PARALLEL TERM GRAPH REWRITING AND CONCURRENT LOGIC PROGRAMS

R. Banach[a][1], **G. A. Papadopoulos**[b][2]

[a]Computer Science Department, Manchester University,
Manchester, M13 9PL, U. K.
[b]School of Information Systems, University of East Anglia,
Norwich, NR4 7TJ, U.K.

Abstract

General term graph rewriting is a powerful computational model, suitable for implementing a wide variety of declarative language paradigms. Here, we address the problems involved in the implementation, on a loosely-coupled architecture, of an intermediate language based on term graph rewriting, DACTL. In general, such problems are severe, so a subset of this language called MONSTR is defined, free from some of the inefficiencies in the original model (such as an excessive necessity for locking). Superficially, much of the expressiveness of the original model is compromised thereby, especially with regard to the implementation of concurrent logic languages. However, transformation techniques that are valid in the context of declarative language translations, and that map fairly general DACTL rule sets to equivalent MONSTR rule sets without endangering the semantics of the former can be defined. These techniques use shared graph nodes that reflect states of computation, and are similar to the logical variable.

1 Introduction

Rewriting theory is a powerful computational model for implementing declarative languages. Modern functional languages based on equations can be seen as specifying sets of rewrite rules (term rewriting systems [8]) that must be applied to reducible subexpressions (redexes) until a final form (the result) is reached. Where possible, common subexpressions can be shared thus saving unnecessary recomputations; this gives rise to graph rewriting systems ([12]). A combination of these two has led to the development of the term graph rewriting model ([4]) and the associated language DACTL ([5]).

DACTL is a compiler target language. At the detailed operational level it is able to accomodate a variety of (often divergent) features such as lazy evaluation, stream AND-parallelism, committed-choice OR-parallelism, sharing of computations, modelling of atomic transactions, dataflow and shared variable synchronisation. As a CTL, its purpose is to decouple the development of high-level declarative languages from that of novel architectures, providing a useful interface between the two. An implementation of a high level language to DACTL can proceed independently from the development of the underlying machine; indeed, such an implementation would not be restricted to a single machine since DACTL itself is designed to be portable to a number of machines. Possible implementations of functional and concurrent logic languages through the DACTL route are discussed in [6,7,9,10,11]. However, these papers address only the top part of the implementation i.e. the mapping of systems of equations or clauses to equivalent systems of DACTL rewrite rules.

1. Email: banach@cs.man.ac.uk
2. Permanent address: Dept. of Computer Science, University of Cyprus, Nicosia, P.O.B. 537, Cyprus.
Email: george@jupiter.cca.ucy.cy

In this paper, we discuss the mapping of DACTL onto a particular distributed architecture, the Flagship machine ([2,14,15]). In fact the Flagship model doesn't play much of a special role here. More or less any distibuted machine model built round message passing primitives would do, but Flagship provided a particular focus for the work. We show that some of DACTL's features (especially the ones related to atomicity of rewrites), although very powerful, are difficult to implement efficiently on such a machine, making DACTL's very expressiveness, an obstacle to its adoption for such architectures. We then go on to define a subset of DACTL (called MONSTR) that can be supported easily and efficiently by the underlying machine. This subset seems at the outset to seriously compromise DACTL's expressiveness, especially with regard to the modelling of non-determinism, a powerful feature of concurrent logic languages. However, we show how general DACTL rule systems, of the kind needed in the implementation of such languages, can be transformed to MONSTR in such a way that the expressiveness and operational semantics of the original rule systems are not compromised in any way. Our technique is particularly applicable to DACTL based implementations of Parlog and GHC. It uses in a rather unsual way, certain shared nodes (referred to as stateholders) that are able to model computational states, and that also subsume the logical variable. This particularly illustrates the power of the latter in process control and synchronisation.

The rest of the paper is organised as follows. The next section discusses DACTL, and the difficulties associated with its implementation. The following section defines MONSTR and shows how it bypasses these problems. The one after shows how DACTL rule sets generated by a Parlog or GHC to DACTL implementation can be mapped to equivalent MONSTR rule sets using stateholders. The advantages and limitations of our method are discussed in the final section along with further work needed to overcome the latter.

2 DACTL

A DACTL graph is a particular type of directed graph. Graphs are normally described using set theoretic machinery, but in this paper our inclination will be more towards concrete syntax. We trust that readers will not be inconvenienced by the slight lack of strict mathematical rigour that this is prone to lead to. Thus the graph contains nodes, and each node has associated with it a number of objects: (1) its unique node identifier that distinguishes it from other nodes which might be otherwise identical, (2) its node symbol, (3) its node marking (one of *****, $\#^n$ ($n \leq 1$), or empty), (4) its sequence of successors, a finite sequence of arcs to other nodes. Each arc in turn has (5) an arc marking (either **^** or empty). We write such a node in the form **x:S[^a** . . . **z]** where **x** is the nodeid, **S** is the symbol and **a** . . . **z** are nodeids of the successors, the arc to the first of which carries the **^** arc marking. Given a collection of such objects, provided the nodeid of each arc refers to a node in the collection, we have a representation of a graph. Mathematically therefore, our graphs are quintuples consisting of the set of nodes (1), and the collection of four attributes pertaining to each node.

Such graphs are rewritten according to DACTL rules. Here is an example:

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

On the left we have the pattern. This is just a rooted DACTL graph written without any of the markings, and allowing the node symbols of certain nodes (e.g. **a** and **b**) to be unspecified; alternatively the special symbol **Any** may be used to label such nodes. Note that certain nodeids have been suppressed without ill effect, by nesting the node definitions. Rule execution proceeds by firstly matching the pattern. This consists of selecting an active node of the graph (one marked with a *****), and finding a rule whose pattern matches the graph at that point. Pattern matching takes symbols into account but ignores markings, and allows unspecified (or **Any**) nodes to match anything. Actually one is allowed more, since there are three pattern operators which can be used to steer the pattern matching. Thus **(P1+P2)** matches if either (sub)-pattern **P1** or **P2** matches, **(P1&P2)** demands that both match, and **(P1-P2)** demands that **P1** matches but **P2** does not.

Having matched the pattern i.e. found a redex, the ***** marking is removed from the root of the redex, and then the RHS of the rule specifies the new nodes to be built by the rule and how they are to be linked into

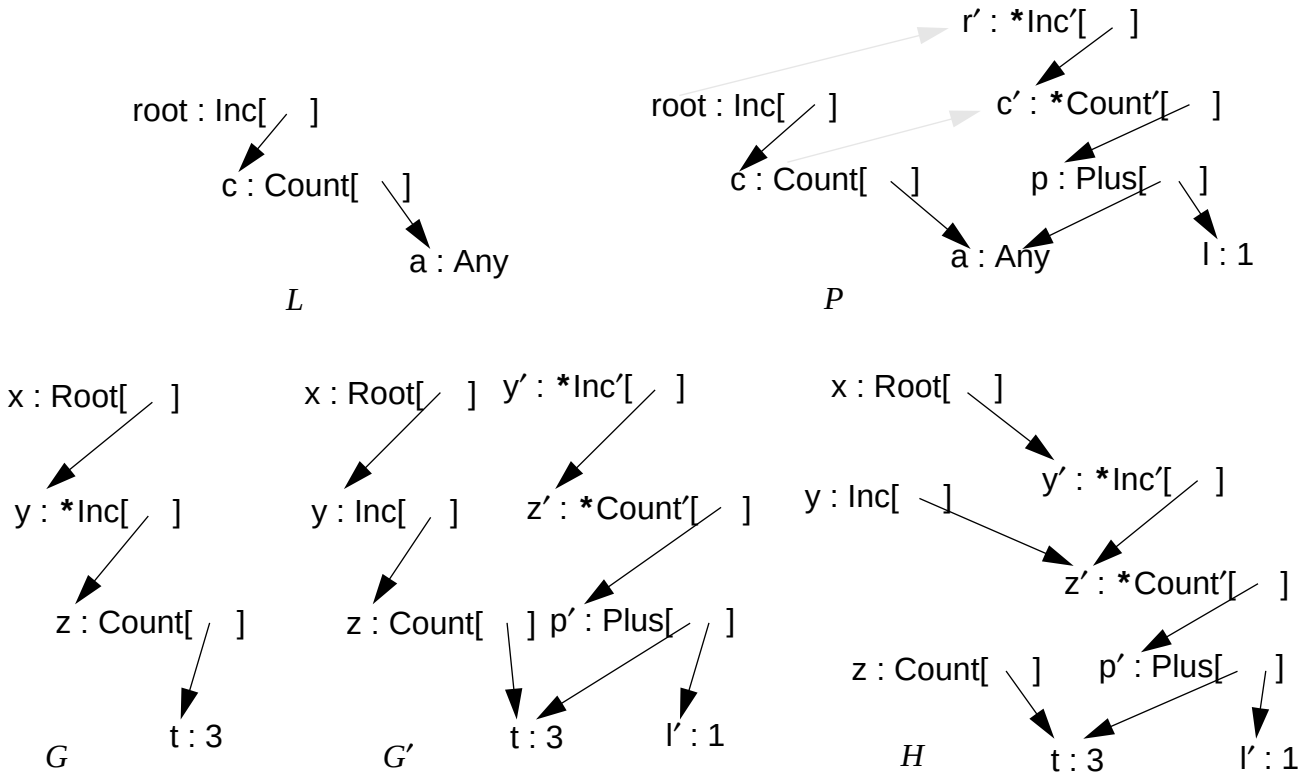
the existing graph (in our example, a node with symbol and marking **#G** with two successors (the nodes matched by **a** and **b**) and a new node **SUCCEED**). The graph structure is further changed by the redirections, which specify that all arcs pointing to the LHS of the redirection be swung over to point to the RHS of the redirection, along with their arc markings. All such redirections are to be performed in parallel. We write a redirection using **=>** which means that the node matched by the root of the pattern is to be redirected to the node following the **=>**. Also **x:=y**, which means that all arcs pointing to (the image of) **x**, are swung over to point to (the image of) **y**, and this syntax allows us to redirect subroot nodes, (in our example the **Var** node is redirected to **SUCCEED**). Finally, there come the activations, which allow us to change the markings on nodes matched by the pattern if they are unmarked, (in our example the node matched by **b**). We write these as e.g. ***q** if the node **q** is otherwise not mentioned on the RHS of the rule.

Rule selection may be done entirely by using the pattern operators, and so the standard rule separator in DACTL is **|**, the non-deterministic selector. However syntactic sugar in the form of a sequential selector, **;** is also provided for convenience, with **|** binding more tightly than **;**. Of course matching may fail, in which case a process called notification occurs. The active ***** marking is removed from the root and all notification arcs pointing at the root (those having the **^** marking), have their **^** markings removed. Whenever the parent of such an arc has a **#**^{n} marking, this is replaced with **#**$^{n-1}$ (where **#**^{0} is to be understood as *****). We thus see that the **#** and **^** markings are useful for suspending a computation until one or more subcomputations have notified.

We examine a simple example of rewriting. Consider the rule:

```
Inc[c:Count[a]] => *Inc′[c′:*Count′[Plus[a 1]], c:=c′
```

The diagrams below provide an illustration of what happens when this rule is used for rewriting. The left hand side of the rule is illustrated by the graph *L*. The graph *P* shows the left hand side included in the graph of the rule considered as a whole. Note that here (and also subsequently), the faint dotted arrows of *P* represent redirections. The rule is applied to the graph *G* which obviously contains an active node which is the root of the redex. The graph *G′* shows the situation once the active marking has been removed from the root and the contractum nodes and arcs have been glued to the redex. Finally the graph *H* shows the state of affairs when the redirections have been performed. There are no activations to worry about in this rule so we are done.



Now consider the following ruleset which shows more of the pattern matching features of DACTL:

```
Select_2_out_of_3[1 2 Any] => *12|
Select_2_out_of_3[Any 2 3] => *23|
Select_2_out_of_3[1 Any 3] => *13;
Select_2_out_of_3[p1:(Var+1) p2:(Var+2) p3:(Var+3)]
        => #Select_2_out_of_3[^p1 ^p2 ^p3];
Select_2_out_of_3[Any Any Any] => *FAIL;
```

Note that any of the first three rules can be selected non-deterministically. However, the **;** at the end of the third rule guarantees that the fourth rule will be tried for matching only when attempted matchings for the rules above it have failed (similarly for the **;** at the end of the fourth rule). Denoting the patterns of the first four rules by **P1**, **P2**, **P3** and **P4**, matching the fourth rule means matching the composite pattern **(P4-(P1+P2+P3))**. Note also the use of pattern union **+** in the fourth rule to check the validity of the procedure's arguments. We will say more about this ruleset shortly when we define MONSTR.

Aside from what has been stated above, DACTL makes no restrictions whatsoever on the structure of rules. Furthermore, DACTL semantics dictates that given an active node in the graph, all of the pattern matching, rule selection, contractum building, redirections, and activations must be performed in a single atomic action, serialisable with all other such atomic rewrites. This is a pretty tall order for an implementation built on a distributed parallel machine, where the graph nodes are scattered around the processing elements, which in turn communicate by message passing — such as Flagship. The only way to guarantee conformity to DACTL semantics in such a situation is to lock every potential pattern node for each rewrite. This is clearly a ludicrous proposition.

3 MONSTR

The Flagship strategy approaches the problem of potentially profligate locking by dramatically curtailing the freedom of general DACTL and gives rise to a sublanguage of DACTL called MONSTR [2,3]. The chief restrictions defining MONSTR, and the motivations behind them are the following:

1. All symbols must be functions, constructors or stateholders. Functions alone may have rules or root redirections. Stateholders alone may have non-root redirections. The root must be redirected in every rule.
2. Symbols must have fixed arities and each function must have a fixed matching template which specifies which (if any — there may be at most one) argument position may be a stateholder, and which others must contain constructors. Only one level matching is permitted, and nodeid equality of **Any** nodes is outlawed.
3. All nodes built by the contractum must be balanced, i.e. if there are *n* notification arcs out of a node, then the node marking must be $\#^n$ and vice versa.
4. No notification arc may point to an idle node (i.e. one with an empty node marking), unless the node is a stateholder.

Condition 1 significantly restricts the potential for undisciplined DACTL nodes to do almost anything, by allowing only certain behaviours for nodes. Condition 2 is a severe restriction on the pattern matching capabilities of DACTL necessitated by the asynchronous nature of Flagship machine semantics. Given that the graph is scattered round the system, the root of a rewrite, which instigates the pattern matching, must either not care at all about what lies at the remote end of an argument pointer, (i.e. it must be an **Any** node), or must demand that it is a constructor to be certain that the arc to it will not change after it has been inspected asynchronously, a possibility that would severely compromise atomicity. MONSTR in fact also permits default rules, of the form

```
F[a:Any  . . . z:Any] => RHS
```

that can be used to ensure that a function node **F** will always be able to match *some* rule, e.g. the last rule of **Select_2_out_of_3**. This just leaves the problem of pattern matching a stateholder argument, which certainly *may* change after inspection. The solution here is to transport the root of the rewrite to the processor containing the stateholder, after which the rest of the arguments may be inspected. The matching of the stateholder and prosecution of the rewrite may then take place in a single atomic action on a single processor. This is possible because of another difference in the Flagship and DACTL semantic models. Flagship refuses to attempt to pattern match any non-idle argument. The root of a rewrite therefore, being active, may be moved onto another processor without harming any other rewrite. A suspended indirection is left in its place. Finally, after the rewrite has happened, activations (or notifications) are instigated by message passing. Conditions 3 and 4, prevent the node and arc markings from easily being used as an information channel. Condition 4 is, strictly speaking, a run-time restriction, but it can be easily enforced by demanding that the contractum of every rule adheres to it and that all redirections are to non-idle nodes unless they are explicitly to a stateholder. In fact our **Inc** rewriting example was a pristine instance of the behaviour of a MONSTR rule.

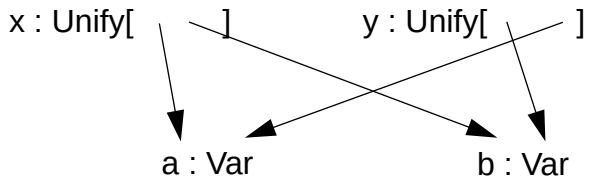
Returning however to the **Select_2_out_of_3** example, we see that it breaks the MONSTR restrictions in two main ways. Firstly, there is no common matching template for the non-default rules. Secondly, the fourth rule creates an unbalanced node in its RHS; **Select_2_out_of_3** will be activated when any one of its arguments is instantiated by other processes assumed to be running in parallel with this one. In general, unbalanced nodes provide a powerful synchronisation mechanism and are a means for expressing non-determinism in DACTL. In the next section we will see how to circumvent the fact that they are forbidden by MONSTR, without losing non-determinism.

MONSTR may thus be viewed in two different lights. Firstly as a sublanguage of DACTL, and secondly as this sublanguage together with the Flagship execution model. The question arises as to the equivalence of these views. There is certainly not space here to discuss this problem properly. Suffice it to say that experience has shown that reasonable programs, for example those produced by the type of concurrent logic language translators considered in this paper, do not suffer from the pathology of yielding inequivalent results when considered as programs running under the two different semantic models. A more detailed treatment of these issues may be found in [1,3]. This experience is particularly heartening in view of the fact that while MONSTR was developed by one of the authors (RB), the logic language translators were developed by the other (GAP), essentially independently, with little more as interface between the two activities than the list of conditions quoted above. This seems to indicate that the MONSTR paradigm is a semantically robust concept and represents a natural way of using graph rewriting as a framework for solving real programming problems.

This good news does not however mean that there are no pitfalls to watch for in special circumstances, in particular the problem of locking. Consider a MONSTR rewrite. The root is in principle locked during rewriting since it is an active node and as such cannot participate in other rewrites (any such other rewrite will block pending the completion of the rewriting of its argument). The stateholder argument is also in principle locked as the rewrite takes place on its processing element and the hardware can easily take suitable precautions. Constructor arguments need no locking as they are purely read-only nodes. This leaves the **Any**s. Must they be locked? Unfortunately, the answer in principle is yes. Consider the following rule, which might well arise as part of a MONSTR implementation of unification.

```
Unify[a:Var b] => *SUCCEED, a:=b
```

We assume that Var is a stateholder. Now suppose we have the graph:



If redirection to a pattern node is implemented by overwriting the redirected node with an indirection to the other, and both rewrites at **x** and **y** proceed without reference to their **Any** arguments, as they would wish to do in a minimal locking regime, then both **a** and **b** would be overwritten with indirections to each other. This is not DACTL semantics.

Does this mean that Flagship should lock **Any** nodes? In reality the answer is no. The example quoted is about the least contrived that can be invented to display this phenomenon. Even so, it assumes two competing unification processes working on the same term graph. This is hardly in the spirit of conventional unification (contrast however, this approach with the problem of commitment via atomic unification in FCP as discussed in the fifth section). Flagship therefore does not lock its **Any**s and avoids paying the performance penalty of doing so. It is considered the responsibility of the programmer or translation system to assure itself that the difficulties just described do not arise.

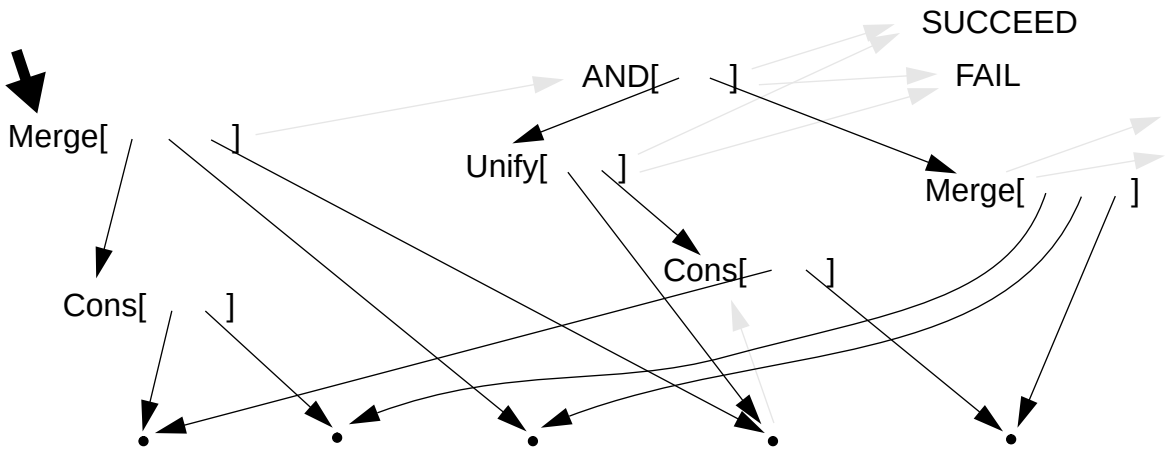
4 Mapping DACTL to MONSTR in Concurrent Logic Programs

The rest of this paper is concerned with the problem of mapping DACTL programs to equivalent MONSTR ones. In general this is an insuperable problem if we discount the obvious but rather puny option of simply coding up a DACTL interpreter in MONSTR. However in the special environment of logic programming, where the programs have considerably more discipline and structure than arbitrary programs which one could write in the DACTL paradigm, there is considerable scope for a systematic, smooth and efficient translation. It is this that we will be concerned with here. Because of space considerations, we will illustrate the main points involved by discussing the behaviour of a series of examples. A more formal translation scheme is implicit in the implementation built by the second author and is described in his doctoral thesis ([11]).

In the sequel we use a Kernel Parlog- or GHC- like syntax. The techniques apply to unsafe GHC programs as well; see, however, the discussion at the end of this section on the implementation of the pointer equality primitive necessary for this. The following non-deterministic merge program illustrates some of the problems that we face in mapping unrestricted DACTL rule sets to equivalent MONSTR ones.

```
merge([U|X],Y,Z) <- Z=[U|Z1], merge(X,Y,Z1).
merge(X,[V|Y],Z) <- Z=[V|Z1], merge(X,Y,Z1).
merge([],Y,Z) <- Z=Y.
merge(X,[],Z) <- Z=X.
```

The general idea for translating programs like the above to DACTL rulesets is to maintain two separate threads of redirections in the computational graph. One thread, implemented primarily by root redirections, looks after the current state of success or failure of the computation. The other thread, implemented by non-root redirections, takes care of the intantiation of logic variables (modelled by stateholders labelled **Var**), via calls to the rules for **Unify**. The diagram below illustrates this schematically. The intention is



to show how an instance of **merge** rewrites to a conjunctive node **AND**, which monitors the progress of a unification and of a newly spawned **merge**. Both the conjunction and unification (and the new **merge** too) will in time rewrite to either **SUCCEED** or **FAIL**. The fat arrow shows where this process starts.

With this insight, and using the techniques described in [10], the translation to DACTL of the above **merge** example is as follows:

```
Merge[Cons[u x] y z] => #AND[^b1 ^b2],
        b1:*Unify[z Cons[u z1:Var]],
        b2:*Merge[x y z1]|
Merge[x Cons[v y] z] => #AND[^b1 ^b2],
        b1:*Unify[z Cons[v z1:Var]],
        b2:*Merge[x y z1]|
Merge[Nil y z] => *Unify[z y]|
Merge[x Nil z] => *Unify[z x];
(Merge[p1 p2 p3] & (Merge[Var Any Any]+Merge[Any Var Any]))
        => #Merge[^p1 ^p2 p3];
Merge[Any Any Any] => *FAIL;
```

This however, is not a valid MONSTR program for the following reasons:

(i) more than one node in the LHS of **Merge** may be a stateholder violating restriction 2;

(ii) the LHS of the fifth rule has pattern operators and therefore violates the fixed matching template requirement of restriction 2; (N.B. Note that the use of the **&** pattern operator in this rule is to introduce names for the arguments of the root, so that they can be refered to in the RHS of the rule; this is a technicality to do with the notion of scope in DACTL patterns);

(iii) the RHSs of the first, second and fifth rules use unbalanced nodes violating restriction 3.

The use of unbalanced nodes in the RHS of the fifth rule ensures that **Merge** will be activated when any of its input arguments is instantiated. **AND** also has unbalanced nodes to detect early failure and kill any remaining computation in the same conjunction; this is explained in detail in [10]. A simple definition of **AND** is shown below:

```
AND[SUCCEED SUCCEED ... SUCCEED] => *SUCCEED;
AND[p1:(Any - FAIL) p2:(Any - FAIL) ... pn:(Any - FAIL)]
        => #AND[^p1 ^p2 ... ^pn];
AND[Any Any ... Any] => *FAIL;
```

The general idea in overcoming the MONSTR restrictions and still being able to express non-determinism, stems from the following idea. Consider a process *A* communicating with *n* other processes B_1 to B_n, in a non-deterministic way. *A* will take appropriate action when it receives some feedback from one or more of the *B* processes. Furthermore, if all the B_i have terminated, *A* may need to take some default action. In the context of concurrent logic languages, *A* can thus be performing input unification, monitoring the execution of an OR-disjunction of guards, or monitoring the execution of an AND-conjunction of goals. *A* will be monitoring its input until it has sufficient information to, say, commit to the appropriate clause (as in the first two cases), or report success/failure (as in the third). Typical default actions for *A* here would be to report failure (for the first two cases), and success (for the third). In the last two cases, a mechanism for terminating unnecessary computations (if some OR-disjunction has succeeded or some AND-conjunction has failed) may be invoked for efficiency reasons.

To express this scenario in MONSTR terms, *A* will be replaced by n+1 processes, A_1 to A_{n+1}, and a stateholder node. Each A_i process ($1 \le i \le n$) is responsible for the outcome of one B_i process; in addition, all A_i processes share the stateholder node which they use for synchronisation and/or communication. The last process A_{n+1}, will be activated only to take the corresponding default action of *A*. The non-deterministic

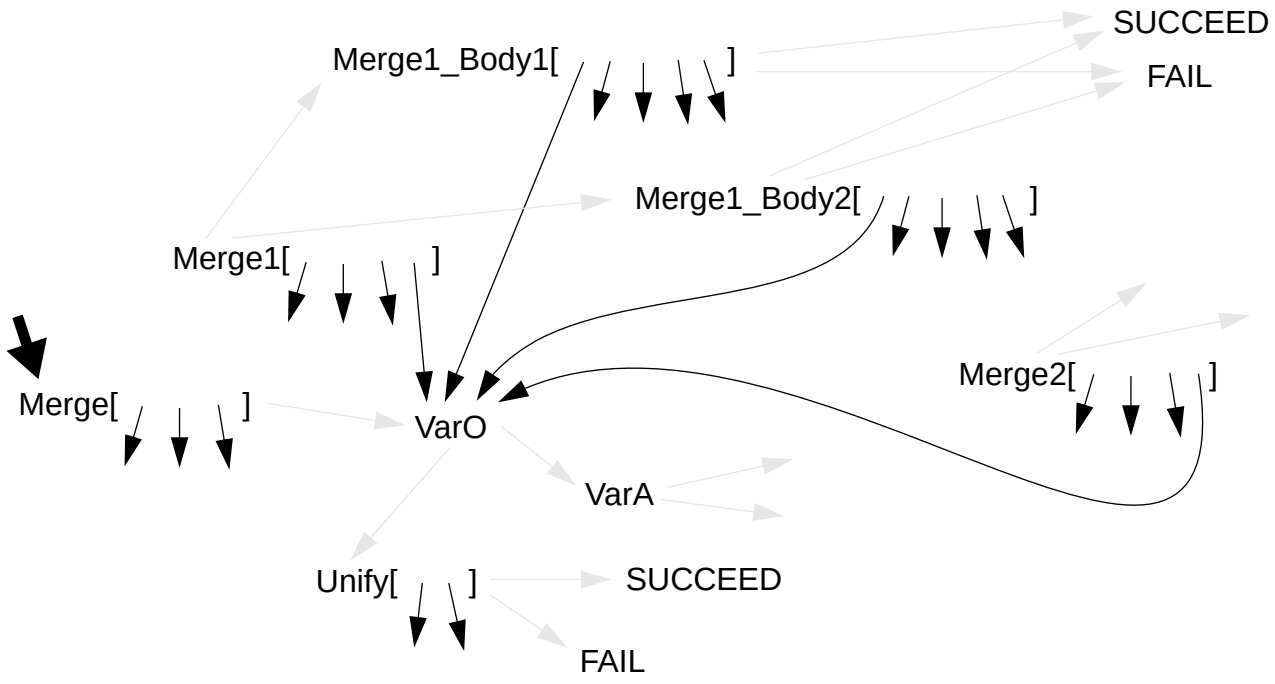
computation is now represented as a piece of graph with a stateholder node at the top which is shared by a number of processes below it. One of these processes will eventually redirect it to the result of the computation. With these considerations in mind, the previous DACTL program for **merge** can be translated to MONSTR as follows:

```
Merge[p1 p2 p3] => result:VarO,
        ##OR2[^o1 ^o2 result],
        o1:*Merge1[p1 p2 p3 result],
        o2:*Merge2[p1 p2 p3 result];

Merge1[Cons[u x] y z result] => Merge1_Body1[result u x y z]|
Merge1[Nil y z result] => *Merge1_Body2[result y z]|
Merge1[p1:Var p2 p3 result] => #Merge1[^p1 p2 p3 result];
Merge1[Any Any Any Any] => *FAIL;

Merge1_Body1[result:VarO u x y z] => *SUCCEED,
        result:=VarA,
        ##AND2[^b1 ^b2 result],
        #Is_SUCCEED[^b1 result],
        #Is_SUCCEED[^b2 result],
        b1:*Unify[z Cons[u z1:Var]],
        b2:*Merge[x y z1];
Merge1_Body1[Any Any Any Any Any] => *FAIL;

Merge1_Body2[result:VarO y z] => *SUCCEED, result:=*Unify[z y];
Merge1_Body2[Any Any Any] => *FAIL;

Merge2[x Cons[v y] z result] => *Merge2_Body1[result x v y z]|
Merge2[x Nil z result] => *Merge2_Body2[result x z]|
Merge2[p1 p2:Var p3 result] => #Merge2[p1 ^p2 p3 result];
Merge2[Any Any Any Any] => *FAIL;

Merge2_Body1[result:VarO x v y z] => *SUCCEED,
        result:=VarA,
        ##AND2[^b1 ^b2 result],
        #Is_SUCCEED[^b1 result],
        #Is_SUCCEED[^b2 result],
        b1:*Unify[z Cons[v z1:Var]],
        b2:*Merge[x y z1];
Merge2_Body1[Any Any Any Any Any] => *FAIL;

Merge2_Body2[result:VarO x z] => *SUCCEED, result:=*Unify[z x];
Merge2_Body2[Any Any Any] => *FAIL;
```

In the above example, **Merge**'s input unification is performed by **Merge1** and **Merge2** which share the overwritable node **result**. The latter two processes work quite independently from each other until they are about to commit to some body. Then they check whether **result** has been overwritten by some other process by checking if its pattern is still **VarO**. If it is, then they either overwrite it with a single body call (as in the case of **Merge1_Body2** and **Merge2_Body2**), or with another overwritable (**VarA**) if there is a conjunction of body calls, and apply the same techniques to the body conjunction. Note that the atomicity of the single non-root redirection allowed in MONSTR guarantees the mutual exclusion required in commitment. The node **result** is therefore used to: (i) provide a communication channel between the cooperating processes, (ii) store the overall result of the computation, (iii) provide a critical region in the commitment phase. In addition, it can be redirected to: (i) a simple data value (e.g. **SUCCEED**, **FAIL**, etc.), (ii) another stateholder node, (iii) a function call. This is a very powerful mechanism, subsuming the Log-

ical Variable, and encapsulating an indivisible quantum of computational state. It is for this reason that we choose to call such overwritable nodes *Stateholders*.

We can give some idea of how these rules behave in another schematic.



To complete the above example, here are the definitions of the auxiliary functions **ORi**, **ANDi** and **Is_SUCCEED**:

```
ORi[FAIL FAIL ... FAIL result:VarO] => *FAIL, result:=*FAIL;

ANDi[SUCCEED SUCCEED ... SUCCEED result:VarA] => *SUCCEED,
        result:=*SUCCEED;
ANDi[Any Any ... Any Any] => *FAIL;

Is_SUCCEED[FAIL result:VarA] => *FAIL, result:=*FAIL;
Is_SUCCEED[Any Any] => *SUCCEED;
```

The next example shows how we overcome the restriction of one level matching and multiple argument input unification:

```
m([f(U)|X],[g(V)|Y],R) <- g1(U,V,R1) | b1(X,Y,R1,R).
m([f(U)|X],[h(V,W)|Y],R) <- g2(U,V,W,R2) | b2(X,Y,R2,R).
```

A possible translation to DACTL follows:

```
M[Cons[F[u] x] Cons[G[v] y] r] => #M_Commit1[^*g1 x y r1 r],
        g1:*G1[u v r1:Var]|
M[Cons[F[u] x] Cons[H[v w] y] r] => #M_Commit2[^*g1 x y r2 r],
        g1:*G2[u v w r2:Var];
M[p1:Var p2:Var p3] => #M[^p1 ^p2 p3]|
M[p1:Var Cons[v:(Var+G[Any]+H[Any Any]) y] p3]
        => #M[^p1 ^#Cons[^v y] p3]|
M[Cons[u:(Var+F[Any]) x] p2:Var p3] => #M[^#Cons[^u x] ^p2 p3]|
M[Cons[u:(Var+F[Any]) x] Cons[v:(Var+G[Any]+H[Any Any]) y] p3]
        => #M[^#Cons[^u x] ^#Cons[^v y] p3];
M[Any Any Any] => *FAIL;
```

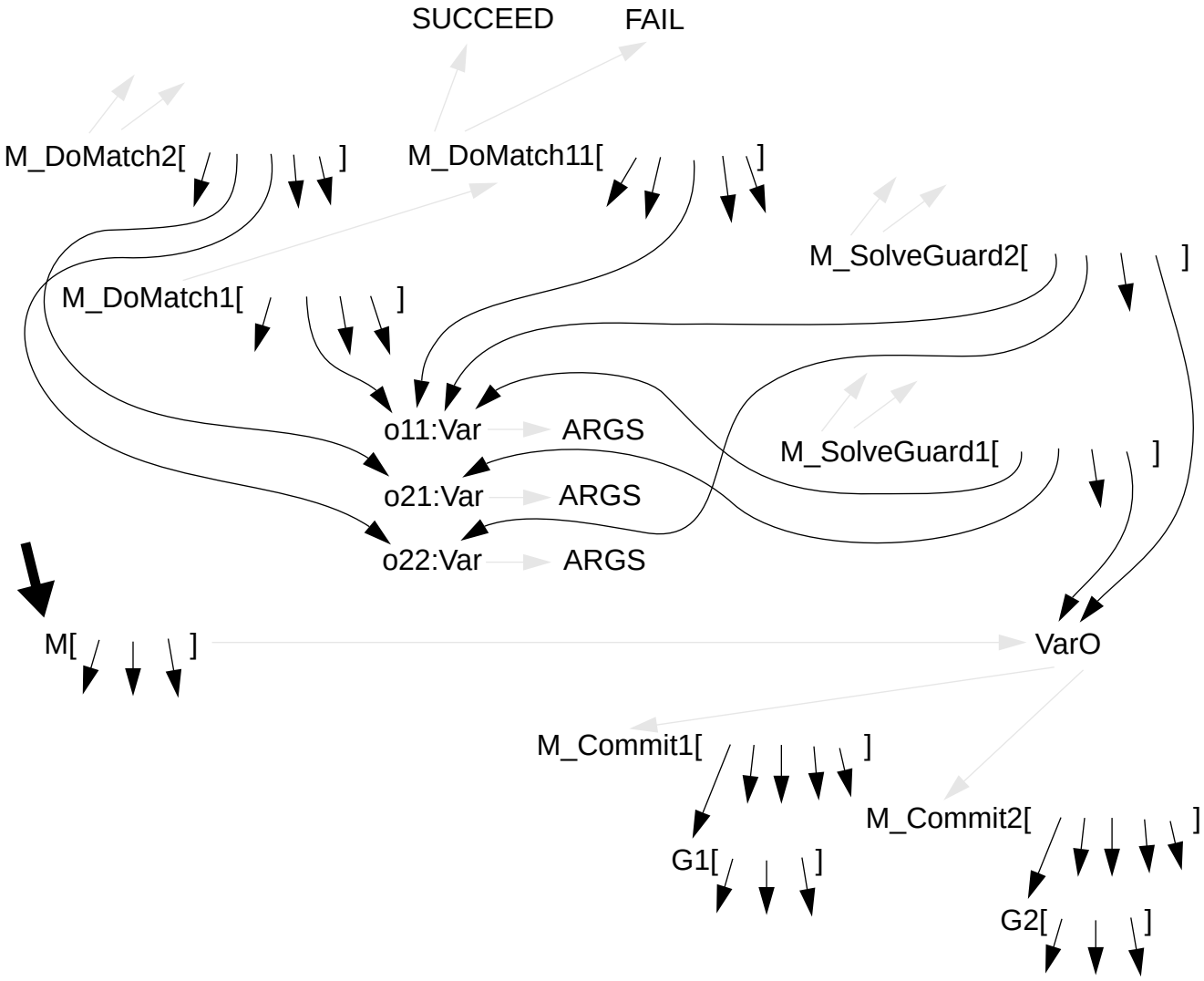
```
M_Commit1[SUCCEED x y r1 r] => *B1[x y r1 r]|
M_Commit1[FAIL Any Any Any Any] => *FAIL;
M_Commit2[SUCCEED x y r2 r] => *B2[x y r2 r]|
M_Commit2[FAIL Any Any Any Any] => *FAIL;
```

Note that DACTL semantics guarantees that input unification will be order independent. The use of so many suspension rules stems from the need to cover every possible legal case for suspension. Although the available pattern operators (and in particular **+**) help in collapsing a number of suspension patterns into one, this number is generally bounded only by all possible combinations of uninstantiated and partially instantiated input arguments, resulting in a large number of suspension rules. A different technique to perform deep pattern matching is to use a set of cooperating processes, each working on a single input argument, thus trading suspension rules for rewrites. Note, however, that this technique is not less efficient that the previous one; matching on deep patterns can be as expensive as performing rewrites. In addition, the resultant code is MONSTR:

```
M[p1 p2 p3] => result:VarO,
        ##Match2[^m1:Var ^m2:Var result],
        ##M_SolveGuard1[^o11:Var ^o21:Var p3 result],
        ##M_SolveGuard2[^o11 ^o22:Var p3 result],
        *M_DoMatch1[p1 o11 m1 m2],
        *M_DoMatch2[p2 o21 o22 m1 m2];

M_SolveGuard1[ARGS2[u x] ARGS2[v y] r result:VarO] => *SUCCEED,
        result:=#M_Commit1[^*G1[u v r1:Var] x y r1 r];
M_SolveGuard2[ARGS2[u x] ARGS3[v w y] r result:VarO] => *SUCCEED,
        result:=#M_Commit2[^*G2[u v w r2:Var] x y r2 r];

M_DoMatch1[p1:Var o11 m1 m2]
        => #M_DoMatch1[^p1 o11 m1 m2];
M_DoMatch1[Cons[f x] o11 m1 m2]
        => *M_DoMatch11[f x o11 m1 m2];
M_DoMatch1[Any Any m1 m2] => *FAIL,
        *ASSIGN[m1 FAIL],
        *ASSIGN[m2 FAIL];

M_DoMatch11[f:Var x o11 m1 m2]
        => #M_DoMatch11[^f x o11 m1 m2];
M_DoMatch11[F[u] x o11 m1 m2]
        => *SUCCEED, *ASSIGN[o11 ARGS2[u x]];
M_DoMatch11[Any Any Any m1 m2] => *FAIL,
        *ASSIGN[m1 FAIL],
        *ASSIGN[m2 FAIL];

M_DoMatch2[p2:Var o21 o22 m1 m2]
        => #M_DoMatch2[^p2 o21 o22 m1 m2];
M_DoMatch2[Cons[gh y] o21 o22 m1 m2]
        => *M_DoMatch21[gh y o21 o22 m1 m2];
M_DoMatch2[Any Any Any m1 m2] => *FAIL,
        *ASSIGN[m1 FAIL],
        *ASSIGN[m2 FAIL];

M_DoMatch21[gh:Var y o21 o22 m1 m2]
        => #M_DoMatch21[^gh y o21 o22 m1 m2];
M_DoMatch21[G[v] y o21 o22 m1 m2]
        => *SUCCEED, *ASSIGN[o21 ARGS2[v y]];
```

```
M_DoMatch21[H[v w] y o21 o22 m1 m2]
        => *SUCCEED, *ASSIGN[o22 ARGS3[v w y]];
M_DoMatch21[Any Any Any Any m1 m2] => *FAIL,
        *ASSIGN[m1 FAIL],
        *ASSIGN[m2 FAIL];

Matchi[FAIL FAIL ... FAIL result:VarO] => *FAIL, result:=*FAIL

ASSIGN[v:Var expr] => *SUCCEED, v:=*expr;
```

Each of the **DoMatch** processes works on a single argument; the result is communicated to the **Solve-Guard** processes which will be activated when all the required input arguments have been instantiated sufficiently. Failure to match a legal pattern is detected and reported immediately. By using some extra processes (not shown here) the process configuration can be short-circuited so that failure to match aborts any remaining computations. Again the essentials are illustrated in the schematic below.



Note that in this case the input patterns are non-overlapping. For procedures having both guards and overlapping (i.e. non-deterministic) input patterns as in **Merge**, a combination of the techniques used in the above two examples is required. Finally, note that a more efficient translation for the above example is possible by performing the matching of input arguments sequentially (say, left to right), collecting matched data structures in the process and suspending where necessary. The advantage here is that computation is more coarse-grained; the disadvantage, of course, is that unification becomes order dependent.

The final example concerns the transformation of programs exhibiting OR-parallelism; here we also show the extra processes required for killing unnecessary work.

```
tree_search(Key,Value,t(_,p(Key',Value'),_))
        <- Key==Key' | Value=Value';
tree_search(Key,Value,t(X,_,Y))
        <- tree_search(Key,Value',X) | Value=Value'.
tree_search(Key,Value,t(X,_,Y))
        <- tree_search(Key,Value',Y) | Value=Value'.
```

Translation to DACTL gives

```
Tree_search[STOP Any Any Any] => STOP;
Tree_search[c key value T[x P[key' value'] y]]
        => #Tree_search_seq[c ^g key value value' x y],
        g:*Eq[key key'];
Tree_search[c key value tree:Var]
        => #Tree_search[^c key value ^tree];
Tree_search[Any Any Any Any] => *FAIL;

Tree_search_seq[STOP Any Any Any Any Any Any] => STOP;
Tree_search_seq[Any SUCCEED Any value value' Any Any]
        => *Unify[value value']|
Tree_search_seq[c FAIL key value value' x y]
        => #Tree_search1[^c c1:Var ^g1 ^g2 value value1 value2],
        g1:*Tree_search[c1 key value1:Var x],
        g2:*Tree_search[c1 key value2:Var y];

Tree_search1[STOP c1:Var Any Any Any Any Any] => STOP, c1:=*STOP;
Tree_search1[c c1:Var SUCCEED Any value value1 Any]
        => *Unify[value value1],
        c1:=*STOP|
Tree_search1[c c1:Var Any SUCCEED value Any value2]
        => *Unify[value value2],
        c1:=*STOP|
Tree_search1[Any Any FAIL FAIL Any Any Any] => *FAIL;
Tree_search1[c c1 g1 g2 value value1 value2]
        => #Tree_search1[^c c1 ^g1 ^g2 value value1 value2];
```

Note the use of unbalanced nodes in **`Tree_search1`** which are used not only to commit non-deterministically to one of the last two clauses for **`tree_search`** but also to kill any remaining computation in the remaining guard. The translation of the above program to MONSTR that preserves this functionality follows (note that we have cheated slightly in allowing one of the **`Tree_search'`** rules below to do two-level pattern matching for the sake of a little brevity).

```
Tree_search[c key value tree] => result:Var,
        #KILL[^c result],
        *Tree_search'[c key value tree result];

Tree_search'[c key value T[x P[key' value'] y] result]
        => *ASSIGN[result
            Tree_search_seq[c key key' value value' x y]];
Tree_search'[c key value tree:Var result]
        => #Tree_search'[c key value ^tree result];
Tree_search'[Any Any Any Any result] => *ASSIGN[result FAIL];
```

```
Tree_search_seq[c key key' value value' x y]
        => result:Var,
        #KILL[^c result],
        #Tree_search_seq'[c ^*Eq[key key']
            key key' value value' x y result];

Tree_search_seq'[Any SUCCEED Any Any value value' Any Any result]
        => *ASSIGN[result Unify[value value']];
Tree_search_seq'[c FAIL key Any value Any x y result]
        => *ASSIGN[result Tree_search1[c key value x y]];

Tree_search1[STOP Any Any Any Any] => *STOP;
Tree_search1[c key value x y] => result:VarO,
        ##OR2[^g1 ^g2 result],
        #KILL[^result c1:Var], #KILL[^c c1],
        #Tree_search1_guard1[^g1 value value1 result],
        #Tree_search1_guard2[^g2 value value2 result],
        g1:*Tree_search[c1 key value1:Var x],
        g2:*Tree_search[c1 key value2:Var y];

Tree_search1_guard1[SUCCEED value value1 result:VarO] => *SUCCEED,
        result:=*Unify[value value1];
Tree_search1_guard1[Any Any Any Any] => *FAIL;

Tree_search1_guard2[SUCCEED value value2 result:VarO] => *SUCCEED,
        result:=*Unify[value value2];
Tree_search1_guard2[Any Any Any Any] => *FAIL;

KILL[Any s:Stateholder] => *STOP, s:=*STOP;

ASSIGN[v:Var expr] => *SUCCEED, v:=*expr;
```

As before, we attempt to indicate the flow of a computation governed by these rules in the schematic on the next page. Note here the use of the **KILL** processes that take the responsibility of monitoring the arrival of any kill signals. Note again the use of the stateholder **result** in allowing only one process to commit. We hope the reader has by now appreciated the functionality hidden within an unbalanced node. Its true power only becomes apparent after the transformation to a set of balanced nodes.

The techniques shown above are sufficient to translate every legal Parlog or safe GHC program to MONSTR without compromising its semantics. However, unrestricted GHC programs, make use of a run time safety test. This can be elegantly implemented in DACTL using pointer equality ([6]). Rules of the form

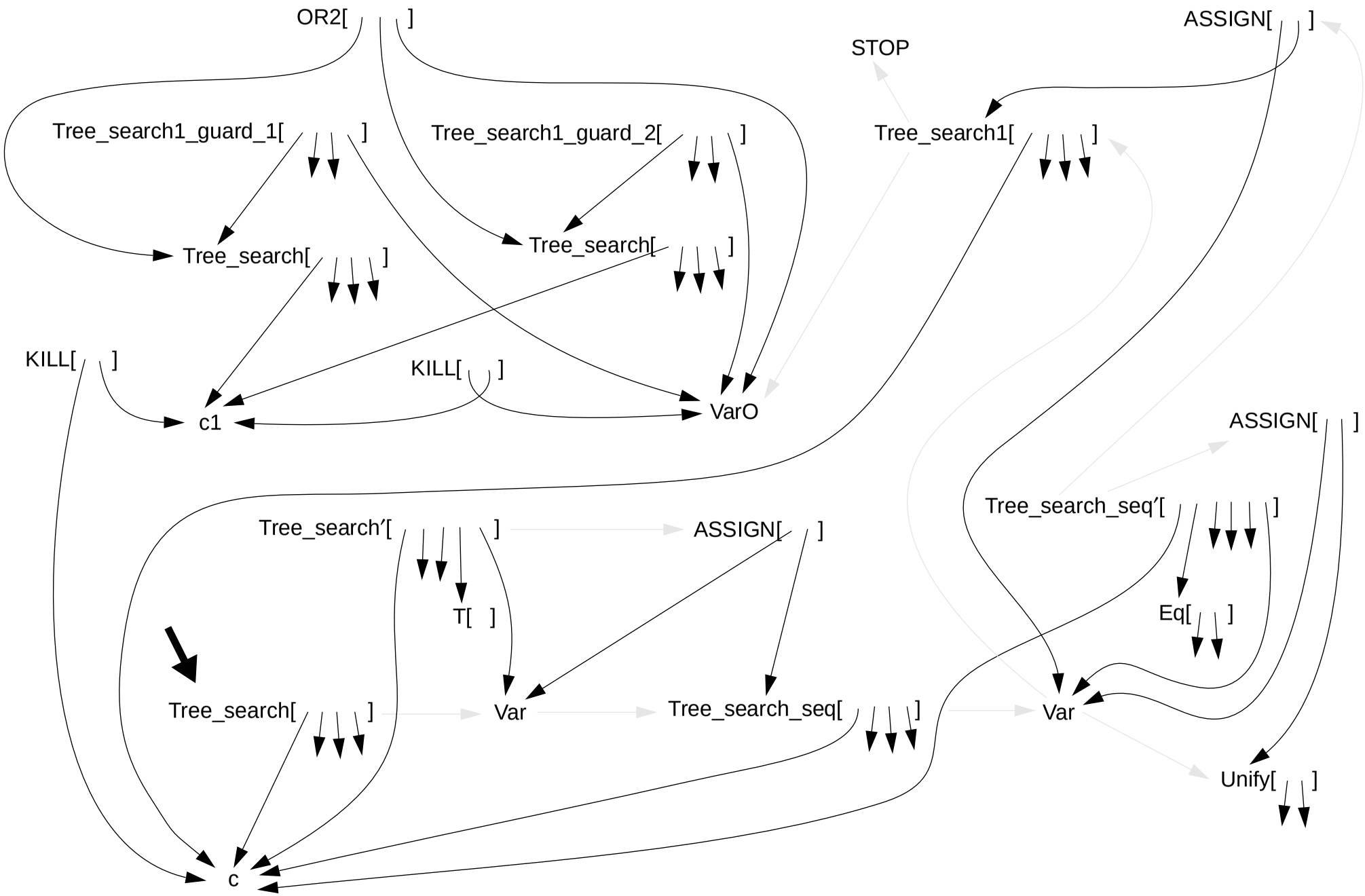
```
Unify[env v:Var[env] t] => *SUCCEED, v:=*t;
Unify[env v:Var[env'] t] => #Unify[env ^v t];
```

check whether the environment where unification is attempted is compatible with the birth place of the variable. Compatibility is checked by representing environments as DACTL nodes and performing pointer equality tests. MONSTR's restriction 2 forbids the general use of pointer equality testing. However, the Flagship machine can support a pointer equality primitive defined as follows:

```
PointersEqual[x:Any x] => *SUCCEED;
PointersEqual[Any Any] => *FAIL;
```

Note that we only test for pointer equality in the unification primitive; the rest of the rulesets for a GHC program would simply carry around the pointers of the variables and predicate calls involved. The above rules for **Unify** can be written as follows:

```
Unify[env v:Var[env'] t] => #IF[^*PointersEqual[env env'] env v t];
```


OR2[ ]
Tree_search1_guard_1[ ]
Tree_search1_guard_2[ ]
Tree_search[ ]
Tree_search[ ]
KILL[ ]
KILL[ ]
c1
VarO
STOP
ASSIGN[ ]
Tree_search1[ ]
Tree_search'[ ]
T[ ]
ASSIGN[ ]
Tree_search[ ]
Var
Tree_search_seq[ ]
c
ASSIGN[ ]
Tree_search_seq'[ ]
Eq[ ]
Var
Unify[ ]

```
IF[SUCCEED Any v:Var[Any] t] => *SUCCEED, v:=*t;
IF[FAIL env v t] => #Unify[env ^v t];
```

Note that the sort of graph generated by the translation of a GHC program to DACTL or MONSTR guarantees that during the test for pointer equality the environment(s) involved will not change. This is important because the notion of pointer equality supported by **`PointersEqual`** is rather weak in the sense that it does not allow the test, and any actions one might want to initiate as a result of the test, to be encapsulated in a single atomic action. This property of the graphs generated holds for the cases where the bodies of clauses are executed only after commitment (which is what happens in practice).

5 Advantages, Limitations and Further Work

We have presented the outline of a parallel implementation of term graph rewriting on the Flagship machine and we explained the need to move from the general DACTL model to a substantially weaker one as defined by MONSTR. We showed how our Parlog and GHC to DACTL implementations can be supported by MONSTR without compromising the semantics and expressiveness of the original DACTL rule sets. However, the resultant MONSTR rule sets are finer grained than their corresponding DACTL ones. The potential additional overhead in executing them must be assesed by means of simulation and/or emulation studies. What made these transformations possible is the fact that Parlog and GHC do not support atomic test-and-set unification. It is the case that DACTL can support a limited form of tell unification ([13]); consider for example, the following FCP(:) program that models a CSP transaction with output guards:

```
p(X,ToC1,ToC2) <- ToC1=hello(X), ToC2=hello(X).
c(Id,hello(X1),_) <- true : Id=X1 | true.
```

This can be translated to DACTL as follows:

```
P[x toc1 toc2] => #AND[^*Unify[toc1 Hello[x]]
        ^*Unify[toc2 Hello[x]]];

C[id Hello[x1:Var] Any] => *SUCCEED, x1:=*id|
C[id Any Hello[x2:Var]] => *SUCCEED, x2:=*id;
(C[p1 p2 p3] & (C[Any (Var+Hello[Any]) Any] +
        C[Any Any (Var+Hello[Any])]))
        => #C[p1 ^p2 ^p3];
C[Any Any Any] => *FAIL;
```

However, the above program cannot be translated to MONSTR; process **`C`** should non-deterministically and atomically select one **`Hello[x]`** message and bind its variable. The above DACTL program can do that but a MONSTR one (using the same techniques as Merge above) would only be able to non-deterministically select one **`Hello[x]`** message; the binding of **`x`** would have to be done after commitment, possibly resulting in failure.

Our treatment of the translation of programs written at a high level of abstraction into MONSTR programs took place in the specific context of concurrent logic programming. Nevertheless, the kind of synchronisation issues that arise naturally in this context are typical of many other programming paradigms, for example in multithreaded sequential programming. Although one can usually construct pathological examples in such lower-level frameworks whose translation into MONSTR raises the kind of difficulties evident in the **`hello`** example above, and worse; it is nevertheless remarkable that "sensible programs" of the type one would actually use to solve problems that arise in practice, seldom if ever display such behaviour. Thus although the translation of arbitrary DACTL programs into efficient MONSTR programs is undoubtedly a problem without a reasonable solution in all cases, in most cases of interest a perfectly reasonable translation can be produced given a suitable understanding of the original program. In this sense, good translation into MONSTR is best initiated from as high a level of abstraction as is convenient;

when the semantics of the situation is available in a transparent form. Our success in achieving a good reprentation of concurrent logic programs is clearly attributable to this strategy. Attempting to analyse the DACTL versions of even reasonable programs statically in order to get a similar level of efficiency in the MONSTR translation will usually be beyond the power of current compiler technology. By the way, MONSTR stands for "a Maximum of One Non-root STateholder per Rewrite", from the second item in its list of defining restrictions in section 3!

References

[1] Banach R., *MONSTR*, (in preparation).

[2] Banach R., Sargeant J., Watson I., Watson P. and Woods V., *The Flagship Project*, Proceedings of the Alvey Technical Conference, Swansea, UK, July 4-7, 1988.

[3] Banach R. and Watson P., *Dealing with State on FLAGSHIP: the MONSTR Computational Model*, in CONPAR-88, Jesshope and Reinhartz eds., Cambridge University Press 1989, pp. 595-604.

[4] Barendregt H. P., Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R.,Plasmeijer M. J. and Sleep M. R., *Term Graph Rewriting*, in PARLE-87, Odijk and Rem eds., LNCS, Springer Verlag Vol 259, pp. 141-158.

[5] Glauert J. R. W., Kennaway J. R., Sleep M. R. and Somner G. W., *Final Specification of DACTL*, Internal Report SYS-C88-11, University of East Anglia, Norwich, UK, 1988.

[6] Glauert J. R. W. and Papadopoulos G. A., *A Parallel Implementation of GHC*, FGCS-88, Tokyo, Japan, Nov. 28 - Dec. 2, 1988, Vol. 3, pp. 1051-1058.

[7] Hammond K., *Implementing Functional Languages for Parallel Machines*, Ph.D. Thesis, University of East Anglia, Norwich, UK, 1988.

[8] Huet G. and Oppen D. C., *Equations and Rewrite Rules: a Survey*, in Formal Language Theory: Perspectives and Open Problems, Book ed., Academic Press, 1980.

[9] Kennaway J. R., *Implementing Term Rewrite Languages in DACTL*, CAAP-88, Nancy, France, Mar.21-24, 1988, Dauchet and Nivat eds., LNCS, Springer Verlag Vol 299, pp. 102-116.

[10] Papadopoulos G. A., *A Fine-Grain Parallel Implementation of PARLOG*, TAPSOFT-89, Barcelona, Spain, Mar. 13-17, 1989, Dias and Orejas eds., LNCS, Springer Verlag Vol 351, pp. 313-327.

[11] Papadopoulos G. A., *Parallel Implementation of Concurrent Logic Languages Using Graph Rewriting Techniques*, Ph.D. Thesis, University of East Anglia, Norwich, UK, 1989.

[12] Peyton Jones S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall,1987.

[13] Shapiro E., *The Family of Concurrent Logic Programming Languages*, Computing Surveys, 1989, Vol 21, pp. 412-510

[14] Watson P. and Watson I., *Evaluating Functional Programs on the Flagship Machine*, FPLCA, Oregon, USA, Sept. 14-16, 1987, Khan ed., LNCS, Springer Verlag Vol 274, pp. 80-97.

[15] Watson I., Woods V., Watson P., Banach R., Greenberg M. and Sargeant J., *Flagship: A Parallel Architecture for Declarative Programming*, 15th International Symposium on Computer Architecture, IEEE, Honolulu, Hawaii, May 30 - June 2, 1988, pp. 124-130.