Object Graph Rewriting: An Experimental Parallel Implementation

John Glauert *
School of Information Systems
University of East Anglia
Norwich NR4 7TJ, UK
Tel: +44 603 592671
Fax: +44 603 507720
jrwg@sys.uea.ac.uk

Abstract

Object Graph Rewriting (OGRe) is a computational model which combines ideas from graph rewriting and process calculi. OGRe has been designed for implementing multi-paradigm languages such as Facile, which combines functional and concurrent programming.

Simple objects, which have states represented by tree-structured terms, respond to messages, also represented as terms. New objects may be created and new messages sent, while the original object either remains unaffected or changes state. The model is inherently concurrent, but fine-grained, so distributed implementation is not necessarily beneficial.

We introduce OGRe with some simple examples and explain how primitives are added which can trigger concurrent execution when appropriate.

1 Introduction and Background

Designers of multi-paradigm languages need to find a common semantic framework in which to unite the paradigms available in the language. At the implementation level it is necessary to consider the characteristics of the computational models concerned. For example: in conventional languages, dataflow, and strict functional languages computation proceeds immediately; in lazy functional languages, computation only takes place when needed, or demanded; computations in concurrent logic languages may block waiting for instantiation of shared variables.

The present work arose out of a study of implementation techniques for Facile [3], a multi-paradigm programming language which combines functional and concurrent programming in a symmetric fashion. The existing practical implementation of the language [7] uses Standard ML to provide the functional part of the language, with concurrency primitives introduced through a library of new system functions. In that implementation the process features of Facile map to threads, and are therefore more expensive to use than the functional features.

In its pure form, Facile enhances the λ–calculus with primitives for process spawning and channel-based communication in the style of CCS [11]. Using the π–calculus [14] and its polyadic form [13], Milner shows that a λ–expression may be translated to a process network which simulates the λ–expression [12]. Similar work was done by Leth [8].

Building on such process translations of the λ–expression new translations were developed to map both the functional and concurrent features of Facile into networks of small processes. This provides a uniform representation of all features of the language, so that the relative cost of using the concurrent features of Facile is much reduced. The ultimate goal is to exploit the inherent concurrency of the process networks in an implementation on a parallel machine.

A number of options were considered when selecting the underlying process calculus. One approach was to use process notations close to the polyadic π–calculus [13] in which communication is name or channel-based. Translations such as those of Milner may be employed as reported in [5] and [4]. In these models the names can be thought of as independent entities which act as brokers, arranging communications between processes. Similar ideas appear in [18].

However, when the translation schemes mentioned above are analysed we find that: the channels used are closely associated with a particular process; this process is the only one to receive messages from the channel; and furthermore, the process receives messages from no other channel. Finally, no process which sends a message needs to synchronise with a remote process able to receive the message.

Exploiting these facts, a model was developed in which communicating agents use asynchronous process-based (or location-based point-to-point) communication. This contrasts with the synchronous channel-based (or name-based) communication of CCS, and the π–calculus. The essence of this process calculus is captured by a rewriting model *Object Graph Rewriting* (OGRe) which can be seen as a very minimal version of Paragon [1]. Although the first application of OGRe has been in the implementation of Facile, the model is general and can be used to implement a number of other computational models rather directly. The characteristics of the model have been chosen carefully in order to allow a very simple implementation of OGRe on existing hardware.

In the next section we introduce the Object Graph Rewriting model of computation. Section 3 provides an outline of the way in which the model may be used to model a number of programming language styles. Section 4 discusses the existing implementation of OGRe and potential for future exploitation of the model. It is argued that the design of OGRe makes it possible to generate an efficient pseudo-parallel implementation on sequential machines or to provide medium- or fine-grain parallel processes for hard-

*This work was supported by the Engineering and Physical Sciences Research Council of Great Britain under grant GR/H 41300.

ware architectures able to exploit low level concurrency.

2 OGRe: A Small Process Language

The *Object Graph Rewriting* (OGRe) language models a computation by a set of named processes which exchange messages. Pattern-directed rewriting rules determine the way in which a process responds to the arrival of a message.

Computation proceeds by picking a message and the process to which it is directed, and replacing them by new processes and messages as specified by a matching rewrite rule.

Figure 1 shows a set of OGRe rules and figure 2 traces a computation based on the same rules.

Each OGRe process has a name taken from an unbounded set of *process names* such that each newly created process has a distinct name.

An OGRe system has a finite set of *state term symbols*, used to construct process states, and a finite set of *data term symbols*, used to construct messages, and also subterms of process states and messages. The symbol sets are disjoint. Each symbol has a fixed arity which determines the number of arguments of terms which are created. Term arguments may also include process names and *primitive data values*, such as integers.

OGRe rules may also use *primitive operation symbols* (disjoint from other symbols) for representing basic functions over primitive data values. Rules also use variables which are bound during pattern matching to process names, data terms, or primitive data values.

OGRe Processes

An example of a declaration an OGRe process with name m is $m : LSubL(p, Int(15))$. The process state is given by the term $LSubL(p, Int(15))$ with a state term symbol and zero or more arguments. Arguments may be process names, primitive data values and data terms. In the example above, the process has state term symbol $LSubL$ and two arguments. The first, p, is a process name and the second, $Int(15)$ is a data term with a primitive data value as argument.

OGRe Messages

An example of a message specification is $a \,!\, R(Int(2))$. Messages are directed to a named process and have contents defined by a data term. In the example, the target process is named a and the data term of the message has symbol R and a single argument $Int(2)$ which represents a boxed integer value.

OGRe Rules

Computation in OGRe is determined by pattern-directed rewriting rules. The pattern for a rule has a pattern for the state term of a process, and a pattern for the data term for a message directed to that process. The pattern contains a variable that will be bound to the name of the root process, and may contain other variables which can match process names, data values, or data terms in the process state or message.

The right hand side of an OGRe rule always contains a redeclaration for the root process named in the pattern. Often, for *functional* processes, the state is unchanged, but in rules for *mutable* processes the arguments and even the symbol of the state term may be changed.

No variable may appear more than once in the pattern, except for the name of the root process which is also the target of the message. A state term may not consist of just a variable. All variables naming processes declared in the body must be unique. The variable naming the root process must be included, but all other variables in the pattern may only be used as arguments to terms. Variables naming the result of applying primitive functions must be unique. All variables occurring only in the body must name results of primitive functions, or processes declared in the body.

Figure 1 illustrates a possible application of OGRe by giving an example set of rules which model a computation by dataflow. This example has been chosen because it illustrates a range of the features of OGRe. It is not intended as a realistic example of how OGRe will be used in practice. A number of other applications are given in the next section.

In dataflow, an operator waits for the arrival of a number of tokens carrying data values. The operator then computes a result and sends it on to a further operator. When modelling dataflow in OGRe in this way, operation nodes become processes and data tokens become messages.

In line with the tagged dataflow model of the Manchester Dataflow Machine [6], we will label tokens as left or right arguments using symbols L and R. When processes representing nodes are created, they will take an argument indicating the node to receive the result. The state symbol of the process indicates which argument of the successor node will be formed by the result token. Hence a process with state $RMul(p)$ creates a result token which will become the right-hand argument for process p.

The first rule

$$\begin{aligned}&e : Start,\ e \,!\, POMTrigger \rightarrow \\ &\quad p : PrintL,\ m : LSubL(p, Int(15)),\ a : RMul(m), \\ &\quad a \,!\, L(Int(6)),\ a \,!\, R(Int(2)),\ e : Start;\end{aligned}$$

will always match the initial configuration of an OGRe computation. In this case it creates three new processes and two new messages, both in this case directed to the same process. The state of the root process, $Start$, is unchanged so the process is functional.

The rules for $RMul$, $RMulL$ and $RMulR$ in figure 1 model dataflow nodes which multiply their arguments and send their result to a destination node as its right-hand argument. The two $RMul$ rules take in the first argument and change the process state to hold the first argument and await the second. The auxiliary rules $RMulL$ and $RMulR$ await right, or left-hand arguments respectively. They compute a result, send it on, and become a $Done$ process.

The processes are *mutable* since the first argument to arrive must be captured and will affect the behaviour when the second arrives. Dataflow nodes which take a literal argument can be modelled by auxiliary nodes with suitable arguments as in the process $m : LSubL(p, Int(15))$ which awaits an argument and will subtract it from 15. This corresponds to an intermediate state which would arise if a process $m : LSub(p)$ received a message $m \,!\, L(Int(15))$ where $LSub$ is defined by analogy with $RMul$.

The rule for $RMulL$

$$\begin{aligned}&r : RMulL(d, Int(a)),\ r \,!\, R(Int(b)) \rightarrow \\ &\quad c = PMul(a, b),\ r : Done,\ d \,!\, R(Int(c));\end{aligned}$$

$$
\begin{aligned}
& e : Start,\ e\,!\,POMTrigger \rightarrow \\
& \qquad p : PrintL,\ m : LSubL(p, Int(15)),\ a : RMul(m), \\
& \qquad a\,!\,L(Int(6)),\ a\,!\,R(Int(2)),\ e : Start; \\
& r : RMul(d),\ r\,!\,L(l) \rightarrow \\
& \qquad r : RMulL(d, l); \\
& r : RMul(d),\ r\,!\,R(l) \rightarrow \\
& \qquad r : RMulR(d, l); \\
& r : RMulL(d, Int(a)),\ r\,!\,R(Int(b)) \rightarrow \\
& \qquad c\ =\ PMul(a,b),\ r : Done,\ d\,!\,R(Int(c)); \\
& r : RMulR(d, Int(a)),\ r\,!\,L(Int(b)) \rightarrow \\
& \qquad c\ =\ PMul(a,b),\ r : Done,\ d\,!\,R(Int(c)); \\
& r : LSubL(d, Int(a)),\ r\,!\,R(Int(b)) \rightarrow \\
& \qquad c\ =\ PSub(a,b),\ r : Done,\ d\,!\,L(Int(c)); \\
& r : PrintL,\ r\,!\,L(v) \rightarrow \\
& \qquad p : Print,\ r : Done,\ p\,!\,Unit(v);
\end{aligned}
$$

Figure 1: OGRe dataflow rules for an arithmetic expression

illustrates the final form of element which can appear on the right hand side of a rule: $c\ =\ PMul(a, b)$ represents a primitive function to multiply data values for incorporation in new processes and messages.

OGRe Rewriting

OGRe computation involves a series of rewrites corresponding to the reception of messages by processes. Each rewrite consumes a message, but may create new messages and processes. Messages correspond to tasks to be performed, and when no further messages exist, computation ceases.

When a process receives a message, the state term of the process and the data term forming the message are matched against the patterns of the OGRe rules. When a match is found, the message is absorbed, the state of the process may be changed, and new process and messages may be generated. Rule application may also apply primitive functions, such as the arithmetic operations on integers.

A process is only involved in computation if it receives messages. If a process is not the destination of an existing message, and its name is not the argument of a state term or data term in some other process or message, then the process can never receive messages and may be garbage collected.

Figure 2 shows the sequence of execution for a dataflow computation which computes $15-(6*2)$ and sends the result to a function *Print*. The result of the multiplication will become the right-hand argument of the subtraction node. The result of the subtraction becomes the left-hand (and only) argument of the print function.

The reader can check that the result would be the same if the right-hand input to a had been considered first. As tokens move through the graph the unreferenced *Done* processes are garbage collected.

Further OGRe Constraints

Whenever the execution configuration contains a message it is possible to rewrite the configuration by considering the contents of the message and the state of the process to which it is directed. An OGRe rewriting rule matches if the state of the configuration process is an instance of the state of the process in the rule pattern, and the contents of the configuration message in is an instance of the contents of the message in the rule pattern. It may be that more than one rule matches in this way, in which case the only admissible rule is the earliest in the ordered set of rules. We place the very strong condition on the rule system that whenever a message is addressed to a process, either in the initial term, or following some rewriting steps, there must be an admissible rule for the message and its target process. This condition avoids the need to queue messages in an implementation.

When the admissible rule has been identified, rewriting takes place as follows: To avoid potential name clashes we make an isomorphic copy of the rule with no process names in common with the execution configuration. We then substitute the name of the configuration process for the name of pattern process throughout the rule and apply to the whole rule the substitution which makes the configuration process and message match the pattern. The rule body may contain primitive operations, and we require that they should have appropriate arguments. The primitive operations are applied to their arguments and replaced by the corresponding results. The final step is to remove the process and message from the configuration and add the body. A new instance of the process will always be present, and will often be an exact copy of the original.

OGRe execution continues by repeated rewriting of process–message pairs until all messages are eliminated. Concurrent rewrites may be performed as long as a given process state takes part in only one rewrite at a time. It is possible to garbage collect a process whose name does not appear in the state of any other process nor as the destination or in the contents of any message.

As has been mentioned, symbols labelling state and data terms are distinct. Symbols have fixed arity. Further, we associate a type with symbols so that a particular argument must always refer to either a process name, a data term, or primitive data. This enables us to check that all messages will be directed to processes able to receive them.

$$
\begin{array}{ll}
 & e : Start,\ e \,!\, POMTrigger \\
\Rightarrow & p : PrintL,\ m : LSubL(p, Int(15)),\ a : RMul(m), \\
 & a \,!\, L(Int(6)),\ a \,!\, R(Int(2)) \\
\Rightarrow & p : PrintL,\ m : LSubL(p, Int(15)),\ a : RMulL(m, Int(6)), \\
 & a \,!\, R(Int(2)) \\
\Rightarrow & p : PrintL,\ m : LSubL(p, Int(15)),\ a : Done \\
 & m \,!\, R(Int(12)) \\
\Rightarrow & p : PrintL,\ m : Done \\
 & p \,!\, L(Int(3)) \\
\Rightarrow & p : Done,\ q : \mathbf{Print} \\
 & q \,!\, \mathbf{Unit}(Int(3))
\end{array}
$$

Figure 2: Evaluation of a dataflow example

3 OGRe Applications

In this section we illustrate the power of OGRe by showing how a number of models of computation may be represented in a direct way.

Modelling Dataflow

The example in the previous section illustrated a model for dataflow in OGRe. The example contained no parallelism, but it is clear that in general there might be many token messages travelling between nodes, and hence many possible concurrent rewritings. The number of tokens must increase when a result is needed by several successors, and a duplication node is used.

In Figure 3, we give the rule for duplicating tokens and an example of how it could be used to form the square of a number. Tracing execution of the example is left as an exercise for the reader!

Similar rules *LLDup*, *RLDup*, and *RRDup* would be needed in other circumstances.

Modelling Communication Channels

The OGRe model is based on asynchronous direct communication between processes. There are no intervening channels as in CCS, or the Polyadic π–Calculus. However, channel-based communication can be modelled by OGRe processes. In Figure 4, a channel is represented by a process which co-ordinates communication between producing and consuming processes. A producer sends a *Put* message to the channel, with the data as argument. A consumer sends a *Get* message with the identity of the process to receive the data. Data is queued until a consumer is known. Unsatisfied requests are queued explicitly until a producer provides data.

The model in Figure 4 supports asynchronous communication since the producer receives no notification that data has been consumed. Unsatisfied messages are stacked, so there is no fairness about this particular model. Surplus data values form a stack, VQ, of available values, while surplus producer names form a stack, RQ, of requests. Observe the order in which requests are satisfied in the example execution in Figure 5. The initial state of the channel holds an empty queue, EQ. The two *Put* actions stack values so that in this trace the *Get* actions receive values in the reverse order.

It should be clear that the model for asynchronous channels could be modified to implement semaphores. No actual data values need be communicated, or stored.

Synchronous communication is possible if we arrange that a notification message, *Sync*, is sent to the producing process when the communication is completed. The producer must include the identity of the process to receive synchronisation when a data value is sent. Further computation by the producer can be blocked until the communication is completed as shown in Figure 6. Note how an unsatisfied request blocks a thread of control as the channel process changes state but does not generate further activity. When a communication completes, a new thread is activated as indicated by the two messages generated, one of which resumes the suspended thread of control.

Modelling State and Logic Variables

The channel model above uses a process to hold the state of the channel as a list of data values or requests. A simple von Neumann storage cell can be implemented with *Set* and *Read* operations as in Figure 7. The *Set* message carries data and the identity of a process to receive acknowledgement in order to provide serialisation. We return the old contents to provide test-and-set functionality.

Figure 8 shows that by adding slightly more sophisticated behaviour, we can model a logic variable in the framework of concurrent logic programming [16]. The model suggested is for illustration only, as the properties of such variables vary from language to language. We will assume that an attempt may be made to *Bind* an unbound variable, $UVar$. This will fail if the variable is already bound. A variable may be *Read*, which will lead to blocking of the computation if the variable is unbound. There is also an extralogical $IsVar$ test for examining the state of the variable. An unbound variable holds a list of suspended readers. The initial state of an unbound variable is $UVar(E)$. The rules for $Free$ are used to release suspended readers. In order to process the rest of the list, the process sends a further message to itself.

$$
\begin{array}{l}
e : Start,\ e\,!\,POMTrigger \to \\
\qquad p : PrintL,\ m : LMul(p),\ d : LRDup(m, m), \\
\qquad d\,!\,L(Int(6)),\ e : Start; \\
p : LRDup(d, e),\ p\,!\,L(x) \to \\
\qquad d\,!\,L(x),\ e\,!\,R(x),\ p : Done;
\end{array}
$$

Figure 3: Duplication of dataflow tokens

$$
\begin{array}{l}
c : Chan(VQ(v, q)),\ c\,!\,Get(r) \to \\
\qquad r\,!\,Data(v),\ c : Chan(q); \\
c : Chan(q),\ c\,!\,Get(r) \to \\
\qquad c : Chan(RQ(r, q)); \\
c : Chan(RQ(r, q)),\ c\,!\,Put(v) \to \\
\qquad r\,!\,Data(v),\ c : Chan(q); \\
c : Chan(q),\ c\,!\,Put(v) \to \\
\qquad c : Chan(VQ(v, q));
\end{array}
$$

Figure 4: Rules for Asynchronous Communication

$$
\begin{array}{ll}
& c : Chan(EQ),\ c!Put(3),\ c!Put(4),\ c!Get(x),\ c!Get(y),\ c!Get(z) \\
\Rightarrow & c : Chan(VQ(3, EQ)),\ c!Put(4),\ c!Get(x),\ c!Get(y),\ c!Get(z) \\
\Rightarrow & c : Chan(VQ(4, VQ(3, EQ))),\ c!Get(x),\ c!Get(y),\ c!Get(z) \\
\Rightarrow & c : Chan(VQ(3, EQ)),\ c!Get(y),\ c!Get(z),\ x!Data(4) \\
\Rightarrow & c : Chan(EQ),\ c!Get(z),\ x!Data(4),\ y!Data(3) \\
\Rightarrow & c : Chan(RQ(z, EQ)),\ x!Data(4),\ y!Data(3)
\end{array}
$$

Figure 5: Asynchronous Communication

$$
\begin{array}{l}
c : Chan(VQ(v, s, q)),\ c\,!\,Get(r) \to \\
\qquad r\,!\,Data(v),\ s\,!\,Sync,\ c : Chan(q); \\
c : Chan(q),\ c\,!\,Get(r) \to \\
\qquad c : Chan(RQ(r, q)); \\
c : Chan(RQ(r, q)),\ c\,!\,Put(v, s) \to \\
\qquad r\,!\,Data(v),\ s\,!\,Sync,\ c : Chan(q); \\
c : Chan(q),\ c\,!\,Put(v, s) \to \\
\qquad c : Chan(VQ(v, s, q));
\end{array}
$$

Figure 6: Rules for Synchronous Communication

$$
\begin{array}{l}
c : Cell(v),\ c\,!\,Set(n, r) \to \\
\qquad r\,!\,Data(v),\ c : Cell(n); \\
c : Cell(v),\ c\,!\,Read(r) \to \\
\qquad r\,!\,Data(v),\ c : Cell(v);
\end{array}
$$

Figure 7: A model for von Neumann cells in OGRe

$$
\begin{aligned}
&c : UVar(q),\ c\,!\,Read(r) \rightarrow \\
&\qquad c : UVar(Q(r,q)); \\
&c : BVar(v),\ c\,!\,Read(r) \rightarrow \\
&\qquad r\,!\,Data(v),\ c : BVar(v); \\
&c : UVar(q),\ c\,!\,Bind(d,r) \rightarrow \\
&\qquad r\,!\,Succ,\ f : Free(q,d),\ f\,!\,Go,\ c : BVar(d); \\
&c : BVar(v),\ c\,!\,Bind(d,r) \rightarrow \\
&\qquad r\,!\,Fail,\ c : BVar(v); \\
&c : UVar(q),\ c\,!\,IsVar(r) \rightarrow \\
&\qquad r\,!\,Succ,\ c : UVar(q); \\
&c : BVar(v),\ c\,!\,IsVar(r) \rightarrow \\
&\qquad r\,!\,Fail,\ c : BVar(v); \\
\\
&f : Free(Q(r,q),d),\ f\,!\,Go \rightarrow \\
&\qquad f\,!\,Go,\ r\,!\,Data(d),\ f : Free(q,d); \\
&f : Free(E,d),\ f\,!\,Go \rightarrow \\
&\qquad f : Done;
\end{aligned}
$$

Figure 8: A model for logical variables in OGRe

Modelling Facile

The main application of OGRe has been in the implementation of Facile. The OGRe model results in a very low-level translation of Facile features in which potential implementation data-structures become visible. For example, Facile channels can be represented as objects which are manipulated in exactly the same way as in the Chemical Abstract Machine [2] proposed for Facile in [9]. The channel representation is closely related to the model for synchronous channel communication above.

Use of process spawning introduces potential concurrency and channels may be used to synchronise and communicate between spawned processes. Otherwise, Facile functions execute strictly sequentially, adopting the same semantics as ML. The translation of the functional subset of Facile is a development of the work reported in [5] and [4]. Within this subset, every rewrite generates precisely one new message so that there is always a single thread of control.

The translation to OGRe is not very direct, involving several OGRe rewrites per function call. However, there is considerable scope for optimisation by symbolic evaluation. In particular, the translations produce code with all primitive values boxed, but optimisation will achieve unboxing in many situations.

4 OGRe Implementation

We will summarise the current state of implementation of OGRe and outline potential for future work. A translator has been written in SML which will convert a λ–expression to OGRe. The input may either be a text file or an SML program by using the SML compiler to generate its internal lambda form. The translator is able to interpret OGRe code or may generate corresponding C for compilation.

Memory Organisation

State and data terms are represented by consecutive memory words containing a symbol followed by arguments. Since OGRe symbols are typed, we can, without loss of generality, insist that all primitive data arguments precede process names and data terms (both of which are represented by machine pointers). This simplifies garbage collection as every term contains one or more data words (there is a symbol at least) followed by zero or more pointers.

Symbols for state terms map to positive integers, distinguishing them from data symbols which use negative integers.

Rule Matching

The current C code provides a reasonably efficient simulation of the fine-grain concurrency of OGRe execution. Code for rules is compiled into a switch statement, and selected according to the state symbol of the process receiving a message.

A task queue is maintained of messages to be processed and the code for the symbol of the topmost target process is executed. Matching proceeds by testing for data symbols as required and building up a table referencing processes and data values bound to pattern variables. When matching succeeds, new processes may be created with state arguments derived from values matched to pattern variables. Messages may be created and queued for further consideration.

Often a rule will generate exactly one new message. This sequential case is optimised so that no manipulation of the queue is required. If more than one message is generated, messages are pushed onto the task queue. If no new messages are generated, the task queue is popped and the next message selected. Execution ceases when the queue becomes empty.

The effect of this is to maintain sequential threads where possible. In the Facile translation, only rules for the concurrency features change the number of messages; the functional part generates code which always produces one new message per rewrite. Generating no messages corresponds to waiting for communication, or process termination. Generating extra messages corresponds to completion of communication, or process creation.

Parallel OGRe Machine

By making a small number of changes to the OGRe implementation we have developed the *Parallel OGRe Machine* (POM). It is an extension of the sequential model discussed above so that if none of its features are used, execution is as before.

We note that the threaded execution discussed above is still of interest with or without a parallel dimension since it allows scope for optimisation and hiding of latency.

We can make some observations about the representation of OGRe processes and data. Firstly, only state terms can be updated by rewriting – all data terms are immutable. (Although this might seem restrictive, mutable arrays, for example, can simply be represented by processes). Secondly, there is no way to create cycles in data terms, except via process states. However, data terms can share subterms and can therefore be DAGs. Thirdly, identical functional processes can be shared, or copied and unshared, without changing the meaning of an OGRe program. This is, of course, not the case for mutable processes.

This means that we could safely export functional processes from one processor to another. When copying process arguments we can regard them as tree-structured. Copying will stop at process names although it would be safe to copy processes which are known to be functional.

Remote Pointers

When a term is transferred from one processor to another, the strategy adopted is to transfer all arguments which are primitive data values and data terms, but processes named in arguments are left where they are. (All that an OGRe rewrite can do with a process name is to copy it or send a message to it. The rewrite can tell nothing about the location or state of the process).

When a term is transferred to a remote machine, any process names it references must be replaced by a *remote pointer*, which is a reference back to the original process. Each POM processor will have a table referencing processes which have remote pointers to them. A remote pointer is a pair (Q, o) where Q is an identifier for the processor and o is the offset in the table on the processor.

The table is needed so that the garbage collector can move nodes around within a processor: the table offset remains unchanged, so the remote processor can still refer to the node by the (Q, o) pair.

If a processor, Q, has a process, n, known by remote pointer (Q, o) then on another processor, P, there will be a *relay process* m. Messages sent to m on P are forwarded to n on Q. The relay process m has the form m : **POMRelay**(Q, o) where both Q and o are data values as shown in Figure 9.

Sending Messages to Remote Processes

The message to be transmitted is sent using normal OGRe processing to a node with function **POMRelay**(Q, o). The code for this is implemented specially, but just appears with the OGRe rules in the switch statement for the program. It will pack the contents of the message (possibly a large term) and send it to the remote processor, Q, indicating the message is for the node at offset o. The receiving processor simply unpacks the message and directs it to the node pointed to by the table entry at offset o.

To pack a message it is sufficient to record the number of data items, then the number of pointer entries, then the data values, and then recursively pack each of the subterms given by pointer values.

If a pointer refers to a data node then we pack the node recursively. If the pointer, r, refers to a process state, then a remote pointer to that process must be created. We allocate a slot, i, in the table on the local processor, P, and place a pointer to r in the slot. To represent the remote pointer we pack the information for **POMRelay**(P, i). If we assume that P and i take an integer each, the representation might be: $[3, 0, POMRelay, P, i]$. When this is unpacked at the receiving node it will create a process which is ready to receive messages and pass them back via the remote pointer to node r.

The receiving processor simply unpacks the message, building a tree structure of nodes, and makes a local task to send it to the node indicated by the slot contents as shown in Figure 10.

Process Spawning

So far we have not explained how processes can be exported to remote processors. We adopt the convention that processes for export are represented by a functional process state which can receive a special message **POMTrigger** which takes no arguments. We assume that all POM processors are loaded up with the full code for the rewrite system.

The fact that **POMTrigger** takes no arguments does not impose restrictions. If a task taking arguments is to be exportable, it can be rewritten to capture the message, create a new process with the original process arguments plus the message as an extra argument, and send **POMTrigger** to it.

When transferring work to remote processors we must also consider load balancing. Not every exportable process need be exported. Instead, it could be handled by the local processor. We will discuss ZAPP [10] style process scheduling later.

Tasks are associated with messages, and exportable tasks with **POMTrigger** messages. The queue of local tasks is used to hold normal messages while a new queue of *exportable* tasks is used for locally-generated **POMTrigger** messages.

If the local queue becomes empty, tasks can be moved from the exportable queue to the local queue. The task is not exported, but is executed locally using exactly the same code. On a single processor all exportable processes are handled locally with almost no penalty associated with the fact that they could have been exported.

On a multiprocessor POM machine, tasks from the exportable queue may be transferred to remote processors: the **POMTrigger** message is removed from the exportable queue; the destination process for the message is packed, generating **POMRelay** nodes for references back to the local processor, and sent to the remote processor. In this case, the root node will be a process state, but is packed nevertheless.

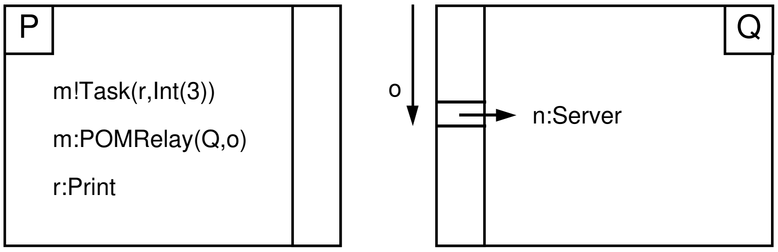

Figure 9: Message to Remote Pointer

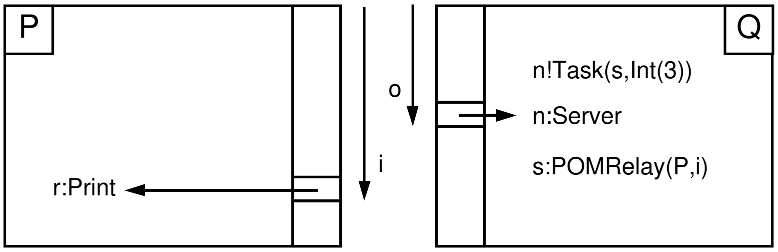

Figure 10: Message Transferred to Remote Processor

At the receiving processor, the message is unpacked and a **POMTrigger** message is placed in the *local* message queue directed to the newly unpacked process state. Since the imported task is in the local queue, it cannot get exported again.

In practice, the process description can be a message sent to a standard **POMSpawn** process on the remote processor. This is a built-in function like **POMRelay**, and is accessible from a standard slot in each local processor table. The mechanism for handling spawned processes is therefore exactly the same as for external messages: the message carries a destination slot; the receiving processor unpacks the message and sends it to the node indicated by the slot (by placing a task in the local message queue).

In the case of spawned tasks, the process description arrives as a message sent to **POMSpawn** which has the unorthodox behaviour of setting up a task which delivers a local **POMTrigger** message to its own "message".

Note that all the messages sent between POM processors can be handled this way: a small set of standard slot locations are reserved for the functions required, and the necessary code is generated in the switch statement used for all OGRe rules.

ZAPP Scheduling

The Zero Assignment Parallel Processor model of Burton and Sleep [10] has a long history, and is rediscovered on a regular basis.

The aspects of the model of interest to us at present are as follows. ZAPP processors try to keep busy, so they only export work when then have enough to keep them going. When they are busy they do not look for work from other processors. Hence work is only moved around when there are some processors with little work to do and others with more than they need. Processors act according to their local level of work. There is no attempt to balance global load in the machine, but this happens as a consequence of local actions.

Each ZAPP processor maintains a list of local or fixed tasks, and a list of exportable tasks. If the overall number of tasks is less than some limit lim_A then the processor sends a message to neighbouring processes to request work. When tasks are imported, they are placed in the local queue, so they will not be moved again. The processor only sends demands for more work after work has been received and only if the number of tasks is below lim_A.

When a processor receives a request for work, it checks to see if it has an exportable task and if the number of tasks is greater than some limit lim_B. If so, it will send back a task from its exportable queue. Otherwise, it stores the request in case lim_B is exceeded in future.

Let us assume that the number of tasks is generally growing in the system. All processors will request work from their neighbours when they start up. Only the starting processor has a task to perform, but will keep hold of it and new tasks spawned until lim_B is reached. It will then send excess exportable tasks to its neighbours which will keep requesting more until they reach lim_A. Only when they reach lim_B will they respond to requests from their neighbours. Once all the processors have more than lim_A tasks, they will stop exchanging tasks.

Often, ZAPP processors have a limited connection topology (say a ring, or grid of processors). Hence the number of neighbours is limited but communication with those neighbours is very quick. With PVM working over TCP/IP, as used in the initial POM implementation, all messages are very slow so all remote processors can be considered to be neighbours.

The current implementation of POM uses PVM to exchange messages between processors. For more efficient handling of messages, it would be interesting to investigate the Active Messages [17] approach. The design decisions of OGRe have been carefully chosen to remove the need for extensive locking, or queuing of messages, since all processes should be ready to handle every message they will be sent.

5 Conclusions

We have presented a model for *Object Graph Rewriting*. OGRe aims to be carefully designed low level intermediate code for implementation of a range of programming languages. The first application of the model is the implementation of Facile in such a way that the cost of concurrent and functional styles are kept in balance. The aim is to achieve good performance on sequential machines, though without discounting parallel implementation.

The *Parallel OGRe Machine* uses the same execution mechanism as on sequential machines, but certain messages are handled differently, causing distribution of work and remote transmission of data.

My thanks to Jeong-Ho Lee who has worked on many of the details of the POM machine design, and to Ian Whittley, who has performed much of the implementation work.

References

[1] Anderson, P., Bolton, D., and Kelly, P. Paragon specifications: Structure, analysis and implementation. In PARLE '92 [15].

[2] Berry, G., and Boudol, G. The chemical abstract machine. In *Proc. POPL '90* (1990), pp. 81–94.

[3] Giacalone, A., Mishra, P., and Prasad, S. Facile: A symmetric integration of concurrent and functional programming. *IJPP 18*, 2 (1989), 121–160.

[4] Glauert, J. R. W. Asynchronous mobile processes and graph rewriting. In PARLE '92 [15], pp. 63–78.

[5] Glauert, J. R. W., Leth, L., and Thomsen, B. A new translation of functions as processes. *SemaGraph '91 Symposium* (1991).

[6] Gurd, J. R., Kirkham, C. C., and Watson, I. The Manchester prototype dataflow computer. *Commun. ACM 26*, 1 (1985), 34–52.

[7] Kramer, A., and Cosquer, F. Distributing facile. *MAGIC Note 12, ECRC* (1991).

[8] Leth, L. *Functional Programs as Reconfigurable Networks of Communicating Processes*. PhD thesis, Imperial College, London, 1991.

[9] Leth, L., and Thomsen, B. Some facile chemistry. *ECRC Technical Report ECRC-92-14* (1992).

[10] McBurney, D. L., and Sleep, M. R. Transputer-based experiments with the zapp architecture. In *Proc. PARLE '87* (1987), vol. 258 of *LNCS*, Springer-Verlag.

[11] Milner, R. A calculus of communicating systems. *LNCS 92* (1980).

[12] Milner, R. Functions as processes. In *Proc. Automata, Languages, and Programming '90* (1990), vol. 443 of *LNCS*, Springer-Verlag.

[13] Milner, R. The polyadic π–calculus: A tutorial. *Technical Report ECS-LFCS-91-180* (1991).

[14] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, Parts I and II. *Technical Report ECS-LFCS-89-85* (1989).

[15] *Proc. PARLE '92* (June 1992), vol. 605 of *LNCS*, Springer-Verlag.

[16] Shapiro, E. Y. The family of concurrent logic programming languages. *Computing Surveys 21*, 3 (1989), 412–510.

[17] van Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture '92* (1992), ACM.

[18] Wise, M. J. Message brokers and communicating Prolog processes. In PARLE '92 [15], pp. 535–549.