Paragon Specifications: Structure, Analysis and Implementation

Paul Anderson[1], *David Bolton*[2] *and Paul Kelly*[3]

[1] Grammatech Inc. One Hopkins Place, Ithaca, New York 14850, USA
[2] Department of Computer Science, City University, Northampton Square, London EC1V 0HB
[3] Department of Computing, Imperial College of Science, Technology and Medicine, London SW7 2BZ

Abstract. Paragon is a notation for specifying object behaviours using sets of rewrite rules, where rewriting is controlled by synchronous and asynchronous message passing, and where objects may be dynamically created as a rewriting side-effect. This paper overviews Paragon, and introduces a simple classification scheme for analysis of Paragon specifications. Restrictions on specifications are discussed in consideration of implementation feasibility and efficiency constraints. Implementation schemes based on the analysis and restrictions are defined. In particular, a translation strategy for static systems is detailed and motivated with a worked example. To reinforce the low-level nature of the derived implementation the translation is defined in terms of a digital hardware description language. Schemes for the implementation of general dynamic systems are also considered.

1 Introduction

Paragon has been applied to the problem of specifying parallel architectures in several ways, and at several levels. It captures the route from a term rewrite system, annotated with control flow information, to a parallel implementation which manages dynamic process creation and control through message passing [BHK89], [BHK90]. The language design synthesises ideas drawn from graph rewriting systems and process algebras, but is distinguished from other notations by its novel mixture of message passing and rewrite systems semantics [BHK91].

This paper examines the structure of Paragon specifications and introduces implementation schemes based on translation to systems of static synchronous parallel processes. Section 2 introduces the Paragon language. Section 3 considers a simple classification scheme, and introduces restrictions on specifications in consideration of implementation issues. Section 4 describes how specifications might be implemented. In particular, a translation strategy to Occam for static systems is detailed and motivated with a worked example. Implementation schemes for general dynamic systems are also discussed. Section 5 concludes by reviewing the results including some interesting insights into the nature of Paragon specifications.

2 Overview of Paragon

A specification is a set of class definitions each of which is defined by type information and a set of rewrite rules. To express control over rewriting, and to introduce the

concept of a process, Paragon rewrite rules may be augmented by a set of messages to be sent should rewriting occur. A rule may therefore forward control to other rules, and hence interaction between class instances is by means of synchronous and asynchronous message passing. Rules can be regarded as being 'triggered' by receipt of an appropriate message, which may carry parameters.

Our starting point in designing Paragon was a notation which incorporated pattern-matching and rewriting of data structures, in the fashion of term-rewriting. This notation has a natural interpretation as "term" graph-rewriting, by interpreting copying in the right hand side of a rewrite rule as pointer, rather than structure, copying [BvEG+87a]. In a general graph-rewriting language, non-root overwriting can be used to model side-effects. In Paragon, message passing is used instead, thereby simplifying the semantics of rewriting [BHK91].

The resulting language differs from DACTL [GKS87] and Lean [BvEG+87b] in making side-effects explicit as message passing and differs from FP2 [SJ89] in allowing messages to be directed towards dynamically created objects.

2.1 Class definitions and rules

A specification in Paragon consists of a set of class definitions, a set of associated data types, and a set of objects representing the initial state of the specified system. Data types are specified using a syntax borrowed from Miranda [Tur85], and define a free algebra with the named constructors. A class definition consists of the structure of the object defined as a data type and a set of rewrite rules which define the behaviour of an object of that class when it receives a message. Classes and data types are written with constructors (possibly with arguments) or as tuples. For example:

data direction ::= North | East | South | West
class ilist ::= Nil | Cons integer ilist
class buffer ::= ⟨**vector** [North **upto** West] router,packet,direction⟩

The syntax of a Paragon rule is given in Fig. 1. A rule consists of a left hand side and a right hand side and has the following form:

$\mathcal{S}$ **given** m(x)
when $\mathcal{G}$
→ $\mathcal{S}'$
then $\mathcal{C}$
where $\mathcal{B}$

This rule is applicable when an object of a particular class whose state matches the pattern $\mathcal{S}$ is passed a message m carrying arguments x. If the guard $\mathcal{G}$ evaluates to true, the rule fires, the object is transformed into $\mathcal{S}'$, and communications $\mathcal{C}$ are generated. Communications take the form obj ! message or obj !! message denoting synchronous and asynchronous message-passing respectively. $\mathcal{B}$ introduces some bindings of names (possibly to functions) used in $\mathcal{S}'$, $\mathcal{B}$ and $\mathcal{C}$. $\mathcal{G}$, $\mathcal{C}$ and $\mathcal{B}$ are all optional. Expressions can contain references to **self**, the object receiving the message, or to **nil**, the null object. New instances of objects are created with the expression **new** (*class name*,*initial state*).

⟨rule⟩ → ⟨lhs⟩ $\underline{\rightarrow}$ ⟨rhs⟩
⟨lhs⟩ → ⟨state⟩ [**given** ⟨message⟩] [**when** ⟨guard⟩]
⟨message⟩ → **name** [(⟨state⟩ { , ⟨state⟩ })]
⟨guard⟩ → **name** = ⟨state⟩
| ⟨predicate⟩ | ⟨guard⟩ ∧ ⟨guard⟩ | ⟨guard⟩ ∨ ⟨guard⟩ | (⟨guard⟩)

⟨rhs⟩ → ⟨state⟩ [**then** ⟨tasks⟩] [**where** ⟨bindings⟩]

⟨tasks⟩ → **name** ! ⟨outgoing⟩ *synchronous*
| **name** !! ⟨outgoing⟩ *asynchronous*
| ⟨tasks⟩ ‖ ⟨tasks⟩ *parallel composition*
| ⟨tasks⟩ ; ⟨tasks⟩ *sequential composition*
| (⟨tasks⟩)

⟨outgoing⟩ → **name** [(⟨expr⟩ { , ⟨expr⟩ })]

Fig. 1. Paragon rules syntax

Rules can also be defined without a **given** clause, with the intention that the rule be applied 'spontaneously' when the state matches $\mathcal{S}$, and $\mathcal{G}$ is true, with no triggering message pass. An example of the use of such rules is given in Sect. 4.4.

2.2 Example

The first example is a parallel version of a simple priority queue which can store integers; later we will examine a more elaborate fault tolerant router specification.

A priority queue is a fifo which accepts input messages tagged with priorities and delivers them in arrival order except that higher priority messages may overtake lower priority ones.

Our implementation uses a linked list of dynamically allocated objects to store queued data. A message tagged with its priority (an integer) is accepted at the head of this list and is propagated along the chain to its appropriate position. The highest priority is represented by the lowest integer. Messages are delivered from the queue by simply taking the head element.

In Paragon we represent the members of the queue using a class `pq` which can represent the empty queue (`EQ` objects) and the non-empty queue (`PQ` objects).

class `pq` ::= `EQ` | `PQ integer pq` {*the queue*}
class `obj` ::= ... {*client class*}

Queue objects receive `put` messages carrying integer parameters representing priorities in an arbitrary order; receipt of a `get` message outputs the largest, which will be at the head of the queue (by sending the message `got` to the client). Each queue instance stores one integer priority; the rest of the queue is represented by reference to another instance.

```
EQ                  given get(o)
→     EQ
                    then          o !! fail
```

```
PQ i q              given get(o)
→     q
                    then          o !! got(i)
```

```
EQ                  given put(i')
→     PQ i' q'
                    where         q' = new(pq, EQ)
```

```
PQ i q              given put(i')
                    when i' < i
→     PQ i' q'
                    where         q' = new(pq, PQ i q)
```

```
PQ i q              given put(i')
                    when i' ≥ i
→     self
                    then          q !! put(i')
```

The first put rule shows the receipt of a message to an empty queue: a new queue object is created to represent the end of the new (one message) queue. The second put rule shows the addition of a higher priority message to the head of the queue, with creation (through **new**) of a new object to represent the head. The final rule shows the propagation of a lower priority message to its proper place in the queue.

Parallelism is achieved here through asynchronous message-passing; several put messages can propagate through a queue at the same time inputting different priority integers. The example is dynamic in allocation of new queue objects as the queue lengthens. Objects named o above are in a queue client class called obj.

3 Structure of specifications

This section introduces a simple classification scheme for analysing Paragon specifications. To ensure they are amenable to implementation by the route described in Sect. 4, we also introduce a series of restrictions on specifications, which reveal further interesting aspects of the nature of Paragon specifications.

3.1 Classification

A *specification* consists of a set of classes C (each defined by structure and rules) and a set of objects O (representing the initial state). An object o that communicates synchronously (asynchronously) with a set of objects O is said to be *synchronous in* (*asynchronous in*) O, and is written $o \in S(O)$ ($o \in A(O)$). A specification with no asynchronous message passing is said to be *synchronous*. Static analysis can

reveal connectivity between objects, following which it is straightforward to trace the objects to which synchronous and asynchronous messages apply.

A message is *total* if, for the class to which it applies, it can always be consumed. That is, the rules admitting the message must cover all the possible values of the target object's class. A message is *partial* if pattern matching or guard evaluation after message receipt can fail, requiring the message to be retained for re-matching. Once again, static analysis of a specification can detect totality, except in the unusual case of complex guards.

An object o (class c) that can create objects which are instances of the classes in C is said to be *dynamic in* C, and is denoted $o \in D(C)$ ($c \in D(C)$). A specification with no objects or classes dynamic in any classes is said to be *static*.

3.2 Restriction

Restrictions arise from consideration of implementation issues as follows:

1. Only one-level pattern matching is allowed. That is, we can match the structure of an object, but not the structure of objects within that object. This can be detected directly from the patterns specified in a set of rules.
   The reason for this restriction is to avoid the implicit critical region which would require locking during pattern matching to avoid inconsistent results. This restriction could be relaxed for **data** objects: since data does not change once it has been constructed the problem of atomic matching does not arise.
   This restriction can also be avoided by transformation of the rules in those cases where synchronisation problems have been shown not to exist (this is related to a problem in compiling DACTL [GKS87]).
2. Only total messages are allowed.
   For partial messages, when pattern-matching and/or guard evaluation fails for all rules implemented by a message procedure, the object processor must retain multiple messages (and message arguments) for each message sort received by the object, and then continually recall the message procedure. Each message must be assigned equal priority in this attempted execution. We discount such an implementation as inefficient and suggest that a viable translation is possible only for total messages.
   Except in the case of complex guards, totality can be detected by static analysis.
3. Data types are not recursive.
   This can be statically checked from type specifications. The restriction applies so that types can be represented in a fixed amount of space, without pointers. It avoids the possibility of sharing of object state. The restriction can, of course, be lifted in some cases.
   Pointer structures *within* an object processor's memory, for example, are not very troublesome, although in general garbage collection may be involved. Garbage collection may be avoided by copying entire data structures instead of copying pointers (intuitively this follows from the immutability of data structures; a more formal treatment could follow [BvEG+87a]).
   In general, however, Paragon allows data structures to be passed in messages, and normally the intended meaning is that sharing of substructure occurs. This raises many problems for a hardware realisation.

4. In any parallel composition of messages, the targets must all be different (e.g. in o1 ! v1 || o2 ! v2, o1 and o2 must be different).
   This avoids conflicts for access to communications channels.
   Since the names of objects may be carried in messages, this cannot be statically detected.

These classifications and restrictions provide some insight into the nature of Paragon specifications; we return to this topic in Sect. 5.

4 Implementation schemes

In this section we discuss implementation schemes for Paragon specifications based on the classification and restrictions described in Sect. 3. First, we consider static, synchronous specifications and present a worked example for a fault tolerant router system. Then we discuss more complex issues arising when a specification involves asynchronous message passing and dynamic object creation (as exemplified by the priority queue example).

The translations presented are intended to preserve the semantics described in [BHK91]. However, the restrictions discussed already define a particular treatment of the atomicity of pattern matching, and the schemes described in this section impose further decisions on, for example, the order of rule matching and the fair treatment of spontaneous rules. These are key issues, of course, and we discuss them within the section and return to them in Sect. 5.

4.1 Hardware implementation for static systems

We begin by showing how static, synchronous Paragon specifications can be implemented using concrete, low-level constructions. To reinforce the low-level nature of this implementation we will describe the implementation in terms of a realisation in digital hardware.

Each of the functions specified in the Paragon system requires a feasible hardware implementation as follows:

Objects. For each object in the system we create an *object processor*. This is a finite state automaton responsible for storing the state of the object, receiving messages directed at the object, and in consequence executing appropriate actions specified by the rules for the object.
A direct hardware implementation is possible when the specification is static so that the number and kind of object processors can be determined.

State. The state of each object is maintained in the corresponding object processor's local memory.
We avoid the need to access remote object's local memory because we have only one-level pattern matching, and data objects do not contain pointers.
Under these restrictions, data and objects share the same representation: a tuple is represented by a register containing a bitfield for each component. If a component is in turn a tuple, its component bitfields appear in-line. A constructor application is treated as a tuple whose first element is a small integer indicating

which constructor. Where alternative constructor applications may appear, the tuple size is that needed for the largest alternative.

Communication. We provide a physical channel for every potential communication between object processors. Each channel is implemented using enough wires to accommodate the largest message sent, with messages represented as indicated for data. Connections are unidirectional, with the receiving object processor acknowledging receipt when it has accepted the message and executed the associated rules. An appropriate block of logic implements the alternative rules for each message sort received.

Determining potential communication paths is troublesome since Paragon allows the name of an object to be communicated in a message. The worst case would require a fully-connected network of channels, which is unacceptable: some kind of shared network would have to be introduced. Data flow analysis can be used to determine which objects can communicate with one another to avoid this in most cases.

This design leads to a translation scheme from Paragon to a hardware description language. To aid explanation we choose a form of Occam [Inm88] suitable for compilation to a silicon layout [MK87]. This paper shows how a suitable subset of Occam can be used to provide a behavioural description for hardware devices, and how this can be translated to a structural hardware description language defining buses, registers etc. The language used is Inmos' own 'HDL'.

We augment the Occam subset with enumerated and record types. These can be readily translated to true Occam BYTE or other types to express the representation of state as described above. The PAR construct will appear only at the outermost level, with the assumption that concurrent processes are implemented using one processing element for each process. For brevity, SEQs are omitted. Object processors will be represented by Occam PROCs; we note [MK87] suggests that these can be implemented by body substitution or by conventional closed procedure calls of known fixed depth.

Communication will be through variant protocol channels defined between object processors (one variant per message sort received). When the 'name' of an object is communicated in messages or stored in an object state it is represented by the channel connected to its object processor. Each message sort received will also be represented as a PROC called by the object processor.

4.2 Object processors

A Paragon specification consists of classes whose behaviour is described in terms of guarded message-receipt and 'spontaneous' rules. We must translate each set of rules for a class into an appropriate set of object processors dealing with both message-receipt and spontaneous rules (spontaneous rules have no given clause, so should be applied whenever the object's state matches the LHS; see Sect. 2.1).

The skeleton description for an object processor with both message-receipt and spontaneous rules is shown in Fig. 2. The processor is described in terms of an internal record state and channel connections. Variant protocol channels $\mathsf{in}_0, \ldots, \mathsf{in}_r$ represent incoming messages (one variant per message sort received by the object), with successful message receipt and execution signalled to the sender via the

```
PROC object-processor (CHAN OF Object-in in0,in1,...,inr,
                       CHAN OF BOOL sack0,sack1,...,sackr,
                       CHAN OF Object-out out0,out1,...,outs)
  Object-state state :
  Set up channels etc as appropriate
  WHILE TRUE
    ALT
      in0 ? message ; arguments
        Deal with spontaneous rule if necessary
        call procedure for message (pass state, and arguments)
        sack0 ! TRUE          Acknowledgement to sender
        :
      inr ? message ; arguments
        Similar to in0
      TRUE & SKIP
        Deal with spontaneous rule if necessary
:
```

Fig. 2. Object processor description

$\mathsf{sack}_0,\ldots,\mathsf{sack}_r$ channels. Channels $\mathsf{out}_0,\ldots,\mathsf{out}_s$ represent objects to which messages are sent during execution of the message procedures, and will be assigned to appropriate internal object state components representing those objects.

Fair treatment of 'spontaneous' rewrites is ensured through polling *both* when there is no message receipt ('TRUE & SKIP' in the Occam idiom) *and* immediately after each receipt.

4.3 Messages and associated rules

Consider the Paragon rule for a message m that communicates n arguments x. This is defined in Paragon by a set of k rules, each of the form $\mathcal{R}_j$, as follows (Sect. 2.1 explains rules and rewrites in detail) :

$$\begin{array}{lll} \mathcal{S}_j & & \textbf{given } \mathsf{m}(\mathsf{x}) \\ & & \textbf{when } \mathcal{G}_j \\ \rightarrow & \mathcal{S}'_j & \\ & & \textbf{then } \mathcal{C}_j \\ & & \textbf{where } \mathcal{B}_j \end{array}$$

We must translate alternative rules for a given message into a procedure in the target language as follows:

TM$[\mathcal{R}]$ maps a set of Paragon rules for a given message onto a procedure.

TO$[\mathcal{S}]$ maps an object on the left-hand-side of a Paragon rule onto a set of statements that will both test if the current object matches and assign values to names in the pattern match (if necessary). If the match is successful, a flag success is set.

TB[$\mathcal{B}$] maps a set of Paragon **where** bindings onto a set of appropriate assignments.

TC[$\mathcal{C}$] will map the set of message passes onto appropriate set of communications via channels. We define a variant protocol channel which implements one variant per message defined for a class, carrying the message arguments in the rest of the protocol.

TG[$\mathcal{G}$] will map the set of Paragon guards onto a logical expression. These guards will operate not only on the variables bound by the message and on the state of the object, but also on the variables bound by the **TO** pattern-match and the required subset of those bound by the **TB where** bindings.

TS[$\mathcal{S}'$] will map an object onto statements in the target language which will rewrite the state so that it appropriately represents the structure of that object.

Figure 3 shows the result of **TM**[$\mathcal{R}$] for a message with k rules communicating n arguments x. The PROC has the same name as the message received and is defined

```
PROC message-name ( Object-state state, Typename x_0, ..., Typename x_{n-1} )
    BOOL success :
    TO[S_0]                          Pattern match
    IF
        success AND TG[G_0]          Guards
            TB[B_0]                  Bindings
            TS[S'_0]                 Change state
            TC[C_0]                  Communications
        TRUE
            TO[S_1]
            IF
                success AND TG[G_1]
                    ...              And so on to translations for k-1
:
```

Fig. 3. Result of translation function **TM**[$\mathcal{R}$] for k rules

in terms of state, which represents the object receiving the message.

Spontaneous rules have a similar translation to replace the phrase *Deal with spontaneous rule if necessary* in the object processor description (Fig. 2).

4.4 Example

The static example is a simplified version of a fault-tolerant packet-switched communications router for WSI [AO88]. A router contains its address expressed as a tuple of its x and y coordinates on the wafer. As a result of an initialisation procedure, a router also records the status of its neighbours. A neighbour can be working, not working ('dud'), or at the edge. A one packet buffer is used to implement a store and forward buffering mechanism to the receiving neighbour router. A packet carries the address of its destination and some status information as well as the payload.

The packet operates in two modes, blocked or unblocked. While it is unblocked it takes the shortest path to its destination. When it becomes blocked by encountering a wall of faulty routers, it stores its displacement from its destination (the integer in the packet type below), and follows the edge of the area of duds until it finds it is closer to its destination than when it became blocked.

4.5 Paragon description

The classes and data types required are as follows:

class router ::= ⟨address,**vector** [North **upto** West] status,buffer⟩
class buffer ::= ⟨**vector** [North **upto** West] router,packet,direction⟩
data packet ::= ⟨address,handedness,blocked,integer,payload⟩
data address ::= ⟨integer,integer⟩
data status ::= Good | Dud | Edge
data direction ::= North | East | South | West
data handedness ::= Left | Right
data blocked ::= Blocked | Unblocked

The route message takes as arguments the incoming packet and direction:

```
⟨a,_,_⟩          given route(⟨a,_,_,_,p⟩,_)          (Home)
→    self
                 then     action on reaching destination
```

```
⟨⟨x,y⟩,stat,buff⟩   given route(⟨⟨xd,yd⟩,h,b,s,p⟩,dir)          (Unblocked)
                    when     stat [prim] = Good ∧ |x-xd| + |y-yd| < s
→    self
                    then     buff ! send(⟨⟨xd,yd⟩,h,Unblocked,999,p⟩,prim)
                    where    prim = primary(⟨x,y⟩,⟨xd,yd⟩)
```

```
⟨⟨x,y⟩,stat,buff⟩   given route(⟨⟨xd,yd⟩,h,b,s,p⟩,dir)          (Blocked)
                    when     |x-xd| + |y-yd| ≥ s ∨ stat [prim] ≠ Good
→    self
                    then     buff ! send(⟨⟨xd,yd⟩,h',Blocked,s',p⟩,newdir)
                    where    prim = primary(⟨x,y⟩,⟨xd,yd⟩)
```

In all of the above rules, the state of the router remains unchanged. The first rule *(Home)* is applicable when the packet has reached its destination address. The second rule *(Unblocked)* matches when the packet is operating in unblocked mode, or when it is closer to its destination than when it became blocked. The optimal forward direction is given by a table lookup performed by the function primary (for brevity not defined here). This is then used to index into the vector of status registers and the vector of neighbouring routers. Rule *(Blocked)* matches when the packet is following the boundary of the area of duds. In this rule newdir, h', and s' represent new direction, new handedness and new displacement respectively. The equations defining these values appear in the **where** clause in a complete version; here they are omitted.

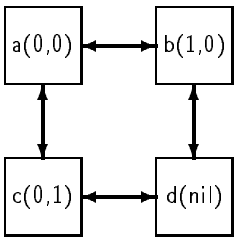
Two further rules for the class buffer implement a simple store and forward buffering mechanism:

⟨sroutes,**nil**,**nil**⟩ **given** send(pack,dir)
→ ⟨sroutes,pack,dir⟩

⟨sroutes,spack,sdir⟩
→ ⟨sroutes,**nil**,**nil**⟩
then sroutes [sdir] ! route(spack,sdir)

The second rule is 'spontaneous'; it does not requires a triggering message. It describes the forwarding action through a message-pass to the appropriate neighbour router identified by indexing the vector of neighbours with the direction sdir stored in the buffer. This allows indefinite delay, thereby modelling asynchronous communication between routers.

An example comprising a four-connected wafer consisting of four routers arranged as a square is shown in Fig. 4, together with the initial object configuration which represents it. For simplicity buffers are not shown in the diagram. To exercise the routing algorithm, in this example router d is non-functional. The vector of neighbour states is given in the order [north,east,south,west].



a = ⟨⟨0,0⟩,[Edge,Good,Good,Edge],Ba⟩
b = ⟨⟨1,0⟩,[Edge,Edge,Dud,Good],Bb⟩
c = ⟨⟨0,1⟩,[Good,Dud,Edge,Edge],Bc⟩
d = **nil**
Ba = ⟨[**nil**,b,c,**nil**],**nil**,**nil**⟩
Bb = ⟨[**nil**,**nil**,d,a],**nil**,**nil**⟩
Bc = ⟨[a,d,**nil**,**nil**],**nil**,**nil**⟩

Fig. 4. A simple specification

4.6 Translation to Occam

We analyse the router example to determine the interconnection topology of the system of required object processors and then apply the translation scheme to the rules. The object processors and topology are trivially given by the following equations:

$$\mathsf{a} \in S(\{\mathsf{Ba}, \mathsf{Pa}\})\ \mathsf{Ba} \in S(\{\mathsf{b}, \mathsf{c}\})\ \mathsf{b} \in S(\{\mathsf{Bb}, \mathsf{Pb}\})\ \mathsf{Bb} \in S(\{\mathsf{a}, \mathsf{d}\})$$
$$\mathsf{c} \in S(\{\mathsf{Bc}, \mathsf{Pc}\})\ \mathsf{Bc} \in S(\{\mathsf{a}, \mathsf{d}\})\ \mathsf{d} \in S(\emptyset)$$

There will normally be one variant per message sort defined for a class. In this specification routers receive the single message route, and buffers the single message send, so the associated protocols therefore contain one variant only:

```
PROTOCOL Buffer-in-Router-out
    CASE send ; Packet ; Direction :
PROTOCOL Buffer-out-Router-in
    CASE route ; Packet ; Direction :
RECORD Buffer-state IS
    [North..West] CHAN OF Buffer-out-Router-in sroutes; Packet spack; Direction sdir :
RECORD Router-state IS
    Address addr; [North..West] Status stat; CHAN OF Buffer-in-Router-out buff :
```

The buffer state is represented by a record, with the four router objects implemented by channels to the appropriate router object processors. Other type declarations are obvious and omitted.

Figure 5 shows the translation to an object processor for class buffer. The buffer

```
PROC buffer-processor(CHAN OF Buffer-in-Router-out input,CHAN OF BOOL bsack,
                      CHAN OF Buffer-out-Router-in N,E,S,W)
  Buffer-state state :
  Packet pack :
  Direction dir :
  state.sroutes [North..West] := [N,E,S,W]   Connect up channels
  WHILE TRUE
    ALT
      input ? send ; pack ; dir
        success := state.spack <> nil AND state.sdir <> nil          Spontaneous
        IF
          success
            state.spack := nil
            state.sdir := nil
            state.sroutes[state.sdir] ! route ; state.spack ; state.sdir
          TRUE
            SKIP
        send (state,pack,dir)
        bsack ! TRUE
      TRUE & SKIP
        success := state.spack <> nil AND state.sdir <> nil
        IF
          success
            state.spack := nil
            state.sdir := nil
            state.sroutes[state.sdir] ! route ; state.spack ; state.sdir
          TRUE
            SKIP
:
```

Fig. 5. Object processor for buffer

processor is connected to a maximum of four routers, one in each direction, and has

one incoming connection.

Figure 6 shows the translated PROC for message route. The buffer is represented by a connection to the appropriate buffer object processor through a channel with a variant protocol for the single message send.

```
PROC route (Router-state state, Packet pack, Direction dir)
    BOOL success : INT disp : Direction prim :
    disp := abs(state.addr.x–pack.addr.y)+abs(state.addr.y–pack.addr.y)
    prim := primary(state.addr,packet.addr)                    Same in both rules
    success := state.addr = pack.addr
    IF
        success
            action on reaching destination
        TRUE
            success := TRUE        No pattern match here
            IF
                success AND state.stat[prim] = Good AND disp < pack.disp
                    pack.blocked := Unblocked
                    pack.disp := 999
                    state.buff ! send ; pack ; prim
                TRUE
                    success := TRUE
                    IF
                        success AND disp >= pack.disp AND state.stat[prim] <> Good
                            Compute new direction newdir, pack.disp, pack.hand
                            pack.blocked := Blocked
                            state.buff ! send ; packet ; newdir
                        TRUE
                            SKIP
:
```

Fig. 6. Translation of the router example

4.7 General systems

This section addresses the complex question of how to implement Paragon specifications whose rules use asynchronous message passing and dynamic object creation (for example, the parallel priority queue in Sect. 2.2).

One approach to dynamic specfications involves analysis to make assumptions about the scope of the required system and possible modification of the specification to make it more amenable to hardware implementation. This is followed by definition of object processors for objects in the dynamic class to match these assumptions. In the case of the priority queue, for example, we might modify the rules for get to 'shuffle up' queue entries leaving an 'empty' entry at the end of the queue. The specified behaviour then more closely matches that of a feasible hardware implementation.

A second approach we describe in more detail defines a source-to-source Paragon transformation to produce a static synchronous specification which can then be translated as defined in Sect. 4.1. We do not suggest particular implementations but describe additional static synchronous systems which are suggestive of conventional solutions using buffers and heaps. Implementation of the more complex communication characteristics produced by the transformations might be (in Occam terms) through passing of indices to shared arrays of channels among the processes.

4.8 Asynchronous message passing

We introduce new objects called *task pools* which will buffer asynchronous communications. All asynchronous communications can be transformed to synchronous communications to a task pool. A task pool is made up of a list of tasks, where a task is defined as a destination, a message and a list of arguments. A task pool can be defined as a synchronous Paragon class receiving the single message add-task. All asynchronous communications, for example a !! m(x), can be translated into T ! add-task(a,m,x), where T is a task-pool. A task pool is required for every object that is asynchronous in any other object. Each task pool associated with an object o must be shared with (i.e. connected to) every object in which o is asynchronous.

4.9 Dynamic specifications

We transform all classes known to be dynamic to use a *heap* to store instances of the class. Each heap will be specific to its class. Individual objects will then be identified by their place in the heap. New objects will be created by allocating a place in the heap. If the specification contains a dynamic class c, we must implement this class with two new objects in the initial state: $H(c)$ which represents the heap for objects of class c; and $P(c)$, known as the *class processor* for c, which will implement the rules for class c.

All objects dynamic in c must be connected synchronously to $H(c)$, and all objects synchronous (asynchronous) in c must be connected synchronously (asynchronously) to $P(c)$.

The heap object receives four messages: new(a), which allocates space for the state of an object and assigns the name-tag for the allocated space to a; free(a), which frees the space with name-tag a; read(x,a), which reads the current object at name-tag a into x; and write(a,x), which rewrites the object at the name-tag a to state x.

The class processor $P(c)$ is a static object to which all messages sent to objects of class c are delivered. Objects of class c are identified by their name-tag in $H(c)$, so the operation of $P(c)$ when it receives a message will be to read the state of the object from $H(c)$, execute the appropriate message procedure, and write the object back if it has been rewritten. Class processors can be defined in the description language in the same way as object processors.

The transformations required on a specification to incorporate heaps and class processors for a dynamic class c in a set of classes C with a typical object a are as follows. All synchronous communications a ! m(x) must be transformed into $P(c)$! m(a,x). All asynchronous communications a !! m(x) must be transformed into

T ! add-task($P(c)$,m,(a,x)), where T is the task pool for C. Bindings containing dynamic object creation through **new** must be transformed to communicate with the heap associated with the class being allocated, so

a = **new** (c, initialstate)

must be transformed into the added communications

$H(c)$! new(a); $H(c)$! write(a,initialstate)

where a is the bound name for the new object in the current rule.

5 Conclusion

Our main purpose in this work was to remove a weak link in the earlier papers applying Paragon to describe the implementation of parallel graph reduction. The objective there was to show how, by a sequence of refinements, a parallel computer architecture could be derived from a specification of its primitive operations as a term rewrite system [BHK91].

The most interesting outcome of the implementation effort has been a much more detailed understanding of the structure of Paragon specifications. This was manifest in the classification and restrictions we imposed on specifications, which seem to identify key issues in any analysis of parallel systems implementation.

For static systems, in addition to the restrictions discussed earlier, we observe that, for a perfectly fair implementation, the given translation requires that exactly one rule be applicable given a particular set of incoming messages (including the empty set, to include spontaneous rewrites). This is rather a strict constraint. Essentially it arises because each object processor has no internal parallelism, so can only test for the applicability of one rule at once, in a predetermined order. This is a generalisation of the rewriting notion of sequentiality. As in functional programming, it very often turns out sequential order of testing is adequate for a particular application. Unlike in the term-rewriting case, the restriction to sequentiality does not result in a deterministic language, since racing between messages can still occur.

We dealt with asynchronous messages and dynamic object creation using a source-to-source Paragon transformation as described in Sect. 4.7. The introduction of "task pools" implies implementation of buffering (lists are used to model buffers in Paragon specifications)—as usual this must be shown to be effective (no overflow, etc) for the application concerned. Of course, garbage collection for dynamic objects is also involved (except in very special cases), and in general this may require cooperation between many of the "class processors" introduced.

To conclude, this paper addressed the problem of bridging the final gap between a low-level Paragon description of an architecture, and the hardware itself. Our experience has been that in the static case the translation is direct enough to support the view that Paragon programs specify hardware, but that for dynamic and asynchronous specifications non-trivial design decisions must be made. The fact that these decisions can be expressed within Paragon lends further support for our use of the language in layered architectural design of parallel systems.

References

[AO88] Paul Anderson and Peter Osmon. A Fault Tolerant Communications Architecture for Wafer Scale Integration. In *Proceedings of the Alvey Technical Conference*, pages 504–507, 1988. City University TCU/CS/1988/13.

[BHK89] David Bolton, Chris Hankin, and Paul Kelly. Parallel object-oriented descriptions of graph reduction machines (extended abstract). In *PARLE'89 Parallel Architectures and Languages Europe*, pages 158–175. Springer Verlag, 1989.

[BHK90] David Bolton, Chris Hankin, and Paul Kelly. Parallel object-oriented descriptions of graph reduction machines. *Future Generations Computer Systems*, 6:225–239, 1990.

[BHK91] D. Bolton, C.L. Hankin, and P.H.J. Kelly. An operational semantics for Paragon: A design notation for parallel architectures. *New Generation Computing*, 9:171–197, 1991.

[BvEG+87a] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. 1987. In [dBNT87, pages 141–158].

[BvEG+87b] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an intermediate language for graph rewriting. 1987. In [dBNT87, pages 159–174].

[dB89] J.W. de Bakker, editor. *Languages for Parallel Architectures.* Parallel Computing. Wiley, 1989.

[dBNT87] J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors. PARLE, *Parallel Architectures and Languages Europe*, volume I. Springer Verlag, June 1987. LNCS 258.

[GKS87] J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep. DACTL: a computational model and compiler target language based on graph reduction. Report SYS-C87-03, school of Information Systems, University of East Anglia, 1987.

[Inm88] Inmos Ltd. *Occam-2 Reference manual.* Prentice Hall International, 1988.

[MK87] David May and Catherine Keane. Compiling occam to silicon. Technical note 23, Inmos Ltd., 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, UK., 1987.

[SJ89] Ph Schnoebelen and Ph Jorrand. *Principles of FP2: Term Algebras for Specification of Parallel Machines.* 1989. In [dB89, pages 223–273].

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Nancy, France.* Springer Verlag, 1985. LNCS 201.

This article was processed using the LaTeX macro package with LMAMULT style