Term Graph Rewriting as a Specification and Implementation Framework for Concurrent Object-Oriented Programming Languages

Richard Banach

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

George A. Papadopoulos*

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

Abstract

The usefulness of the generalised computational model of Term Graph Rewriting Systems (TGRS) for designing and implementing concurrent object-oriented languages, and also for specifying and reasoning about the interaction between concurrency and object-orientation (such as concurrent synchronisation of methods or interference problems between concurrency and inheritance), is examined in this paper by mapping a state-of-the-art functional object-oriented language onto the MONSTR computational model, a restricted form of TGRS specifically designed to act as a point of reference in the design and implementation of declarative and semi-declarative programming languages especially suited for distributed architectures.

1. Introduction

Lately there seems to be an agreement between designers and implementors of concurrent or parallel languages that object-oriented programming should be the major programming discipline enforced by a language model. In particular, a programmer should design his program components and the associated interaction between them, by means of objects communicating by sending and receiving messages, and exploiting all the features of OOP which are seen as assisting him in writing programs which "behave properly" in a parallel environment.

The above mentioned approach is advocated in nearly the whole programming language spectrum, ranging from functional languages, to concurrent constraint languages and of course also imperative languages ([6,7,8]). Thus, a number of issues related to reasoning about the correct and efficient execution of a program are now transposed from the level of, say, accessing single variables (as in the case of concurrent constraint or imperative languages), to the more abstract level of accessing an object by means of invoking associated methods. These issues are still of paramount importance since "pure" OOP remains essentially an imperative concept even for the case of object-oriented declarative languages where "non monotonic" behaviour such as destructive assignment is used to model objects with state ([6,8]). Thus, although such imperative behaviour may be well hidden within the functionality of an object, unintended object access can still cause problems, such as deadlock, at the level of (concurrent) method invocations.

The purpose of the current work is to exploit the generalised computational model of Term Graph Rewriting ([10]) and in particular the MONSTR model and associated compiler target language ([1]), in mapping the behaviour of programs written in UFO ([8,9]), a state-of-the-art object-oriented functional language, onto equivalent sets of MONSTR rewrite rules. MONSTR is a computational model designed specifically for the needs of parallel machines and indeed it has played the role of a "machine language" for at least one such architecture ([11]). More to the point, in this work we provide a set of TGR-based abstractions related to the fundamental operations with objects (object creation, method invocation, delegation and inheritance mechanisms, etc.) the aim being: (i), to provide an implementation route by means of mapping such abstractions onto the intermediate MONSTR formalism; and most importantly perhaps (ii), to reason about the behaviour of a program using these abstractions by revealing its run-time properties through an examination of the respective properties of the equivalent MONSTR rule system. In addition, a variety of prototype MONSTR based implementations of object methods can be examined, the aim being to study issues related to correctness and efficiency of execution, deadlock prevention, compiler optimisations, etc.

The rest of the paper is organised as follows: the next two sections introduce MONSTR and UFO and provide

*** Current attachment: Department of Interactive Systems, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Email: george@cwi.nl**

This work was done during the stay of the second author at GMD and CWI supported by an 1994 ERCIM fellowship.

some more incentives for this work; the following few describe mapping techniques from UFO to MONSTR and discuss a number of issues related to the understanding of the behaviour of the code; the paper ends with some conclusions and pointers to related and further work.

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.

2.1 The MONSTR Language

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.

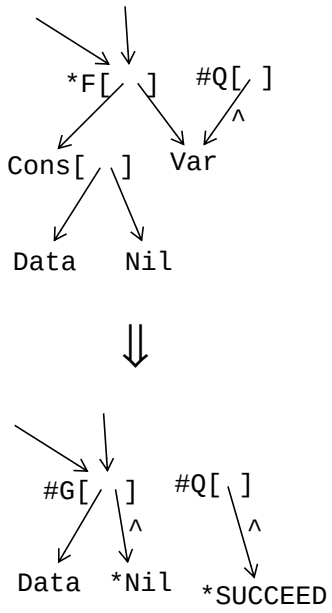
```
F[Cons[a b] s:Var]
=> #G[a ^*b], s:=*SUCCEED;
```

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 unlabeled children; such undefined nodes may match anything. Note that the pattern is shallow; this is fundamental to MONSTR as large patterns demand large scale locking to ensure atomicity.

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

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

The final tasks of a MONSTR rewrite are to make the root inactive (idle); 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 the diagram below.



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

The above assumed that there was a rule which matched. 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.

To make this a computational model suited to distributed machines, a number of rules are imposed on the syntactic structure of systems so that some useful runtime properties hold. Rather briefly:

- Symbols are divided into functions (which have rules but are otherwise unmatched), stateholders (which have no rules but are non-root redirectable) and constructors (which may just be matched).
- Functions have fixed matching templates, one level deep, with at most one stateholder in a prespecified position, and avoiding pointer equivalence testing of undefined nodes.[1]

• All nodes are balanced (i.e. have a $\#^n$ marking iff they have *n* notification out-arcs).
• All arcs are state saturated (i.e. if notification arcs, then either the child node is a stateholder, or is non-idle).
• The root is always redirected in a rule, and LHSs of redirections are not activated unless also the RHSs of other redirections.

The balancedness and state saturatedness conditions are provable runtime properties of execution graphs of systems that adhere to some simple restrictions. Otherwise the restrictions mainly address the requirements of pattern matching, and of garbage collection. See [1,2] for a fuller discussion.

2.2 Using MONSTR for Specification

MONSTR is fail-safe in that the restrictions that it imposes, guarantee that systems are implementable on distributed architectures. Nevertheless, when using it as a specification medium, certain of the designer's objectives may make particular locality assumptions reasonable. Under such circumstances, one may avail oneself of more powerful pattern matching in certain rules, knowing that overall system structure will make sure that no extreme demands are made of the underlying hardware. We will typically make use of this below, when we assume that the instance data associated with an object is localised at the object.

Furthermore, it is obviously more convenient to deal with bigger rewrites than small ones. However when we do this, we must always be clear whether (a): we expect the bigger rules to be taken at face value for locality reasons, (b): the bigger rules are being used as a readable shorthand for a truly more finegrained implementation. In the latter case, knowledge of rule overlaps, and of the worst possible concurrency to be expected during pattern matching a redex, can significantly influence the robustness needed in a strategy for breaking down the pattern matching into true one-level MONSTR rewrites. What must be avoided, is the possibility that schedules of finegrained rewrites cause unintended effects incompatible with the coarsegrained rules, if the latter are regarded as being specificational.

When considering a concurrent object-based language, the rule system generated by its translation to MONSTR can be used not only as an implementation route for the language in question, but also as a description of the intended operational semantics. This is because MONSTR is effectively a high-level machine language. The rule system generated by translation, is not only a representation of the original program but can also be quite an accurate representation of the underlying run-time data structures that must be set up and manipulated during the execution of the program; including rules that perform operations such as message queuing, protecting the internal state of objects during concurrent execution of their operations, synchronisation mechanisms, structure of objects, etc.

Thus the mapping of a concurrent OOP language to a model such as MONSTR provides among other things the ability to examine various ways of implementing object behaviour, and a framework for comparing different object-based languages. In this paper we tackle the first of these issues by showing how the concurrent OOP functional language UFO can be mapped onto MONSTR.

3. UFO

UFO ([8,9]) is a state-of-the-art concurrent functional language, essentially an object-based extension to SISAL, combining the data-flow and the actor models of computation. As such, it is particularly suited for parallel implementation and it is a natural candidate for being mapped onto MONSTR. UFO features among other things: class encapsulation, inheritance, array manipulation, strong typing and stateful computations. Here we concentrate on the basic principles of mapping a UFO object to a MONSTR rule system, paying particular attention to the interaction of state with concurrency, and leaving the discussion of the remaining issues for a future paper.

The first example is part of a purely declarative (stateless) object representing a complex number.

```
class Complex(re,im:Float)
mult(c:Complex):Complex is
   Complex(re*c.re-im*c.im,
           re*c.im+im*c.re)
square:Complex is
   self.mult(self)
end
```

The definition of the class comprises two parts, the declaration of the instance values, and of the class's methods and their implementations. All expressions in the body of a function are executed in parallel and their only means of synchronisation is through data dependencies. The special keyword `self` refers to the object in question and `x.f` is equivalent to `f(x)` but provides a message passing OOP style.

The next example, a counter object, will serve to introduce UFO's stateful objects.

```
stateful class Counter (initial:Int)
{ state i=initial }
proc inc (n:Int): Int is
  do new i=i+n return new i od
read: Int is i
end
```

The above code defines an object class whose instances comprise an imperative hidden variable `i` representing the

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

object's state initialised to the parameter `initial`, a declarative function `read` and an imperative procedure `inc` which updates the state accordingly. Note the UFO semantic feature that demands that updates to state are only allowed on a single-assignment basis, and the new values are referred to by using the `new` keyword. As a result, evaluation of the `return`ed expression above, is delayed until a reference to the new value is available. If stronger sequentialisation than that is needed, UFO provides the `finally` keyword, which forces all lexically preceding expressions to return a value, before any following evaluations can commence.

The final example shows how UFO supports conditional messages.

```
stateful class MyCounter
   inherits Counter
** inherits initial and i
accept read when i>0
proc dec is do new i=i-n return new i od
end
```

Note that the message `read` will only be served as long as the counter has a positive value; otherwise, it will be queued until the condition is satisfied.

4. Mapping UFO to MONSTR

To map a UFO program to a MONSTR rule system, a fresh instance of a class is created by a rule of the form:

```
New<i>ClassName</i>[<i>params</i>]
=> *self:Channel_Empty,
   #<i>ClassName</i>[self ^state],
   state:*InitVal<i>ClassName</i>[<i>params</i>];
```

where the italicised parts represent parametric information that varies from class to class. `self` is a stateholder representing a communication channel between the object and the rest of the world, *`params`* denotes the parameters for computing the object's initial instance values via the function `InitVal`*`ClassName`*, and `state` is one or more nodes corresponding directly to the object's state variables. MONSTR stateholders are used to capture the various possible communication patterns between the object and its environment, encompassing conditional message acceptance, etc. Ironically, we will see shortly that because of the clean way that the OO paradigm handles state when combined with single assignment, representing an object's state does not actually require stateholders!

Method invocation is modelled by sending an appropriate message to the object's channel, and inheritance is modelled by delegation and copy semantics. Thus the function symbol *`ClassName`* has a number of rules which handle: message input, spawning of subcomputations for the method invoked, and any channel locking required.

So, a possible translation of the class `Complex` to MONSTR (allowing two level matching because of the localisation of the method data at the object's input channel) is as follows:

```
NewComplex[r i]
=> *self:Channel_Empty,
   *Complex[self r i];

Complex[self:Channel[
   Mult[re im ans]] r i]
=> *Complex[self r i],
   self:=*Channel_Empty,
   newself:##NewComplex[
       ^##SUB[^*MUL[r re] ^*MUL[i im]]
       ^##ADD[^*MUL[r im] ^*MUL[i re]]],
   #Assign[ans ^newself];

Complex[self:Channel[Square[ans]] r i]
=> *Complex[self r i],
   self:=*Channel_Empty,
   newself:##NewComplex[
       ^##SUB[^*MUL[r r] ^*MUL[i i]]
       ^##ADD[^*MUL[r i] ^*MUL[i r]]],
   #Assign[ans ^newself];

Complex[self:Channel_Empty r i]
=> #Complex[^self r i];
```

where

```
Assign[s:STATE t] => *OK, s:=*t;
```

A number of issues are noteworthy regarding the above piece of code. Firstly, the instantiation rule has been optimised, as no non-trivial computation is needed to create the initial state from the parameters.

Secondly, message sending to an object is implemented asynchronously: anyone (with a reference to the object's input channel) can send it a message by launching a `Send`. This communicates with it via the following rules:

```
Send[c:Channel_Empty message]
=> *OK, c:=*Channel[message];

Send[c message] => #Send[^c message];
```

These rules say that a message arriving at an empty object is able to wake it and get processed immediately, while if the channel is busy, the `Send` has to wait until the message currently being processed has released the channel. Note that the rules for `Send`, must strictly adhere to the MONSTR restrictions, as by definition, no locality assumptions can be made about the business of message sending.

Thirdly, note that the queueing mechanism for messages is very natural. It is the suspension mechanism of the graph rewriting system itself. For this to be acceptable, i.e. for it

to agree semantically with the UFO definition which speaks of a concrete message queue, we need to convince ourselves that there will be no observable differences between our nondeterministic message processing strategy and a real queue. This only becomes an issue in the presence of conditional message acceptance, where processing of a message may be deferred as a result of its failing to satisfy some acceptance criterion; and this may happen several times. With a concrete queue, testing is always done in the same order, whereas with a nondeterministic strategy, this needn't be the case. For the two to be equivalent, we at least need there to be no observable consequences of performing a conditional message test. UFO aims for this by stipulating that the conditions of the test involve only *simple expressions* which are free of side effects.

Fourthly, note that computation within a method is constrained only by data dependency, but the object will only be able to service another message when the current method invocation has reset the input channel, and updated any state required (unnecessary in the example above). Since this resetting can happen before all the method's computation has completed, a number of methods can in principle be executing in parallel on the same object.

We now turn our attention to the counter example whose MONSTR implementation is shown below. Note again that we use more powerful pattern matching than the default "one stateholder and one level" for matching the condition of the object, since we know that the input channel and its contents will be localised at the object. In fact since the non-emptiness of the input channel locks the object for the exclusive use of the object rules themselves, the pattern matching itself can quite easily be broken down into several true MONSTR pattern matching steps which visit the various arguments in turn and collect the required information incrementally. This is because objects are not schizophrenic: there is only ever *one* function node in the graph per channel node that is responsible for the pattern matching on that channel node. This follows easily from the form of the RHS of rules for class instantiation.

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

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

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

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

Counter_Inter[self state]
=> *Counter[self state],
   self:=*Channel_Empty;
```

Note that the above structure of rules and of execution displays the fluidity one anticipates in a fully asynchronous model. When one makes use of the sequencing facilities provided by the `finally` keyword, extra machinery is needed to conform to the desired semantics, as is apparent when one considers the following contrived definition of an add function.

```
sillyadd(a,b:Int):Int is
{
    c=Counter(0);
    c!inc(a); c!inc(b)
 finally
    c.read
}
```

This is easy to translate to MONSTR given our previous work on the `Counter` object, since all of the procedures that need to update their state before the expression following `finally` can evaluate, return explicit values. The arrival of these values can be exploited to achieve the required synchronisation. Given this, `sillyadd` could be translated to MONSTR as follows:

```
SillyAdd[a b]
=> #Ind[^ans:STATE],
   ##Finally2.1[^ans1:STATE
                ^ans2:STATE s],
   c:*NewCounter[0],
   *Send[c Inc[a ans1]],
   *Send[c Inc[b ans2]],
   s:Send[c Read[ans:STATE]];
```

where

```
Ind[a] => *a;

Finally2.1[x y s] => *OK, *s;
```

If it were not the case that `Inc` returned a value, e.g. if there was no `return` in the declaration, and thus it was implemented as

```
Counter[self:Channel[Inc[value]] state]
=> #Counter_Inter[self
                  ^*ADD[value state]];
```

then the implementation would have to create an acknowledging version in order to implement `sillyadd`, viz. a version almost identical to that already given:

```
Counter[self:Channel[
   Inc_Ack[value] ack_chan] state]
=> #Counter_Inter[self
      ^newstate:*ADD[value state]],
   #Finally1.1[^newstate z],
   z:Assign[ack_chan OK];
```

A question that arose in the design of UFO is whether stateless classes would be able to inherit from stateful ones. If the answer were yes, then the MONSTR code for even stateless objects like `Complex` might actually need explicit acknowledging as above. A more natural answer is no, whereupon there is no need for this to happen, and this is the decision embodied in the UFO definition. Nevertheless note that even for stateful objects, the extra acknowledging mechanism is only rarely required, i.e. only when non-value-returning procedures are the synchronisers before a `finally` expression; as the normal conventions for unlocking an object by resetting its channel only when it is permissible to start processing a new message, ensure that an appropriate degree of atomicity is enforced.

As a further example, let us consider the `MyCounter` class, and the implementation of the conditionally accepted `read` function which only reads if the value of the counter is non-negative. The MONSTR translation yields the rules

```
MyCounter[self:Channel[
   Read[ans_chan]] state]
=> #MyCounter_Inter[self ^*GE[state 0]
      state ans_chan],
   *Counter[self state],
   self:=*Channel_Empty;

MyCounter_Inter[self True
   state ans_chan]
=> *OK, ans_chan:=state;

MyCounter_Inter[self False
   state ans_chan]
=> *Send[self Read[ans_chan]];
```

Several things are noteworthy about this translation. Firstly, method processing is again nondeterministic. Secondly, as specified by the UFO definition, the object is unlocked as soon as a copy of the state is taken, because only read access is required in completing the method call. This copying is implemented by just creating a reference to the state, as the particular state value itself is an instance of an immutable entity (a natural number), which will never be changed. Such a strategy works because creation of a new state value is always accompanied by the spawning of a fresh `Counter` function node which is to be the new state's unique parent. Under the circumstances, it is not necessary to redirect the old state to the new state as there is no parent of the old state that is not garbaged when all method processing pertaining to that particular state value completes. The correctness of this is linked to the invariant mentioned previously. In more detail, the following paragraph describes the key invariant informally.

> **At any instant of the computation, each object is represented by: a single input channel, one or more state values which are constructors, and a single object function node which is one of the input channel's parents and which is the unique parent of the state values. Processing of a method results in the creation of fresh constructor values (if update is required) and a fresh object function node, whose children are the input channel node and the (new) constructors. The input channel is redirected to a reset value, and the old function and old state values are garbaged.**

Thus the evolution of the state of an object is actually expressed by means more reminiscent of declarative programming, i.e. by using constructors, albeit a succession of them, rather than stateholders. The reason we are able to do this is because of the hiding of the state within the object, and the success of the approach is a particularly vivid manifestation of how elegantly state can be handled when the OO paradigm is combined with single assignment in a clean manner. Of course there must be *some* part which is non-declarative; it is to be found in the handling of the input channel, which *does* require a stateholder and non-root redirection.

It is important to realise what the above invariant does *not* say as well as what it does. It does not demand that all object processing is sequential. Many methods can run in parallel, working in principle on different values of the object's state, but methods are accepted sequentially, and there is always a uniquely determined state value that each new method call refers to. The processing of the next method call is delayed (by not resetting the `self` channel) until enough of the new state value is available; i.e. when the data dependencies arising from the `return` expression of the state update computation have been satisfied.

Thirdly, referring again to the example, it is obviously the case that if the acceptance test were not to terminate (something that we can safely disregard in the case of the `GE` function), all further method processing by the counter would block, and the counter object would deadlock. Indeed it is difficult to see how one can prevent the possibility of such deadlocks without restricting the allowable acceptance tests to a provably terminating set of expressions. UFO's insistence that acceptance tests contain only simple expressions encourages this, but does not guarantee it. Of course similar remarks hold whenever it is not guaranteed that a computation that ultimately updates the state of an object will terminate, as updating also requires the object to remain locked. The situation becomes

particularly fragile when state values and values used in acceptance tests are non-ground.

5. Towards UFO☺

Building on our previous remarks, we notice that the UFO definition, which stresses that concurrency should be maximised, (being at least at an abstract level limited only by data dependency), is in a sense too conservative in this regard, in keeping the `self` channel locked until the new state value is available. For consider the following.

The discipline that a message is left intact in a channel until explicitly cleared by the object, is enforced by the rules for `Send`, which cause `Send`s at a busy channel to wait patiently. On the other hand, when an object responds to a method call, once the message has been pattern matched and the appropriate rule selected, the object function node takes no further interest in the `self` channel apart from needing to unlock it sometime, since the structure of what needs to be done subsequently is encoded in the structure of the selected rule. This decouples the channel discipline from the state update discipline. Therefore the channel can be unlocked earlier, immediately after matching.

The authors thus introduce here a semantic variant of UFO, which we christen UFO☺, which features more concurrency by unlocking the `self` channel earlier. In fact in UFO☺, channel unlocking for stateless and stateful classes is done in exactly the same way. The MONSTR translations of UFO programs with UFO☺ semantics are different. Here are two examples. Firstly, the early release version of the `Inc` proc for `Counter`.

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

The channel is released immediately, but the RHS `Counter` node, being created suspended, will ignore any new message that a waiting `Send` might install while the updated value is being computed, until the computation returns and wakes the `Counter`. Transmission of the result to the caller is done asynchronously by the `Assign` which must also wait for the new value. In line with the intuition that minimising unnecessary dependencies creates simplicity, we note that there is now no need for a separate `Counter_Inter` function.

As another example, consider the explicitly acknowledging version of `Inc` according to UFO☺ semantics; again there is no intermediate function.

```
Counter[self:Channel[
   Inc_Ack[value] ack_chan] state]
=> #Counter[self ^y:*ADD[value state]],
   self:=*Channel_Empty,
   #Finally1.1[^y z],
   z:Assign[ack_chan OK];
```

6. On Queues

As we remarked before, the UFO definition explicitly speaks about the queueing of method calls. In an ideal world, where space and time are unlimited, and all conditional acceptance tests can be relied on to have no side effects and to terminate, there shouldn't be any observable differences between non-deterministic message processing and an explicit queue for a single execution - for the absence of side effects and the guarantee of termination make it impossible for an external observer to know whether any particular message has arrived and has been unsuccessfully tested one or more times, or whether it is still in transit.

This is related to the question of what communication model is appropriate for UFO-like languages. Most high level asynchronous languages specifically avoid making assumptions about the communication model for fear of making undesirable, expensive and constraining demands of implementors. These may come about surreptitiously because in the real world resources are limited, and the above-mentioned guarantees do not apply absolutely.

In view of these remarks, and to get closer to the letter of the law on UFO, let us examine the incorporation of an explicit message queue in our translations. As an example we show how conditional message acceptance can be handled at the MONSTR level by translating part of the `MyCounter` example. Note that the introduction of queues requires new machinery, hence the `Q`-prefixes.

```
NewQMyCounter[init]
=> *self:QChannel[qhead],
   qhead:QNil,
   *QMyCounter[self qhead qhead init];
```

(*The queue is initialised empty.*)

```
QSend[self:QChannel[qnil:QNil] mess]
=> *OK, nq:QNil,
   self:=QChannel[nq],
   *Assign[qnil Channel[mess nq]];
```

(*The higher granularity* `QSend` *function for a queue.*)

```
QMyCounter[self qhd curr:Channel[
   mess:Read[ans] nxt] state]
=> #IF[^*GT[state 0] thn els],
   thn:QMyCounter_Inter[self qhd
      curr state],
   els:QMyCounter[self qhd nxt state];

QMyCounter[self qhd curr:QNil state]
=> *MyCounter[self qhd qhd state];

QMyCounter_Inter[self qhd curr:Channel[
```

```
   mess:Read[ans] nxt] state]
=> *QMyCounter[self qhd curr state],
   curr:=*nxt, *Assign[ans state];
```

where

```
IF[True t e] => *OK, *t;
IF[False t e] => *OK, *e;
```

(*The* `QMyCounter` *function runs round the queue looking for a message it can respond to*.)

Note that the queue reduces concurrency. Until the result of the conditional test is known, we do not know whether to remove the current message from the queue or not; and we dare not launch a fresh `QMyCounter` early in case it runs right round the queue and processes the same message twice, ruining the main invariant (our simple queue suffers from a busy waiting overhead, though more sophisticated code can be written to avoid this).

7. Using the Translations

Above we have seen that the finegrained rule based approach of MONSTR is well suited to discussing precise details of synchronisation for the primitives of higher level languages, due to its simple yet rigorous operational semantics. The strategy may be extended to give translations of the whole of (programs in) a high level language into collections of finegrained rules. This has certainly been done in the past ([4,5]).

However this approach, when carried out uncritically, can yield a mass of finegrained rules, which quickly become difficult to comprehend for humans, and inefficient to execute by machine. Our recommendation is that apart from the use of MONSTR for clarifying finegrained details of synchronisation etc., if appropriate justification in terms of locality of action is provided, larger granularity primitives may be designed to act as basic building blocks for implementations. All could have their semantics defined via translation into MONSTR rule collections, and their appropriateness could be judged by the ease or otherwise of the resulting serialisability proofs.

8. Conclusions

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 of and implementation techniques for objects, etc. In [7] it is argued that there is a need to develop semantic frameworks capable of allowing reasoning about the way various features of concurrent OOP languages operate and to provide a common point of reference in comparing various such languages.

In this paper we have demonstrated the utility of MONSTR as such a framework. Although we have concentrated on a specific language model (UFO), we can show MONSTR's applicability to many "semi-declarative" languages (languages beyond the functional world, i.e. that support state), such as concurrent constraint ones ([6]). Language features mapped onto equivalent sets of MONSTR rewrite rules can be reasoned about in the same rigorous way as has already been done for process calculi ([3]), and as has been glimpsed above in the discussion of our object invariant. Furthermore, one is also able to reason precisely about variants of a single language, as can be done in our case by examining the different MONSTR code needed to describe different versions of UFO (e.g. [8,9], and as shown in our fleeting discussion of UFO☺), especially wrt the interaction between concurrency, state and method invocation.

Acknowledgment: The authors would like to thank John Sargeant for comments on an earlier version of the paper.

9. References

[1] R. Banach, "MONSTR: Term Graph Rewriting for Parallel Machines", in [10], pp. 243-252.

[2] R. Banach, "MONSTR I — Fundamental Issues and the Design of MONSTR", submitted to the *JUCS*, 1995.

[3] R. Banach, J. Balazs and G. A. Papadopoulos, "A Translation of the Pi-Calculus into MONSTR", *Journal of Universal Computer Science*, Springer Verlag, Vol. 1, No. 6, 1995, pp. 335-394.

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

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

[6] M. Henz, G. Smolka and J. Wurtz, "Object-Oriented Concurrent Constraint Programming in Oz", *PPCP*, MIT Press, Cambridge, MA, 1994, pp. 27-48.

[7] O. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents", *OOPSLA/ECOOP '90*, ACM Computer Society Press, Ottawa, Canada, Oct. 21-25, 1990, pp. 38-43.

[8] J. Sargeant, "Uniting Functional and Object-Oriented Programming", *1st JSST*, Kanazawa, Japan, Nov. 4-6, 1993, LNCS 742, Springer Verlag, pp. 1-26.

[9] J. Sargeant, C. Kirkham and S. Hooton, "*UFO 1.0 Reference Manual*" (DRAFT), UFO Group, Department of Computer Science, University of Manchester, 1995.

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

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