MONSTR: TERM GRAPH REWRITING FOR PARALLEL MACHINES

R. Banach

Computer Science Department, Manchester University,

Manchester, M13 9PL, U.K.

Abstract

Term graphs locally look like terms, but globally have a more general directed graph structure. A rewriting model gives term graph rule systems an operational semantics and makes them into a general model of computation which one might hope to implement directly on a suitable architecture. Unfortunately, if the architecture is not basically sequential, the demands of the model are too great to make direct implementation a reasonable proposition. Other models may be defined that have better properties and which may be proved equivalent to the original under suitable conditions.

1 Introduction

Term graphs and their rewriting model were first described as such in Barendregt et al. (1987). Since then, two main directions of research have been identified. The first treats a term graph as some sort of representation of an associated term, and seeks to see how results about TRSs carry over to the graph world (see eg. Barendregt et al. (1987), Farmer et al. (1989), or Kennaway et al. (1990)). The second treats term graph rewriting more as a model in its own right and seeks to identify issues of independent interest. The present work fits into the latter category and the primary issue addressed is the adoptability of generalised term graph rewriting systems (TGRSs) as a fundamental computational model at the architectural level. This is a desirable thing to do since many currently fashionable programming styles (e.g. functional or logic) have direct and transparent translations into the TGR framework.

Unfortunatley, the practicalities of implementing the model on a parallel machine (an architectural style that is also currently fashionable), are such as to make the objective infeasable in its raw form. This is surprising because there are many implementations of fashionable programming styles on fashionable architectures that actually perform acceptably well. This indicates that there must be features of the full TGR model whose power is not being utilised fully when one translates such programs into the TGR form, and moreover, that these are the more difficult features of the model to implement. This hypothesis turns out to be correct, and gives rise to a program of work to indentify the little used, expensive aspects of general TGRSs and to exclude them by carefully redefining TGR languages and/or their semantic models. Some aspects of this work are surveyed in this paper.

There is far too little space available to present all the details, let alone all the proofs. These can be found in Banach (1992). The rest of the paper is organised as follows. In section 2, we present an abstract version of a concrete TGR language, DACTL. In section 3 we present a sublanguage, MONSTR, and justify its selection on architectural grounds. In section 4, we discuss a number of semantic models for the MONSTR sublanguage, arguing that these more closely reflect the way a realistic parallel machine might execute MONSTR programs. We present, usually without proof, the main soundness results that map computations according to these more operational models, into computations according to the original semantics, thus giving a handle on correctness.

2 Abstract DACTL

We present here a rather abstract and slightly simplified version of the language DACTL (Glauert et al. (1988a, 1988b)). The abstraction takes us away from the tedium of concrete syntax and enables us to concentrate on the semantic issues. We assume we are given an alphabet $\mathbf{S} = \{S, T\ldots\}$ of **node symbols**. We write **SeqN** for the set of sets of naturals of the form $\{1\ldots n\}$, where $\{1\ldots 0\} = \emptyset$.

Definition 2.1 A(n abstract DACTL) **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 $N \to \mathbf{S}$,

(3) α is a map $N \to N^*$,

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

(5) ν is a map $N \to \{\varepsilon, \hat{}\ \}^*$,

such that for all $x \in G$, $\mathrm{dom}(\alpha(x)) = \mathrm{dom}(\nu(x)) \in$ **SeqN**.

Informally, for each node we have its node symbol $\sigma(x)$ and its sequence of successors $\alpha(x)$. The node carries a **node marking** $\mu(x)$, and each arc to a successor (say the k'th, $\alpha(x)[k]$) carries an **arc marking**, $\nu(x)[k]$. 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 $x \in G$ (which we write in preference to $x \in N(G)$), the domain of $\alpha(x)$ or of $\nu(x)$ is called the **arity** of x, $A(x)$. We refer to an arc of the graph using the notation (p_k, c) to indicate that c is the k'th child of p, i.e. that $c = \alpha(x)[k]$ and $k \in A(p)$.

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

We assume there is a special node symbol Any, not considered to be in $\mathbf{S}$.

Definition 2.2 A **rule** R is a quadrouple (G, r, Red, Act) where

(1) G is a graph, except that some nodes of G may be labelled with Any, and

$$\sigma(n) = \mathrm{Any} \Rightarrow \alpha(n) = \varepsilon$$

(ε is the empty sequence, empty node marking, empty arc marking, according to context).

Nodes n with $\sigma(n) = \mathrm{Any}$ are called **implicit** whereas other nodes are **explicit**.

(2) r is a node of G called the **root**, and all implicit nodes are accessible from the root via a path of length at least one (so the root is explicit). If $\sigma(r) = S$, then R is a **rule for** S. For all nodes x accessible from (and including) r, $\mu(x) = \varepsilon$ and $\nu(x)[k] = \varepsilon$ for all $k \in \mathrm{A}(x)$.

(3) Red is a set of pairs, (called **redirections**) of nodes of G such that if $(x, y) \in Red$, then x is explicit and accessible from r. If $(x, y), (u, v) \in Red$ then $x = u \Rightarrow y = v$ and $x \neq u \Rightarrow \sigma(y) \neq \sigma(v)$. For $(x, y) \in Red$, x is called the **LHS** and y the **RHS** of the redirection.

(4) Act is a set of nodes (called **activations**) of G such that for all $x \in Act$, x is accessible from the root r.

The subgraph of G accessible from (and including) the root is called the **pattern** of the rule, and nodes of G not in the pattern are called **contractum** nodes. More generally a pattern is a rooted graph in which there may be some implicit nodes. By a graph we will normally mean an object without implicit nodes and not necessarily endowed with a root.

We note at this point that condition (3) on rules, is a simplification of what is permitted by the general DACTL model. The more complex possibilities allowed by the DACTL definition concern the so-called pattern calculus which is too far removed from our present concerns to be included here.

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

(1) $h(r) = t$

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

So a matching is a graph structure homomorphism in which Any nodes can match anything but explicit nodes must behave well. The same defintion will suffice for matching patterns to other patterns or, omitting mention of roots, for matching graphs to other graphs.

Matching allows us to describe the rewriting model for TGRSs.

A **system** is just a set of rules. Rewriting proceeds via two stages, rule selection and rule execution.

Definition 2.4 Let X be a graph, $t \in X$ a node of X such that $\mu(t) = *$, and $\mathbf{R}$ a system. Let $Sel = \{R \mid$ there is an $R \in \mathbf{R}$ such that there is a matching $h : P \to X$ of the pattern P of the graph G of the rule R to X at $t\}$. **Rule selection** is some (otherwise unspecified) process for choosing a member of Sel assuming it is non-empty. It clearly only has force when there is more than one member of Sel. The chosen R makes t the root of the **redex** $h(P)$ and R the **selected rule** that **governs** the rewrite.

We note that the above is an oversimplification compared to the DACTL definition, the differences again involving the pattern calculus.

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 X, t, $R = (G, r, Red, Act)$ and h given as above.

Definition 2.5 Contractum building adds a copy of each contractum node of G to X. Node markings for such nodes are taken from G. Copies of arcs of G from contractum nodes to their successors are added in such a way that there is a graph structure homomorphism h' from the whole of G to the graph being created which agrees with h on P. Arc markings are again taken from G. Call the resulting graph X' and let $i_{X,X'}$ be the natural injection.

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

Definition 2.7 Activation takes the nodes $y = i_{X',X''}(x)$ for $x = h'(u)$ with $u \in Act$, and if $\mu(y) = \varepsilon$ then the node marking is changed to $*$. Unless $t'' = i_{X',X''}(h'(r))$ was one of these nodes, its marking is changed to ε. Call the resulting graph Y, and define $i_{X'',Y}$ as the natural injection.

The result of the rewrite is the graph Y. For a more detailed exposition of contraction building and redirection, see Barendregt et al. (1987) 3.6 (i)-(ii), or Glauert et al. (1988a) for a more detailed exposition of all the phases in a concrete syntax setting. Note that our notion of activation is a little different than in Glauert et al. (1988a), the pattern calculus being at the heart of the matter, and also that we have combined root quiescence and activation in our version of the activation phase, whereas root quiescence is done right at the beginning of the Glauert et al. (1988a) definition. The reason for this is that it yields significant technical simplifications when graph markings are involved in inductive proofs of properties of executions (although the present paper provides no evidence for this, due to its succinctness).

Suppose now that Sel is empty. Then instead of a rewrite, notification takes place.

Definition 2.8 Notification is the process whereby the node marking $\mu(t)$ is changed to ε, and for all arcs $(p_k, t) \in X$, if the arc making $\nu(p)[k]$ is ^, then it is changed to ε, and if the node marking $\mu(p)$ is $\#^n$ (for $n \geq 1$) it is changed to $\#^{n-1}$, with $\#^0$ being understood as $*$. Call the resulting graph Y and let $i_{X,Y}$ be the natural injection.

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

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

Definition 2.10 An **execution** of a system **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 the sequence, G_{i+1} results from G_i by either rewriting or notification at some active node of G_i. Graphs occuring in executions are called **execution graphs**.

By composing the various maps $i_{X,X'}$, $i_{X',X''}$ or $r_{X',X''}$, etc., we can track the history of a node through an execution of the system. We thus have $i_{X,Y}(x)$ as the node which is the copy in Y of $x \in X$, and $r_{X,Y}(x)$ as the node of Y that x got redirected to (with $r_{X,Y}(x) = i_{X,Y}(x)$ where no redirection took place). In fact more is true, since for a rewrite, the $i_{X,Y}(x)$ and $r_{X,Y}(x)$ maps are the crucial elements that enable Y to be characterised as the unique way (up to isomorphism) of completing a certain universal construction. However it would take us too far from the objectives of this paper to discuss this here. See Banach (1991) for an account of this topic. Note that no node is ever destroyed.

So much for the definitions which give us the basic operational semantics of DACTL. The main problem with this semantics from a architectural point of view, is that an entire rewrite or entire notification must take place as a single atomic action. This places quite a burden on a parallel implementaion, when it has to cope with rewrites or notifications that might overlap with other rewrites or notifications. In particular, whatever an implementation does, must in some acceptable way be equivalent to a serialisable sequence of rewrites and notifications.

3 MONSTR

Now we present an abridged TGR language, MONSTR, which curtails some of the problems inherent in the DACTL definition. The definition is in terms of a list of restrictions on the general structure of DACTL.

First, we partition the symbol alphabet **S** into **F** $\cup$ **C** $\cup$ **V**, where **F** consists of **functions** which have rules but which cannot occur at subroot positions of patterns of rules; and **C** and **V**, consisting of **constructors** and **variables** (or **stateholders**), neither of which can occur at root positions of patterns of rules and therefore neither of which have rules; in addition constructors are not permitted to occur as the LHS of a redirection.

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

We insist that there is at least one default rule for every function symbol, and we allow a normal rule to be selected in preference to a default rule whenever either will match, but the latter aspect will be of no significance below.

Definition 3.1 MONSTR graphs, rules and systems must conform to the following list of restrictions.

(1) Symbols have fixed arities, i.e. the map $x \rightarrow A(x)$ factors through $\sigma(x)$, and thus $A(x) = A(\sigma(x))$.

(2) For each $F \in$ **F** there is a subset $M(F) \subseteq A(F)$ such that $k \in M(F)$ iff for any normal rule for F with root r, $\alpha(r)[k]$ is explicit.

(3) For each $F \in$ **F** there is a subset $\Sigma(F) \subseteq A(F)$, at most a singleton, such that if $k \in \Sigma(F)$, then for any normal rule for F with root r, $\sigma(\alpha(r)[k]) \in$ **C** $\cup$ **V**. Otherwise, for explicit $\alpha(r)[l]$, $l \neq k$, $\sigma(\alpha(r)[l]) \in$ **C**.

(4) For each rule, any grandchild of the root is implicit.

(5) For every rule, no implicit node may have more than one parent in the pattern.

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

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

(8) For every rule with root r, $(r, t) \in Red$ for some t.

(9) For every rule, if $(x, y) \in Red$ with $\mu(y) = \varepsilon$ and $y \notin Act$ then y is explicit and $\sigma(y) \in \mathbf{V}$.

(10) For every rule, if $(y, z) \in Red$, then $y \notin Act$ unless $(x, y) \in Red$ for some x.

Despite the size of this collection of restrictions, a useful and expressive model of computation remains. In particular it is trivial to simulate a Turing Machine, and although this fact is of no further interest to us, we outline in 3.3 below how it is done as an example of MONSTR rewriting.

The restrictions on symbols and rules introduced in the preamble to 3.1 are there to impose some discipline on the generality of DACTL; designating symbols to be either functions, which perform rewrites but do nothing else; or constructors, which hold values but do nothing else (in particular they do not get redirected); or stateholders, which as their name implies, hold updatable values and can model standard notions of state.

Restrictions (1) – (5) are largely "geographical" in nature. They ease hardware construction, but also have an impact on problems arising from concurrent executions (see the discussion of the finegrained Monstr semantic model below). Restriction (10) permits a smooth implementation of redirection by overwriting. Restrictions (6) – (9) have rather more immediate consequences; which also happen to aid overwriting.

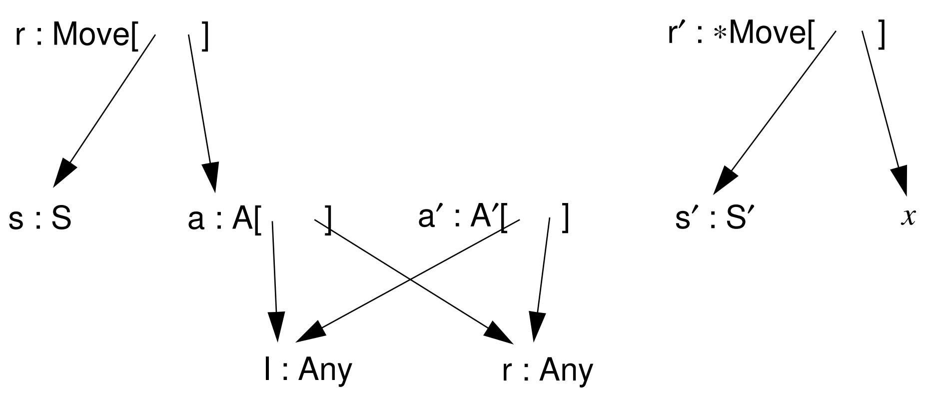
Theorem 3.2 (Fundamental Properties) If a DACTL system obeys restrictions (6) – (9) of definition 3.1 then every execution graph is

(1) balanced,

(2) state saturated.

Example 3.3 **A** Turing Machine consists of a set of quintuples of the form (S, A, S', A', x) where as usual, S and S' are the old and new internal state, A and A' are the old and new tape symbols, and x is one of L, R, or N, representing left right or no movement of the tape head. Fig 1. depicts a typical rule of a MONSTR system simulation of the machine. The function node r : Move, is the root of the pattern for such a rule and represents the read/write head of the machine. Its children are s : S, a constructor representing the current internal state with empty arity, and a : A, a stateholder representing the current tape symbol whose left and right children l and r are implicit nodes intended to match the unspecified left and right neighbours of a : A on the tape. The action of the rule creates a new active Move node r′ ready for the next transition, a new internal state node s′ : S′, and a new tape node a′ : A′. The redirections redirect the old to the new Move node (although this will have no actual effect on the execution graph), and more importantly, a to a′. If, ignoring garbage and the Move and S nodes, the execution graph consists of a representation of the tape as a doubly linked list, this redirection splices the new value into the list, given that the left and right children of a′ are l and r respectively. Note that the second child of r′ is x, which stands for one of l, r or a′, depending on the desired motion of the read/write head.

We assume the tape to be delimited by special left and right endmarkers and there must be a corresponding pair of rules like that just described to deal with the cases where either of these is encountered. The initial rule must set up the initial configuration of the tape with its endmarkers, and an active Move node with out-arcs pointing to the initial internal state and the starting square of the tape. These are rather boring technicalities with which we will not detain the reader.



$Red = \{(\mathsf{r}, \mathsf{r}'), (\mathsf{a}, \mathsf{a}')\},\ Act = \varnothing$

Fig 1.

4 Alternative Semantic Models

Having defined the MONSTR sublanguage of DACTL, we can now explore the consequences of executing MONSTR systems according to different operational semantics. The models we consider are the suspending MONSTR model, the finegrained MONSTR model, fully coercing models, and serialisable weak models.

4.1 The Suspending Monstr Model

The suspending Monstr model addresses one of the simpler problem areas of the DACTL semantic model, namely the fact that a rewrite can proceed independently of the markings of the subroot redex nodes. This is indeed problematic for implementations, because implementations that feature concurrent rewriting of distinct but perhaps overlapping redexes may well want to manipulate the graph in a more finegrained way than the atomicity of the rewriting paradigm would allow, and this would tend to destroy serialisability.

For $F \in \mathbf{F}$ we call $M(F)$ the **map** of F. Since all $M(F)$ arguments of F redexes are explicit, they must be examined during pattern matching. We insist that such arguments are **idle** i.e. have the ε node marking, before rewriting can occur. If this is not the case, then a suspension occurs.

Definition 4.1.1 Let $t \in X$ be an active function node and let $m > 0$ where m is the cardinality of $\{k \in M(\sigma(t)) \mid \mu(\alpha(t)[k]) \neq \varepsilon\}$. Then the **suspension** of t (on its non-idle amputated map arguments) changes $\mu(t)$ to $\#^m$ and for each k such that $k \in M(\sigma(t))$ and $\mu(\alpha(t)[k]) \neq \varepsilon$, $\nu(t)[k]$ is changed to ^. Call the resulting graph Y and let the natural injection be $i_{X,Y}$ as before.

Suspending MONSTR executions are thus sequences of graphs generated from an initial graph by rewrites, notifications and suspensions, and compared to DACTL executions, impose additional dependencies between rewrites. We note that since the root node of a suspension aquires as many # markings as it has out-arcs which aquire a ^ marking, balancedness is preserved and so suspending Monstr executions enjoy the same fundamental properties as DACTL executions.

Theorem 4.1.2 Let G_i be a suspending MONSTR execution graph. Then there is a DACTL execution graph with graph structure isomorphic to that of G_i.

Theorem 4.1.3 Suppose some suspending MONSTR execution terminates in a final graph where all the node and arc markings are ε. Then there is some DACTL execution that produces the same final graph.

Theorem 4.1.4 Suppose for some suspending MONSTR execution, for each execution graph, the subgraph consisting of nodes and arcs having non-ε markings is acyclic. Then if the execution terminates, there is some DACTL execution that produces the same final graph.

4.2 The Finegrained MONSTR Model

Despite its suspension events, the suspending MONSTR model still insists that rewrites are done in a single atomic action, where they occur. The finegrained MONSTR model breaks this atomicity of rewriting down into more primitive steps. On the whole, actions involving both of the pair of nodes at the two ends of an arc, are performed using message passing, and so correspond to more than one atomic action in the model. As the precise details are rather intricate we will content ourselves with a rather more informal presentation than hitherto.

We start with the notifications. In finegrained MONSTR these are split into two sorts of action. **Notification issuing**, which quiesces the root and issues notification messages to the relevant parents; and **notification events** where such messages arrive and alter the node marking from $\#^n$ to $\#^{n\text{-}1}$ (or $*$). We have not said when the relevant arc marking gets altered. In reality it may be changed as part of either type of action, but doing so temporarily breaks one or other of the fundamental properties 3.2. This a technical nuisance, though not serious since parts of execution graphs where the invariant breaks down correspond 1 – 1 with partially completed notifications (or other actions whose finegrained versions also temporarily break the invariant).

Rewrites are broken up in a somewhat more complex manner. If the function labelling the root of the redex has no normal rules, then the rewrite moves immediately to the rewrite action below. Otherwise rewriting starts with **read issuing**, in which if t is the root of the redex and m is the cardinality of *C-args* $= M(\sigma(t)) - \Sigma(\sigma(t))$, then a read message is issued to each *C-args* child of t, $\mu(t)$ is changed to $\#^m$, and for each $k \in$ *C-args*, $\nu(t)[k]$ is changed to ^. This breaks fundamental property 3.2.(b). When such a message arrives at its destination, a **read event** takes place, in which if the argument is idle (say it is the k'th), and a constructor, $\nu(t)[k]$ is changed to ε, and $\mu(t)$ changes from $\#^l$ to $\#^{l\text{-}1}$. If the argument is idle but not a constructor, the same thing happens but in addition a 'FAIL' is recorded at the root of the redex. Here $\#^0$ is $*$ as before and if $\mu(t)$ indeed becomes $*$, then the rewrite is ready to perform the rewrite action below. If the argument is non-idle the read event is retried when the argument notifies. We see how finegrained MONSTR suspensions are a more abstract model of these argument reading preliminaries, which are themselves a more abstract model of what happens at the machine level.

The **rewrite** action performs the heart of the rewrite and consists of three stages, all taking place within the confines of a single atomic action. Firstly, if any FAILs have been recorded for the root of the redex, a default rule rather than the original normal rule is selected for matching and execution. This is because although the children at *C-args* positions of the root may well have been redirected to constructor form by this time, the asynchronous nature of the argument reading process means that the rewriting mechanism is unaware of this fact.

Secondly, assuming no FAILs, the pattern is matched including any stateholder argument $\alpha(t)[\Sigma(\sigma(t))]$. By hypothesis, enough arguments have been read to yield a verdict for any normal rule for $\sigma(t)$. All this proceeds unless the stateholder argument itself is non-idle in which case the rewrite suspends until notification by the stateholder argument.

Thirdly, assuming no FAILs or suspension on the stateholder, a rule is selected in the usual way. The rule to be executed now having been fixed by one means or another, the contractum is built, the redirections are performed, and activation messages are issued to any nodes matched to pattern nodes in the activation set of the rule. Finally the root of the redex can be quiesced unless it turned out to be one of the nodes to be activated. This completes the description of rewrite actions.

The final type of action of interest is the **activation event**, in which an activation message arrives at its destination, and if the node marking of the destination is ε, changes it to $*$, activating the node.

Finegrained MONSTR executions are thus generated by sequences of actions of the kind described above. With sufficient attention to detail the following results can be obtained.

Definition 4.2.1 Let R be a default rule with root r, and suppose the cardinality of $M(\sigma(r))$ is m. Suppose there is exactly one contractum node c, with $\sigma(c) = \sigma(r)$, $\mu(c) = \#^m$, $\alpha(c)[k] = \alpha(r)[k]$ for $k \in A(r)$, $\nu(c)[k] = \hat{}$ for $k \in M(\sigma(r))$, and $\nu(c)[k] = \varepsilon$ for $k \in A(\sigma(r)) - M(\sigma(r))$. Suppose $Red = \{(r, c)\}$ and $Act = \{\alpha(r)[k] \mid k \in M(\sigma(r))\}$. Then R is a **refiring rule**.

Definition 4.2.2 Let R be a normal rule with rule with root r and suppose $\sigma(\alpha(r)[k]) \in \mathbf{V}$ for $k \in \Sigma(\sigma(r))$. Suppose there is exactly one contractum node c, with $\sigma(c) = \sigma(r)$, $\mu(c) = \#$, $\alpha(c)[k] = \alpha(r)[k]$ for $k \in A(r)$,and $\nu(c)[k] = \varepsilon$ for $k \in A(\sigma(r)) - \Sigma(\sigma(r))$ and $\nu(c)[k] = \hat{}$ for $k \in \Sigma(\sigma(r))$. Suppose $Red = \{(r, c)\}$ and $Act = \varnothing$. Then R is a **suspension rule**.

Theorem 4.2.3 Suppose for some system that

(1) every default rule for a symbol with non-empty arity is a refiring rule,

(2) every normal rule either,

   (a) does not match a stateholder (even if its $\Sigma(\sigma(\text{root})) \neq \varnothing$), or

   (b) matches a stateholder (in its $\Sigma(\sigma(\text{root}))$ position) and redirects it to a non-idle node, or

   (c) is a suspension rule.

Suppose also that

(ActF) no activation message ever finds its destination to be an idle function node.

Then if a finegrained MONSTR execution terminates, there is a suspending MONSTR execution of the system that produces the same final graph, *modulo garbage*.

This is an opportune moment to briefly discuss garbage. Since nodes of the graph are never deleted during any execution step, copious quantities of garbage are generated. We define the live nodes of the graph to be *base* live nodes (which are active nodes, and constructors specifically designated as such), or nodes accessible from live nodes via ε-marked arcs, or nodes that can access live nodes via ^-marked arcs. All others are garbage. This definition turns out to be sound.

4.3 Fully Coercing Models

Rewrites in the previous two models wait for active subcomputations to terminate (i.e. notify) before proceeding. This is the idea behind suspension. In this sense the two models are coercing in a rather weak sense. The fully coercing versions of the two models are coercing in a stronger sense, namely that they actively instigate subcomputations when they chance upon inactive subcomputations during pattern matching. Since constructors are regarded as ground objects, and stateholders are ground until some rewrite chooses to redirect them, the only nodes that denote the presence of inactive subcomputations are idle function nodes. Thus in the fully coercing version of suspending MONSTR, the root of a pattern match suspends not only on non-idle $M(\sigma(\text{root}))$ arguments, but also on those $M(\sigma(\text{root}))$ arguments which are idle functions, which are simultaneously made active. In the fully coercing version of finegrained MONSTR, a similar thing happens when a read event encounters an idle function node. Instead of recording a FAILure, the event activates the function and completion of pattern matching is delayed until the child subcomputation notifies.

Of course if no contractum node of any rule of a system is an idle function, then any semantic model that is at least as coercing as suspending MONSTR is fully coercing.

Executions according to coercing models may be viewed as certain executions of a modified rule set (no longer MONSTR though). These can be defined by mapping a pattern of a conventional rule into all its possible truncations, together with all possible configurations of activating unmatched nodes. The intention is to model all possible ways in which part of the pattern may exist, with some of the unmatched nodes being labeled by idle functions which are thus activated.

The major point of significance concerning the fully coercing models follows.

Theorem 4.3.1 For fully coercing finegrained MONSTR, theorem 4.2.3 holds but without requiring the (runtime) condition (ActF).

4.4 Serialisable Weak Models

In all of the previous models, executions have been serial since the semantic models have been derived from the DACTL rewriting model, and rewriting models are conventionally sequential. On the other hand, the ultimate objective is to model the actions of an underlying parallel machine whose operation is in no way sequential. Sequentiallity thus plays two conflicting roles in our study of operational semantics. On the one hand, it is an objective, because we want to reduce the concurrent operation of the parallel machine to some serial rewriting sequence of the original DACTL semantic model if we can; on the other hand it is an obstacle, because for technical simplicity, we would like to work in the graph world as much as possible, but that world only offers us the sequential rewriting paradigm, which thus makes the modelling of concurrent behaviour problematic.

To overcome the obstacle, we cannot avoid looking at the way a putative machine would represent and manipulate the graph. Such an examination would reveal the areas of conflict and hence of potential interference between different execution primitives. One could then lift the criteria necessary for non-interference, to the graph world and study them in the more convenient abstract setting.

For the particular architectural model we have in mind, the Flagship model (Watson et al. (1987a, 1987b)), the problems boil down to the study of the interference of distinct redirections, given a minimal locking strategy for graph nodes. To see this, it is enough to review the definition of the redirection phase, which makes it plain that if the execution graph nodes matched to the LHS and RHS of a redirection are identical, then the redirection is a null action. To enforce this in a world of coded up graph representations with concurrently operating agents modifying the graph, requires both an identity test for node representations, and locking of appropriate parts of the coding to enable redirections to be performed atomically. Both of these are luxuries that an efficient implementation of the rewriting model would rather do without. The study of serialisable weak models thus turns into the search for criteria that ensure that a system will execute safely in the absence of mechanisms in the semantic model to determine identity and enforce atomicity of redirection.

5 Conclusions

The reduction of the atomic semantics of the MONSTR subset of DACTL to a semantics more in keeping with the behaviour of a concrete architectural model such as the Flagship machine turns out to be a complex process. Most people who have attempted the reduction of abstract semantic models to more realistic ones would agree with this view. In fact it is not possible to do the reduction for the full MONSTR sublanguage as theorem 4.2.3 might be taken to imply. It turns out that all points of difference between DACTL and machine semantics give rise to counterexamples (usually quite easy to construct) that yield different results according to which semantic model is used. Nevertheless these counterexamples are usually sufficiently pathological to inspire a search for criteria under which equivalence can be demonstrated, and to date, the search has been a fruitful one. See Banach (1992) for details.

References

Banach R. (1991), DACTL Rewriting is Categorical. *These proceedings.*

Banach R. (1992), MONSTR, *in preparation*.

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

Farmer W.M., Watro R.J. (1989), Redex Capturing in Term Graph Rewriting, *in* Computing with the Curry Chip.

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

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

Kennaway J.R., Klop J-W., Sleep M.R., de Vries F-J. (1990), Transfinite Reductions in Orthogonal Term Rewrite Systems, CWI draft.

Watson P., Watson I. (1987a), Evaluating Functional Programs on the Flagship Machine, *in* proc. FLCA-87, Khan G. *ed*., Springer, LNCS **274** 80 - 97.

Watson I., Woods V., Watson P., Banach R., Greenberg M., Sageant J. (1987b), Flagship: A Parallel Architecture for Declarative Programming, *in* 15th Annual International Symposium on Computer Architecture, Hawaii, A.C.M.