Object Graph Rewriting

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

November 27, 2000

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 agents, which have states represented by tree-structured terms, respond to messages, also represented as terms. New agents may be created and new messages sent, while the original agent 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 then provide a formal definition.

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 [4], a multi-paradigm programming language which combines functional and concurrent programming in a symmetric fashion. The existing practical implementation of the language [11] used 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.

A process calculus 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 is reported here. OGRe 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. The following section provides a formal definition. Section 4 provides an outline of the way in which the model may be used to model a number of programming language styles. Section 5 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 hardware 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 agents which exchange messages. Pattern-directed rewriting rules determine the way in which a agent responds to the arrival of a message.

Computation proceeds by picking a message and the agent to which it is directed, and replacing them by new agents 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 agent has a name taken from an unbounded set of *agent name variables* such that each newly created agent has a distinct name.

An OGRe system has a finite set of *agent term symbols*, used to construct agent states, and a finite set of *data term symbols*, used to construct messages and also subterms of agent 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 agent 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 agent names, data terms, or primitive data values.

OGRe agents

An example of a declaration of an OGRe agent with name m is $m : LSubL(p, Int(15))$. The agent state is given by the term $LSubL(p, Int(15))$ with an agent term symbol and zero or more arguments. Arguments may be agent names, primitive data values and data terms. In the example above, the agent has agent term symbol $LSubL$ and two arguments. The first, p, is a agent name variable 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 agent and have contents defined by a data term. In the example, the target agent 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 a list of pattern-directed rewriting rules. The pattern for a rule has a pattern for the agent term of an agent, and a pattern for the data term for a message directed to that agent. The pattern contains a variable that will be bound to the name of the root agent, and may contain other variables which can match agent names, data values, or data terms in the agent term or message.

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

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 Section 4.

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

Figure 1: OGRe dataflow rules for an arithmetic expression

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 agents and data tokens become messages.

In line with the tagged dataflow model of the Manchester Dataflow Machine [9], we will label tokens as left or right arguments using symbols L and R. When agents representing nodes are created, they will take an argument indicating the node to receive the result. The agent term symbol of the agent indicates which argument of the successor node will be formed by the result token. Hence a agent with state $RMul(p)$ creates a result token which will become the right-hand argument for agent 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 agents and two new messages, both in this case directed to the same agent. The state of the root agent, $Start$, is unchanged so the agent 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 agent 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 $Done$ agent.

The agents 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 agent $m : LSubL(p, Int(15))$ which awaits an argument and will subtract it from 15. This corresponds to an intermediate state which would arise if an agent $m : LSub(p)$ received a message $m\,!\,L(Int(15))$ where $LSub$ is defined by analogy with $RMul$.

The rule for $RMulL$

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

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 agents and messages.

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

Figure 2: Evaluation of a dataflow example

OGRe Rewriting

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

When an agent receives a message, the agent term 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 agent may be changed, and new agents and messages may be generated. Rule application may also apply primitive functions, such as the arithmetic operations on integers.

It may be that more than one rule matches, in which case the first to match in the list of rules is chosen.

It may be that no rule matches, in which case the message remains unabsorbed but may become part of a future redex if the agent changes state due to another rewrite.

An agent is only involved in computation if it receives messages. If an agent is not the destination of an existing message, and its name is not the argument of an agent term or data term in some other agent or message, then the agent 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* agents are garbage collected.

3 Formal Definition of OGRe

Comment 3.1 *Computations can be regarded as an optimisation. I will therefore omit them from the first version below. I will also (later) try to show that they can be transformed away. The transformed version will take more "steps" unless (which I might) I regard a computation as a "step".*

Computations are of the form

$$n = F(o, p)$$

example

$$n = IAdd(3, 4)$$

and we can assume that they only appear in programs in rules of the form:

$$r : Add(o,p), r!Dest(s) \rightarrow r : Add(o,p), n = IAdd(o,p), s!Res(n)$$

We could consider the computations as being expressed by an equivalent rule without computations:

$$r : Add(3,4), r!Dest(s) \rightarrow r : Add(3,4), s!Res(7)$$

and so on for all possible arguments of IAdd.

The syntax covers both OGRe rules (the program code) and OGRe data (the program execution state). OGRe data will contain only agent name variables (and hence no computations or primitive operation systems).

Comment 3.2 *There is a question whether there should be a distinction between primitive data (such as binary numbers 3, 4, ... and characters 'a', 'b', ...) and other data values without arguments (such as True, False, perhaps).*

In the low-level implementation these are distinguished and not allowed to mix in the same argument position of a symbol. Nor are primitive data values permitted as contents of messages (they must be boxed). For now, I will not make this distinction since the implementation simply provides additional context dependent constraints. I will try to refine the definition to express these additional constraints later.

A further aspect of the implementation is that the types of symbols (arities and classes of arguments) are mostly derived from context. Here, we assume that they are specified in advance.

Classes

agent class AC

data class DC

Symbols

agent term symbols ATS of class AC

data term symbols DTS of class DC

Agent term symbols are disjoint from data term symbols. Each ATS and DTS is associated with a sequence of classes. The length of the sequence gives the arity of the symbol and the classes constrain the form of the arguments (see below).

Variables

agent name variables ANV of class AC

data term variables DTV of class DC

Data term variables are disjoint from agent name variables.

Expressions

expressions $E ::= P \mid E, E$

processes $P ::= A \mid M$

agent $A ::= n : AT$, where $n \in ANV$

messages $M ::= n!DT$ where $n \in ANV$

agent terms $AT ::= F \mid F(TA)$, where $F \in ATS$

data terms $DT ::= G \mid G(TA)$, where $G \in DTS$

term arguments $TA ::= DV \mid TA, TA$

data values $DV ::= DT \mid x \mid n$, where $x \in DTV$ and $n \in PNV$

Expressions contain a number of processes (agents and messages). Agents are named agent terms. Messages are data terms directed to agents. Agent terms and data terms may have data terms and variables as arguments

Constraints: In AT and DT the arities and classes of symbols are respected. If a symbol argument is of class AC then it takes the form $n \in ANV$. If a symbol argument is of class DC then it takes the form DT or $x \in DTV$.

Definitions: We consider bound and free occurrences of variables in expressions. The function *occurs* gives the set of variables which *occur* in an expression. The only binder in expressions is $n : AT$, the agent definition form, which binds all occurrences of n in the expression. The function *bound* yields the set of variables *bound* in an expression. The function $free$ yields the set of variables that occur *free* in an expression.

$$
\begin{aligned}
occurs(E_1, E_2) &= occurs(E_1) \cup occurs(E_2) \\
occurs(n : AT) &= \{n\} \cup occurs(AT) \\
occurs(n!DT) &= \{n\} \cup occurs(DT) \\
occurs(F) &= \{\} \\
occurs(F(TA)) &= occurs(TA) \\
occurs(G) &= \{\} \\
occurs(G(TA)) &= occurs(TA) \\
occurs(TA_1, TA_2) &= occurs(TA_1) \cup occurs(TA_2) \\
occurs(x) &= \{x\}, x \in DTV \\
occurs(n) &= \{n\}, n \in PNV \\
bound(E_1, E_2) &= bound(E_1) \cup bound(E_2) \\
bound(n : AT) &= \{n\} \\
bound(n!DT) &= \{\} \\
free(E) &= occurs(E) \setminus bound(E)
\end{aligned}
$$

We say that definitions in E are *linear* if every $n \in bound(E)$ is bound by only one agent definition in E. An expression is *closed* if there are no free occurrences of variables, and *open* otherwise. We say that occurrences in a construct are *linear* if every variable occurs at most once.

Rules

rules $R ::= E_l \rightarrow E_r$

Definitions: We extend earlier concepts to rules. E_l takes the role of the *pattern* for the rule. Free variable occurrences in E_l bind variables in E_r.

$$
\begin{aligned}
occurs(E_l \rightarrow E_r) &= occurs(E_l) \cup occurs(E_r) \\
bound(E_l \rightarrow E_r) &= occurs(E_l) \cup bound(E_r)
\end{aligned}
$$

Constraints: E_l is an expression in E satisfying the following: E_l is of the form $n : AT, n!DT$ where occurrences in $n : AT$ and $n!DT$ are linear. $free(n : AT) \cap free(n!DT) = \emptyset$. $free(E_r) \subseteq occurs(E_l)$. $occurs(E_l) \cap bound(E_r) = n$

Properties: The pattern E_l has no repeated occurrences, except for n. Recall that n is bound in $n : AT$. Occurrences in E_l bind occurrences in E_r and a rule contains no free variables. There is an agent in E_r with the name n. Hence E_r cannot be empty. No other agent defined in E_r can have a name which occurs in E_l and hence bound by pattern matching.

Rewriting

An OGRe program starts with a ruleset (a list of rules) and an OGRe expression (the initial state). Execution proceeds by rewriting steps as described below until no further rewriting is possible.

When applying rules, expressions are considered equivalent modulo ordering of processes. A redex is a pair of an agent and a message from the program state which is an instantiation of E_l from an OGRe rule. Its contractum is the corresponding instantiation of E_r.

Ruleset $RS ::= R; \mid R; RS$

Program State $PS ::= E$

Constraints: PS is a closed expression. Definitions in PS are linear.

Properties: Hence there are no data term variables in PS.

A rule R of form $E_l \to E_r$, *matches* an expression E of form $m : AT, m!DT$ if there is a substitution σ with domain $bound(E_l \to E_r)$ such that $\sigma(E_l) = E$. The contractum is $\sigma(E_r)$.

A *rewrite step* takes a program state E, E' to state $\sigma(E_r), E'$, where ruleset RS contains a rule R of form $E_l \to E_r$ and σ is a substitution such that $\sigma(E_l) = E$ and definitions in $\sigma(E_r), E'$ are linear. Further, R is the first rule in RS that matches E.

Properties: E is of form $m : AT, m!DT$. σ maps agent names to agent names and data term variables to data terms. σ applies to any agent names in E_r that are not in E_l and is chosen so that these map to new and distinct agent names not found in the program state. This ensures that definitions will be linear in the resulting state.

Rewriting continues with the new program state. Rewriting terminates when no further steps can be made.

Comment 3.3 *I don't think that Jeong-Ho needs to worry about the rest of this, but it is included as part of my attempt to formalise some other issues.*

Garbage Collection

An agent can be garbage collected if it can never take part in future rewrites. This will be the case if no message could be created which will be sent to the agent because there is no occurrence of the agent name in the rest of the program state. It may be that a set of agents contains a cycle of mutual occurrences, but none of the set occurrs in the rest of the program state. In this case, the whole set can be removed.

Let the program state PS be partitioned into two expressions E_L, E_G where E_G contains only agents. If $free(E_L) = \emptyset$ then agents in E_G can take part in no further rewrites, and can be removed. E_G will be termed *garbage* in PS. (By this definition, messages can never be garbage).

Let a program state PS be expressed as E_L, E_G where E_G is the maximal set of agents for which E_G is garbage in PS. We term E_L the *live component* of PS.

Replacing a program state by its live component does not affect the future choice of rewriting steps during execution.

Functional Agents

We will use the term functional for agents that never change their state as a result of rewriting.

A rule is said to be *functional* if it has the form $m : AT, m!DT \to m : AT, E$. The effect of rewriting according to such a rule will leave the agent in the redex unchanged.

An agent term symbol F is said to be *functional* if every rule of the form $m : F(AT), m!DT \to E$ is functional. An agent A is said to be *functional* if the agent term has a functional symbol. An agent which is not functional is said to be *mutable*.

$$
\begin{aligned}
&e : Start,\ e\,!\,POMTrigger \rightarrow \\
&\quad p : PrintL,\ m : LMul(p),\ d : LRDup(m,m), \\
&\quad d\,!\,L(Int(6)),\ e : Start; \\
&p : LRDup(d,e),\ p\,!\,L(x) \rightarrow \\
&\quad d\,!\,L(x),\ e\,!\,R(x),\ p : Done;
\end{aligned}
$$

Figure 3: Duplication of dataflow tokens

Since a functional agent will never change its state, a message sent to such an agent is either an immediate candidate for taking part in a rewrite, or can never become one. A message which cannot take part in a rewrite could be regarded as an error and will certainly remain part of the program state in all subsequent rewrites.

Deadlock

A program is *deadlocked* if it terminates, but messages remain. If a message is sent to a functional agent, but cannot form part of a redex, then the program can only terminate in deadlock. Other forms of deadlock are possible involving messages to mutable agents. If a program terminates without deadlock then it can be seen that all the remaining agents are garbage.

4 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 agents. There are no intervening channels as in CCS, or the Polyadic π–Calculus. However, channel-based communication can be modelled by OGRe agents. In Figure 4, a channel is represented by an agent which co-ordinates communication between producing and consuming agents. 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 agent 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.

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

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

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 agent when the communication is completed. The producer must include the identity of the agent 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 agent 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.

These rules illustrate a particular class of OGRe programs in which there are rules which enable agents to absorb every message sent to them. The OGRe model allows for messages that do not match to remain until a match is possible. Figure 7 shows a much simpler ruleset which implements synchronous communication by absorbing Put messages only when the channel is empty, and Get messages only when a Put has been absorbed.

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

Figure 6: Rules for Synchronous Communication

$$
\begin{aligned}
&c : Chan(EQ),\ c\,!\,Put(v,s) \rightarrow \\
&\qquad c : Chan(VQ(v,s)); \\
&c : Chan(VQ(v,s)),\ c\,!\,Get(r) \rightarrow \\
&\qquad r\,!\,Data(v),\ s\,!\,Sync,\ c : Chan(EQ);
\end{aligned}
$$

Figure 7: Simplified Rules for Synchronous Communication

$$
\begin{aligned}
&c : Cell(v),\ c\,!\,Set(n,r) \rightarrow \\
&\qquad r\,!\,Data(v),\ c : Cell(n); \\
&c : Cell(v),\ c\,!\,Read(r) \rightarrow \\
&\qquad r\,!\,Data(v),\ c : Cell(v);
\end{aligned}
$$

Figure 8: A model for von Neumann cells in OGRe

Modelling State and Logic Variables

The channel model above uses an agent 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 8. The *Set* message carries data and the identity of an agent to receive acknowledgement in order to provide serialisation. We return the old contents to provide test-and-set functionality.

Figure 9 shows that by adding slightly more sophisticated behaviour, we can model a logic variable in the framework of concurrent logic programming [20]. 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 ignore further *Bind* messages if the variable is already bound. A bound variable may be *Read*. There is also an extralogical $IsVar$ test for examining the state of the variable.

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 agents which are manipulated in exactly the same way as in the Chemical Abstract Machine [2] proposed for Facile in [13]. 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,

$$
\begin{aligned}
&c : UVar,\ c\,!\,Bind(v,r) \rightarrow \\
&\qquad r\,!\,Succ,\ c : BVar(v); \\
&c : BVar(v),\ c\,!\,Read(r) \rightarrow \\
&\qquad r\,!\,Data(v),\ c : BVar(v); \\
&c : UVar,\ c\,!\,IsVar(r) \rightarrow \\
&\qquad r\,!\,Succ,\ c : UVar; \\
&c : BVar(v),\ c\,!\,IsVar(r) \rightarrow \\
&\qquad r\,!\,Fail,\ c : BVar(v);
\end{aligned}
$$

Figure 9: A model for logical variables in OGRe

adopting the same semantics as ML. The translation of the functional subset of Facile is a development of the work reported in [7] and [5]. 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.

5 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 agent 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 agent terms map to positive integers, distinguishing them from data term 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 agent term symbol of the agent receiving a message.

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

If no rule matches, the message is queued with the agent to which it was sent. If the agent is later rewritten, any queued messages are returned to the task queue in case they match the new agent state.

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.

6 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.

References

[1] Anderson, P., Bolton, D., and Kelly, P. Paragon specifications: Structure, analysis and implementation. In PARLE '92 [19].

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

[3] Boudol, G. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

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

[5] Glauert, J. R. W. Asynchronous mobile processes and graph rewriting. In PARLE '92 [19], pp. 63–78.

[6] Glauert, J. R. W., Kennaway, J. R., and Sleep, M. R. Dactl: An experimental graph rewriting language. In *Proc. 4th International Workshop on Graph Grammars and their Applications to Computer Science* (1991), vol. 532 of *LNCS*, Springer-Verlag.

[7] Glauert, J. R. W., Leth, L., and Thomsen, B. A new translation of functions as processes. In Sleep et al. [21], pp. 269–282.

[8] Goldsmith, R. G., McBurney, D. L., and Sleep, M. R. Parallel execution of concurrent clean on zapp. In Sleep et al. [21], pp. 283–301.

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

[10] Honda, K., and Tokoro, M. An object calculus for asynchronous communication. In *Proc. ECOOP'91* (1991), Geneva.

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

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

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

[14] 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.

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

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

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

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

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

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

[21] Sleep, M. R., Plasmeijer, M. J., and van Eekelen, M. C. J. D., Eds. *Term Graph Rewriting: Theory and Practice.* Wiley, 1993.

[22] 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.