Asynchronous Mobile Processes and Graph Rewriting

J. Glauert[*]
School of Information Systems
University of East Anglia, Norwich NR4 7TJ, UK
jrwg@sys.uea.ac.uk

To appear in Proceedings of Parle92

Abstract

Honda and Tokoro provide a formal system for communicating systems developed from Milner's π–calculus. Unlike other formalisms, their work is based on asynchronous communication primitives.

This paper proposes some minor but practically significant extensions to a model based on asynchronous communication and shows how the resulting system may be mapped very directly onto a graph rewriting system.

While the model based on asynchronous communication permits the most direct translation, a related model using synchronous communication may be implemented in a similar manner.

1 Introduction

The calculus of communicating systems of Milner [18] has provided a fertile theory for the study of concurrent process networks. CCS can describe regular systems, but cannot express systems with an arbitrary dynamic structure.

1.1 Synchronous and Asynchronous Process Calculi

There have been two main approaches to extending CCS to model process networks which can evolve dynamically. One approach supports higher-order processes which may be communicated as values. An example is the language CHOCS of Thomsen [21]. Another approach allows the communication of link names which then act as references to processes. The early work by Engberg and Nielsen on ECCS [7] provided the foundation for the π–calculus [20] and LCCS developed by Leth [17]. Influenced by the Chemical Abstract Machine of Berry and Boudol [4], Milner has recast parts of the π–calculus allowing an elegant representation of functions as process networks [19]. The subset language omits both matching and summation, and provides a replication construct in place of recursively defined agents.

While all these calculi are based on a synchronous model of communication, Honda and Tokoro base an object-oriented formalism on asynchronous communication [14]. They develop an asynchronous concept of bisimulation and provide a mapping from the language of Milner [19] to their asynchronous form. This mapping, and other uses of the asynchronous formalism, rely on techniques for sequentialisation of events. One contribution of the current paper is to show that great simplifications can be made to these techniques if the messages communicated over links are allowed to be tuples rather than single values.

[*]Partially supported by ESPRIT BRA 3074: Semagraph

1.2 Practical Application of Process Calculi

A number of studies have been made of the relationship between process notations and functional programming. Early work by Kennaway and Sleep used the LNET model [15] and Thomsen used the higher-order processes of CHOCS [21, 22]. Milner [19], and Leth [17], show how an arbitrary lambda expression may be converted to a network of processes whose behaviour simulates reduction of the original λ–expression. Such work has been primarily motivated by a desire to explore the theoretical relationship between the formalisms, rather than to provide an efficient implementation of the Lambda Calculus.

There is increasing interest in using concepts from process calculi as the basis for practical implementations of concurrent programming languages. The language Facile [8] aims to integrate functional and process styles. Concurrency primitives have been proposed for Standard ML [5]. The author is collaborating with the Facile project at ECRC to study implementation techniques for such integrated languages. Early work considers encoding the functional components of a program as a network of processes so that the complete language may be converted to process form. [13] describes a new translation of the λ–Calculus into a process notation which has advantages over those proposed by Milner or Leth as it supports a mixture of lazy and eager evaluation strategies using techniques similar to [6]. While Milner and Leth consider only the pure λ–Calculus, it is shown how constant data values and their operations may be included in the translations.

In this paper we explore the relationship with asynchronous communication models and show that with such a model, process networks may be translated very naturally into a generalised graph rewriting system (GRS). A translation to the practical GRS language Dactl [9] is presented.

2 A Process Notation

The process notation described in this section has features corresponding to those of ECCS [7], the π–Calculus [20] and LCCS [17]. All may be regarded as extensions to CCS allowing the communication of link names and hence allow dynamic process networks to be generated.

In common with [14], however, we adopt an asynchronous communication model and concentrate on features corresponding to the subset of the π–Calculus used in [19]. We will name the asynchronous language ACPL for Asynchronous-Communication Process Language. It has a synchronous counterpart named SCPL.

2.1 Syntax of ACPL

The forms of agent, or process, allowed are equivalent to those of [14] but using the syntax of LCCS. In the syntax below, P and Q are agents, A an agent identifier, x a link, and y a value or link name:

Asynchronous Output $x!y$

Send value y on link x.

Input $x?y.\,P$

Receive a value on link x and bind the value to y in subsequent behaviour P.

Restriction $P\backslash x$

x is the name of a link which may only be used within P. For many purposes this may be seen as the declaration of the link x.

Parallel Composition $(P|Q)$

P and Q continue concurrently and may interact via shared links. Binary composition is illustrated but an arbitrary number of agents may be composed including zero which gives inaction.

Agent Definition $P : A(x_1, \ldots, x_n) = Q$

A is an agent identifier of arity n which may be used in P. $x_1, \ldots, x_n$ are distinct names and may be free names in Q. Q may contain agent identifiers, including A, and free names from P.

Defined Agent $A(y_1, \ldots, y_n)$

A corresponding agent definition of the form $A(x_1, \ldots, x_n) = Q$ must be in scope. The defined agent behaves like $Q\{y_1/x_1, \ldots, y_n/x_n\}$.

Hence parallel composition and repetition is supported, but not summation (choice). ? and \ are name binding constructs yielding the obvious notion of free and bound names.

We will allow communication of tuples of values as atomic events:

Asynchronous Tupled Output $x!(q, r)$

Send pair of values q and r on link x.

Tupled Input $x?(q, r).\, P$

Receive a pair of values on link x. Bind the first value to q and the second to r in subsequent behaviour P.

Such a pair of input values may be bound to a single name, as long as the name is only used as an output value later. All our examples will involve communication of pairs of values, although our model would permit communication of arbitrary tuples. There is an issue of how to type messages and links, but we do not persue it here.

Note that in this notation output actions are always followed by inaction as in [14]. Hence other techniques are required to make subsequent computation depend on reception of a message by another agent.

2.2 Syntax of SCPL

The synchronous notation differs only in the notation for output actions:

Synchronous Output $x!y.\, P$

Send value y on link x and continue with behaviour P.

Synchronous Tupled Output $x!(q, r).\, P$

Send pair of values q and r on link x and continue with behaviour P.

The operational semantics of the notations will not not discussed in detail here since [14] and [19] provide a sound basis.

2.3 Sequentialisation and Tupled Communication

Under synchronous as well as asynchronous calculi there is a need for care when multiple values are to be be exchanged between agents. In an example from [20] agent P is to communicate the pair of values (u, v) to *either* agent Q *or* R. We may express communication of a pair of values directly in ACPL by:

$$\begin{aligned}
&(P|Q|R) : \\
&\qquad P = x!(u,v) : \\
&\qquad Q = x?(y,z) \,.\, Q' : \\
&\qquad R = x?(y,z) \,.\, R'
\end{aligned}$$

Since the pair of values (u, v) is transmitted as an indivisible action, either Q receives the pair, or R does. This considerably simplifies the code needed by other models as will be seen.

If we are restricted to communicating single values we might think of writing:

$$\begin{aligned}
&(P|Q|R) : \\
&\qquad P = x!u \,.\, x!v : \\
&\qquad Q = x?y \,.\, x?z \,.\, Q' : \\
&\qquad R = x?y \,.\, x?z \,.\, R'
\end{aligned}$$

However, it is possible that u might be received by Q and v by R, or vice versa. Note that this is a program in SCPL.

To solve this problem, Milner *et al* develop the concept of *molecular actions* in which a temporary private channel is used as a capability to ensure that a single agent receives both values:

$$\begin{aligned}
&(P|Q|R) : \\
&\qquad P = (x!t \,|\, t!u \,.\, t!v)\backslash t : \\
&\qquad Q = x?s \,.\, s?y \,.\, s?z \,.\, Q' : \\
&\qquad R = x?s \,.\, s?y \,.\, s?z \,.\, R'
\end{aligned}$$

s is not free in Q' or R'.

Under an asynchronous communication regime, we cannot use output actions to guard further actions, but sequentialisation is possible. As in [20], Honda and Tokoro communicate a private channel to the recipient. However, in their solution to the problem, the recipient sends back a series of link names on which the pair of data values (u, v) will be transmitted:

$$\begin{aligned}
&(P|Q|R) : \\
&\qquad P = (\, x!t \,|\, t?a.\, (\, a!u \,|\, t?b.\, b!v \,)\,)\backslash t : \\
&\qquad Q = x?s.\, (\, s!c \,|\, c?y.\, (\, s!c \,|\, c?z.\, Q' \,)\,)\backslash c :
\end{aligned}$$

Only Q is shown, R being similar. s and c are not free in Q'. Note that sequentialisation is achieved within ACPL and hence without using output actions as guards.

2.4 Output Guards using Asynchronous Communication

In all the examples above we have assumed that the process P is inactive after the pair of values has been transmitted, as required by ACPL. The use of molecular actions in the style of [20] required the use of output guards, but the scheme of [14] shows that such guards can be avoided.

However, the synchronous languages do permit output guards so it is of interest to see if ACPL can model the same behaviour. We will consider messages consisting of a single value, u, bound to y in the recipients. Using SCPL the example will be:

$$\begin{aligned}&(P|Q|R) :\\ &\qquad P \;=\; x!u\,.\,P' :\\ &\qquad Q \;=\; x?y\,.\,Q' :\end{aligned}$$

Honda and Tokoro show that it is indeed possible to model this behaviour with an asynchronous language. In essence, their translation adds an extra acknowledgement link name to the values transmitted in a message. The recipient sends a dummy value to this channel when all values in the message have been received. The sender guards its future behaviour with an input guard receiving on the acknowledgement link:

$$\begin{aligned}&(P|Q|R) :\\ &\qquad P \;=\; (\,x!(u,r)\,|\,r?z.\,P'\,)\backslash r :\\ &\qquad Q \;=\; x?(y,k)\,.\,(\,k!d\backslash d\,|\,Q'\,) :\end{aligned}$$

r and z are not free in P'. k is not free in Q'.

ACPL allows transmission of tuples, so the code above is sufficient. [14] communicate simple values, so the translation is a good deal more involved. The code requires the transmission of a pair of values which would be encoded as follows:

$$\begin{aligned}&(P|Q|R) :\\ &\qquad P \;=\; (\,x!t\,|\,t?a.\,(\,a!u\,|\,t?b.\,b!r\,)\,|\,r?z.\,P'\,)\backslash r\backslash t :\\ &\qquad Q \;=\; x?s.\,(\,s!c\,|\,c?y.\,(\,s!c\,|\,c?k.\,(\,k!d\backslash d\,|\,Q'\,)\,)\,)\backslash c :\end{aligned}$$

It should be clear that a model supporting communication of tuples allows a much more concise solution to these problems in both synchronous and asynchronous forms. Such a model is also more practical since fewer communications are required. With tupled communication the number of messages exchanged is 1 in the synchronous case and 2 in the asynchronous case. Communicating a pair of simple values takes 3 message exchanges for a molecular action in the synchronous case and 8 in the asynchronous case (not illustrated).

As the number of values to be communicated increases, the savings of the tupled style get larger. Of course, the message size increases when using tupled communication, but under typical implementation schemes, messages have to be quite substantial before the cost of transmitting the data values becomes comparable with setup cost.

3 Translation of Lazy λ–Calculus

As an example we will build on [19] which provides encodings of the λ–Calculus in the π–Calculus. Only closed terms are considered. The encodings do not reduce the bodies of abstractions. This is not a problem when considering the application of these techniques to functional programming where functional normal forms are not usually of interest.

The encodings simulate particular reduction strategies. We will consider the encoding of the lazy λ–Calculus [1] which only performs β–reduction on the outermost redex on the left-spine, in other words, normal-order reduction takes place. The pure λ–Calculus is not of great practical use. Constants, such as boolean and integer data values, and operations on them must be added to make a practical functional language.

3.1 An Extension to Milner's Lazy λ–Calculus Scheme

The translation scheme is parameterised by the name of a link which will provide a characteristic handle on the process network for the translated expression. In the scheme below, x is a variable, c is an integer constant, M and N are λ–expressions, and Inc is the operator returning the successor of its argument:

$$\begin{array}{rcl} [\![x]\!]_u & = & x!u \\ [\![c]\!]_u & = & u?v\,.\,v!c \\ [\![\lambda x.\,M]\!]_u & = & u?(x,v)\,.\,[\![M]\!]_v \\ [\![M\,N]\!]_u & = & (\,[\![M]\!]_v \mid (v!(t,u) \mid R:\ R\ =\ t?w.\,([\![N]\!]_w \mid R))\backslash t\,)\backslash v \\ [\![Inc(M)]\!]_u & = & (\,[\![M]\!]_a \mid (a!s \mid s?m\,.\,u?v\,.\,v!(m+1))\backslash s\,)\backslash a \end{array}$$

An abstraction expects to receive a pair providing a link for its argument and a link to form the handle on the result of the application. The code for an application sends such values to the function which will receive the pair when it has become an application. Evaluation of the argument does not proceed until the application requests the argument by giving it an argument link w via the t channel. The argument, but not the function, may be used several times and hence a recursive defined agent is used.

Constants must be modelled by process networks which behave correctly within the translation scheme. Abstractions and variables wait to be given the names of links on which to communicate, while applications evolve until they simulate abstractions. As a data value may be an argument, it must behave in the same way as an abstraction. Processes representing constants will expect to be sent the name of a link and will respond with the value along the link. An Inc operator requests the value of its argument by giving it a result channel s and receiving the value m. It then acts exactly like the translation of a constant, awaiting its own result channel on u and returning the value $m + 1$ as required.

3.2 An Example Translation

We will consider translation of a very simple expression:

$$\begin{array}{rcl} [\![(\lambda x.Inc(x))\,3]\!]_u & = & (\,f?(x,q).\,(x!a \mid (\,a!s \mid s?m.\,q?v.\,v!(m+1)\,)\backslash s)\backslash a \\ & & \mid f!(t,u) \\ & & \mid R : R = t?w.\,(w?v.\,v!3|R)\,)\backslash f\backslash t \end{array}$$

A λ–expression acting as a program is expected to reduce to a constant, rather than an abstraction. To extract the value from a program, P, we would produce a network of the form:

$$(\,[\![P]\!]_u \mid Read(u)\,)\backslash u \ :\ Read(w) \ \ = \ \ (\,w!c \mid c?v\,.\,Print(\,v\,)\,)\backslash c$$

and $Print$ will display the final result.

4 Process Notation and Graph Rewriting

Process networks in the asynchronous style described in [14] and in ACPL may be easily translated into generalised graph rewriting systems. The discussion below uses the practical graph rewriting language Dactl [9]. Dactl provides a notation for describing computational objects in terms of directed graphs and for describing programs in terms of pattern-directed graph transformation rules. Although the syntax is different, the model of graph rewriting is the same as in LEAN [3]. A restricted form of this rewriting model underlies Term Graph Rewriting [2].

Dactl may be used as a vehicle for comparing implementation techniques and computational strategies since it provides fine control of the rewriting process. Also, since it has a well-defined operational semantics, it may be used as an intermediate code for language implementation.

Studies using Dactl have shown that implementations of a range of declarative languages may be made using graph rewriting as a common computational model [10]. In addition to translation of functional programming and term rewriting languages [16], which are traditionally associated with graph rewriting, Dactl has been used successfully to translate full GHC on which is based the kernel language adopted by the Japanese Fifth Generation Computer Systems project [11].

Graph rewriting also shows promise for supporting the integration of different programming styles. [12] reports early work on integration of functional and concurrent logic languages, while the work reported in [13] and this paper arises from a study of the integration of functional and process-based programming.

4.1 The Graph Rewriting Language Dactl

The nodes of a Dactl graph are labelled with a symbol which indicates that the node plays the role of an *operator* at the root of a rule application, a *data constructor*, or an *overwritable*. The role of operators and constructors will be familiar. The novel feature is the use of overwritable nodes which may be modified as a side-effect of rule application. This enables Dactl to express many more computations than the conventional graph rewriting used to implement functional languages. Overwritables may model von Neumann storage cells, semaphores, and the logic variable. Overwritable nodes will be used in this paper to model the contents of communication channels.

Each node has a distinct identifier. From a node leads a sequence of arcs to successor nodes. Arcs point from an operator node to its arguments, or from a data constructor to its sub-terms. A Dactl graph may be represented by listing the definitions of the nodes giving their identifier, symbol, and a sequence of identifiers for the successor nodes. Repetition of identifiers is used to indicate sharing in a graph:

```
a: Append[ c c ],
c: Cons[ o n ],
o: 1,
n: Nil
```

Symbols are integers or identifiers starting in upper-case, while node identifiers start in lower-case. A node definition may replace one of the occurrences of the node identifier, and redundant identifiers may be removed allowing the equivalent shorthand form:

```
a: Append[ c:Cons[ 1 Nil ] c ]
```

Dactl rules contain a *pattern* to be matched and a body, or *contractum*, to replace the occurrence of the pattern in the graph, or *redex*. Patterns are Dactl graphs but may contain node identifiers lacking a definition which will match an arbitrary node.

The contractum of a rule contains new graph structure to be built, which may reference nodes matched by the pattern, and one or more *redirections*, which indicate that the source of the redirection should be overwritten by the target. In classic term rewriting rules there will always be a redirection overwriting the root of the redex with the root of the contractum. In such rules the pattern and contractum are separated by the symbol `=>`. This is shorthand for a form with an explicit overwrite using `->` as separator.

Computation under the Dactl model proceeds by identifying a subgraph matching the pattern of a rewriting rule and replacing it by the contractum of the rule. If more than one rule matches, an arbitrary choice may be made about which rule to apply; fairness is not assumed. To control the order of evaluation, attempts to match the rules against the graph only begin at *active* nodes. Such nodes are marked with a $*$ in the representation of a Dactl graph. The pattern of a rule contains no such markings, since the matching process is insensitive to markings, but the contractum may use markings to nominate further nodes at which rewriting may take place. If multiple active nodes arise they may be considered in any order, or even in parallel if there is no conflict between possible rewritings.

Nodes may also be created *suspended* waiting for *notification* that a successor has been rewritten to a stable form. This enables a rule to create a dataflow graph in which certain nodes are active and will produce results which awaken parent nodes once all arguments are available. Suspension is indicated by one or more # markings. Each notification removes one suspension, the node becoming active when the last suspension is removed. Notification takes place when an active node is considered for rewriting, but no rule matches. This is typically because the node in question is a constructor or overwritable. Arcs which will form notification paths are marked with ˆ. The following examples show some rules with markings and the execution sequence for functional versions of append, and a version using the logic variable.

```
RULE
{1}  FunApp[Cons[u x] y] => #Cons[u ^*FunApp[x y]];
{2}  FunApp[Nil y]       => *y;
{3}  FunApp[p1 p2]       => #FunApp[^*p1 p2];
```

The example is a strict function which rewrites to a data constructor node, `Cons`, which is suspended waiting for completion of a recursive invocation of the function before notifying the parents of the original node. The final rule applies if the first argument is not yet in the form of a list. The argument is activated and the `FunApp` application will be reconsidered when the argument has been rewritten.

Using explicit redirections we could write the first rule as follows:

```
{1}  r:FunApp[Cons[u x] y] -> c:#Cons[u ^*FunApp[x y]], r:=c;
```

We will illustrate evaluation of a Dactl graph using an example with sharing, indicating the rule applied at each step, or a hyphen when no rule matches and notification occurs:

```
       *FunApp[ c:Cons[ o:1 n:Nil ] c ]
=>{1}  #Cons[ o:1 ^s:*FunApp[n:Nil c:Cons[o n]] ]
=>{2}  #Cons[ o:1 ^c:*Cons[o n:Nil] ]
=>{-}  *Cons[ o:1 c:Cons[o n:Nil] ]
=>{-}  Cons[ o:1 c:Cons[o n:Nil] ]
```

The computation terminates when no active nodes remain.

The following rules illustrate a version of append from logic programming. Here the result of the rewrite indicates success or failure of a predicate and results are communicated by instantiation of shared variables.

```
RULE
{1}  LogApp[Cons[u x] y r:Var] => *LogApp[x y w:Var], r:=*Cons[u w];
{2}  LogApp[Nil y r:Var]       => *SUCCEED, r:=*y;
{3}  LogApp[p:Var q r:Var]     => #LogApp[^p q r];
```

The result of the first rule depends on the success of a recursive use of the predicate. The variable `r` is bound to a `Cons` node whose second argument is a new variable which will be instantiated by the recursive call. The `Cons` node is made active in order to notify any computation suspended waiting for the variable to be instantiated. The final rule illustrates the case when the critical first argument is not instantiated. The call suspends waiting for some other computation to instantiate the variable (using one of the first two rules, for example).

We illustrate these rules in action with two calls to `LogApp` which share a variable:

```
       *LogApp[Nil Cons[1 Nil] v:Var], *LogApp[v Cons[2 Nil] w:Var]
=>{3}  *LogApp[Nil Cons[1 Nil] v:Var], #LogApp[^v Cons[2 Nil] w:Var]
=>{2}  *SUCCEED, v:*Cons[1 Nil], #LogApp[^v Cons[2 Nil] w:Var]
=>{-}  *SUCCEED, *LogApp[v:Cons[1 Nil] Cons[2 Nil] w:Var]
=>{1}  *SUCCEED, w:*Cons[1 x], *LogApp[Nil Cons[2 Nil] x:Var]
=>{2}  *SUCCEED, w:*Cons[1 x], x:*Cons[2 Nil], *SUCCEED
=>...
```

In the first step, the right-hand `LogApp` predicate finds a variable and suspends until it is instantiated. The second step instantiates the variable as a side-effect. The third step finds no match on the list value assigned to variable `v`, and reactivates the suspended `LogApp` predicate.

As a final illustration, we consider the modelling of buffered communication channels. A channel will consist of an overwritable `Chan` whose argument is a list acting as a stack of available values. To output to the channel, a rule simply overwrites the channel to contain a list prefixed with the new value. To input from a channel, a rule must test for available input. If the channel contains the empty list, the operation is suspended until input is available. Otherwise the channel is rewritten containing the tail of the original list.

Some example rules in this style are:

```
RULE
{1}  Put[c:Chan[v] d] -> c:= *Chan[Cons[d v]];
{2}  Get[c:Chan[Cons[d v]]] -> *Use[d], c:=*Chan[v];
{3}  Get[c] -> #Get[^c];
```

In the illustration of the rules in action below the first attempt to read from the channel suspends on finding the empty channel `Chan[Nil]`. Note the final state with the channel empty again:

```
       *Get[c],  *Put[c 2],  c:Chan[n:Nil]
=>{3}  #Get[^c], *Put[c 2],  c:Chan[n:Nil]
=>{1}  #Get[^c],  c:*Chan[Cons[2 n:Nil]]
=>{-}  *Get[c],   c:Chan[Cons[2 n:Nil]]
=>{2}  *Use[2],   c:Chan[n:Nil]
=>...
```

4.2 Translating ACPL Networks to Standard Form

Before describing the translation of a ACPL process network to a GRS, it will be shown that any network may be expressed in a standard form which makes heavy use of defined agents. If A stands for an agent identifier then we will define a sub-language of ACPL in which the only forms allowed are:

$$P ::= x!y \,\Big|\, x?y.A \,\Big|\, (A \mid \cdots \mid A) \,\Big|\, A \,\Big|\, P\backslash x \,\Big|\, P : A = P$$

Furthermore, we only permit the agent definitions to be at the outermost level (no agent definitions within agent definitions), and they may contain no free variables. Any ACPL expression can be converted to this form as will be shown below. Equivalent forms of the translation of the example $\lambda x.Inc(x))\,3$ are shown for illustration: The original program is:

$$\begin{aligned}
&(\,(\,f?(x,q).\,(x!a \mid a!s \mid s?m.\,q?v.\,v!(m+1)\,)\backslash s\backslash a\\
&\quad \mid f!(t,u)\\
&\quad \mid R:\; R = \;t?w.\,(w?v.\,v!3 \mid R)\\
&\quad)\backslash f\backslash t\\
&\mid \mathit{Read}(u)\\
&)\backslash u:\\
&\mathit{Read}(w) = \;(\,w!c \mid c?v.\,\mathit{Print}(v)\,)\backslash c
\end{aligned}$$

Adding additional agent definitions to satisfy the reduced syntax, and also using an agent definition Put for all output operations:

$$\begin{aligned}
&P : P =\\
&\qquad (\,Q:\; Q =\\
&\qquad\quad (\,A:\; A = \;f?(x,q).\,B:\; B = \;(\mathit{Put}(x,a) \mid \mathit{Put}(a,s) \mid C:\\
&\qquad\qquad C = \;s?m.\,D:\; D = \;q?v.\,\mathit{Put}(v,(m+1)))\backslash s\backslash a\\
&\qquad\quad \mid \mathit{Put}(f,(t,u))\\
&\qquad\quad \mid R:\; R = \;t?w.\,E:\; E = \;(F:\; F = \;w?v.\,\mathit{Put}(v,3) \mid R)\\
&\qquad\quad)\backslash f\backslash t\\
&\qquad \mid \mathit{Read}(u)\\
&\qquad)\backslash u:\\
&\qquad \mathit{Read}(w) = \;(\,\mathit{Put}(w,c) \mid G:\; G = \;c?v.\,\mathit{Print}(v)\,)\backslash c:\\
&\qquad \mathit{Put}(x,v) = \;x!v
\end{aligned}$$

Adding parameters to avoid free variables:

$$\begin{aligned}
&P : P =\\
&\qquad (\,Q(u):\; Q(u) =\\
&\qquad\quad (\,A(f):\; A(f) = \;f?(x,q).\,B(x,q):\; B(x,q) = \;(\mathit{Put}(x,a) \mid \mathit{Put}(a,s) \mid C(s,q):\\
&\qquad\qquad C(s,q) = \;s?m.\,D(q,m):\; D(q,m) = \;q?v.\,\mathit{Put}(v,(m+1)))\backslash s\backslash a\\
&\qquad\quad \mid \mathit{Put}(f,(t,u))\\
&\qquad\quad \mid R(t):\; R(t) = \;t?w.E(w,t):\\
&\qquad\qquad E(w,t) = \;(F(w):\; F(w) = \;w?v.\mathit{Put}(v,3) \mid R(t))\\
&\qquad\quad)\backslash f\backslash t\\
&\qquad \mid \mathit{Read}(u)
\end{aligned}$$

$$
\begin{aligned}
&)\backslash u :\\
&\mathit{Read}(w) = (\, \mathit{Put}(w,c) \mid G(c) :\ G(c) =\ c?v.\, \mathit{Print}(v)\,)\backslash c :\\
&\mathit{Put}(x,v) :\ \mathit{Put}(x,v) =\ x!v
\end{aligned}
$$

All definitions can now be pulled to the top level:

$$
P : \quad
\begin{aligned}
P &= (\, Q(u) \mid \mathit{Read}(u)\,)\backslash u :\\
Q(u) &= (\, A(f) \mid \mathit{Put}(f,(t,u)) \mid R(t)\,)\backslash f\backslash t :\\
A(f) &= f?(x,q).\, B(x,q) :\\
B(x,q) &= (\, \mathit{Put}(x,a) \mid \mathit{Put}(a,s) \mid C(s,q)\,)\backslash s\backslash a :\\
C(s,q) &= s?m.\, D(q,m) :\\
D(q,m) &= q?v.\, \mathit{Put}(v,(m+1)) :\\
R(t) &= t?w.\, E(w,t) :\\
E(w,t) &= (\, F(w) \mid R(t)\,) :\\
F(w) &= w?v.\, \mathit{Put}(v,3) :\\
\mathit{Read}(w) &= (\, \mathit{Put}(w,c) \mid G(c)\,)\backslash c :\\
G(c) &= c?v.\, \mathit{Print}(v) :\\
\mathit{Put}(x,v) &= x!v
\end{aligned}
$$

4.3 Translating ACPL Networks to Graph Rewriting Systems

It will be seen that ACPL programs in standard form have a very simple format. It is possible to convert agent definitions in standard form directly to Dactl rules. We will give the translated Dactl code and then describe the translation:

```
RULE
  INITIAL -> *Q[u], *Read[u], u: NewChan;
  Q[u] -> *A[f], *Put[f Pair[t u]], *R[t], f: NewChan, t: NewChan;
  A[f:Chan[Cons[Pair[x q] r]]] -> *B[x q], f:=*Chan[r];
  A[f:Chan[Nil]] -> #A[^f];
  B[x q] -> *Put[x a], *Put[a s], *C[s q], s: NewChan, a: NewChan;
  C[s:Chan[Cons[m r]] q] -> *D[q m], s:=*Chan[r];
  C[s:Chan[Nil] q] -> #C[^s q];
  D[q:Chan[Cons[v r]] m] -> #Put[v ^*IAdd[m 1]], q:=*Chan[r];
  D[q:Chan[Nil] m] -> #D[^q m];
  R[t:Chan[Cons[w r]]] -> *E[w t], t:=*Chan[r];
  R[t:Chan[Nil]] -> #R[^t];
  E[w t] -> *F[w], *R[t];
  F[w:Chan[Cons[v r]]] -> *Put[v 3], w:=*Chan[r];
  F[w:Chan[Nil]] -> #F[^w];
  Read[w] -> *Put[w c], *G[c], c: NewChan;
  G[c:Chan[Cons[v r]]] -> *Print[v], c:=*Chan[r];
  G[c:Chan[Nil]] -> #G[^c];
```

All Dactl programs start with a graph containing a single node with the symbol `INITIAL`, which replaces P above. Processes generally correspond to GRS terms. The `*` marks an active process or rewritable term. Parallel composition, used by Q, B, E, and *Read* is very straightforward and appears as little more than a change of syntax. Restriction, used in Q, B, and *Read* corresponds to declaration of new links denoted by the term `NewChan`. Links are represented by the overwritable symbol `Chan` which contains a stack of messages which have been sent on the link. `IAdd` is the primitive function for adding integers.

All output actions use the primitive `Put` introduced earlier. To output a pair, the `Pair` constructor is used, as in `Q`. Processes guarded by input actions become two rules: the first matches a non-empty channel, binds the value received, and rewrites the channel having extracted the value; the second applies if the channel is empty and blocks until input *is* available. When a pair is input, (for example in `A`), the pattern matches a pair of values for use in the body of the action.

A new channel is defined as the following pattern:

```
PATTERN
  NewChan = Chan[Nil];
```

The processes `Put` and `Print` are defined as follows:

```
RULE
  Put[c:Chan[v] d] -> c := *Chan[Cons[d v]];
  Print[v] -> *PrintF["Result: %d" v];
```

`Put` just inserts the new message at the head of the stack of messages held by the link.

The overall style of execution contrasts strongly with the more conventional Term Graph Rewriting [2] approach. Instead of representing an expression as a single rooted term, this style represents sub-expressions as independent unrooted terms linked by shared references to nodes representing channels. Execution corresponds more to the actions of the Chemical Abstract Machine [4] than a traditional graph reduction machine.

A translator has been written in ML which takes λ–expressions and converts them to an internal form of ACPL. The process network is converted to standard form, as described above, and is mapped to Dactl. The Dactl code may be executed using an interpreter or compiler developed at UEA. The example above is as produced by the translator, with machine-generated identifiers replaced by more readable ones.

The mapping from λ–expressions to ACPL used here is based on extensions to the Lazy scheme in [19]. The translator also implements a new scheme reported in [13].

4.4 Translating SCPL Networks to Graph Rewriting Systems

It has been shown that communication in the synchronous language can be modelled in the asynchronous language by sending an acknowledgement channel. Using the earlier example we can provide a translation to Dactl.

The SCPL network:

$$\begin{aligned} S(v) &= (\,P(v,x)\,|\,Q(x)\,)\backslash x : \\ P(u,x) &= x!u.\,P' : \\ Q(x) &= x?y\,.\,Q'(y) \end{aligned}$$

can be modelled by the ACPL network:

$$\begin{aligned} S(v) &= (\,P(v,x)\,|\,Q(x)\,)\backslash x : \\ P(u,x) &= (\,x!(u,k)\,|\,k?z.\,P'\,)\backslash k : \\ Q(x) &= x?(y,k)\,.\,(\,k!z\backslash z\,|\,Q'(y)\,) \end{aligned}$$

Translating the ACPL program yields Dactl rules such as:

```
RULE
  S[v] -> *P[v x], *Q[x], x:NewChan;
  P[u x] -> *Put[x Pair[u k]], *A[k], k:NewChan;
  A[k:Chan[Cons[z r]]] -> *P', k:=*Chan[r];
  A[k:Chan[Nil]] -> #A[^k];
  Q[x:Chan[Cons[Pair[y k] r]]] ->
        *Q'[y], x:=*Chan[r], *Put[k z], z:NewChan;
  Q[x:Chan[Nil]] -> #Q[^x];
```

Hence the communication mechanism can be used in the normal way. However, optimisations can be made if it is known that a link is being used purely for synchronisation:

- No special value z needs to be communicated on the synchronisation channel k
- The synchronisation channel will be used only once so need not be rewritten once used
- A simpler overwritable structure will suffice in place of a complete link

The Dactl code can be simplified to the form below. The synchronisation value takes the initial value `Wait` when created by P and is overwritten by `Free` when the message is received by Q:

```
RULE
  S[v] -> *P[v x], *Q[x], x:NewChan;
  P[u x] -> *Put[x Pair[u k]], *A[k], k:Wait;
  A[k:Free] -> *P';
  A[k:Wait] -> #A[^k];
  Q[x:Chan[Cons[Pair[y k:Wait] r]]] -> *Q'[y], x:=*Chan[r], k:=*Free;
  Q[x:Chan[Nil]] -> #Q[^x];
```

Further, should it be desired, a mixture of synchronous (acknowledged) and asynchronous (unacknowledged) communications can be used. In this case the Dactl code receiving a message would distinguish between classes of values in the message queue.

The final example below involves a process P which uses synchronous communication and R which waits for no acknowledgement. Instead of using the constructor `Cons` to build the message stack, we use either `ACons` for asynchronous messages or `SCons` for synchronous messages. For this example specialised rules are used for output.

$$\begin{aligned} S &= (\,P(x)\,|\,Q(x)\,|\,R(x)\,)\backslash x : \\ P(x) &= x!2.\,Print(3) : \\ Q(x) &= x?y.\,(\,Q(x)\,|\,Print(y)\,) : \\ R(x) &= (\,x!4\,|\,Print(5)\,) \end{aligned}$$

```
RULE
  S -> *P[x], *Q[x], *R[x], x:NewChan;
  P[x:Chan[s]] -> x:=*Chan[SCons[2 s k:Wait]], *A[k];
  A[k:Free] -> *Print[3];
  A[k:Wait] -> #A[^k];
  R[x:Chan[s]] -> x:=*Chan[ACons[4 s]], *Print[ 5];
  Q[x:Chan[SCons[y r k:Wait]]] ->
          *Q[x], x:=*Chan[r], k:=*Free, *Print[y];
  Q[x:Chan[ACons[y r]]] -> *Q[x], x:=*Chan[r], *Print[y];
  Q[x:Chan[Nil]] -> #Q[^x];
```

5 Conclusions

This paper has explored some practical extensions to the theory of process calculi with asynchronous communication in the style of [14].

Although the process language is rich enough to support all the facilities used in [19], the language lacks the choice (sum) operators and hence avoids the need to retract communication offers. It also adopts a different approach to the repetition construct proposed there. [14] show how the repetition construct might be expressed in ACPL, but the Dactl translation would compute indefinitely, generating increasing numbers of copies of the repeated process through internal actions. Instead, we use recursively defined agents as in earlier models and rely on such processes having dependencies on external actions to prevent an explosion of the process network.

We have not explored data-dependent matching constructs at this stage. [14] provide a theoretically elegant approach to selections, but we feel that a more direct approach will be required for our rather practical purposes. The match construct of the π–calculus may be suitable.

It has been shown that the language ACPL based on asynchronous communication principles may be mapped to a graph rewriting notation. This opens up the question of whether graph rewriting might lead to a suitable low-level implementation route for such process notations, and for programming languages integrating functional and process styles. It is also shown that the requirements of synchronous communication may be satisfied with comparatively little overhead.

Acknowledgements

The author would like to thank Alessandro Giacalone, Lone Leth, and Bent Thomsen of ECRC, Munich, and members of the Dactl project at UEA for many useful discussions and much support, and to acknowledge the financial support of the ESPRIT programme through the Semagraph BRA.

References

[1] S. Abramsky: *The Lazy Lambda Calculus.* Chapter 4 in D. Turner (ed.), *Research Topics in Functional Programming*, pp. 65-116, Addison Wesley. (1988)

[2] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, & M.R. Sleep: *Term Graph Rewriting.* Proc. PARLE 87, Springer LNCS 259, p141-158. (1987)

[3] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, & M.R. Sleep: *Lean: An Intermediate Language Based on Graph Rewriting.* Parallel Computing, Vol 9, p163-177. (1989)

[4] G. Berry & G. Boudol: *The Chemical Abstract Machine.* POPL 90, p 81-94. (1990)

[5] D. Berry, R. Milner, & D.N. Turner: *A semantics for ML concurrency primitives.* Laboratory for Foundations of Computer Science, Edinburgh University. (1991)

[6] F.W. Burton: *Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.* TOPLAS, Vol 6, No 4, p159-174. (1984)

[7] U. Engberg & M. Nielsen: *A Calculus of Communicating Systems with Label Passing.* Report DAIMI PB-205, Computer Science Department, University of Aarhus. (1984)

[8] A. Giacalone, P. Mishra, & S. Prasad: *Facile: A Symmetric Integration of Concurrent and Functional Programming.* IJPP, Vol 18, No 2, p121-160. (1989)

[9] J.R.W. Glauert, J.R. Kennaway, & M.R. Sleep: *Dactl: An Experimental Graph Rewriting Language* Proc. 4th International Workshop on Graph Grammars, Bremen, 1990. Springer LNCS 532. (1991)

[10] J.R.W. Glauert, K. Hammond, J.R. Kennaway & G.A. Papadopoulos: *Using Dactl to Implement Declarative Languages.* CONPAR88, Manchester, UK, Sept. 12-16, Vol. 1, pp. 89-94. (1988)

[11] J.R.W. Glauert & G.A. Papadopoulos: *A Parallel Implementation of GHC.* Proceedings International Conference on Fifth Generation Computer Systems. ICOT, Tokyo, December 1988. (1988)

[12] J.R.W. Glauert & G.A. Papadopoulos: *Unifying Concurrent Logic and Functional Languages in a Graph Rewriting Framework.* Proceedings, 3rd Panhellenic Computer Science Conference, Athens, May 1991. (1991)

[13] J.R.W. Glauert, L. Leth & B. Thomsen: *A New Translation of Functions as Processes.* SemaGraph '91 Symposium, December 1991. (1991)

[14] K. Honda & M. Tokoro: *An object calculus for asynchronous communication.* Proc. ECOOP'91, Geneva, July 1991. (1991)

[15] J.R. Kennaway & M.R. Sleep: *Expressions as Processes.* Proc. Lisp and FP, August 1982, p.21-28. (1982)

[16] J.R. Kennaway: *Implementing Term Rewrite Languages in Dactl.* Theor. Comp. Sci. 72, p.225-250. (1990)

[17] L. Leth: *Functional Programs as Reconfigurable Networks of Communicating Processes.* Ph.D Thesis, Imperial College, London University. (1991)

[18] R. Milner: *Calculus of Communicating Systems.* Springer LNCS 92. (1980)

[19] R. Milner: *Functions as Processes.* Automata, Languages, and Programming, Springer LNCS 443. (1990) Also as: Technical Report, INRIA Sophia Antipolis, June 1989.

[20] R. Milner, J. Parrow, & D. Walker: *A Calculus of Mobile Processes.* Parts I and II. TR ECS-LFCS-89-85, Edinburgh University, June 1989. (1989)

[21] B. Thomsen: *A Calculus of Higher Order Communicating Systems.* Proceedings of POPL 89, pp. 143-154, The Association for Computing Machinery. (1989)

[22] B. Thomsen: *Calculi for Higher Order Communicating Systems.* Ph.D. Thesis, Imperial College, London University. (1990)