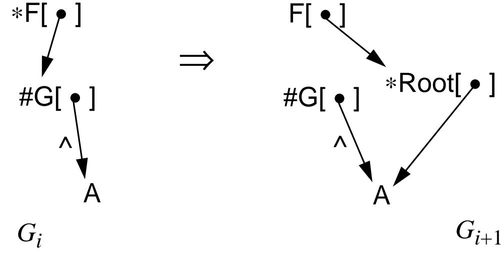

Fig. 9 A garbage node becomes live.

Definition 5.5 With the usual terminology, a matching $g : L \rightarrow G$ is a standard redex, iff the follwing hold.

(1) $g(root)$ is active.

(2) For all explicit nodes x of L other than *root*, $g(x)$ is idle.

(3) All arcs in $g(L)$ are normal.

Remark 5.6 For a matching to be a candidate for rewriting, definition 5.5.(1) must hold anyway, and in a balanced graph, a redex is standard iff definition 5.5.(1) and 5.5.(2) hold.

The next theorem says that garbage remains garbage after rewrites of standard redexes.

Theorem 5.7 Let $g : L \rightarrow G$ be a standard redex, rewritten to produce a graph H by a rule $D = (P, root, Red, Act)$. Then

(1) If x is a garbage node of G, then $i_{G,H}(x)$ is a garbage node of H.

(2) If (p_k, c) is a garbage arc of G, then $(i_{G,H}(p)_k, r_{G,H}(c))$ is a garbage arc of H.

Proof. It is easy to see that the whole of $g(L)$ for a standard redex is live. If q is a node not in $g(L)$, then if (q_k, z) is a notification arc with $z \in g(L)$ then q is live; similarly if (z_k, q) is a normal arc with $z \in g(L)$ then q is live. Each proof of liveness that depends on the liveness of $g(L)$ and proves the liveness of a node or arc outside of $g(L)$, must involve a step like one of these two cases, shown below in [Fig. 10]. We call these the redex-emergent steps, and the arcs involved, the redex-emergent arcs.

Consider the garbage node x in G. There is no proof of liveness of x in G so $x \notin g(L)$. After contractum building, all proofs of liveness in G remain valid after being mapped to G' because of the injection $i_{G,G'}$ which preserves markings. New proofs of liveness may have been created involving the contractum nodes, but none of them can prove $i_{G,G'}(x)$ live. For suppose not. To do so such a proof would have to follow a semipath from a contractum node to $i_{G,G'}(x)$. Since such a semipath must pass through $g'(L)$, we would have a redex-emergent step in the proof. Since all redex-emergent steps are unchanged from G, the final part of such a proof would correspond with the final part of a proof in G, and x would be live in G, a contradiction.

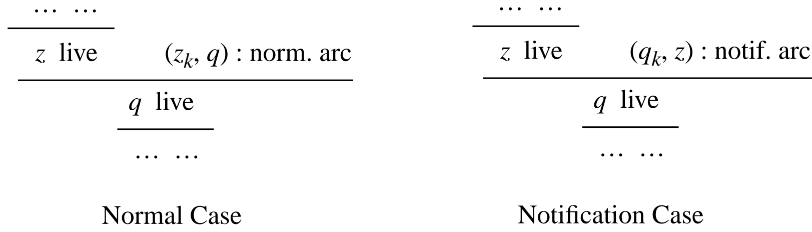

Fig. 10 Redex-emergent steps in proofs of liveness;
z is live because it is in the redex.

After redirection, all previous proofs not mentioning redex or contractum nodes remain unchanged, since for arcs not containing a redex or contractum node, $i_{G',G''}$ extends to a marking-preserving homomorphism. Call $g''(P)$ the extended redex for brevity. By the previous remark, any proof of liveness of (the $i_{G,G''}$ image of) x in G'' must involve an extended-redex-emergent step. Refering to [Fig. 10], there are two cases.

The Normal Case: Here we note that a normal arc (z_k, q) with z in the extended redex must have z in $g''(L)$ since redirection does not affect the tails of arcs. Thus the final part of the proof would correspond with the final part of a proof in G', and $i_{G,G'}(x)$ would be live in G', a contradiction.

The Notification Case: Here we note that a notification arc (q_k, z) in G'' is the $(i_{G',G''}, r_{G',G''})$ image of a notification arc (q_k, z^*) of G'. If z^* did not get redirected, then z^* is in $g'(L)$ as this is the only part of the extended redex in G' accessible from outside $g'(P)$. But then $g'(L)$ is live, and so the final part of the proof would correspond with the final part of a proof in G' and $i_{G,G'}(x)$ would be live in G'. If z^* did get redirected to z, then z^* is in $g'(L)$ since LHSs of all redirections are. Once more we would find a proof with a redex-emergent step involving (q_k, z^*), showing that $i_{G,G'}(x)$ was live in G'.

We conclude that the $i_{G,G''}$ image of x remains garbage in G''. (Note also that the redirection phase might have destroyed some proofs of liveness if there were non-root redirections, since the LHSs of such redirections frequently become inaccessible from the root of the redex.)

Finally the root quiescence and activation phase. If $h(root)$ is idle in H, some proofs of liveness that exist for G'' are destroyed, which cannot make the $i_{G,H}$ image of x live. If some nodes of $h(L)$ are activated, some new proofs of liveness in H without counterparts in G'' might be created. However, all such activated nodes are in $h(L)$. Thus any liveness proof utilising an extended-redex-emergent step involving such a node will have a final part for which a corresponding liveness proof in G' can be found, by tracing back through the redirection phase. So whatever nodes are made live this way, the $i_{G,H}$ image of x is not one of them. We conclude that x is garbage in H.

For a garbage arc (p_k, c), we argue that at least one of p or c is garbage and thus outside of $g(L)$ in G. By the preceding, its $i_{G,H}$ image is still garbage in H. If p is the garbage node, then $(i_{G,H}(p)_k, r_{G,H}(c))$ is obviously garbage. If c is the garbage node, then be-

cause c is outside of $g(L)$, $r_{G,H}(c) = i_{G,H}(c)$, the latter of which is garbage in H, giving the conclusion. We are done. ☺

So standard redexes behave well with respect to garbage preservation. Further we have the following result.

Theorem 5.8 Let G be a graph and t an active node of G. Suppose a notification step is performed at t to produce graph H. Then

(1) If x is a garbage node of G, then $i_{G,H}(x)$ is a garbage node of H.

(2) If (p_k, c) is a garbage arc of G, then $(i_{G,H}(p)_k, r_{G,H}(c))$ is a garbage arc of H.

Proof. For a notification from t, let the notification redex consists of all notification arcs (z_l, t) and their constituent nodes, since these are the only nodes and arcs manipulated by the notification. The redex-emergent arcs are therefore normal arcs (t_m, q), normal arcs (z_m, q), and notification arcs (q_m, z). All such arcs are live in G since t is active in G.

We recall that for notifications, $r_{G,H} = i_{G,H}$. In H, the $i_{G,H}$ image of t is idle, and the $i_{G,H}$ images of all arcs (z_l, t) are normal. The $i_{G,H}$ images of redex-emergent arcs (z_m, q) and (q_m, z) are live iff their corresponding z is live, and the $i_{G,H}$ images of redex-emergent arcs (t_m, q) are live iff t is live.

For a given z, unless in G, $\mu(z) = \#$ or $\mu(z) = *$, whence in H, $\mu(i_{G,H}(z)) = *$, $i_{G,H}(z)$ will not be live unless there is an alternative "nonlocal" proof of this, not involving the marking on $i_{G,H}(z)$ or the $i_{G,H}$ image of the arc (z_l, t). Similarly for t.

Notification may therefore destroy proofs of liveness of the redex-emergent arcs, and thus proofs of liveness of other nodes, which depend on the liveness of the notification redex in G. Since the $i_{G,H}$ images of no redex-emergent arcs are live that were not so before, no node may be proved live that could not be proved so before, so notification may create but not destroy garbage, and if x, or (p_k, c) was garbage in G, its $i_{G,H}$ image is garbage in H. ☺

Theorem 5.9 Executions whose steps consist only of rewrites of standard redexes, and of notifications, preserve garbage.

We see therefore the importance of standard redexes. The semantic models we will consider later in this investigation, are such that only standard redexes are ever rewritten.

The fact that garbage is regarded as computationally irrelevant means that we can construct an equivalence relation on graphs, such that G_1 and G_2 are equivalent iff their live subgraphs are isomorphic. This gives us one of many possible notions of equivalence for finite executions; viz. $[G_0, G_1 \ldots]$ and $[G'_0, G'_1 \ldots]$ are equivalent provided their final graphs have isomorphic live subgraphs. This is a fairly coarse equivalence, and many finer ones, based on various notions of bisimulation, are of course possible. We will not pursue these further at this point.

Returning to the running example of [Section 3], we see that [Fig. 3] contains no garbage, but that rewriting creates some garbage by the time we reach [Fig. 7]. Removing the garbage (which in fact does not cause any loss of conformance to definition 3.1), results in [Fig. 11]. We can also easily check that the redex in the running example was indeed a standard redex.

We end this section with a couple of lemmas that will be useful in the next section.

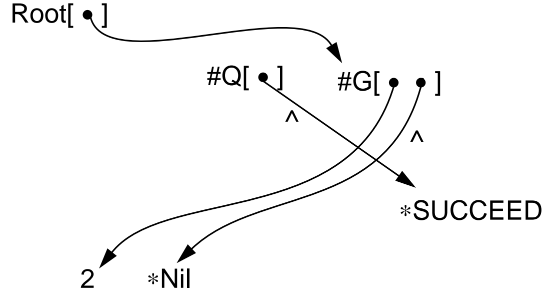

Fig. 11 Garbage collection in the running example.

Lemma 5.10 (Overwriting lemma) Let G be a balanced graph, and let $g : L \to G$ be a standard redex for a rule $D = (P, root, Red, Act)$. Denoting the result of contractum building using a prime, as usual, let

$$Red_g = \{(g'(a), g'(b)) \mid (a, b) \in Red, g'(a) \neq g'(b)\}$$

Let $(g'(a), g'(b)) \in Red_g$ and suppose

(1) $\sigma(a) \neq$ Root.

(2) There is no $(g'(c), g'(d)) \in Red_g$ with $g'(d) = g'(a)$.

(3) $g'(a) \notin g'(Act)$.

Then $h(a)$ is garbage in the graph H produced by the rewrite.

Proof. By (1), $\sigma(g'(a)) \neq$ Root, whence $h(a)$ in H cannot be proved live by clause (1) of definition 5.1. Since $g'(a)$ is redirected, a is explicit whence $g'(a)$ is either idle or the root of the redex in G' since the redex is standard. Thus by (3), $\mu(h(a)) \neq *$ in H whence $h(a)$ cannot be proved live by clause (2) of definition 5.1. Because by (2), $g'(a)$ is the LHS but not the RHS of a redirection, $h(a)$ has no in-arcs (normal or otherwise) in H, hence cannot be proved live by clause (3) of definition 5.1. Finally, since a is explicit, the redex is standard, and G is balanced, $g(a)$ is not suspended in G, so $g(a)$ has no notification out-arcs in G, hence $h(a)$ has none in H and $h(a)$ cannot be proved live by clause (4) of definition 5.1. ☺

Lemma 5.11 (Moving lemma) Let G be a balanced graph, and let $g : L \to G$ be a standard redex for a rule $D = (P, root, Red, Act)$. Denoting the result of contractum building using a prime, as usual, let Red_g be as in lemma 5.10 and let $(g'(a), g'(b)) \in Red_g$ satisfy

(2) There is no $(g'(c), g'(d)) \in Red_g$ with $g'(d) = g'(a)$.

Then $h(a)$ has no in-arcs in H.

Proof. Part and parcel of the previous proof. ☺

6 ABSTRACT PACKET STORE REWRITING, AND OVERWRITING

In the previous sections, we discussed some properties of our graph rewriting model at what could be called an abstract syntax level. In this section we move closer to a machine representation, though still a representation more suited to a serial execution model than the distributed store model we are ultimately interested in.

Definition 6.1 We assume given a store Σ consisting of store locations ranged over by l etc. Each store location l may contain a packet. A packet at location l may be regarded as a data structure, containing

(1) a node marking $\mu(l)$, as for graph nodes,

(2) a symbol $\sigma(l)$, taken from **S**,

(3) a sequence of items $\alpha(l)$, each item being either a store location l', or BLANK,

(4) a set of return addresses $\rho(l)$, each return address (or reversed pointer), being a pair consisting of a store location l' together with an index k into the sequence of items $\alpha(l')$, and written $l'.k$.

The correspondence between term graphs G, and packet store representations $\Pi(G)$ is straightforward enough. In words, node markings and symbols have direct analogues in packets. Normal arcs correspond to items of a packet, or forward pointers, pointing at other store locations; while notification arcs correspond to reversed pointers pointing at a BLANK item in the parent packet, and residing in the return address set of the child packet. The essential difference between the graph and packet worlds is thus the treatment of notification arcs. Here is the formal statement.

Definition 6.2 A packet store contains a simple representation of a graph $G = (N, \sigma, \alpha, \mu, \nu)$ iff there is an injective map $\pi : N_G \to \Sigma : x \mapsto \pi(x)$ such that there is a packet at each location in $\mathrm{rng}(\pi)$ such that

(1) $\mu(\pi(x)) = \mu(x)$.

(2) $\sigma(\pi(x)) = \sigma(x)$.

(3) For all $k \in A(x)$,
$$\alpha(\pi(x))[k] = \textbf{If } \nu(x)[k] = \varepsilon \textbf{ Then } \pi(\alpha(x)[k]) \textbf{ Else } \text{BLANK}$$

(4) $l'.k \in \rho(\pi(x)) \Leftrightarrow$ there is a $z \in N$ such that $\pi(z) = l'$, $\alpha(z)[k] = x$, $\nu(z)[k] = \wedge$.

To enable the smooth implementation of rewriting below, we introduce a new node symbol Ind, not in **S**, (extending the signature of σ above to $\mathbf{S} \cup \{\mathsf{Ind}\}$), and use it as the label for an indirection packet, transparent (when idle) to any pointer tracing mechanisms in the packet store. In this section, Ind packets will always be idle, and have empty return address sets. Chains of such Ind packets, even chains that merge, are perfectly permissible provided they do not loop. We do not write down the obvious invariant for this. To fully dereference an Ind chain we introduce the notation α^+ defined by

$$\alpha^+(l)[k] = \textbf{If } \sigma(\alpha(l)[k]) = \mathsf{Ind} \textbf{ Then } \alpha^+(\alpha(l)[k])[1] \textbf{ Else } \alpha(l)[k]$$

which is well founded because there are no Ind loops. When representations of graphs might contain such Ind packets, we no longer call them simple.

[Fig. 12] illustrates a graph and a non-simple packet store representation of it. Note that we draw the return address set near the front of the packet, so that it occurs in a fixed place in each packet, for convenience.

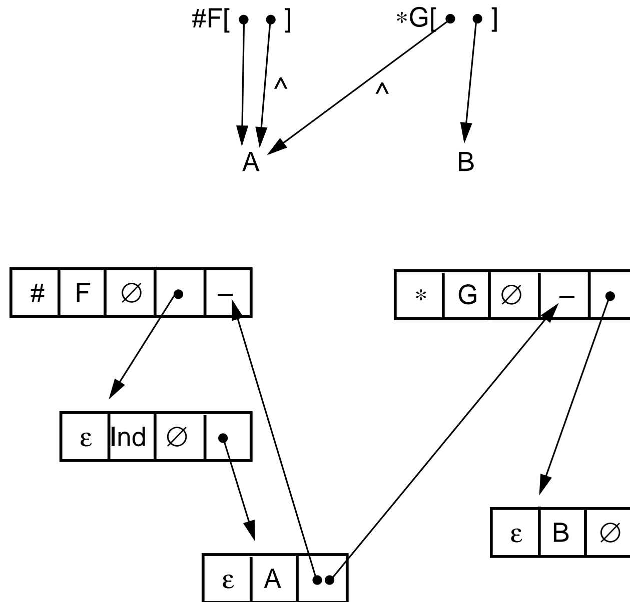

Fig. 12 A graph and a non-simple packet store representation.

Note that the reversed pointer representation of notification arcs in the packet store is ideally suited to the needs of a mark-scan garbage collector designed to implement the notion of liveness in definition 5.1. (The mark phase of such a collector is of course nothing more than the animation of an adequately large set of possible proofs of liveness in an execution graph. More specifically, the packet store structure explains why we do not deduce liveness of notification arcs from parent to child rather than the converse (or both); a packet store collector would have no means of locating such a child from the parent.)

If garbage is to be collected in the packet store, we need only represent the live subgraph of any particular graph. Since our implementation of rewriting in a packet store is intended to reuse certain garbage packets when it is safe to do so, we will adhere to this view subsequently. A notion of representation that is restricted to the live subgraph might thus go as follows.

Definition 6.3 A packet store contains a live representation of a graph $G = (N, \sigma, \alpha, \mu, \nu)$ iff there is an injective map $\pi : N_{\mathrm{LSG}(G)} \to \Sigma : x \mapsto \pi(x)$ such that there is a packet at each location in $\mathrm{rng}(\pi)$ such that

(1) $\mu(\pi(x)) = \mu(x)$.

(2) $\sigma(\pi(x)) = \sigma(x)$.

(3) For all $k \in A(x)$,

$$\alpha^{+}(\pi(x))[k] = \ \textbf{If}\ \nu(x)[k] = \varepsilon\ \textbf{Then}\ \pi(\alpha(x)[k])\ \textbf{Else}\ \mathrm{BLANK}$$

(4) $l'.k \in \rho(\pi(x)) \Leftrightarrow$ there is a $z \in N$ such that $\pi(z) = l'$, $\alpha(z)[k] = x$, $\nu(z)[k] = \wedge$.

Note that this is almost identical to definition 6.2. The admittance of Ind chains is revealed by the + superscript in clause (3), (omitting this superscript gives the definition of simple live representations). Also the correspondence works well because proofs of liveness follow normal arcs (forward pointers) from parent to child, and notification arcs (reversed pointers) from child to parent, as already noted. Obviously, possibilities between definitions 6.2 and 6.3, corresponding to choosing various subsets of $\mathrm{Gar}(G)$ to be represented along with $\mathrm{LSG}(G)$, exist.

Our translation of rewriting will attempt to manipulate only packets belonging to the redex itself, to avoid having to alter distant parts of the packet store, perhaps not easily located. It will also reuse as many redex packets as possible to minimise the creation of garbage. Of course such a strategy cannot yield a complete absence of garbage in the packet store, as knowing exactly which redex packets become garbage during a rewrite is impossible without a global search of the store; after all, a given redex packet may have many or no live pointers (perhaps reversed) to it from outside the redex. Thus a genuine implementation will need to incorporate a garbage collector, but this is outside the scope of papers in this series.

Avoiding global searches of the packet store during rewriting, or avoiding the elaborate caching mechanisms that might obviate the need for such searches, will be thus be suboptimal. However, we can regard the implementation of notification arcs by reversed pointers, as a caching mechanism designed for the benefit of notification steps in definition 6.4 below if we wish.

To ensure that we only need to manipulate the redex packets, we will need to restrict to the rewriting of standard redexes in balanced graphs, and insights such as the overwriting lemma 5.10, will prove invaluable. Of course the fact that a redex is standard and balanced, means that all of it can be located straightforwardly by following forward pointers from the root, an important aspect for a genuine implementation.

As a matter of notation we distinguish the various maps of different packet store representations by subscripting in the obvious way, eg. $\pi_G : N_G \to \Sigma$ and $\pi_H : N_H \to \Sigma$, for representations of a graph G and (say) the result of a rewrite H.

We warm up with the much simpler construction for notifications. Here all we need to do is reverse the return addresses in the notifying node, and change some packet markings.

Definition 6.4 Let G be a graph and $t \in G$ be an active node where notification is performed yielding graph H. Let $\Pi(G)$ contain a packet store representation of at least LSG(G). Define $\Pi(H)$ as follows.

$\Pi(H)$ is as $\Pi(G)$ (in particular insofar as $\pi_G : N_G \to \Sigma = \pi_H : N_H \to \Sigma$, allowing us to drop the subscripts on π), except that:

(1) $\mu_H(\pi(t)) = \varepsilon$.

(2) $\rho_H(\pi(t)) = \varnothing$.

(3) For all $l.k \in \rho_G(\pi(t))$,

(a) $\mu_H(l) =$ **If** $\mu_G(l) = \#$ **Then** $*$
Else If $\mu_G(l) = \#^n$ (with $n \geq 1$) **Then** $\#^{n-1}$
Else $\mu_G(l)$

(b) $\alpha_H(l)[k] = \pi(t)$

Theorem 6.5 Definition 6.4 provides a correct implementation of notifications in that if $\Pi(G)$ contains a packet store representation of LSG(G), then $\Pi(H)$ contains a packet store representation of LSG(H).

Proof. Fairly self-evident given the simple nature of notifications. ☺

Before we give the construction for rewriting, some descriptive motivation is in order.

We wish to avoid having to change distant parts of the packet store representation of a graph during rewriting. Therefore, where a pointer of some sort (either forwards or reversed), points at a location in the store, and the node the location represents is to be redirected, it is preferable that the packet representing the RHS of the redirection be made to overwrite the packet representing the LHS. In general this is impossible as the following non-balanced example shows.

Example 6.6 (Bottom Avoiding Merge) Consider the rules shown in [Fig. 13] within a packet store representation. From an examination of the contractum part of these rules, we see that typically, a BAM node is created singly suspended on a pair of notification arcs. Thus as soon as either of these arcs notifies, the BAM node is active and can rewrite, despite having another out-arc which is a notification arc, and which in principle could notify at any moment. In general, the BAM nodes will also have normal in-arcs from other parts of the graph, which wish to consume the lists being nondeterministically merged.

In a packet store representation, a BAM packet will thus have both forwards and reversed pointers pointing at it. Consider a BAM rewrite. In order that distant forwards pointers directed at the LHS BAM node do not need to be altered, the contractum Cons packet, to which the LHS BAM packet is redirected, should overwrite the LHS BAM packet. However, in order that distant reversed pointers directed at the LHS BAM do not need to be altered, the contractum BAM packet, to which they should point after the rewrite, should overwrite the LHS BAM. Clearly we can't have both of these possibilities. Restricting to standard redexes in balanced graphs avoids these problems, as there are never any reversed pointers targeted at any redirectable node (i.e. LHS of a redirec-

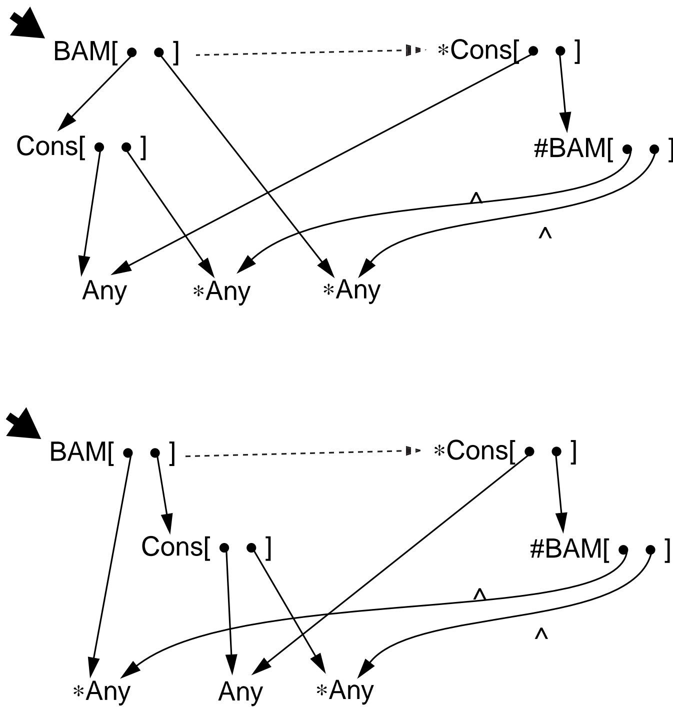

Fig. 13 Bottom Avoiding Merge.

tion). This opens the door to definition 6.7. (How to do bottom avoiding merges under such restrictions will be described later.)

First some more preamble. Consider the redirections *Red*, of a rule $D = (P, root, Red, Act)$. Given a node $a \in L$, either a is redirected (i.e. $(a, b) \in Red$ for some b) or not. If so, then b is either redirected or not, and so on. In general starting with a, we generate a maximal chain of redirections. If this chain is finite, it either ends with a contractum node, or with a LHS node which is not itself redirected. If the chain is infinite, we have a cycle of redirections (assuming P is finite). Examples are given in [Fig. 14.(a1) – (c1)] of each of the cases, showing the structure of a sample set of redirections in *Red*. Contractum nodes are shown open while LHS nodes are shown filled in; and an arrow represents a redirection, eg. the node represented by the packet A at store location 1, is to be redirected to the node represented by packet B at store location 2.

Assume the redex is standard and balanced and for simplicity that the matching is injective. Thus we know that there are no reversed pointers pointing at any of the LHSs of redirections, as all arcs of the redex are normal.

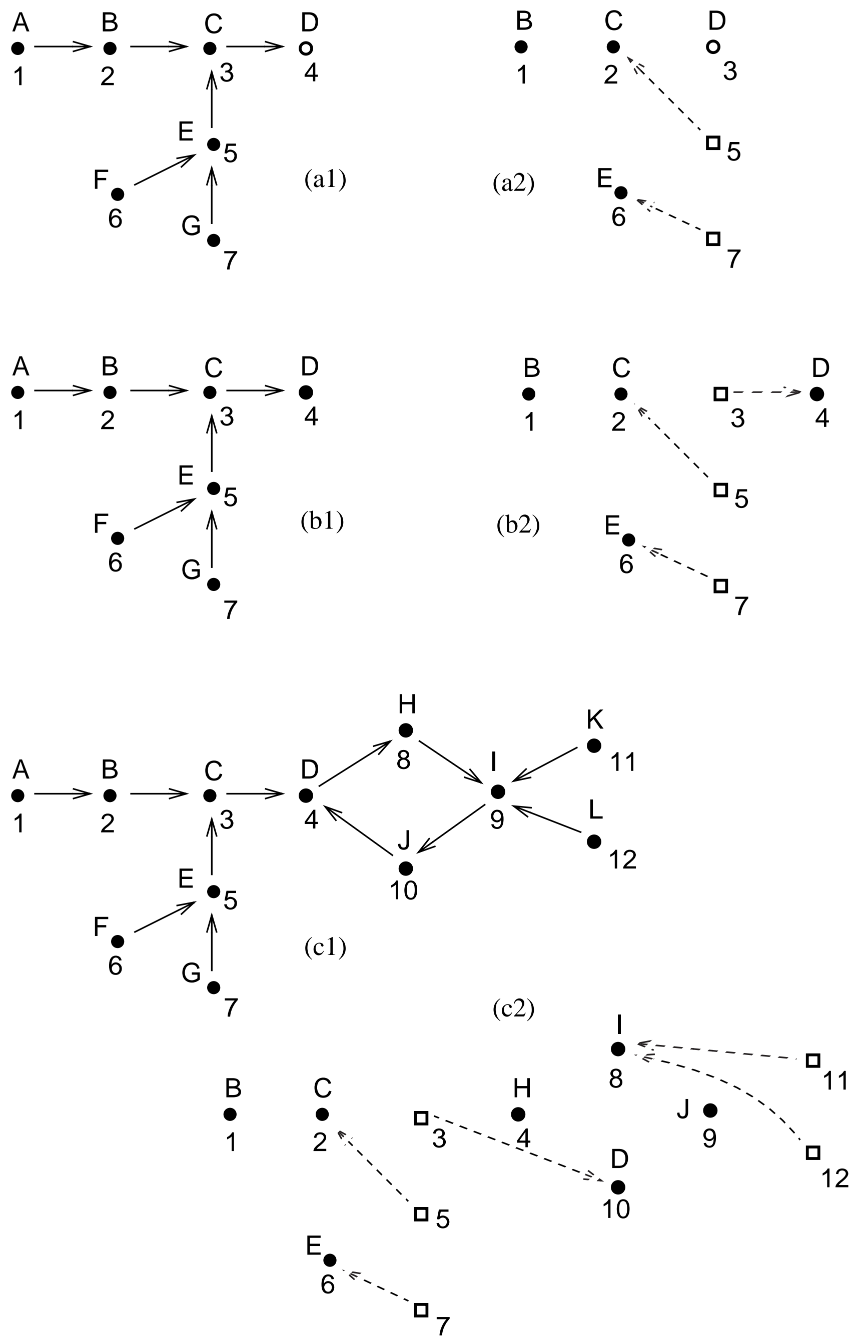

Fig. 14 Some multiple redirections: (a1) – (c1), before; (a2) – (c2), after.

Consider Fig. [14.(a1)]. To redirect C to D, it is sufficient to overwrite location 3 with D right away instead of allocating location 4 to D. To redirect A to B, it is likewise sufficient to overwrite location 1 with B, given that B is itself to be redirected. Since both B and E are to be redirected to C, it is not clear which of locations 2 or 5 should be overwritten with C. Doing both is not a good idea as it destroys the injectivity of the representation of (live sub)graphs. To overcome this we overwrite location 2 with C, and location 5 with an Ind packet, pointing at location 2. Forward pointers pointing at 5 will be deflected to 2 where they will find C as required. Redirecting F and G to E is similar. Note that overwriting 7, 6, 1 with Ind, E, B respectively, relies crucially on the overwriting lemma 5.10. (We assume that all the hypotheses for 5.10 are in fact satisfied).

We remark that this takes care of forward pointers pointing at A, B, etc. but not of reversed pointers in A, B etc. pointing at other nodes. If X is redirected to Y, the reversed pointers of X must be accumulated into the return address set of Y. Similarly when several nodes are redirected to Y, all of their reversed pointers must be accumulated in Y. If Y is simultaneously to be redirected to Z, Y's original reversed pointers must be moved to Z, before Y acquires X's reversed pointers.

Now consider [Fig. 14.(b1)] and the redirection of C to D. Both C and D are existing packets and may have forwards pointers directed at them. There are two possibilities. One could overwrite 4 with an Ind pointing at 3 and overwrite 3 with D; or one could overwrite 3 with an Ind pointing at D, given that C is to overwrite 2. The latter of these has one crucial advantage. D may be a packet representing a node matched to an implicit node of the left subpattern of the rule. Even in a standard redex of a balanced graph such a packet could be suspended and have reversed pointers targeted at it. So we want to leave D where it is. The rest of [Fig. 14.(b1)] can be dealt with similarly to [Fig. 14.(a1)], including the accumulation of reversed pointers, which all end up at the representative of the RHS of the redirection.

[Fig. 14.(c1)] involves a cycle of mutually targeted redirections. To deal with this, we move the packets in the cycle back one place in the cycle; thus D is to be found at 10, J at 9 etc. Ind packets must be targeted at the new locations of D and I from locations 3 and 11, 12 respectively, to handle redirections to D and I from outside the cycle. Reversed pointers are handled in the usual manner.

In [Fig. 14.(a2) – (c2)], we show the result of these implementation decisions. A square with a dashed out-arc represents an Ind packet, and the reversed pointer manipulations are not shown.

The above assumed that the redex was injective. If not, we must take care not to overwrite a location with an Ind pointing (directly or via other Inds) to itself, as this would yield a packet store which did not represent any (live sub)graph. The above arguments and insights must thus be applied to the set $\{(g'(a), g'(b)) \mid (a, b) \in \mathit{Red}, g'(a) \neq g'(b)\}$ as in lemma 5.10.

Going further, we notice that if a node is the LHS but not the RHS of a redirection, but that it is not garbaged by virtue of being either activated, or being Root-labelled, then in the result of a rewrite of a standard redex, it gets left with no in-arcs by the moving lemma 5.11. We can exploit this in the packet store in the presence of overwriting, by moving the corresponding packet before it gets overwritten; i.e. we allocate a fresh lo-

cation for it and copy the packet contents (except for the return address set), allowing its original location to be reused as if the packet were indeed garbaged.

The following unusually long definition describes a packet store transformation that is intended to implement a rewrite taking on board all of our preceding insights.

Definition 6.7 Begin Let $g : L \rightarrow G$ be a standard redex in a balanced graph G, of a rule $D = (P, root, Red, Act)$. Using a prime to indicate the result of the contractum building stage of a rewrite as usual, let

$$Red_g = \{(g'(a), g'(b)) \mid (a, b) \in Red,\ g'(a) \neq g'(b)\},$$
$$Act_g = \{g'(a) \mid a \in Act\},$$
$$Rem_g = \{g'(a) \mid g'(a) \in L_g, \text{ and there is no } (g'(c), g'(d)) \in Red_g \text{ with } g'(d) = g'(a), \text{ and either } \sigma(a) = \mathsf{Root} \text{ or } g'(a) \in Act_g\}.$$

Let $\Pi(G)$ be a packet store representation of at least the live subgraph of G.

We will give the rather imperative-looking description of the transormation in two phases. Phase 1 will deal with contractum building and redirection, and will be split into three subphases. Phase 2 will deal with root quiescence and activation. Phase 1 will generate the packet stores $\Pi_{1.1}$, $\Pi_{1.2}$, $\Pi_{1.3}$, while Phase 2 will generate the packet store Π_2.

Phase 1.

Informally, the three subphases of Phase 1 are as follows. In the first, we allocate locations to all contractum nodes which are not the RHS of a redirection, but leave the packets at those locations partly incomplete; also we allocate new locations for nodes in Rem_g, which cannot be garbaged, and copy the relevant packet contents. In the second, we deal with all nodes involved in redirection, again leaving some packets incomplete. In the third, we complete the incomplete packets left over from the first two subphases.

Subphase 1. For each contractum node p say, of $g'(P)$ which is not the RHS of a redirection in Red_g, choose a distinct unused store location l say, and create a packet there. The packet marking and symbol are taken from the node p. The sequence of items will be filled in later, and the return address set is initialised to $\varnothing$. Also for each node in Rem_g, choose a fresh distinct unused store location l say, and create a packet there whose contents are identical to the existing packet which represents it in the representation $\Pi(G)$, apart from the return address set, which is assigned to $\varnothing$.

Subphase 2. Define a redirection cluster to be a pair $(\{x_1 \ldots x_n\}, y)$ where $(x_1, y) \ldots (x_n, y) \in Red_g$, and there is no other redirection in Red_g with RHS y. The set $\{x_1 \ldots x_n\}$ is called the LHS set of the cluster and y is called the RHS.

All redirection clusters are handled similarly and simultaneously, and there are four cases depending on the nature of y.

Case (1): y is a node in a cycle of redirections (eg. D in [Fig. 14.(c1)]). Then x_1 (say) is also in the cycle. If x_1 is represented by packet X_1, at $l_1, \ldots, x_n$ by packet X_n at l_n, and y by Y at m, then the marking, symbol and item list from Y are moved to l_1, and all the return address sets from $X_1 \ldots X_n$ are amalgamated with the return address set of X_1 at l_1. Locations $l_2 \ldots l_n$ are overwritten by Ind packets pointing to l_1. In symbols, the changes from $\Pi_{1.1}$ are given by:

(a) $\mu_{1.2}(l_1) = \mu_{1.1}(m)$,

(b) $\sigma_{1.2}(l_1) = \sigma_{1.1}(m)$,

(c) $\alpha_{1.2}(l_1) = \alpha_{1.1}(m)$,

(d) $\rho_{1.2}(l_1) = \bigcup_{i=1\ldots n} \rho_{1.1}(l_i)$,

(e) For $i = 2 \ldots n$: $\mu_{1.2}(l_i) = \varepsilon$, $\sigma_{1.2}(l_i) = \mathsf{Ind}$, $\alpha_{1.2}(l_i) = [l_1]$, $\rho_{1.2}(l_i) = \varnothing$.

Case (2): y is a node of $g'(L)$ which is itself to be redirected (eg. B in [Fig. 14.(b1)]), but which is not in a cycle of redirections. The transformation is as for case (1) except that none of the x_i's are in a cycle either.

Case (3): y is a node of $g'(L)$ which is not itself redirected (eg. D in [Fig. 14.(b1)]). In this case we overwrite $l_1 \ldots l_n$ with Inds pointing to m, where Y is to remain. The return address sets from $l_1 \ldots l_n$ are accumulated into Y. In symbols, the changes from $\Pi_{1.1}$ are given by:

(a) $\mu_{1.2}(m) = \mu_{1.1}(m)$,

(b) $\sigma_{1.2}(m) = \sigma_{1.1}(m)$,

(c) $\alpha_{1.2}(m) = \alpha_{1.1}(m)$,

(d) $\rho_{1.2}(m) = \rho_{1.2}(m) \cup \bigcup_{i=1\ldots n} \rho_{1.1}(l_i)$,

(e) For $i = 1 \ldots n$: $\mu_{1.2}(l_i) = \varepsilon$, $\sigma_{1.2}(l_i) = \mathsf{Ind}$, $\alpha_{1.2}(l_i) = [m]$, $\rho_{1.2}(l_i) = \varnothing$.

Case (4): y is a contractum node (eg. D in [Fig. 14.(a1)]). In this case we overwrite l_1 (say) with a packet representing y. The packet marking and symbol are taken from the node y. The sequence of items will be filled in later (although this could at a pinch be done now). The return address set is the amalgamation of the return address sets from $l_1 \ldots l_n$. Locations $l_2 \ldots l_n$ are overwritten by Ind packets pointing to l_1. In symbols, the changes from $\Pi_{1.1}$ are given by:

(a) $\mu_{1.2}(l_1) = \mu(y)$,

(b) $\sigma_{1.2}(l_1) = \sigma(y)$,

(c) $\rho_{1.2}(l_1) = \bigcup_{i=1\ldots n} \rho_{1.1}(l_i)$,

(d) For $i = 2 \ldots n$: $\mu_{1.2}(l_i) = \varepsilon$, $\sigma_{1.2}(l_i) = \mathsf{Ind}$, $\alpha_{1.2}(l_i) = [l_1]$, $\rho_{1.2}(l_i) = \varnothing$.

Subphase 3. In this subphase we construct representatives of all arcs (p_k, c) whose parent node p is a contractum node; represented by a packet at l by virtue of either subphase 1 or subphase 2 case (4). The child node of such an arc can be either another contractum node or a redex node. In building the representatives of such arcs, we fill in the item lists of representatives of contractum nodes, and insert reversed pointers into the return address sets of representatives of both contractum nodes and redex nodes. Doing so requires consideration of several cases as follows.

Consider a normal arc (p_k, c) in $g'(P)$.

• If c was itself a contractum node not the RHS of a redirection in Red_g, then $\alpha_{1.3}(l)[k] = \pi_{1.3}(c) = \pi_{1.2}(c) = \pi_{1.1}(c)$, the location selected for c in subphase 1.

• If c was a contractum node that was the RHS of a redirection, $\alpha_{1.3}(l)[k]$ is the location of the packet that c overwrote.

• If c was in $g'(L)$ then $\alpha_{1.3}(l)[k] = \pi_{1.1}(c)$, the original location of c. (If c was redirected, either the target packet, or an Ind pointing to the target will be at location $\pi_{1.1}(c)$, as required. If not, c's packet will still be where it was before.)

Now consider a notification arc (p_k, c) in $g'(P)$. In all cases, BLANK is put into $\alpha_{1.3}(l)[k]$.

• If c was itself a contractum node that was not the RHS of a redirection, then a return address $l.k$ is added to $\rho_{1.2}(\pi_{1.2}(c))$, and when all of these have been accumulated, the result is $\rho_{1.3}(\pi_{1.3}(c))$, where $\pi_{1.3}(c) = \pi_{1.2}(c) = \pi_{1.1}(c)$.

• If c was a contractum node that was the RHS of a redirection, $l.k$ is added to the return address set at the location that c overwrote.

• If c was in $g'(L)$, there are three subcases. If c was not redirected, $l.k$ is added to $\rho_{1.2}(\pi_{1.2}(c))$, where $\pi_{1.2}(c) = \pi_{1.1}(c)$. If c was redirected, and the packet representing the target is now at $\pi_{1.2}(c) = \pi_{1.1}(c)$, $l.k$ is added to $\rho_{1.2}(\pi_{1.2}(c))$. If c was redirected, and an Ind packet is at $\pi_{1.1}(c)$, $l.k$ is added to $\rho_{1.2}(\alpha_{1.2}(\pi_{1.1}(c))[1])$, i.e. into the return address set at the location pointed to by the Ind.

Phase 2.

Phase 2 completes the new packet store representation Π_2, incorporating the effects of root quiescence and the activations.

Let $x \in g'(L)$.

If x is neither the LHS or the RHS of any redirection in Red_g, then the packet location l representing x remains unchanged and

$$\mu_2(l) = \begin{array}{l} \textbf{If } x \in Act_g \text{ and } \mu_{1.3}(l) = \varepsilon \textbf{ Then } * \\ \textbf{Else If } x = g'(root) \text{ and } x \notin Act_g \textbf{ Then } \varepsilon \\ \textbf{Else } \mu_{1.3}(l) \end{array}$$

If x is the LHS but is not the RHS of any redirection in Red_g, then if all the hypotheses of the overwriting lemma 5.10 were satisfied for x, $h(x)$ is garbage in the graph H produced by the rewrite, so that x need not have a representing packet in the resulting packet store, and indeed $\pi_{1.1}(x)$ has already been overwritten in subphase 1.2, so we need do nothing. Otherwise, if $x \in Rem_g$, then prior to overwriting, a copy was made in subphase 1.1 of $\pi_G(x)$ at say location $l' = \pi_2(x)$. We merely need to perform the root quiescence and activations at this new location viz.

$$\mu_2(\pi_2(x)) = \begin{array}{l} \textbf{If } x \in Act_g \text{ and } \mu_{1.3}(l') = \varepsilon \textbf{ Then } * \\ \textbf{Else If } x = g'(root) \text{ and } x \notin Act_g \textbf{ Then } \varepsilon \\ \textbf{Else } \mu_{1.3}(\pi_2(x)) \end{array}$$

Finally if x is the RHS but is not the LHS of any redirection in Red_g, then our implementation strategy left its location l unchanged; therefore

$\mu_2(l) =$ **If** $x \in Act_g$ and $\mu_{1.3}(l) = \varepsilon$ **Then** $*$
Else If $x = g'(root)$ and $x \notin Act_g$ **Then** ε
Else $\mu_{1.3}(l)$

Apart from the changes mentioned directly or indirectly above, the packet store is to remain unaltered.

End (of definition 6.7).

Theorem 6.8 Let $D = (P, root, Red, Act)$ be a rule and $g : L \to G$ be a standard redex in a balanced graph G. Suppose a rewrite of the redex by the rule produces a graph H. Then definition 6.7 provides, up to garbage, a correct implementation of rewriting in the sense that if $\Pi(G)$ is a correct representation of at least LSG(G) in a packet store, then the construction in definition 6.7 generates $\Pi(H)$, a correct representation of at least LSG(H) in the packet store.

Proof. If $\Pi(G)$ is a correct representation of LSG(G), then

(1) For each node x of LSG(G), there is a distinct representative $\pi_G(x)$, with appropriate marking and label.

(2) For each normal arc (p_k, c) of LSG(G), there is a (possibly trivial) indirection chain, starting at $\pi_G(p)$, such that $\alpha^+_G(\pi_G(p))[k] = \pi_G(c)$.

(3) For each notification arc (p_k, c) of LSG(G), there is a reversed pointer $\pi_G(p).k$ in $\rho_G(\pi_G(c))$, and $\alpha_G(\pi_G(p))[k] =$ BLANK.

We have to sure the same holds for $\Pi(H)$. This reduces to a particularly uninspiring case analysis of all the possible ways that live nodes and arcs arise in H, and we will omit the details, hoping that readers are sufficiently convinced by the description and examples above. Save for the following comments.

On the whole, definition 6.7 follows the abstract definition of rewriting, except where redirections are concerned. Since we take care to detect and not implement identity redirections (whether given already at the syntactic level, or merely generated as a byproduct of a noninjective redex matching), the mechanism of overwriting the LHS packet of a redirection with the target, or with an Ind that points to it, will not yield a cycle of Inds provided the representation of LSG(G) was correct. On this basis, the implementation of the various cases for redirection clusters in definition 6.7 can be shown to yield a correct representation of redirection, given that a representation of a balanced standard redex has no reversed pointers pointing at any of its packets, except perhaps for some packets matched to implicit nodes of the pattern.

Accepting that contractum building and redirection are correctly implemented, it is easy to check that root quiescence and activation are also correctly implemented. ☺

Of course it must not be supposed that the construction of definition 6.7 is the only, or the most efficient possible one for implementing rewriting, though it is intended to be reasonably efficient; there are a few optimisations that one may consider including. However all of them increase the complexity beyond its already substantial level.

Example 6.9 In [Fig. 15] we show how the construction of definition 6.7 implements the Fac[5] rewrite illustrated as the first step of [Fig. 1.(b)]. [Fig. 15.(a)] shows the

starting packet store, [Fig. 15.(b)] shows the situation after subphase 1.1, i.e. after allocation of packets to contractum nodes not involved in redirection, [Fig. 15.(c)] shows the situation after subphase 1.2, i.e. after a contractum packet has overwritten the LHS root packet, and [Fig. 15.(d)] shows the situation after subphase 1.3, i.e. after forwards and reversed pointers representing contractum nodes' arcs have been fixed up. There are no activations in this rewrite, and as the root packet has been destroyed, root quiescence is null. Hence phase 2 is vacuous and is not shown.

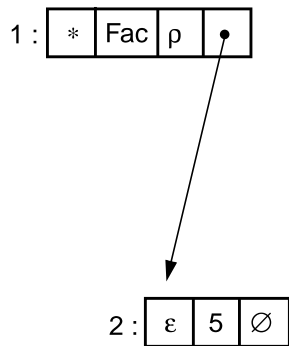

Fig. 15.(a) Starting packet store for the Fac[5] rewrite.

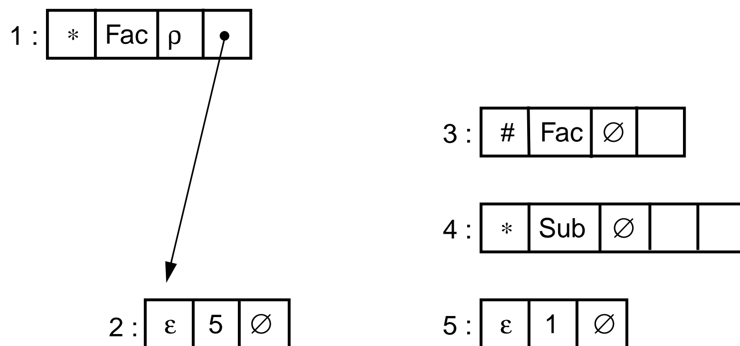

Fig. 15.(b) After subphase 1.1; allocation of certain contractum packets.

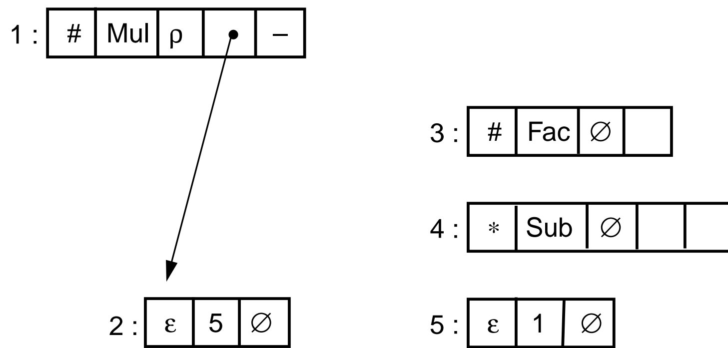

Fig. 15.(c) After subphase 1.2; overwriting of the root packet.

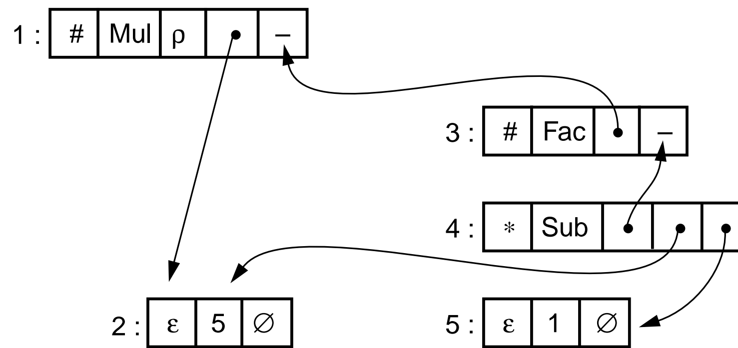

Fig. 15.(d) After subphase 1.3; pointer fixup for contractum packets.

7 VOLATILE PACKETS, PACKET MOBILITY, AND THE FIREWALL PRINCIPLE

In this section we describe some features of packet store rewriting which are extremely useful when we consider a distributed packet store and a more finegrained rewriting model. For technical convenience, we express such features in the flat packet store and coarse grained rewriting world of [Section 6] whenever we can.

Suppose there is a packet at location $t \in \Sigma$ such that no forward or reversed pointer points to it. Then the packet can be moved to some other unused location $t' \in \Sigma$ say, without destroying the correctness of the representation of any graph that the store rep-

resents. Packets with this property are called volatile since they are free to move around. (The reader will notice that we have exploited this property already in the packet store implementation of [Section 6] via the moving lemma 5.11.)

The easy movement of packets around the packet store is clearly very beneficial when the packet store is fragmented, as it would be in a distributed system, so we examine techniques for packet movement in a little more detail.

In general, unless access to the packet store is strictly via a sequence of non-overlapping atomic actions, and global pointer fix-ups can be done cheaply, arbitrary movement of packets around the store can be extremely expensive in terms of synchronisation costs and pointer adjustments. So we look to exploit special cases, where these costs can be reduced.

The main reason for wanting to move packets in a distributed packet store, is to distribute work to idle processors. Therefore the main candidates for movement are active packets at the roots of redexes, and perhaps the packets that they might need for rewriting. Suppose we want to move an active packet at location *t*. If it is volatile, we just move it. If not, and we are determined to move it anyway for whatever reason, then we can leave behind an Ind packet to act as target for the packet's incoming pointers. This is viable provided the packet was balanced to start with (i.e. had no incoming reversed pointers). Furthermore, to minimise pointer fixup, we can make the Ind suspended and make the connection to the newly volatilised packet via a reversed pointer from the packet. The transformation is shown in [Fig. 16], for an active packet labelled with a symbol F.

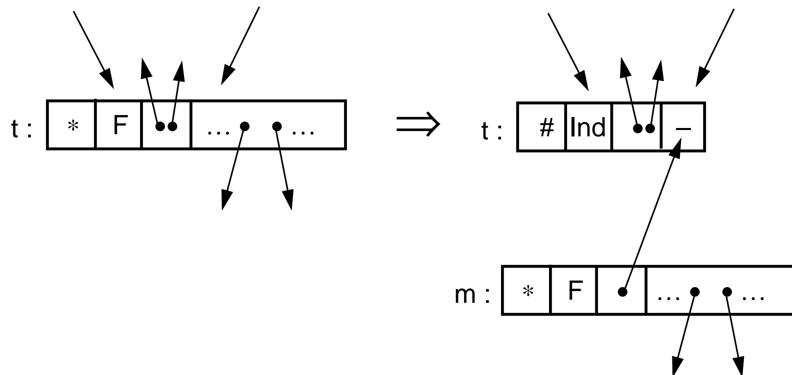

Fig. 16 Volatilising a non-volatile active packet

Note that the original return address set remains at location *t*. This is a design decision which is somewhat arbitrary at this stage, but leads to the question of what happens when the suspended Ind is notified. However, if we allow such an Ind to become active, and then to behave like any other packet for which there are no rewrite rules, i.e. to notify in its own turn, then notifications reach the original intended recipients, and suspended Inds act as intermediaries for reversed pointers in just as natural a manner as

idle Inds act as intermediaries for forwards pointers. Note the crucial role played by balancedness in all of this. The situation is illustrated in [Fig. 17].

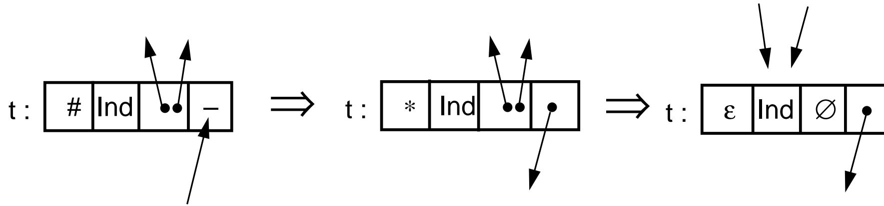

Fig. 17 Notification via a suspended Ind.

Our design for volatilising an active packet has the following side effect. Referring to [Fig. 16], the volatile version of the packet at location *m* can no longer be found by searching along pointers directed at *t*, since there is no pointer from *t* to *m*. This means that some atomic action which needed to access the packet at *t*, can no longer do so once the packet has been volatilised. Since we do not wish the semantics of a rewrite system to depend on whether packets are moved around using previously agreed mechanisms designed specifically for this purpose, we are led to enunciate the:

FIREWALL PRINCIPLE:

> Non-idle packets are to be regarded as inaccessible by rewriting primitives for which they are not the root packet, (with the exception of a few specific cases).

The kind of exceptions we envisage above, are eg. the initial contact that a rewriting agent makes with a packet, that establishes that the packet is indeed non-idle. Obviously this must be regarded as a form of access to the packet that is unavoidable, even if a decision not to access the packet further is taken subsequently.

The firewall principle is really a design objective for semantic models for packet store rewriting. Clearly the original DACTL semantics, when translated to a packet store, violates it since it permits the rewriting of non-standard redexes. One of the goals of this series of papers is to design alternative semantic models that respect the principle, without destroying correctness with respect to DACTL semantics. Pursuing this train of thought further casts the firewall principle in the role of provider of opportunities for alternative semantic models. If a semantic model wishes to avail itself of some action that would temporarily destroy correctness with respect to eg. DACTL semantics, then perhaps if the action is concealed behind a firewall of non-idle packets, the violation of correctness can be hidden from view until it has completed. Packet volatilisation above, may be viewed in this light; and a further most important instance of the exploitation of the principle will be described in [Section 9].

8 COPYING AND STATE SATURATEDNESS, CONSTRUCTORS AND STATEHOLDERS

Now we examine another problem arising from distributed implementations but which is expressible in the flat packet store world. Suppose an active packet at the root of a standard redex wishes to pattern match its explicit argument packets. Suppose some of them are not on the same processor as the root (suppose they occur distributed over several processors).

Since the redex is standard, we know that the arguments are idle, so moving them towards the root using the mechanism of [Section 7] is not possible; at least not without some additional threat to correctness, as leaving a non-idle Ind in place of a previously idle packet creates a firewall where there was none before, which exceeds the provisions of the previously announced firewall principle. Moving the root to the arguments is not possible in general either, since there may be several distinct processors holding the arguments. To avoid global pointer fixup problems we can try to copy the required arguments to the processor containing the root; but for this to work, we need to take a couple of precautions.

The first is that we must be sure that the argument packet will not occur as the LHS of a redirection performed by some concurrent rewriting agent. If it did, then the original packet and the copy would get out of step, leading to great problems in ensuring semantic consistency of the overall execution sequence. The copied argument must thus be a constructor. For simplicity, we can demand that all constructor packets be recognisable as such from an examination of the symbol labelling them, and we further stipulate that there are no rules for constructor symbols. So we designate a subset of $\mathbf{S}$ named $\mathbf{C}$, as the set of constructor symbols.

The second precaution concerns the return address set of the argument packet. If this is non-empty, possible problems arise if the argument receives an activation. Either we take measures to always direct the activation at the original packet, a nuisance if there is a local copy available; which following the notification by the constructor prompted by the activation, would again get out of step with the original. Or we forbid idle constructors to have non-empty return address sets. We have already studied a mechanism that enables us to ensure this. We just ensure that all execution graphs are state saturated by using theorem 4.4. This again makes use of a subset of symbols, the stateholders or variables $\mathbf{V}$, the only symbols permitted to label packets which are both idle, and have non-empty return address sets. We achieve our objective by stipulating that

$$\mathbf{C} \cap \mathbf{V} = \varnothing$$

While we are about it, we stipulate that there are no rules for stateholder symbols either. Thus the only distinction between constructors and stateholders is that stateholders can have non-empty return address sets while idle, and that they are redirectable, though only at non-root positions of a redex. Because of the latter fact, stateholders are of course not copyable.

Forbidding constructors to have non-empty return address sets when idle has a further beneficial consequence. In a balanced graph, a packet which is idle but has a non-empty return address set is really encoding the state of a synchronisation primitive, as any suspended parent will not be notified, and thus will not be able to rewrite, until the idle packet has been activated by some other subcomputation. Aesthetically, this is not a

task one would normally give to a constructor, so we restrict this role to stateholders, whose advertised properties make them fit for the job.

9 REDEX SIZE, STATEHOLDERS, AND THE MONSTR PRINCIPLE

This section explores the problem of redex size and structure, an issue that is only problematic in a concurrent distributed implementation. We use the word “redex” in the general sense to describe the set of packets needed for a given atomic action of a rewriting model, eg. notification.

The DACTL model makes no restrictions on the size of the LHS of a rule; the left subpattern can be abitrarily large. For standard redexes, this raises no problem in a flat packet store world with a serial execution model, but in a concurrent and distributed implementation, obvious problems occur when redexes with fairly arbitrary structure overlap. In general, prohibitive synchronisation costs must be paid in order to guarantee adherence to DACTL semantics in such an environment.

We wish to avoid such costs, and look for ways of circumventing them, bearing in mind also that a genuine implementation might choose to provide some hardware support for pattern matching of redex packets.

We note that the redex structure for a notification is fairly simple already. We just have an active root packet, and a number of waiting arguments, accessible from it by reversed pointers. One could almost call this a standard redex. In a balanced graph, the arguments are to be converted from one non-idle state to another, and we can exploit the firewall principle to hide the change of state and the precise moment it takes place from unwelcome eyes. Notification can therefore be implemented by sending suitable messages to the waiting arguments and the latency incurred thereby can be hidden from view. In a later paper in this series dealing with fine grained rewriting in detail, the above will be shown more formally. For now we say nothing more about notifications.

For rewrites, we have made some progress in permitting idle constructors to be copied to the root of the redex before matching. Presumably some caching mechanism retains a reference to the original location of a copy, to enable true graph matching to be performed. Allowing redexes of arbitrary depth entails several phases of copying, as packets at deeper levels can only be located when their parents have been retrieved, so we choose to allow only one level of constructor matching, hence of copying.

However the above leaves stateholders somewhat out in the cold as they cannot be copied. If there are several stateholder children of the root, then we have an impasse, since if they are distributed over several processors, there is no single place that all the required packets could meet. If there is only one though, we could exploit the volatilisation trick of [Section 7] to move the root of the rewrite to the processor containing its stateholder child, and only then proceed with the copying of constructors. This is the “most important example of exploitation of the firewall principle” mentioned at the end of [Section 7].

We have by the above argument, arrived at the essential structure of MONSTR patterns, to be described more formally in [Section 11], and at the intended method of dealing with them at the lower level of granularity of packet store rewriting whose understanding is our ultimate goal. We can encapsulate this in:

THE MONSTR PRINCIPLE:

> A root of a rewrite is permitted to require at most one stateholder child and perhaps several constructor children for pattern matching. All other nodes of the left subpattern must be implicit. The intended mechanism for matching is to move the root to the stateholder, and then to copy the constructors.

Incidentally, this also explains the MONSTR acronym: a Maximum of One Non-root STateholder per Rewrite. [Fig. 18] shows a typical MONSTR redex.

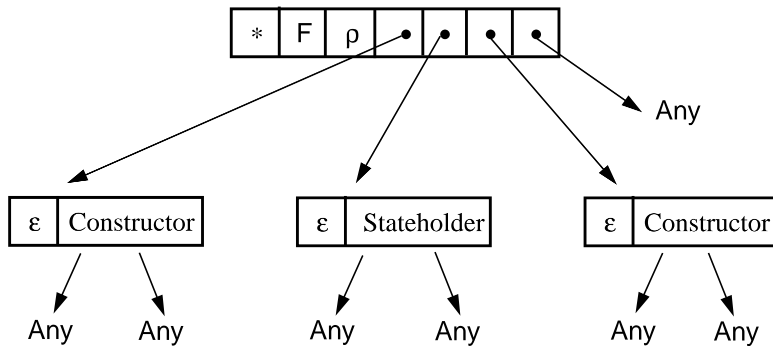

Fig. 18 The structure of a typical MONSTR redex.

To maximise the opportunities for hardware support during pattern matching, we will further insist that all rules for a symbol must match the same argument positions; and that any stateholder argument is likewise present in a fixed position. Thus effort is not wasted on accessing arguments that might not be needed, or on trying to establish where the stateholder argument might be. This will be formalised in the definition of [Section 11].

The scenario sketched above is the most we can accomplish without contravening the precepts described previously. If we were to permit ourselves to wander beyond the confines of the firewall prinicple as described in [Section 7], by moving idle stateholders, say leaving suspended Inds in their place, a much larger class of patterns would become available to us. The price to be paid would be a greater risk of deadlock during pattern matching, as previously accessible packets were replaced by suspended Inds, blocking other accesses by the firewall principle, perhaps rather indiscriminately. The correctness problems caused by such a rewriting model would prove greater, and the costs of the hardware support for making such pattern matching efficient would probably not be justified in terms of any imagined indispensability of the expressivity of pattern matching gained thereby.

10 MESSAGES, ATOMICITY, AND FINEGRAINED REWRITING

In a genuine distributed system, many tasks are accomplished using message passing. In the context of a distributed packet store a message can be viewed as a special kind of volatile packet, typically one with exactly one forwards pointer and exactly one reversed pointer. Whether the direction of travel of the message is along the forwards pointer or along the reversed pointer will depend on the purpose of the message.

At higher levels of abstraction, the indivisible atomic actions of a rewrite model of computation will often involve several packets. In a distributed packet store, there is no guarantee that all the packets for a given action will be in the same processor, so, ignoring for the moment issues of correctness, it is necessary to break such actions down into smaller subactions involving only one packet at a time, and to connect the subactions using messages.

We give an example of this process in [Fig. 19] and [Fig. 20]. In [Fig. 19], we show a higher level atomic action involving two packets, both of which are altered by the action. Typically one of the packets will be the instigator of the action, say it is root.

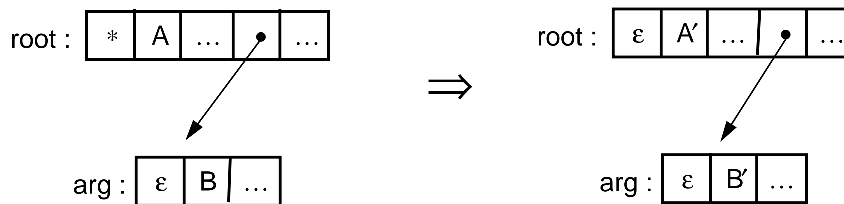

Fig. 19 A coarsegrained atomic action.

We break the action down into smaller pieces in [Fig. 20]. Thus as the instigator of the action, the root packet will send out a message to its argument, instructing it to change appropriately. [Fig. 20.(a)] shows the message on its way. On arrival, the appropriate change in the argument takes place, and a response message is sent back to the root, illustrated in [Fig. 20.(b)]. [Fig. 20.(c)] shows the end result.

Two points emerge from this. Firstly, since messages can be regarded as volatile packets, low level rewriting can be modelled by a suitable suite of primitives in our higher level flat packet store world. This is useful for studying such low level systems. And secondly, since the root started out active, and was suspended during the course of the intermediate actions, the firewall principle enables us to conceal some of the lower level actions from undue interference by other unrelated actions. (However, since the argument was idle, this concealment is not a perfect guarantee of correctness with respect to higher levels of abstraction).

Fig. 20 A finegrained version of the atomic action of [Fig. 19].

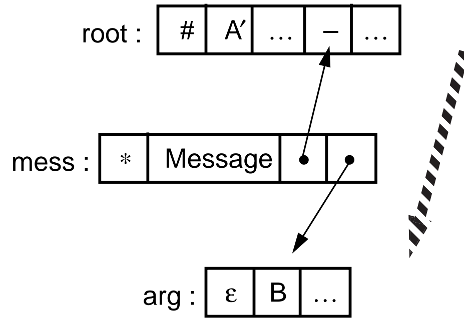

Fig. 20.(a) The root sends out a message in the direction shown.

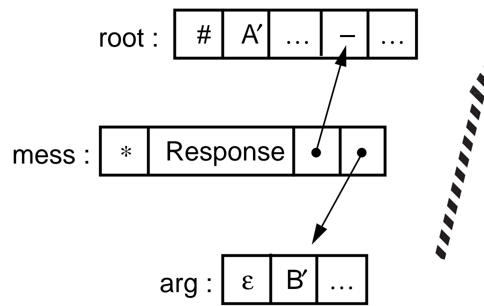

Fig. 20.(b) The message arrives at the argument; a response is sent out.

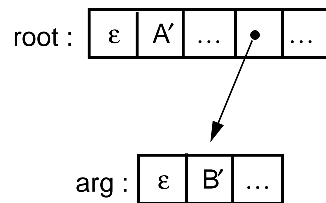

Fig. 20.(c) The response arrives and the action completes.

With this preamble, we are now able to sketch informally the low level distributed packet store rewriting model for what will be the MONSTR sublanguage of DACTL. The detailed study of the low level model will appear in a later paper in the series.

As before, the main tasks are the implementation of rewrites and notifications. We start with the notifications which are simpler and consist of the following two kinds of action.

Notification_Send. An active packet which is to notify, say X at location l, changes its marking to ε and sends out a Notification message to each return address $p.k$ in its return address set. The message contains a pointer to l as well as the return address $p.k$.

Notification_Receive. A Notification message with return address $p.k$ and pointer l arrives at its destination packet P at location p. The pointer l from the message is installed in the k'th item of P, and if P is suspended, its suspension marking is decremented; being changed to $*$ if the decrement removes the last suspension.

The combination of these two kinds of action is clearly a fragmenting of the more coarsegrained notion of notification in [Section 6]. Rewrites require a more complex breaking up into the following actions.

Move_to_Stateholder. On the assumption that the root packet of the rewrite has a stateholder argument (to be determined syntactically by the definition in [Section 11]), and the stateholder is in a remote processor, the root is volatilised and moved to that processor. Otherwise a null action.

Constructor_Request. For each constructor argument of the root such that the argument is in a remote processor, (which arguments these are, is again determined syntactically by the definition in [Section 11]), a Constructor_Request message is sent to the argument. The root becomes suspended on the requisite number of replies. Each message contains the pointer whose constructor is sought, and the return address of the root item whence it originated.

Constructor_Suspend. A Constructor_Request message arrives at its destination, which it finds to be a non-idle packet. The message packet itself becomes suspended, its pointer becoming a return address which is added to the return address set of the non-idle destination.

Constructor_Resume. A suspended Constructor_Request message is notified. It subsequently tries to reaccess the destination. (This action is really just like a Notification_Receive action).

Constructor_Respond. A Constructor_Request message arrives at its destination, which it finds to be an idle packet. A Constructor_Response message is formed. It contains a pointer to the destination and a reversed pointer to the originating root item, and travels back to the root. The body of the message contains either a copy of the argument if the argument is a constructor, or NONCONSTRUCTOR if the argument proved to be something other than a constructor.

Constructor_Receive. A Constructor_Response message arrives back at the root. The body of the message is cached locally, and the root is notified. If the last suspension on the root is removed thereby, the root is ready to perform the rewrite proper.

Stateholder_Suspend. A root of a rewrite is on the same processor as its one potential stateholder argument, and it has local copies of any other constructors (or NONCON-

STRUCTORS) required at other argument positions. The root attempts to lock the argument at the stateholder position but finds it non-idle. The root suspends waiting for the argument to notify.

Stateholder_Resume. A root of a rewrite which is waiting for the argumented at its stateholder position to notify, is notified. (Once more like a Notification_Receive action).

Perform_Rewrite. A root of a rewrite is on the same processor as its one potential stateholder argument, and it has local copies of any other constructors (or NONCONSTRUCTORS) required at other argument positions. The root attempts to lock the argument at the stateholder position and finds it idle. The lock succeeds. The root then pattern matches its explicit arguments to select an appropriate rule. Once the rule is selected, contractum building and redirection are performed essentially according to the recipe in [Section 6]; also root quiescence if necessary. If (exceptionally), the root is in the activation set of the rule, the activation is done there and then on the root packet. Otherwise, for each full pattern arc of the rule governing the rewrite whose child node is in the activation set of the rule, an Activation message is sent to the requisite packet representative in the redex. An Activation message contains a pointer to the destination packet, and if the arc to which it corresponds, is a notification arc from a contractum node, the message also contains a reversed pointer to the contractum packet item in question. (Clearly this can result in several Activation messages being sent to the same packet). The lock on the stateholder argument is released.

Activation_Receive. An Activation message arrives at its destination. If the message contains a reversed pointer, it is inserted into the return address set of the destination. If the destination packet is idle, its marking is changed to active.

The fragmenting of rewrites into the finegrained atomic actions above, clearly leads to substantial correctness problems, since in general, there is nothing to prevent non-serialisable schedules of low level primitives from arising in an execution sequence. By a non-serialisable schedule, we mean one whose constitutent finegrained atomic actions cannot be reordered into an order consistent with a sequence of coarsegrained rewrite and notification actions performed according to the prescriptions in [Section 6], without changing the result of the execution.

Having described MONSTR rewriting at both the coarsegrained and finegrained levels, it is instructive to see that both models display the essential features of "interaction" [Lafont (1990)]. In an interaction system, computation takes place only when two agents meet each other at their "principal ports"; whereupon a graph rewrite (of a kind different to those described in this paper) takes place. It is interesting that considerations of locality on the one hand (for MONSTR), and of cut elimination on the other (for interaction systems) led to such superficially similar looking solutions.

At the coarsegrained MONSTR level, one can view a MONSTR rewrite as an interaction between the root packet and the stateholder, the fixed stateholder position of the root serving as its principal port, and the locking of the stateholder itself acting to select which in-arc will serve as the stateholder's principal port for the rewrite. The admittance of constructors into the scheme of things can be viewed as a mild generalisation of the interaction paradigm, or as something that can be translated away at the coarsegrained level, by preceding the interaction with the stateholder by a sequence of interactions with the constructors. See also [Banach and Papadopoulos (1996b)].

At the finegrained level proper, we see the correspondence with the interaction paradigm even more clearly. Most of the atomic actions can be viewed as interactions between a packet and a message. Those that are not, are the initiating actions, which could be manipulated into interactions between packets and Activation messages if required, and the Perform_Rewrite actions which could be changed so that each preceding Constructor_Receive action was a separate interaction with the root, and encoded its data directly into the root packet, leaving the Perform_Rewrite action as simply an interaction between root and stateholder. Generally speaking, similar "linear" ideas have gained popularity in many spheres of contemporary computing research; and it is curious to note that the historical roots of both the present work (see references cited already), and of interaction nets (linear logic, the precursor of interaction net theory [see Girard (1987)]), both date back to around the same time.

11 THE MONSTR SUBLANGUAGE

In preceding sections we have identified a number of properties of graphs, rewrites and execution sequences, that singly and in combination, have appeared desirable for various reasons, but particularly for our lower level finegrained concurrent distributed packet store rewriting model. Here they are.

(1) Balancedness.

(2) State Saturatedness.

(3) The Overwriting Lemma.

(4) The MONSTR Principle.

(5) The Firewall Principle.

(6) Standard Redexes.

(7) Finegrained Rewrites and Notifications.

The first four of these can be addressed by simple syntactic restrictions, and these are the design desiderata that determine the definition of the MONSTR sublanguage defined below. The last three are semantic issues that contravene the semantics of DACTL rewriting and generate much more intricate correctness problems to be studied in depth subsequently.

The syntax of MONSTR is defined by a series of restrictions on alphabets, symbols, rules and systems as follows.

Restriction 11.1 (Alphabets) The alphabet of symbols $\mathbf{S}$, is the disjoint union of three subalphabets

$$\mathbf{S} = \mathbf{F} \cup \mathbf{C} \cup \mathbf{V}$$

$\mathbf{F}$ is the alphabet of function symbols. A function symbol may label the root of the left subpattern L of a rule, but not any subroot node of L. Function symbols may label the LHS of a redirection.

$\mathbf{C}$ is the alphabet of constructor symbols. A constructor symbol may label a subroot node of the left subpattern of a rule, but not the root. Constructors may not label the LHS of a redirection.

V is the alphabet of stateholders, or variables. A stateholder symbol may label a subroot node of the left subpattern of a rule, but not the root. Stateholders may label the LHS of a redirection.

The above formalises our previous remarks about **C** and **V**, and also introduces **F**, the functions, which are the only symbols for which there are rules. The restrictions separate the possible behaviours at root and at subroot nodes: the functions act as instigators of rewrites, the constructors encode immutable values, while the stateholders are able to model notions of updatable state, and to play a central role in the coding of synchronisation primitives.

Restriction 11.2 (Symbols)

(1) For each $S \in \mathbf{S}$, there is a set of natural numbers $A(S)$, in every case an initial segment of the naturals from 1, or empty.

(2) For each $F \in \mathbf{F}$, there is a subset of $A(F)$, $\mathrm{Map}(F)$.

(3) For each $F \in \mathbf{F}$, there is a subset of $\mathrm{Map}(F)$, $\mathrm{State}(F)$, in every case either a singleton or empty.

(4) $\mathsf{Root} \in \mathbf{C}$.

The above maps each symbol S to its arity $A(S)$. The intention is that all S-labelled nodes are to have the same arity. For functions F, $\mathrm{Map}(F)$ is the set of argument positions at which all normal rules for F (see below), will always need to pattern match. Similarly $\mathrm{State}(F)$, if non-empty, contains the position at which any stateholder argument of F must occur in a normal rule for F. Clause (4) states that Root is a constructor, assuring one of the hypotheses of the overwriting lemma 5.10.

Definition 11.3 (Normal and Default Rules) Let $F \in \mathbf{F}$. A rule for F such that each child of the root is a distinct implicit node is called a default rule for F. Otherwise the rule is a normal rule.

Note that with fixed arities, a default rule for F will always succeed in matching its left subpattern to any active F-labelled node of a graph, precisely because no-non trivial conditions need to be satisfied by the children of the root of the redex.

Restriction 11.4 (Rules) Let $D = (P, root, Red, Act)$ be a rule with left subpattern L. Then

(1) Each node has the arity dictated by its symbol, i.e.

$$\text{For all } x \in P,\ A(x) = A(\sigma(x))$$

(2) Each normal rule for a symbol matches the same set of arguments of the root, i.e. if $\sigma(root) = F$, and D is a normal rule then

$$\alpha(root)[k] \text{ is explicit } \Leftrightarrow k \in \mathrm{Map}(F)$$

(3) A rule for a function may match at most one stateholder, and then only in a fixed position; all other explicit arguments must be constructors, i.e. if $\sigma(root) = F$, and D is a normal rule then

$$\sigma(\alpha(root)[k]) \in \mathbf{V} \Rightarrow k \in \mathrm{State}(F)$$

(4) All grandchildren of the root are implicit, i.e. for all $k \in A(\sigma(root))$, and $j \in A(\sigma(\alpha(root)[k]))$

$$\alpha(\alpha(root)[k])[j] \text{ is implicit}$$

(5) Implicit nodes of the left subpattern have only one parent in the left subpattern, i.e. if $y \in P$ is implicit, there is precisely one $x \in L$ such that for some $k \in A(x)$, $y = \alpha(x)[k]$.

(6) Every $x \in P$ is balanced, i.e.

$$\mu(x) = \#^n \text{ (for } n \geq 1) \iff |\{k \mid \nu(x)[k] = \wedge\}| = n$$

(7) Every arc (p_k, c) of P is either state saturated or activated, i.e.

$$\nu(p)[k] = \wedge \text{ and } \mu(c) = \varepsilon \implies \sigma(c) \in \mathbf{V} \text{ or } c \in Act$$

(8) The root is always redirected, i.e. for some $b \in P$

$$(root, b) \in Red$$

(9) No arc can lose state saturatedness through redirection, i.e.

$$(a, b) \in Red \text{ and } \mu(b) = \varepsilon \implies \sigma(b) \in \mathbf{V} \text{ or } b \in Act$$

(10) A node which is the LHS but not the RHS of a redirection should be garbaged by a rewrite whenever possible, i.e.

$$(b, c) \in Red \text{ and } b \in Act \implies \text{there is a } b \neq a \in L \text{ such that } (a, b) \in Red$$

Theorem 11.5 (Desirable Properties) When all rules used, conform to Restriction 11.4, induction over executions yields many desirable properties. (We do not show these in detail, given the theorems already proved in preceding sections.)

Restriction 11.4.(1) makes all execution graph nodes respect the arities of their symbols.

Restrictions 11.4.(2) and (3) make sure that the pattern matching requirements of each redex, depend solely on the symbol at the root.

Restriction 11.4.(4) prevents children of the root occurring as children of their siblings in the rule. (N.B. This prevents a finegrained packet rewriting model from having to check whether grandchild pointers point to the correct packet in a second phase of pattern matching. From a theoretical viewpoint though, the condition could be weakened to asserting that every non-root node of the left subpattern was accessible as either a child or grandchild of the root, permitting the above.)

Restriction 11.5.(5) ensures that not even pointer equivalence is required for matching any redex node that is not determined by Map($\sigma(root)$).

Restriction 11.4.(6) yields balancedness of execution graphs via theorem 4.2.

Restrictions 11.4.(7) – (9) yield state saturatedness of execution graphs via theorem 4.4.

Restriction 11.4.(10) ensures that the overwriting lemma 5.10, applies to most redirections in practice, enabling the representation of rewriting by the packet store manipulations of [Section 6], with minimal need for the weaker moving lemma 5.11.

Restriction 11.6 (Systems, Rule Selection) For each $F \in \mathsf{F}$ there is a pair of sets ($\mathcal{N}_F$, $\mathcal{D}_F$), where $\mathcal{N}_F$ consists of normal rules for F, and $\mathcal{D}_F$ is non-empty and consists of just default rules for F. In an execution, when a chosen root t is identified and it is labelled by $F \in \mathsf{F}$, rule selection is performed according to the following procedure:

> **If** some rule from $\mathcal{N}_F$ matches the chosen root t
> **Then** $Sel = \{D \in \mathcal{N}_F \mid D \text{ matches at } t\}$
> **Else** $Sel = \mathcal{D}_F$

Choice of rule from *Sel* is nondeterministic as before.

Restriction 11.6 gives us a simple priority mechanism that always enables a normal rule for a symbol F to be used in preference to a default rule in situations where both would match. That we can always rewrite at a chosen root is ensured by the non-emptiness of $\mathcal{D}_F$. The full DACTL language enables similar and more sophisticated rule selection decisions to be made by virtue of a more powerful pattern matching calculus which we eschew.

The above list of restrictions constitutes the definition of the syntax of user-defined MONSTR systems. We have mentioned in passing that pointer equality testing (of implicit nodes) is avoided in the pattern matching repertoire of MONSTR; nevertheless it is often desirable in many programming situations. To allow for this, MONSTR includes some builtins, namely a specific function symbol PointersEqual, with two rather obvious rules. In concrete syntax, the single rule in $\mathcal{N}_{\mathsf{PointersEqual}}$ is

```
PointersEqual[ x x ] => *True
```

and the single rule in $\mathcal{D}_{\mathsf{PointersEqual}}$ is

```
PointersEqual[ x y ] => *False
```

The meaning of these rules should be obvious (and can be deduced from the comments about concrete syntax in the examples below). In a coarsegrained implementation, the normal rule will match iff its two pointer arguments dereference to the same node. In a distributed finegrained implementation, messages are sent out along the two argument pointers, and depending on the addresses of the locations that they report back, the root is redirected to True or False.

Obviously these rules give a fairly weak notion of pointer *inequality* in the presence of multiple concurrent agents. Given the semantics of redirection, once two pointers dereference to the same place, that situation will persist irrespective of what happens subsequently. But if they are reported to be different, then not much can be deduced in general, as by the time the False result is inspected and some consequent action initiated, some other redirection might have made them equal. See also [Lindstrom (1986)].

The above completes the definition of the syntax of MONSTR. As mentioned, MONSTR systems may be executed according to DACTL semantics, or according to other semantic models. The study of these other semantic models constitutes the semantic side of the MONSTR issue, and only by exploring it can we eventually establish other desirable properties that show correctness of executions performed according to these other models. A brief sketch of these questions follows in the next section.

Example 11.7 (Fac Revisited) We set out the Fac example of [Fig. 1], as a formal MONSTR system. We use the concrete syntax of DACTL, explaining the possibly less obvious features as we go.

S consists of
- **F** = {Fac, Mul, Sub, Initial}
- **C** = {0, 1, 2, 3, …} i.e. the naturals
- **V** = ∅

Rules for Sub

Rules in N_{Sub}

```
Sub[ 0 0 ] => *0 |
Sub[ 1 0 ] => *1 |
```

… etc. … i.e. the normal delta rules for subtraction.

(N.B. The notation *0 in the RHS of the first rule indicates a contractum node, to be created active. The notation => is a separator between LHS and RHS, and also indicates that the root of the LHS is to be redirected to the first node on the RHS. The vertical bar is used as a rule separator; it is DACTL's notation for a rule separator where there are no priorities among the rules.)

Default rule in D_{Sub}

```
Sub[ x y ] => ##Sub[ ^*x ^*y ]
```

(N.B. The notation ##Sub[^*x ^*y] indicates that the two implicit arguments of the root of the redex are to be activated, and the root of the contractum is to suspend, waiting for both to notify (along the notification arcs implicitly created). The rule illustrates a typical stucture for a default rule, embodying the slogan, "if none of the normal rules matched, attempt to induce the arguments to rewrite, in the hope that their reduced form will match one of the normal rules". Note the DACTL concrete syntax convention whereby identifiers starting with a lowercase letter are node identifiers, which can be used to express sharing as above, while identifiers starting with an uppercase letter are node symbols. Nodes mentioned by identifier alone are assumed to be implicit, as above.)

Rules for Mul

Rules in N_{Mul} … the normal delta rules for multiplication.

Default rule in D_{Mul}

```
Mul[ x y ] => ##Mul[ ^*x ^*y ]
```

Rules for Fac (first version)

Rule in N_{Fac}

```
Fac[ 0 ] => *1
```

Default rule in D_{Fac}

```
Fac[ n ] => #Mul[ n ^#Fac[ ^*Sub[ n 1 ] ] ]
```

(N.B. Note the fairly obvious notation for nesting node descriptions, using explicit node identifiers (eg. n) only where multiple references to the same node are needed.)

Rules for Fac (second version)

Rules in N_{Fac}

```
Fac[ 0 ] => *1   |
Fac[ n:1 ] => #Mul[ n ^#Fac[ ^*Sub[ n 1 ] ] ]   |
Fac[ n:2 ] => #Mul[ n ^#Fac[ ^*Sub[ n 1 ] ] ]   |
… etc. …
```

Default rule in D_{Fac}

```
Fac[ n ] => #Fac[ ^*n ]
```

(N.B. Note that when both a node identifier and its symbol need to be quoted, the syntax is x:S. In this version, arguments which are naturals are explicitly matched. Obviously a more compact notation could be invented for this, eg. Fac[n:Nat], as exists in DACTL, but we want to keep the syntax of MONSTR as light as possible in these papers.)

Rules for Initial

Default rule in D_{Initial}

```
Initial => *Fac[ 5 ]
```

The rewriting of the Initial node using either set of rules for Fac will clearly yield the computation of [Fig. 1].

Example 11.7 (Bottom Avoiding Merge Revisited) This example shows how the synchronisations modelled by unbalanced BAM nodes in example 6.16 can be expressed in MONSTR. Essentially one translates the nondeterminism arising from the receipt of individual notifications by a once suspended BAM node, into the nondeterminism of choice of redex to be rewritten among a family of overlapping redexes.

The scenario features a Consumer of the merged lists, and a pair of Producers each independently producing Items. The Producers and Consumer interact via a stateholder which is accessed by Reader and Writer agents connected to the Consumer and Producers respectively. The stateholder acts like a bounded buffer with one slot. When the stateholder is Full (and contains an Item), the Reader may read it for the benefit of the Consumer making it Empty, and Writers acting on behalf of Producers have to wait. When the stateholder is Empty, both Writers may attempt to write an Item to it, making it Full, and the nondeterminism of selecting the chosen redex root, serves as the source of nondeterminism in the merge. The merge is bottom avoiding since a Writer will only attempt to access the stateholder when the corresponding Producer has actually generated an Item to put into it. [Fig. 21] shows a possible execution graph of this system, omitting garbage.

S consists of

F = {Producer, Consumer, Reader, Writer, Initial}
C = {Item, Cons, Nil}
V = {Empty, Full}

Rules for Producer

Default rule in D_{Producer}

```
Producer => *Cons[ Item *Producer ]
```

(N.B. A Producer generates a list of Items.)

Rules for Consumer

Rule in N_{Consumer}
```
Consumer[ Cons[ h t ] ] => #Consumer[ ^*t ]
```

(N.B. A Consumer consumes a list of Items.]

Default rule in D_{Consumer}
```
Consumer[ x ] => #Consumer[ ^x ]
```

Rules for Reader

Rules in N_{Reader}
```
Reader[ s:Full[ x ] ] => *Cons[ x #Reader[ ^y:Empty ] ] , s := *y   |
Reader[ s:Empty ] => #Reader[ ^s ]
```

(N.B. If the stateholder is Full, a Reader can extract its Item, making it Empty, and causing Writers to be subsequently notified. Otherwise it must wait. The customary notation for assignment (:=) is used to indicate the required non-root redirection.)

Default rule in D_{Reader}
```
Reader[ x ] => #Reader[ ^x ]
```

Rules for Writer

Rules in N_{Writer}
```
Writer[ Cons[ h t ] s:Empty ] => #Writer[ ^*t u:Full[ h ] ] , s := *u   |
Writer[ x:Cons[ h t ] s:Full ] => #Writer[ x ^s ]
```

(N.B. If the stateholder is Empty, a Writer can insert an Item into it, making it Full, and causing any Reader to be subsequently notified. Otherwise it must wait.)

Default rule in D_{Writer}
```
Writer[ x y ] => ##Writer[ ^*x ^*y ]
```

Rules for Initial

Default rule in D_{Initial}
```
Initial =>  #Consumer[ ^#Reader[ ^s:Empty ] ] ,
            x:#Writer[ ^*Producer s ] ,
            y:#Writer[ ^*Producer s ]
```

(N.B. Initial creates the basic configuration of two Producers and one Consumer.)

Readers (of this paper), will quickly realise that the essential mechanism at work in this example, is very similar to the one for semaphores given in [Fig. 2], except that transmission of data is taking place here. The semaphore rules are MONSTR rules provided we assign symbols to classes **F**, **C**, **V** appropriately. (This is why we previously said that we would want Free to be a stateholder.) Readers will further realise that similar systems would solve many of the classical synchronisation problems, such as the readers and writers problem, or the bounded buffer. Indeed it turns out that the underlying programming pattern here is at the heart of the wide applicability of MONSTR noted at

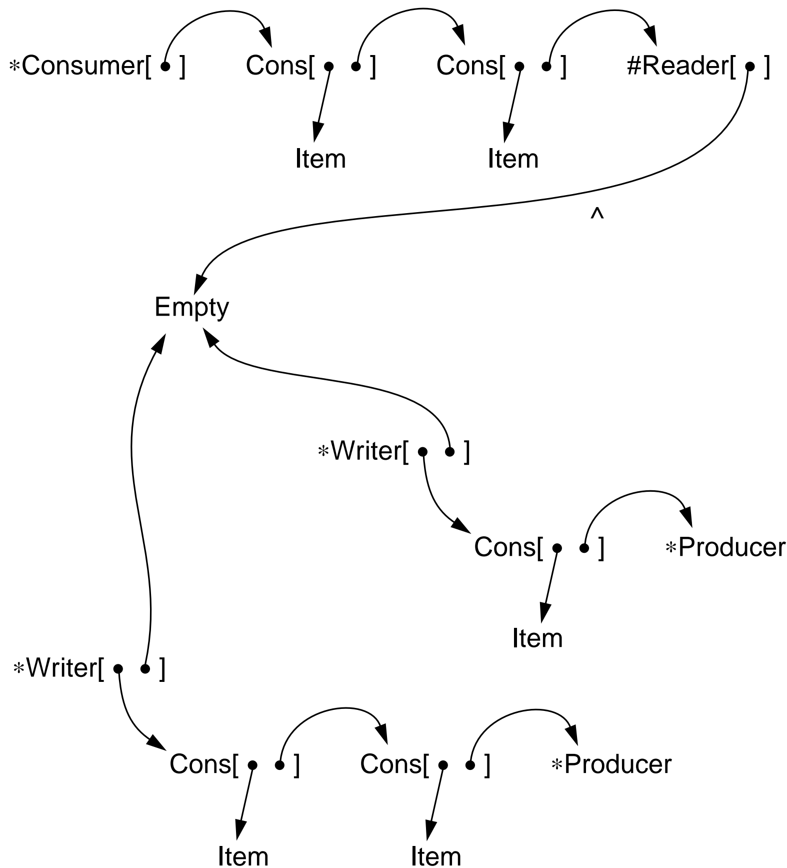

Fig. 21 A possible configuration of the MONSTR version of Bottom Avoiding Merge. Note the two overlapping Writer redexes.

the end of the introduction. And lastly we note that the running example of a rewrite in [Section 2] can be viewed as a rewrite of a MONSTR rule in the same way.

12 BRIDGING THE SEMANTIC GAP — OUTLOOK

It is clear that the gap between the coarsegrained DACTL rewriting model, and the finegrained packet store rewriting model is considerable. Rather than attempt to find a way of bridging the gap in one span, it is best to use the divide and conquer heuristic to break the task up into more manageable pieces. Accordingly, the study of MONSTR semantics will consist of the definition of a number of semantic models for executing MONSTR systems, each embodying some feature that brings it one step closer to the ultimate model of interest, and then of the study of the circumstances under which each

new model yields results equivalent to those of its predecessor. In this manner we gradually descend from DACTL semantics to the finegrained packet store model.

Because the gap between the two semantic models is so large it is futile to hope that everything derivable under the original DACTL semantics will be derivable in the lower level models; so that expecting completeness results in that sense is pointless (though because all our models have full computational power, one can give completeness by "coding up" DACTL behaviours using some form of simulation — but this is hardly what we are after).

We must therefore be satisfied with soundness results, so that a lower level model accurately reproduces an acceptable subset of behaviours of the higher level model. The soundness results will normally state that under suitable syntactic constraints, the new semantics behaves acceptably with respect to the old. The difference between such syntactic restrictions and those say of [Section 11], is that the restrictions gain their effect only in collusion with the new semantics, whereas those of [Section 11] mostly yielded desirable results within the remit of the original semantics. We now sketch some of these semantic models.

12.1 The Suspending MONSTR Model

Probably the easiest semantic feature in the low level model to address is the firewall principle. This entails formalising what happens when a rewrite attempts to rewrite a (potential) redex which is not standard, i.e. where the chosen root has some explicitly matched arguments which are not idle. The suspending MONSTR semantic model prescribes a suspension event under such circumstances. The root of the rewrite becomes suspended on all its non-idle explicitly matched arguments, all the requisite arcs become notification arcs, and the rewrite cannot retry until all the non-idle arguments have notified. Otherwise, suspending MONSTR semantics is identical to DACTL semantics, so the only other events the model admits are notifications, and rewrites of standard redexes. It is fairly clear that the desirable syntactic properties of executions discussed in [Section 11], are unaffected by this change.

The main problem which suspensions introduce is that of potential deadlock. Two (or more) overlapping non-standard redexes may become suspended on each other's non-idle nodes and become deadlocked in a way that DACTL rewrites, insensitive to non-root markings would not. We can borrow techniques from deadlock theory to give a reasonable account of safe systems.

12.2 The Finegrained MONSTR Model

The finegrained MONSTR model expresses in graph terms, the features of the low level packet rewrite model described in [Section 10]. Remaining in the graph world is technically convenient for semantic investigations. Thus at least in principle, there are message nodes, and special kinds of rewrite and notification for them etc. However most of this detail can be disregarded. A typical message will only be relevant to its source and destination nodes; and it is only the actions that take place at the source and at the destination, the fact that they are separated in time, and the extent to which they are visible to other actions taking place in the execution graph, that makes a difference to the result computed by an execution. The firewall principle helps a lot in reconciling computations generated by the finegrained model with those generated by the suspending model.

The main problem which the breaking up of larger atomic actions into smaller ones generates, is one of non-serialisable schedules, as has been mentioned before. Certain schedules of finegrained actions cause changes of state incompatible with coarsegrained rewriting, to be visible to the computation at large, and which the firewall principle is incapable of concealing. Controlling these in the presence of the sharing of subgraphs can pose quite a challenge for serialisability theory.

12.3 Serialisable Weak Models

One feature that is inherent in both preceding semantic models is the accurate implementation of redirection, in particular the non-implementation of identity redirections that arise because the LHS and RHS of a redirection match the same node. In reality this requires an identity test, and this in turn requires access to non-Map(σ(*root*)) nodes when these are the RHS of a redirection, complicating pattern matching. When concurrent rewriting is taking place, this has to be strengthened with some locking mechanism to prevent cycles of Inds being generated by the simultaneous effecting of two mutually directed redirections, eg. $x := y$ and $y := x$. Doing either of these alone is safe, but doing both simultaneously without locking, can in a packet store implementation, generate a cycle of Inds.

This is a problem at both coarsegrained and finegrained levels of abstraction. Its study involves looking at how data dependencies can evolve in an execution of a system, and static analysis of a system can yield the insight that dangerous situations such as the one described, do not arise. In reality, an architecture that truly only examines Map(σ(*root*)) nodes in a redex, must have some assurance that the above problems will not arise in any rewrite with a RHS of a redirection at one of the non-Map(σ(*root*)) nodes of the redex.

12.4 Coercing MONSTR Models

One can say that the suspending MONSTR model is coercing in a weak sense in that redexes wait for the subcomputation at non-idle matched nodes to notify before proceeding. One can strengthen this behaviour so that the pattern matching process, when it encounters an idle function, where it expects a constructor or stateholder, actually activates the function and suspends on the subcomputation, in the hope that the subcomputation will reduce to constructor or stateholder form. This kind of coercing semantics is useful in the implementation of many functional languages, which adhere to just such a strategy. There is also some real benefit for the finegrained rewriting model, where the coercing behaviour strengthens the hand of the firewall principle and affords considerable simplification to the main soundness result.

12.5 Non-MONSTR Models

As well as semantic models expressly designed for executing MONSTR systems, we can consider some generalisations. For instance there is the proposal at the end of [Section 9] for dealing with larger patterns containing more than one stateholder. If we agree that in such a model, stateholders are temporarily made non-idle during the rewrite, whether by being substituted by suspended Inds or otherwise, we can examine the resulting semantics both at coarsegrained and at finegrained levels.

13 CONCLUSIONS

In the previous sections, we have set out the DACTL rewriting model, and the low level packet store rewriting desiderata that led to the series of restrictions in [Section 11]; these in turn led to the possibility that the MONSTR sublanguage that they defined, might be executable on a distributed store architecture with reasonable efficiency and reasonable semantics. In the immediately preceding section, we outlined some aspects of the semantic task that needs to be addressed to justify this view. Subsequent papers in this series will grapple with the murky details.

Acknowledgements

In the early days of MONSTR, during the Flagship project, many people contributed useful ideas that eventually evolved into the design described in this paper. MONSTR would certainly not have happened without their contributions. It is a pleasure to acknowledge Ian Watson, Paul Watson, Viv Woods, John Sargeant, Mark Greenberg, John Glauert, Richard Kennaway, Ronan Sleep, George Papadopoulos, Nic Holt, Steve Leunig, Tom Thompson and Paul Townsend. Others whom fading memories have caused to be omitted, are offered heartfelt apologies by the author.

References

[Banach (1993)] 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).

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

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

[Banach and Papadopoulos (1995b)] Banach R., Papadopoulos G., A Highly Parallel Model for Object-Oriented Concurrent Constraint Programming. *in*: Proc. IEEE ICA^{3}PP-95, **1**, 61-70, IEEE Computer Society Press, (1995).

[Banach and Papadopoulos (1996a)] Banach R., Papadopoulos G., Expressing Runtime Structure and Synchronisation in Concurrent Object-Oriented Languages with MONSTR. *in*: Proc. IFIP FMOODS-96, 16pp., Chapman Hall, (1996), *to appear.* (See also the Prelim. Proc. pp. 373-388.)

[Banach and Papadopoulos (1996b)] Banach R., Papadopoulos G., A Study of Two Graph Rewriting Formalisms: Interaction Nets and MONSTR. *Submitted to*: J. Programming Languages, (1996).

[Banach et al. (1995)] Banach R., Balazs J., Papadopoulos G., A Translation of the Pi-Calculus into MONSTR, J.UCS **1**, 335-394, (1995).

[Barendregt et al. (1987)] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R Kennaway, M.J. Plasmeijer, M.R. Sleep., Term Graph Rewriting. *in*: Proc. PARLE-87, de Bakker, Nijman (eds.), LNCS **259**, 141-158, Springer, (1987).

[Darlington and Reeve (1981)] Darlington J., Reeve M., ALICE — A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. *in*: Proc. ACM FPLCA, ACM Press, (1981).

[FPLCA] Proceedings of various ACM FPLCA Conferences, ACM Press, 1980-1990.

[Fasel and Keller (1986)] Fasel J.H., Keller R.M. (eds.), Graph Reduction. Proc. Santa Fe Workshop 1986, LNCS **279**, Springer, (1986).

[Girard (1987)] Girard J-Y., Linear Logic, Theoretical Computer Science **50**, 1-102, (1987).

[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).

[Glauert et al. (1990)] Glauert J.R.W., Kennaway J.R., Sleep M.R., DACTL: An Experimental Graph Rewriting Language. *in*: Graph Grammars and their Application to Computer Science, Ehrig, Kreowski, Rozenberg, (eds.), LNCS **532**, 378-395, Springer, (1990).

[Lafont (1990)] Lafont Y., Interaction Nets. *in*: Proc. Seventh A.C.M. Symposium on Principles of Programming Languages, 95-108, ACM, (1990).

[Lindstrom (1986)] Lindstrom G., Implementing Logical Variables on a Graph Reduction Architecture. *in*: Proc. Graph Reduction, LNCS **279**, 382-400, Springer, (1986).

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

[Treleaven et al. (1982)] Treleaven P.C., Brownbridge D.R., Hopkins R.P., Data Driven and Demand Driven Computer Architecture. Computing Surveys **14**, 93-143, (1982).

[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).

[Woods (1986)] Woods J.V. (ed.), Fifth Generation Computer Architectures. Proc. IFIP TC 10 Working Conference, Manchester 1985, North-Holland, (1986).