1

æ

Expressing Runtime Structure and Synchronisation in Concurrent OO Languages with MONSTR

R. Banach
Department of Computer Science, University of Manchester, Manchester, M13 9PL, U.K., banach@cs.man.ac.uk

G. A. Papadopoulos
Department of Computer Science, University of Cyprus, CY-1678, Nicosia, Cyprus., george@turing.cs.ucy.ac.cy

Abstract

The extended term graph rewriting formalism of MONSTR is described, together with some of its more important rigorously established properties, particularly regarding serialisability and acyclicity. This basis is used for giving a convenient description of the global runtime structure of a concurrent object oriented language. The formalism proves especially convenient for describing very precisely a variety of intended synchronisation properties of objects in a concurrent OOL, and this flexibility is illustrated by considering a variety of possible operational semantics for a simple counter object. A lower bound object example illustrates that even more extreme synchronisation properties for objects may be contemplated without stretching the capabilities of the MONSTR formalism. The presentation is independent of any specific high level OOL.

Keywords

Object Oriented Languages, Object Synchronisation, Term Graph Rewriting, MONSTR, Distributed Processing, Serialisability.

1 INTRODUCTION

Recently a number of proposals have been put forward with the aim of combining concurrency and object-orientation. They differ in many aspects regarding the way they handle the issues pertaining to this combination, such as degree of concurrency allowed not only between objects but also within an object (and how the internal state of the latter can be protected), synchronisation mechanisms e.g. locks, wait queues, synchronisation counters or activation conditions, process structures, and implementation techniques for objects, etc. In Papathomas (1990) six different categories of OO languages are identified. It has

also been argued (Nierstrasz and Papathomas 1990) that there is a need to develop semantic frameworks for reasoning about the way various features of concurrent OO languages operate, and to provide a common point of reference in comparing various such languages.

In the current work we exploit the generalised computational model of Term Graph Rewriting (Sleep et al. 1993) and in particular the MONSTR model and associated compiler target language (Banach 1993, 1996a-d), to develop an abstract formal framework for reasoning about some of the above mentioned issues. We pay particular attention to the way the global runtime structures are set up and to how various synchronisation properties of objects can be supported.

The rest of the paper is organised as follows. The next section introduces MONSTR, paying particular attention to properties of particular interest in the present context: atomicity of rewrites, serialisability and acyclicity. The next section provides a simple but useful abstract representation of what constitutes an object. The next section comprises the main part of the paper and discusses the way inter- and intra-object interaction takes place by examining a variety of possible concurrent semantics for a specific example, a counter object and a mild generalisation of it, a lower bound object. Note that the presentation is independent of any specific high level OOL. The paper ends with some concluding remarks.

2 MONSTR

One of the main advantages in using a rule based rewriting model of computation for specifying properties of systems is that one important issue, namely the atomicity of primitive actions, is made precise automatically; i.e. each rule must execute as an atomic action. When this approach is used for distributed systems, sufficient thought must go into the design of the permitted rules, in order that the synchronisation capabilities of a distributed system are not unduly taxed. MONSTR is a rule based language that was designed with distributed systems in mind and in fact it has been implemented on at least one such architecture (Watson et al. 1988).

2.1 MONSTR Rewrites

The fundamental objects of MONSTR are term graphs. A term graph, is a directed graph where the nodes are labelled with symbols, assumed of fixed arity, and each node has a sequence of out-arcs to its child nodes. The nodes and arcs of term graphs are marked to control rewriting strategy as we will see below. The term graph that represents the instantaneous state of the computation is modified by the application of some rule. Let us look at a rule in action, to see what happens during a rewrite.

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

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

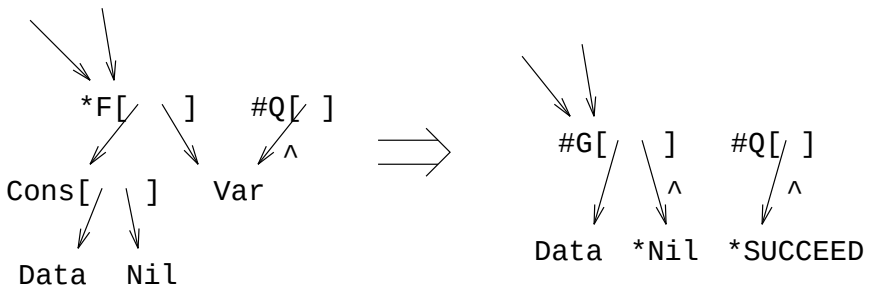

Figure 1 A MONSTR rewrite.

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

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

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

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

The above assumed that there was a rule which matched. If not then notification occurs. This is an alternative atomic action to rewriting, in which the root becomes idle, and for all its all its `^`-marked in-arcs (notification arcs), the `^` marking is removed, and the number of suspensions (`#`'s) in the parent node's marking is decremented (with $\#^0 = *$). In this manner subcomputations can signal their completion to their parents.

2.2 MONSTR Syntactic Restrictions

To make the above a computational model suited to distributed machines, a number of restrictions are imposed on the syntactic structure of systems so that some useful runtime properties hold. We paraphrase from Banach (1996a), where there is a thorough study of why these are appropriate.

Alphabets and Symbols. The alphabet of symbols $\mathbf{S}$, is the disjoint union of three subalphabets $\mathbf{S} = \mathbf{F} \uplus \mathbf{C} \uplus \mathbf{V}$ where: $\mathbf{F}$ is the alphabet of function symbols which may

label the root of the LHS L of a rule, but not any subroot node of L, and which may be the LHS of a redirection. **C** is the alphabet of constructor symbols which may label a subroot node of the LHS of a rule, but not the root, and which may not be the LHS of a redirection. **V** is the alphabet of stateholders, or variables. A stateholder symbol may label a subroot node of the LHS of a rule, but not the root. Stateholders may label the LHS of a redirection. Each $S \in \mathbf{S}$ has a (fixed) arity $A(S)$. For each $F \in \mathbf{F}$, there are subsets $\text{State}(F) \subseteq \text{Map}(F) \subseteq A(F)$, with $\text{State}(F)$ either a singleton or empty. `Root` $\in \mathbf{C}$.

In the formal model, for a node x in a graph, we write $\sigma(x)$ for its symbol, $\alpha(x)$ for its sequence of child nodes, with $\alpha(x)[k]$ its k'th child, $\mu(x)$ for its marking, and $\nu(x)[k]$ for the marking on its k'th out-arc. Also rules are of two kinds: those which pattern match (normal rules); and those which do not (default rules). Every function symbol must have at least one default rule. In the definition of a rule, implicit nodes are the ones that may match anything; others are called explicit.

Rules. Let $D = (P, root, Red, Act)$ be a rule, with $root$ the root of the LHS L, P a graph which is the union of the LHS and RHS of the rule, (with all implicit nodes accessible from root), redirections Red, (where the LHS of each redirection is an explicit node of L, and where each node of L is the LHS of at most one redirection), and activations Act. Then

1. Each node of P has the arity dictated by its symbol, i.e. 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 $\alpha(root)[k]$ is explicit $\Leftrightarrow k \in \text{Map}(F)$.
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 $\sigma(\alpha(root)[k]) \in \mathbf{V} \Rightarrow k \in \text{State}(F)$.
4. All grandchildren of the root are implicit, i.e. for all $k \in A(\sigma(root))$, and $j \in A(\sigma(\alpha(root)[k]))$, $\alpha(\alpha(root)[k])[j]$ is implicit.
5. Implicit nodes of the LHS have only one parent in the LHS (left linearity), 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. $\mu(x) = \#^n$ (for $n \geq 1$) $\Leftrightarrow |\{k \mid \nu(x)[k] = \hat{}\}| = n$.
7. Every arc (p_k, c) of P is either state saturated or activated, i.e. $\nu(p)[k] = \hat{}$ and $\mu(c) = \varepsilon \Rightarrow \sigma(c) \in \mathbf{V}$ 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. $(a, b) \in Red$ and $\mu(b) = \varepsilon \Rightarrow \sigma(b) \in \mathbf{V}$ 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. $(b, c) \in Red$ and $b \in Act \Rightarrow$ there is a $b \neq a \in L$ such that $(a, b) \in Red$.

Desirable Properties of Rewriting. By convention rewriting always starts with a single active node labelled `Initial`. When all rules used conform to the restrictions given above, induction over executions yields many desirable properties. Namely:

*This explains the MONSTR acronym. It stands for: a Maximum of One Non-root STateholder per Rewrite

- All execution graph nodes respect the arities of their symbols.
- The pattern matching requirements of each redex, depend solely on the symbol at the root (and so can be delegated to simple hardware).
- No pointer equivalence is required for matching any redex node, that is not evident from $\mathrm{Map}(\sigma(root))$, (ditto).
- All execution graphs are balanced and state saturated.
- The overwriting lemma (Banach 1996a, Lemma 5.10), applies to most redirections, in practice enabling the convenient representation of rewriting by packet store manipulations, (and particularly the representation of redirection by packet overwriting).

To ensure that rewriting conforms to the exigencies of MONSTR garbage collection, an issue we will not discuss further here, and for the operational convenience of distributed hardware, we must restrict rewrites only to redexes where the explicitly matched children of the root are idle, i.e. $\mu(\alpha(t)[k]) = \varepsilon$. If not, a suspension takes place, where the root of the rewrite becomes suspended on the non-idle matched children, waiting for them to notify. This is described in Banach (1996b). The next definition states the circumstances under which each kind of atomic action is performed.

Execution Steps. Let G be a graph and t an active node of G, at which we wish to perform a step. The kind of step to be performed is determined as follows.

If $\sigma(t) \in \mathbf{C} \cup \mathbf{V}$
Then Perform a notification at t
Else **If** For all $k \in \mathrm{Map}(\sigma(t))$, $\mu(\alpha(t)[k]) = \varepsilon$ (and $\nu(t)[k] = \varepsilon$)
Then Perform a rewrite using a rule chosen nondeterministically from Sel, where Sel is the set of normal rules that match at t if there are any, otherwise any default rule for the root symbol.
Else Perform a suspension at t

2.3 Serialisability

The above described the atomic semantics of MONSTR. Though LHSs of rules are small, there is still quite a potential demand for locking in a distributed implementation, as matching, redirection and activation are all required to be performed in a synchronised manner. In principle we want a distributed implementation to be able to work as follows.

Since both the function symbol at the root, and the stateholder symbol at its fixed stateholder position represent mutable computational state, it would be hard in a general computational framework, to avoid having to update them simultaneously if an overall serial semantics is desired. So we allow an active function node to migrate to its single stateholder argument (whose position is known statically, which thus enables the migration to be performed by autonomous hardware). Note that the suspension mechanism alluded to above conceals this movement from other rewrites anyway. Once the active function node was in the processor containing the stateholder, the other arguments, being constructors, could be copied asynchronously. Rule matching could then take place, with the building of new nodes and redirection being done atomically. Finally activation messages could be sent to any nodes requiring activation, wherever they may be.

Notifications would be performed by message passing, while constructor fetch messages happening upon non-idle nodes would suspend, as in the atomic suspension mechanism.

The above sounds innocuous enough, and it seems plausible that such a rewriting mechanism ought to be equivalent to a serial model, but in fact there is still plenty of scope for constructing non-serialisable shedules. Nevertheless, the counterexamples that one can construct invariably seem rather pathological, and it turns out that one can recover serialisability under rather mild assumptions. We make three preliminary definitions.

Resuspending Rule. A normal rule for a function `F` is a resuspending rule for `F` and `S` iff the rule explicitly matches `S` at its stateholder position, and the only RHS node is another `F`, suspended once on the matched stateholder `S`. For example:

```
F[ a b:C1 c:S d:C2 e ] => #F[ a b ^c d e ]
```

Refiring Rule. A default rule for a function `F` is a refiring rule iff the only RHS node is another `F`, suspended on all the explicitly matched arguments, which are activated. For example, for the same `F` as above:

```
F[ a b c d e ] => ###F[ a ^*b ^*c ^*d e ]
```

Resuspending Property. Let $\mathcal{R}$ be a MONSTR system. Let $D = (P, root, Red, Act)$ be a rule of $\mathcal{R}$. Let (p_k, c) be an arc of the graph P of D. Suppose that if (p_k, c) is a notification arc and c is an idle stateholder, then $\sigma(p)$ is a function symbol, k is its stateholder position, c is not the RHS of a redirection, and every rule for $\sigma(p)$ in $\mathcal{R}$ in which the root matches $\sigma(c)$ is a resuspending rule for $\sigma(p)$ and $\sigma(c)$. If the preceding holds for all such arcs in all rules of $\mathcal{R}$; and if furthermore all redirections of stateholders are to non-idle nodes, then $\mathcal{R}$ has the resuspending property.

Lemma. Let $\mathcal{R}$ be a MONSTR system with the resuspending property and let $\mathcal{H} = [H_0, H_1, \ldots]$ be an execution of $\mathcal{R}$ according to either atomic or finegrained semantics. Then in every graph H_i in $\mathcal{H}$, every notification arc (p_k, c) whose child c is an idle stateholder, has $\sigma(p) \in \mathbf{F}$, $k \in \mathrm{State}(\sigma(p))$, and all rules for $\sigma(p)$ which match $\sigma(c)$ in position k are resuspending rules for $\sigma(p)$ and $\sigma(c)$.

Strongly Resilient System. Let $\mathcal{R}$ be a MONSTR system. Then $\mathcal{R}$ is a strongly resilient system if:

1. $\mathcal{R}$ has the resuspending property.
2. No rule $D = (P, root, Red, Act)$ of $\mathcal{R}$ contains an idle function node in $P - L$.
3. Each default rule in $\mathcal{R}$ for every symbol F with $\mathrm{Map}(F) \neq \emptyset$ is a refiring rule.

Serialisability Theorem. Let $\mathcal{R}$ be a strongly resilient system and let $\mathcal{H} = [H_0, H_1, \ldots]$ be an execution of $\mathcal{R}$ according to finegrained semantics. Then $\mathcal{H}$ is serialisable; i.e. there is an atomic execution $\mathcal{G} = [G_0, G_1, \ldots]$ of $\mathcal{R}$ that corresponds to $\mathcal{H}$ in a natural way.

Of course the above is very vague; we have not said what we mean by "corresponds ... in a natural way", any more than we have been precise in the definition of the finegrained model. Here we merely say that all the "important" events in $\mathcal{H}$ can be identified in $\mathcal{G}$, so that $\mathcal{G}$ can be said to "compute the same thing" as $\mathcal{H}$. "Important" events include at least all rewrites by nonresuspending normal rules. (These are the rules that cause changes in the structure of the execution graph.) We do not have space to say more here; for further details see Banach (1996c).

In fact, by tinkering a little with the operational semantics of MONSTR, a stronger version of the theorem can be shown, dispensing with the need for clauses 2 and 3 in the definition of strongly resilient systems (see Banach 1996d). In any case idle functions are rarely used in practice, except for translating case selections, whereupon the subcomputations for each alternative can be created idle, awaiting activation when the selector subcomputation completes. Such instances are easily judged to be safe, or one can reprogram them to avoid the use of idle functions if necessary (see below).

Compositionality. The above gives us a notion of serialisability type for a strongly resilient system $\mathcal{R}$, being $(Sus, NonSus)$ where

$Sus = \{(F, S) \mid \mathcal{R}$ has a resuspending normal rule for F and $S\}$, and
$NonSus = \{(F, S) \mid \mathcal{R}$ has a non-resuspending normal rule for F and $S\}$

and where $Sus \cap NonSus = \emptyset$. Systems $\mathcal{R}_1$ and $\mathcal{R}_2$ are serialisability compatible iff $(Sus_1 \cup NonSus_2) \cap (Sus_2 \cup NonSus_1) = \emptyset$, and the serialisability type of $\mathcal{R}_1 \cup \mathcal{R}_2$ is $((Sus_1 \cup Sus_2), (NonSus_1 \cup NonSus_2))$.

Serialisability is vital for a distributed implementation of rewriting in a rewrite rule based language, as it is almost impossible to design systems that are rule based implementations of the expected behaviour of anything at all, if one has to take into account at every step all the potential finegrained executions, in case they throw up pathological shedules. In some cases good behaviour is relatively evident when the data dependencies in the system are clearly visible (as is usually the case for the case selections discussed above), but in a dynamically evolving system, particularly one involving dynamic binding which allows the system structure to evolve in unexpected ways, a solid serialisability result is very reassuring, and frees the system designer from worrying about low level finegrained implementation detail, while simultaneously freeing the architecture designer from the prohibitive costs of excessive locking.

2.4 Acyclicity

In reasoning about data dependencies in a system, acyclicity is often a useful bonus, but it is interesting that it is neither necessary nor sufficient for serialisability. All the phenomena that render pathological counterexamples non-serialisable, can be realised within always acyclic systems. (Nevertheless the correspondence between a finegrained execution and its serialisation can in some cases be rendered more simply if the system is always acyclic.)

If a graph is acyclic, and we perform some redirections, then the result can contain a cycle only if there is cycle of paths such that the head of each path is redirected to the tail of the next (and cyclically). From this we can get the following.

Acyclicity Theorem. Suppose for a rule $D = (P, root, Red, Act)$, P is acyclic, and for each redirection (a, b) in Red, either (i), b is not in L and is not an ancestor of a; or (ii), b is a descendant of a. Then the rule preserves acyclicity.

Corollary. Suppose for a rule $D = (P, root, Red, Act)$, P is acyclic, and for each redirection (a, b) in Red, b is not an ancestor of a. Suppose for the non-root redirection (c, d), d is in L but is not a descendant of c. Suppose in the rewrite of a graph G, a is matched to x and b to y. Suppose there is no path from y to x in G. Then the rewrite preserves acyclicity.

Note that the theorem relies on static properties of rules and as such, systems consisting

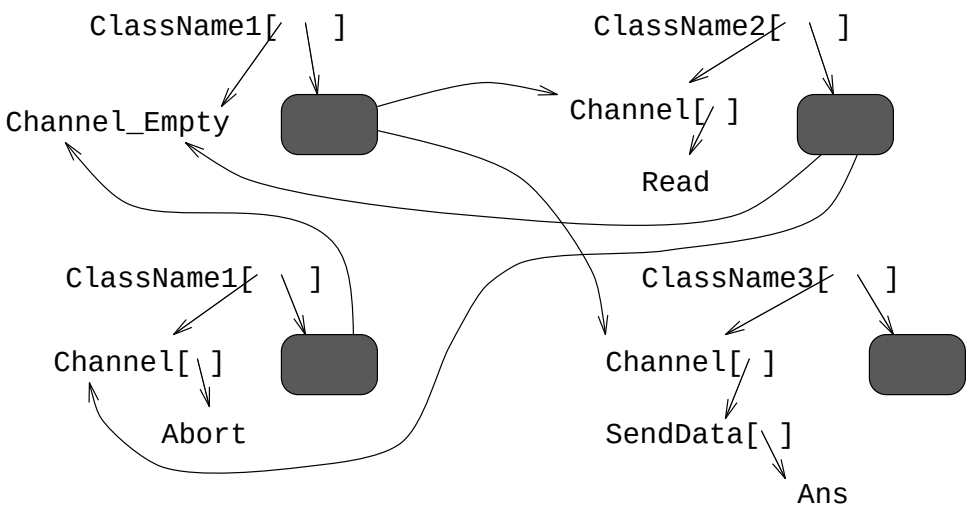

Figure 2 A collection of MONSTR objects.

purely of rules complying with its hypotheses guarantee acyclicity of all execution graphs, and are composable with each other. The corollary however relies on properties of the matchings that may arise during an execution, and these must be established by more global analyses. So we do not get immediate composability in the latter case.

3 CONCURRENT OO PROGRAMMING VIA MONSTR

We are proposing that MONSTR provides a good foundation for defining the behaviour of concurrent object oriented systems. One can translate a high level OOL into MONSTR, and then use the operational semantics of the resulting rule system to define the behaviour of the OOL. Different details in the translation yield different operational behaviours for the rule system, hence a different operational definition for the OOL. We claim that the kinds of behaviour that one might reasonably want an OOL to display, can in fact be captured fairly naturally in MONSTR systems. The general idea in this approach can be seen in Figure 2.

The Figure illustrates four objects, two of which are instances of `ClassName1`, and one of `ClassName2` and of `ClassName3`. The objects themselves consist of three pieces each. The first, an object's interface to the outside world, is its "self" channel node, bearing either the `Channel_Empty` symbol, or in the state `Channel[message_contents]`. The second is a function node with symbol usually named after the class, e.g. `ClassName1`. This function controls overall activity in the object, and is the "owner" of the self channel node. The third is the shaded blob, representing everything else about the object; e.g. instance variables, partially completed subcomputations pertinent to the object, references to (the self nodes of) other objects, etc. For example, the upper `ClassName1` object has references to `ClassName2` and `ClassName3`, while the lower `ClassName1` object has a reference to the upper `ClassName1` object only. The `ClassName2` object has references to both `ClassName1` objects. And the `ClassName3` object refers to no other object.

The key invariant is that for each object channel node, the owner is unique, being the only node with a reference to the object's channel node that does not come from within a shaded blob. Objects call upon each other to perform work by posting messages to each others' channel nodes, and the processing of a message is a method invocation. For example, the lower `ClassName1` has been called on to abort, while `ClassName2` has been called on to read. `ClassName3` has been called on to send data, and has provided a channel labelled `Ans` in which to send the response.

The protocol observed at each self channel node, expressed by the rules for the various symbols involved, can accurately describe the synchronisation properties of concurrent method invocation in a parallel environment, via the rigorous atomic operational semantics of MONSTR. The natural thing is that various objects having a reference to some particular object, say $Obj1$, are responsible for sending it messages via message send functions. These successfully lodge the message in the self channel, but only if the channel was empty. Likewise, the unique $Obj1$ owner is responsible for extracting and processing the messages, but can only do so if it finds the channel not empty. Other possibilities exist, e.g. modelling an explicit message queue, if required by the semantics of the high level OOL under consideration. Other aspects of synchronisation within and between objects, can also be accurately and unambiguously described by appropriate choice of MONSTR rules as we shall see below. The suspension markings, for overtly programmed sequencing, and the run-time suspension mechanism, both play a role in this. Finally, the serialisability theorem reassures us that the primitives used in defining a concurrent OOL via MONSTR are in sympathy with what a realistic implementation might hope to achieve, even if its implementation philosophy is quite different from term graph rewriting.

From the point of view of providing a definition of some OOL, the very small LHSs of MONSTR rules are sometimes an inconvenience. When there is good justification, it is reasonable to relax these restrictions, but when this is done, the justification should always be presented, as a check that too much would not be demanded of an implementation. Thus deep LHSs, or ones involving more than one stateholder could be regarded as acceptable, provided it was made clear what locality considerations justified the assumption that such rules could be executed atomically; in particular how such rules would not break the serialisability properties of the ruleset as a whole. We shall sometimes make use of such devices below.

4 STRUCTURE AND SYNCHRONISATION VIA MONSTR

4.1 Message Sending

Let us look in detail at some rules that embody the general principles discussed above. We start with the module for message sending. This consists of the rules:

```
Send[c:Channel_Empty message] => *OK , c:=*Channel[message] ;
Send[c:S[..] message] => #Send[^c message] ;
Send[c message] => #Send[^*c message] ;
```

where `S` is any symbol in ($\mathbf{V} - \{$`Channel_Empty`$\}$). In future we will abbreviate collections of rules such as are represented by the middle line above by writing the obvious shorthand form `Send[c:(V - Channel_Empty) message] => ...` A real implementation would obviously include the ability to pattern match simple potentially infinite collections of rules such as these by using negative matching tests. Note that the serialisability type of the `Send` module is

$$(\{(\texttt{Send}, \texttt{S}) \mid \texttt{S} \in (\mathbf{V} - \{\texttt{Channel_Empty}\})\}, \{(\texttt{Send}, \texttt{Channel_Empty})\}).$$

We can see that if `message` contains only nodes that cannot access the destination channel node, e.g. it contains only simple method name constructors, having descendants which are at worst some constructor parameters, or which contain in addition only response channels created for the purpose by the sending object, then the rules above (and specifically the first of them) preserve acyclicity. On the other hand, if references to objects are being passed around, then the dynamic nature of typical OO systems means that message may contain as a parameter a reference to the destination channel c, and message arrival would entail the creation of a cycle. This is not necessarily harmful in itself. An object $O1$ may pass around to other objects, various object references, including itself. This activity causes no harm even if $O1$ does not know that it is referring to itself while doing this, (perhaps having only an indirection to the location of the object it is passing around).

Posting methods to unknown object references is potentially more dangerous. Say $O1$ sends a method request to some other object $O2$, of which it knows nothing other than (perhaps only the location of an indirection to) its location, (and thus which might unknowingly perhaps be $O1$ itself). If $O1$ does not need to wait for a response from $O2$, then it is unlikely that problems will arise even if $O1 = O2$. However if $O1$ does need to wait for a response, then some kind of deadlock might well occur if $O1 = O2$. This depends critically on the semantics of method processing in the OOL of interest. Typically, the instance variables of $O1$ will be locked for the duration of method processing, to ensure a unique serial semantics associating instance variable values with method requests processed; so $O1$ will refuse to service the message it sent itself and will deadlock. However if weaker coherence between instance variable values and method requests received is acceptable, then there may be a way in which the knot tied when $O1 = O2$ would not strangle the computation. We will allude to this briefly below.

4.2 A Counter Object

Let us look at the rules for the inside of an object to illustrate some of these points. Below are the MONSTR rules for a simple imperative counter object. They are numbered on the right for ease of future reference.

```
NewCounter[init]                                                    [1]
=> *self:Channel_Empty , *Counter[self init] ;

Counter[self:Channel[Read[ans_chan]] state]                         [2]
=> *Counter[self state] , self:=*Channel_Empty
   *Assign[ans_chan state] ;
```

```
Counter[self:Channel[Inc[value ans_chan]] state]                    [3]
=> #Counter[self ^newstate:*ADD[value state]] ,
   self:=*Channel_Empty , #Assign[ans_chan ^newstate] ;

Counter[self:Channel_Empty state]                                   [4]
=> #Counter[^self state] ;

Counter[self state]                                                 [5]
=> #Counter[^*self state] ;
```

where

```
Assign[s:V t] => *OK , s:=*t ;
Assign[s t] => #Assign[^*s t] ;
```

Rule [1] is the rule used to create a fresh counter object from within the code for some other object *OthO* say. The object *OthO* creates an initial value `init` for the new counter instance, and gives it as a parameter to a node `*NewCounter[init]` that it creates in the RHS of some rule for the method it is currently evaluating. The rule for `NewCounter` is an example of a rule which is a default rule for a function symbol F such that $\mathrm{Map}(F) = \emptyset$. As such it is not forced by the serialisability theorem to be a refiring rule, the main reason being that the function symbol `NewCounter` requires no pattern matching, and therefore there are no potential race conditions to impede serialisability arising from the values of parameters at different moments.

Such rules are useful as they add mobility to a system. In the default execution model for MONSTR, stateholders are normally idle, and are referred to from many points in the graph, being the seat of shared state. Therefore they are deemed immobile, and moving them requires programmed higher level synchronisation in general. However active functions, which in the general case must move to their stateholder argument anyway, and are rendered unmatchable by other rewrites by virtue of being non-idle, are mobile. So a `NewCounter` node being active, is able to relocate with ease, aiding load balancing, before it becomes the `self` channel node of the new counter object. Note that references to `NewCounter` can be passed around with impunity by the creating object *OthO*, as the target of such references will be unobservable until the `self` channel is instantiated.

Rule [4] shows the object waiting for the arrival of a method call. It is a resuspension rule for `Counter` and `Channel_Empty` and corresponds to the fact that Send can only install a message if the channel is in the `Channel_Empty` state. So the synchronisation works as expected, and the serialisability theorem assures us that there are no races in a suitable distributed implementation.

The preceding two rules, [2] and [3], show some basic method call processing. Note firstly that they feature deeper pattern matching than is permitted by restriction 4 for MONSTR rules. This is a convenient shorthand to increase readability, as hinted at above. A system conforming more faithfully to the letter of the law would replace rules [2] and [3] with:

```
Counter[self:Channel[mess] state]                                   [m]
=> *Counter_Match[self mess state] ;
```

```
Counter_Match[self:Channel[a] Read[ans_chan] state]                    [2′]
=> *Counter[self state], self:=*Channel_Empty ,
   *Assign[ans_chan state] ;

Counter_Match[self:Channel[a] Inc[value ans_chan] state]               [3′]
=> #Counter[self ^newstate:*ADD[value state]] ,
   self:=*Channel_Empty, #Assign[ans_chan ^newstate] ;

Counter_Match[self mess state]                                         [6]
=> ##Counter[^*self ^*mess state] ;
```

That this works as required relies critically on the fact that each channel node has a unique owner, this being the `Counter` or `Counter_Match` function node in this case. Because the only other nodes allowed to rewrite a channel node are `Send` functions, and these must wait when the channel is occupied, the owner is at liberty to break down the pattern matching into several shallow phases without fearing any race conditions. Subsequently, once the message in the channel has been decoded, the RHS of rules [2′] or [3′] represent the computation of the method involved.

Let us look at [2′]. In its RHS, the `self` channel is reset, and the owner rewrites to a `Counter` function. These are done as part of the atomic action of rewriting. Also an `Assign` function is created to asynchronously assign `ans_chan` to the current value of the counter. That this can indeed be done asynchronously is a consequence of the fact that each object's blob, which contains the value of its instance variables, has a unique parent, the object's owner, and that when method processing completes, a new owner is created. The new owner does not need to refer to the old values of the instance variables, so these need never be redirected to any new values created by subsequent message processing. Thus the asynchronous assignment is safe.

In fact the history of the instance variables through a computation may be represented by a sequence of constructors if a clean single assignment discipline for creation of new instance variable values is adhered to within method computations. (Such a discipline is specified in UFO (Sargeant 1993); of course if a less clean story within method computations pertains, then stateholders may be required. This depends on the desired semantics for method computations.)

For contrast let us see how a less asynchronous assignment discipline would work, in which we forced the counter object to wait until the assignment completed successfully. We would replace [2′] by

```
Counter_Match[self:Channel[a] Read[ans_chan] state]                    [2″]
=> #Counter_Inter[self ^a state] , self:=*Channel_Empty ,
   *a:Assign[ans_chan state] ;

Counter_Inter[self OK state]                                           [ci]
=> *Counter[self state] ;

Counter_Inter[self a state]                                            [7]
=> #Counter_Inter[self ^*a state] ;
```

In this specification, we have again chosen to unlock the channel early, by having the redirection `self:=*Channel_Empty` within rule [2″], rather than postponing it till rule [*ci*], where it would be done at the same time that the owner rewrote to a `Counter` node. The early method allows a little more concurrency, as a waiting `Send` could install the next message before the `Counter` instantiated. (Of course no notice would be taken of the new message until the `Counter` was ready to do so.) Finally, if we were certain that locality considerations justified it, we could include the redirection of `ans_chan` to state as a third redirection in rule [2] or its analogues, instead of using the `Assign` function, though such an overreaching of the MONSTR restrictions would need a thorough case for support.

Let us now look at rule [3′]. The synchronisation discipline embodied within it is again a fairly natural one. The channel is reset early as before, but the new `Counter` function is now suspended waiting for the computation of the new state value (via the `ADD[value state]`), to terminate. Assuming that `ADD` works directly on integer constructors and yields an integer constructor, is consistent with our remarks above, that the sequence of instance variable values over time, can be consistently represented by a sequence of constructors in the graph. Once more the assignment of the new value to the response channel is done by an asynchronous `Assign` function which waits for the new constructor to appear.

As previously, various other synchronisation disciplines can, be imagined. Noting that MONSTR operational semantics specifies that a non-idle node cannot be pattern matched, we can give a more eager definition of the increment method, replacing rule [3′] by rule [3″] below

```
Counter_Match[self Inc[value ans_chan] state]                    [3″]
=> *Counter[self newstate:*ADD[value state]] ,
   self:=*Channel_Empty, *Assign[ans_chan newstate] ;
```

In this version, computation proceeds before the new state value has been instantiated. Unlike the programmed suspensions on `newstate` previously, we now pass round references to the uninstantiated value, relying on the dynamic suspension mechanism for proper synchronisation. By contrast a much more sequential definition can be described by

```
Counter_Match[self:Channel[a] Inc[value ans_chan] state]         [3‴]
=> #Counter_Inter[self ^a newstate:*ADD[value state]] ,
   self:=*Channel_Empty, a:#Assign[ans_chan ^newstate] ;
```

This version which uses the same `Counter_Inter` function as before, demands that the new value be computed first, then that the assignment complete successfully, and only then that the owner rewrites to a new `Counter` function. Meanwhile the `self` channel is reset early as before. The reader will agree that with a slightly different `Counter_Inter` function, the `Assign` and `Counter_Inter` rewrites could be permitted to proceed concurrently if both were suspended on the outcome of the `ADD`. And various alternative possibilities exist for resetting the `self` channel if required.

We observe that for all of these cases, the serialisability types of the various `Counter` and `Counter_Match` functions are

```
({(Counter(_Match), Channel_Empty)},
{(Counter(_Match), Read), (Counter(_Match), Inc)})
```

which are serialisability compatible with the `Send` module, as we would wish. Note finally that as with most well typed MONSTR systems, the default rules demanded as a fail safe measure in the syntactic definition of MONSTR are in fact never used.

4.3 Less Coherent Semantics for an LBound Object

All the above variations maintained the invariant that there was a precise 1-1 correspondence between instance values and the sequence of method calls that produced them, even if sometimes the values could be passed around before they were fully instantiated. Suppose by contrast that an application required a lower bound object for some quantity where there would be a system bottleneck if the object always had to wait for the next value to be computed. For instance a parallel alpha-beta search might make use of such an object to provide a safe if suboptimal bound for tree pruning. Rules for such an object (using deep pattern matching for brevity) might appear as follows.

```
NewLBound[init]                                                    [8]
=> *self:Channel_Empty , *LBound[self CurrVal[init]] ;

LBound[self:Channel[Read[ans_chan]] state]                         [9]
=> *LBound[self state] , self:=*Channel_Empty ,
   #Assign[ans_chan ^*Deref[state]] ;

LBound[self:Channel[Upd[value ans_chan]] state]                    [10]
=> *LBound[self state] , self:=*Channel_Empty ,
   #IF2.1[^#GT[value ^a:*Deref[state]] then1 then2 else1] ,
   then1:Assign[state CurrVal[value]] ,
   then2:Assign[ans_chan value] ,
   else1:Assign[ans_chan a] ;

LBound[self:Channel_Empty state]                                   [11]
=> #LBound[^self state] ;

LBound[self state]                                                 [12]
=> #LBound[^*self state] ;
```

where

```
Deref[CurrVal[x]] => *x ;
Deref[x] => #Deref[^*x] ;

IF2.1[True then1 then2 else1] => *OK , *then1 , *then2 ;
IF2.1[False then1 then2 else1] => *OK , *else1 ;
IF2.1[x then1 then2 else1] => #IF2.1[^*x then1 then2 else1] ;
```

Now the state value has to be enclosed in a stateholder `CurrVal` as a call of the `Upd` method will lead to its typically being updated after the method call has relinquished

control. In general the method processing of several `Upd` calls may be in progress at once, and they are not guaranteed to be inspecting the most up to date lower bound available by any means. So now there is a many-1 map between the sequence of calls arriving at the self channel of the object and the sequence of values of instance variables in the object, (as observed by the calls).

Note further the idle function nodes `then1`, `then2`, `else1`, in the RHS of the `Upd` method rule. On the face of it they break the serialisability theorem, but as discussed in section 2, they are being used to implement case analysis, and it is not too hard to argue that the computation specified by the rule is serialisable. We argue thus. By inspection of the RHS of rule [10], the only parent of the idle functions is the `IF2.1`, node whose first argument determines which of them receives an activation. Since no other node can access the idle functions, no race conditions can arise from any delay consequent upon not performing the activations atomically.

Finally we note that if an `LBound` object were (rather bizarrely) to send itself an `Upd` method call from within (a suitably enhanced specification of) an `Upd` call, it would not deadlock, because of the non-serial semantics of instance variable update. The same cannot be said for any version of the `Counter` object with a souped up `Inc` method, because the serial instance variable update discipline used in the `Counter` definitely *would* cause deadlock.

5 CONCLUSIONS

In this paper we have made use of an intermediate formalism to study the semantics of concurrent OO languages (COOLs). In particular, we developed a simple but useful abstraction of what constitures an object and we showed how this can be mapped onto the MONSTR computational model. We then discussed a number of issues pertaining to synchronisation of concurrent method invocations between and within an object by lifting the relevant discussion from the level of a COOL to that of a set of MONSTR rewrite rules. This generalises our initial work (Banach and Papadopoulos 1995b), where we focused on the object oriented functional language UFO (Sargeant 1993). It is important to note that our simple object abstraction renders the current work independent of any particular COOL. This means that MONSTR provides good neutral ground for comparing different synchronisation semantics for COOLs, whether existing or proposed. Therefore, during the process of designing a new COOL, whether it is an extension of some existing base language or a brand new one, the language designer can use the proposed framework to study cheaply, various aspects of the language's semantics (Nierstrasz and Papathomas 1990, Papathomas 1989). Other uses of this framework are also possible. For instance, one could use MONSTR as a means of comparing similar features offered by different languages, and could thus study issues related to expressiveness or interaction of these features.

Moreover, not only can our work serve as a specification of language features, (and an executable one at that via an implementation of MONSTR); but it can be used as an implementation framework for COOLs by developing fully fledged language translators to MONSTR, where emphasis is on optimising the run-time activities of the generated graph rewrite rule sets. This is a traditional and extensively tested approach, having been used for a variety of language formalisms (Banach and Papadopoulos 1993, 1995a). A full

translation gives the added benefit of allowing rigorous reasoning about all aspects of a language; e.g. the way Banach et al. (1995) gives an alternative perspective on process calculi. And there is no reason why we could not apply the principles of our approach to other similar families of languages such as the concurrent constraint ones (Henz et al. 1994).

REFERENCES

Banach R. (1993), MONSTR: Term Graph Rewriting for Parallel Machines, in *Term Graph Rewriting: Theory and Practice*, Sleep et al (eds.), Wiley, pp. 243–252.

Banach R. (1996a), MONSTR I — Fundamental Issues and the Design of MONSTR, *Journal of Universal Computer Science*, to appear.

Banach R. (1996b), MONSTR II — Suspending MONSTR Semantics and Independence, submitted to *Journal of Universal Computer Science*.

Banach R. (1996c), MONSTR III — Finegrained Semantics and Serialisability, in preparation.

Banach R. (1996d), MONSTR IV — Coercing Semantics and Serialisability for Resilient Systems, in preparation.

Banach R., Balazs J., Papadopoulos G. A. (1995), A Translation of the Pi-Calculus into MONSTR, *Journal of Universal Computer Science*, **1** (6), 335-394.

Banach R., Papadopoulos G. A. (1993), Parallel Term Graph Rewriting and Concurrent Logic Programs, *Proc. WPDP '93*, Sofia, Bulgaria, May 4-7, 303–322.

Banach R., Papadopoulos G. A. (1995a), Linear Behaviour of Term Graph Rewriting Programs, *Proc. ACM SAC '95*, Nashville TN., USA, Feb. 26-28, ACM Press, 157–163.

Banach R., Papadopoulos G. A. (1995b), Term Graph Rewriting as a Specification and Implementation Framework for Concurrent Object Oriented Programming Languages, *Proc. MPPM '95*, Berlin, Germany, Oct. 9-12, IEEE Press, 151–158.

Henz M.,Smolka G., Wurtz J. (1994), Object-Oriented Concurrent Constraint Programming in Oz, *Proc. PPCP*, MIT Press, Cambridge MA., 27–48.

Nierstrasz O., Papathomas M. (1990), Viewing Objects as Patterns of Communicating Agents, *Proc. OOPSLA/ECOOP '90*, ACM Press, Ottawa, Canada, Oct. 21-25, 38–43.

Papathomas M. (1989), Concurrency Issues in Object-Oriented Languages, in *Object Oriented Development*, ed. D. Tsichritzis, Centre Universitaire d' Informatique, University of Geneva, 207–245.

Sargeant J. (1993), Uniting Functional and Object-Oriented Programming, *Proc. 1st JSST*, Kanazawa, Japan, Nov. 4-6, LNCS **742**, Springer Verlag, 1–26.

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

Watson I., Woods V., Watson P., Banach R., Greenberg M., Sargeant J. (1988), Flagship: A Parallel Architecture for Declarative Programming, *Proc. 15th ISCA*, Hawaii, May 30-June 2, 124–130.