Graph reduction implementation of a production system

Stephen Murrell and Robert Plant*

The paper explores the implementation of rule-based pattern-directed inference systems on parallel computers. The paper discusses one of these approaches in detail, the use of a graph-reduction machine such as ALICE. The technique is illustrated through two example domains: automobile fault diagnosis and organic psychiatric mental disorders. The paper discusses extensions to the graph reduction technique as applied to knowledge-based systems, including partitioning, time considerations and input data types. The paper shows that the graph-reduction technique has significant advantages for knowledge-based system implementation over conventional approaches, and it demonstrates that this programming style is amenable to knowledge engineering domains.

Keywords: parallel processing, graph reduction, production systems, ALICE system

This paper explores the implementation of rule-based pattern directed inference systems on parallel computers.

Traditionally, speed has been a major problem with rule-based production systems. In order to overcome this, three approaches are possible: improvements in implementation, faster computers, and the use of multiple processors. In this paper we take the third path.

There are a great many different styles of parallel computation, all of which could potentially be applied to the problem of rule based production systems. We chose to use graph reduction[1-3] in our investigation for its simple flexibility. There are two other approaches to parallel processing that also have strong promise for future investigation:

- *Transputer-based methods:* These are a popular and flexible approach[4-6] to parallel processing. Some decision-based and diagnostic problems[6] have been implemented in this style, but these have been implementations specific to a particular problem, and they show no apparent scope for generalisation.
- *Hypercube or cosmic cube fixed-topology systems:* Although these are ideal[7] for vector processing and other regular programs, this approach may lack the flexibility of the other approaches, but as an established technology it should not be ignored.

Our investigation into the applicability of the graph reduction approach is based on the ALICE system[†] (the Applicative Language Idealized Computing Engine)[1]. In the first section of this paper we discuss the background to parallel processing and graph reduction machines. Additionally, a helpful introductory example showing graph reduction applied to a simple mathematical problem is given in Reference 1. The use of graph reduction for knowledge-based systems is shown through two examples in the second section of the paper. Firstly, a small example is given for the diagnosis of automobile faults that shows the complete reduction process, followed by a more complex problem in the domain of organic psychiatric mental disorders. After the fundamental techniques of graph reduction applied to knowledge-based systems have been illustrated, the third section discusses extensions to that base, including the partitioning of knowledge bases and the use of non-Boolean inputs, and timing factors. In the final section of the paper, we discuss future research directions and advantages that these techniques have for the knowledge engineering community.

Department of Computer Science and Mathematics, University of Miami, Coral Gables, FL 33124, USA
*Department of Computer Information Systems, University of Miami, Coral Gables, FL 33124, USA
Paper received 2 November 1993. Revised paper received 8 July 1994. Accepted 27 July 1994

[†]We do not use ALICE itself, but a local implementation (MALICE) which follows the original very closely.

ALICE architecture

A program in an ALICE system consists not of the traditional ordered sequence of uniform instructions, but of a set of 'packets' connected in a graph, or tree-like structure. Each packet represents either an operation to be performed, or an item or structure of data. The connections, or arcs, between packets encode dependency, dataflow paths, the order of execution, and the shapes of data structures.

An ALICE system consists of a number of logically identical processing agents sharing and independently executing the same program in parallel. These processing agents, and their interfaces with the shared program store, are designed in such a way that the execution time for any operation has no component which is dependent upon the number of processors currently operating, and ideally so that an individual processing agent can, with minimal delay, be connected to, or disconnected from, a system without disrupting the operations of that system. A consequence of this design is that, if a single processing agent is engaged in a complex computation, a second agent may, at any time, be 'plugged in', and (provided that there is enough work for two agents) halve the required processing time. Similarly, an agent may be 'unplugged' from a system in which it is no longer required, and reconnected to another, or even used as an individual workstation.

Operations of processing agent

When a processing agent is running, it will continually cycle through the familiar 'fetch–execute' sequence. First, it finds in the shared program store (called the 'packet pool') any packet that is suitable for execution, and removes that packet, or marks it in some way, so that no other agent will select the same one. The agent then inspects the contents of the packet, to find which operation it represents, and what the parameters of that operation are to be (they may either be other packets, or simple constants), and it performs the operation by executing a sequence of instructions found in a preloaded program store.

Performing the operations indicated by a single packet will generally involve some or all of the following:

- extracting information from argument packets,
- performing simple, low-level operations (such as arithmetic calculations),
- updating information in other packets,
- creating new packets to represent new data structures, or newly required subcomputations,
- removing obsolete packets,
- causing external actions (such as user input or output),
- replacing the original packet (as ALICE is a graph reduction machine, this must be an 'update-in-place' operation which will leave the surrounding graph structure intact, and the newly created subgraph installed as an integral part of that structure, completely replacing the original packet).

PRODUCTION SYSTEMS THROUGH GRAPH REDUCTION

In this section of the paper we present an approach to the use of graph reduction in the creation of knowledge-based systems. In order to facilitate this, we will develop a knowledge-based system from a decision table, a form of intermediate representation often used by knowledge engineers in the development of knowledge-based systems[8-11].

The approach that we follow is to take the decision table form as an input to a program that translates that table into a graph reduction program that can be run on an ALICE machine (see *Figure 1*).

We will now consider each of these processes in greater detail.

Production of graph-reduction program

The domain upon which our system is based in this first example is trivial, and yet it is designed to illustrate several important aspects of the graph-reduction approach to knowledge-based system implementation. We start our development with a decision table, in this case an automobile fault diagnosis table, consisting of only three conditions and three possible actions (this is shown in *Figure 2*).

Creating a decision table from which to commence our development aids system verification and validation, and is an area of active research[12-15]. However, a full discussion of this is beyond the scope of this paper.

The decision table is processed by a Pascal program which mechanically translates the table into an ALICE program. This has several advantages: the correctness of the system is maintained, and automatic transformation saves labor and time for the knowledge engineer, and aids knowledge engineers who have little experience in developing graph-reduction systems. The full ALICE program resulting from transforming the decision table is given in Appendix 1.

The ALICE program in Appendix 1 is composed of nine parts, each of which describes how to reduce a different type of packet: "initial", "query", "positive", "negative", "and2", "and3", "or2", "and3", and

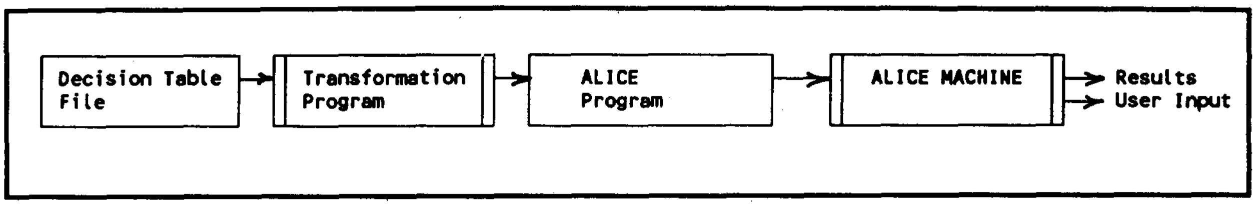
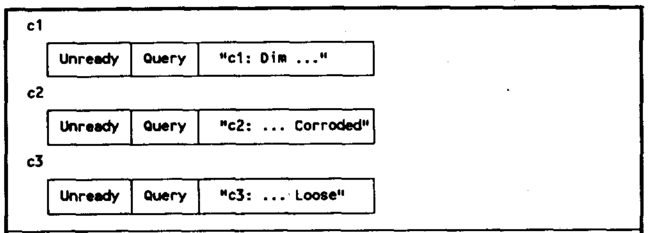


Figure 1 Decision table for graph-reduction transformation process

C1: Dim Headlights	Y	Y	Y	Y
C2: Battery Terminals Corroded	Y	Y	N	N
C3: Battery Terminals Loose		Y	Y	N
A1: Clean Battery Terminals	X			
A2: Tighten Battery Terminals		X	X	
A3: Recharge Battery				X

Figure 2 Decision table for graph-reduction system



```
new(c1) id=query st=unready arg1="c1: Dim Headlights"
new(c2) id=query st=unready arg1="c2: Battery Terminals Corroded"
new(c3) id=query st=unready arg1="c3: Battery Terminals Loose"
```

Figure 3 Condition packets

"conclude". The first part (between 'to reduce initial' and 'to reduce query') is designed to construct the graph over which the graph-reduction system will be run. In this system we create the graph from the queries backwards to the conclusions, in a backward chaining style. First, three condition packets are created, as shown in *Figure 3*.

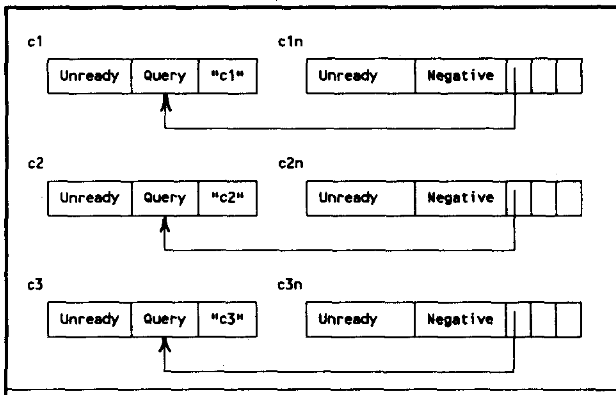
Each packet, when and if it is picked for reduction, is reduced according to the sets of rules provided in the ALICE program. Just as the section headed 'to reduce initial' describes how the initial default seed packet is replaced to produce the machine's correct initial configuration, a later section of the program labelled 'to reduce query' describes in detail how each of these new query packets should be reduced. The identifiers *c*1, *c*2, and *c*3 simply provide a means of referring back to these packets later.

The status field of a packet determines when a packet may be selected for reduction. Only packets with a status of 'ready' may ever be reduced. These query packets will remain unprocessed until their statuses are changed.

If a query packet is ever picked for reduction, the rules (shown in full in Appendix 1) specify that the associated string should be printed, and the user invited to type a response. If the response is 'yes', then the query packet is changed from a computational packet to a data packet representing the value TRUE. If the answer is 'no', then the value is FALSE. Any other response is ignored, and the packet is made available for a second reduction, thus causing the question to be asked again later.

The second aspect of system creation specified in the 'to reduce initial' section is the creation of the *second packet layer* (see *Figure 4*).

This second layer provides negated forms of the conditions. If one of these packets is ever picked for reduction, it will be unable to proceed until its argument (the condition packet that it refers to) has already



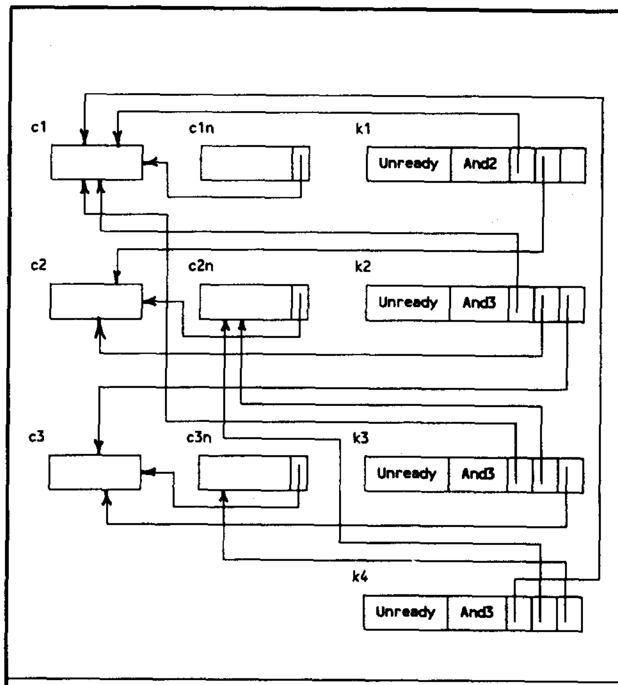
```
new(c1n) id=negative st=unready arg1=c1
new(c2n) id=negative st=unready arg1=c2
new(c3n) id=negative st=unready arg1=c3
```

Figure 4 Second layer propagation packets

been reduced to a data packet. Until that happens, this packet is suspended, and the argument packet's status is changed to 'ready' so that it may be selected. This is how demand for a computation is propagated through the graph of packets.

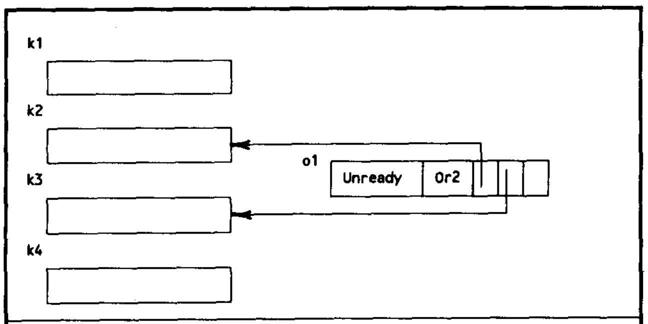
When a 'negative' packet is eventually reduced, it is rewritten as a data packet with the opposite logical value to that of its condition.

The third layer, which represents the left-hand sides of the rules, calculates the conjunction of the relevant conditions (e.g. the packet *k*3, shown in *Figure 5*, represents the condition $C1 \wedge \neg C2 \wedge C3$). Similarly, the fourth layer, which only exists for right-hand sides that are activated by more than one conjunction of conditions, consists of logical disjunctions (see *Figure 6*).



```
new(k1) id=and2 st=unready arg1=c1 arg2=c2
new(k2) id=and3 st=unready arg1=c1 arg2=c2 arg3=c3
new(k3) id=and3 st=unready arg1=c1 arg2=c2n arg3=c3
new(k4) id=and3 st=unready arg1=c1 arg2=c2n arg3=c3n
```

Figure 5 Third layer conjunction packets



```
new(o1) id=or2 st=unready arg1=k2 arg2=k3
```

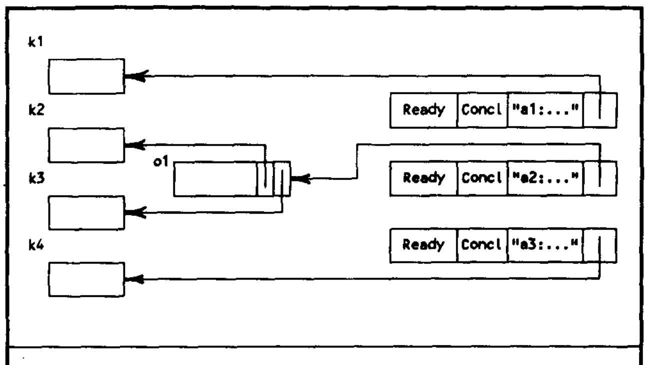
Figure 6 Fourth layer disjunction packets

"and" and "or" packets behave in fundamentally the same way as the 'negative' packets; their arguments are changed from 'unready' to 'ready' if they have not already been reduced to data, and, once the arguments are data, these packets are also rewritten as either TRUE or FALSE data packets according to the operation that they represent. In this implementation, complete evaluation of Booleans is performed. Once one of the arguments of an "and" packet reduces to FALSE, the other packets are still evaluated. This is just a simplification, and certainly not a fixed feature of the graph reduction method.

Having created all of the necessary left-hand side conditions with their logical conjunctions and disjunctions, we can now create packets that represent the right-hand sides of the rules, the conclusions (see *Figure 7*).

These last three packets are not given identifiers because no other packets will need to refer to them; they are the roots of independent computations. The final clause of each, "rc=1", sets their reference counts. ALICE uses reference counts to determine whether or not a packet is still in use. Reference counts are normally calculated automatically, and, if a packet's reference count ever reaches zero, the system recycles it, through a process of garbage collection. Any packet that is deliberately not referred to by any others must be protected from garbage collection by being given an artificially nonzero reference count.

It is by this mechanism that the system recognizes the packet *cln* as unnecessary, and removes it.



```
new id=conclude st=ready arg1=k1 arg2="a1: Clean Battery Terminals" rc=1
new id=conclude st=ready arg1=o1 arg2="a2: Tighten Battery Terminals" rc=1
new id=conclude st=ready arg1=k4 arg2="a3: Recharge Battery" rc=1
```

Figure 7 Creation of right hand side conclusion packets

Reducing the production system graph

Once the graph structure has been created (it is shown in its entirety in *Figure 8*), the ALICE system can proceed to reduce the graph on the basis of the input data from the user.

Whenever a processor is free, a packet with a ready status is selected at random; initially, there are only three possibilities:

new id=conclude st=ready arg1=*k*1
arg2="*a*1: Clean Battery Terminals" rc=1

new id=conclude st=ready arg1=*o*1
arg2="*a*2: Tighten Battery Terminals"rc=1

new id=conclude st=ready arg1=*k*4
arg2="*a*3: Recharge Battery" rc=1

In a machine with a single processor, one of these ready packets is selected. In a multiprocessor system, as many packets as there are processors may be selected simultaneously. For example, the packet representing the conclusion 'tighten battery terminals' could be selected. As its one required argument (the "or" packet) has not yet been reduced to a result, the conclusion is suspended, and the argument is made ready. This process of propagating readiness continues until the queries are ready, and all higher nodes are suspended. In *Figure 9*, which shows only the subtree for one column of the decision table, at this point, the queries would all be 'ready', and all packets to the right would be 'suspended'.

When a query is selected, the corresponding question is asked, and the packet is reduced to data. Any packets that were previously suspended to wait for it are made ready again. In this way, definite results are propagated up the tree until a conclusion is finally reselected. Referring again to *Figure 9*, execution terminates when the conclusion has been printed and all the packets to the left turned to 'data'.

Graph reduction for psychiatric domain

Having demonstrated the utility of the graph reduction approach to knowledge-based systems programming through the small example in the previous two sections, we will now show the application of the technique in a larger domain, that of organic psychiatric disorders. This domain has shown itself to be applicable to the traditional techniques of rule-based expert systems development[16], and it is sufficiently complex to have warranted examination by many researchers[17,18]. In this paper, we will consider only a subsection of the American Psychiatric Association's classification scheme for mental disorders as stated in its *Diagnostic and Statistical Manual of Mental Disorders* (DSM-III-R). The section we will consider is that for organic mental disorders, for which the decision table shown in *Figure 10* can be created from the DSM-III-R classifications.

The creation of the graph-reduction program was achieved by following the methodology illustrated in *Figure 1*, passing the decision table through the Pascal preprocessor in order to produce the ALICE program which is given in Appendix 2. This program reduces in

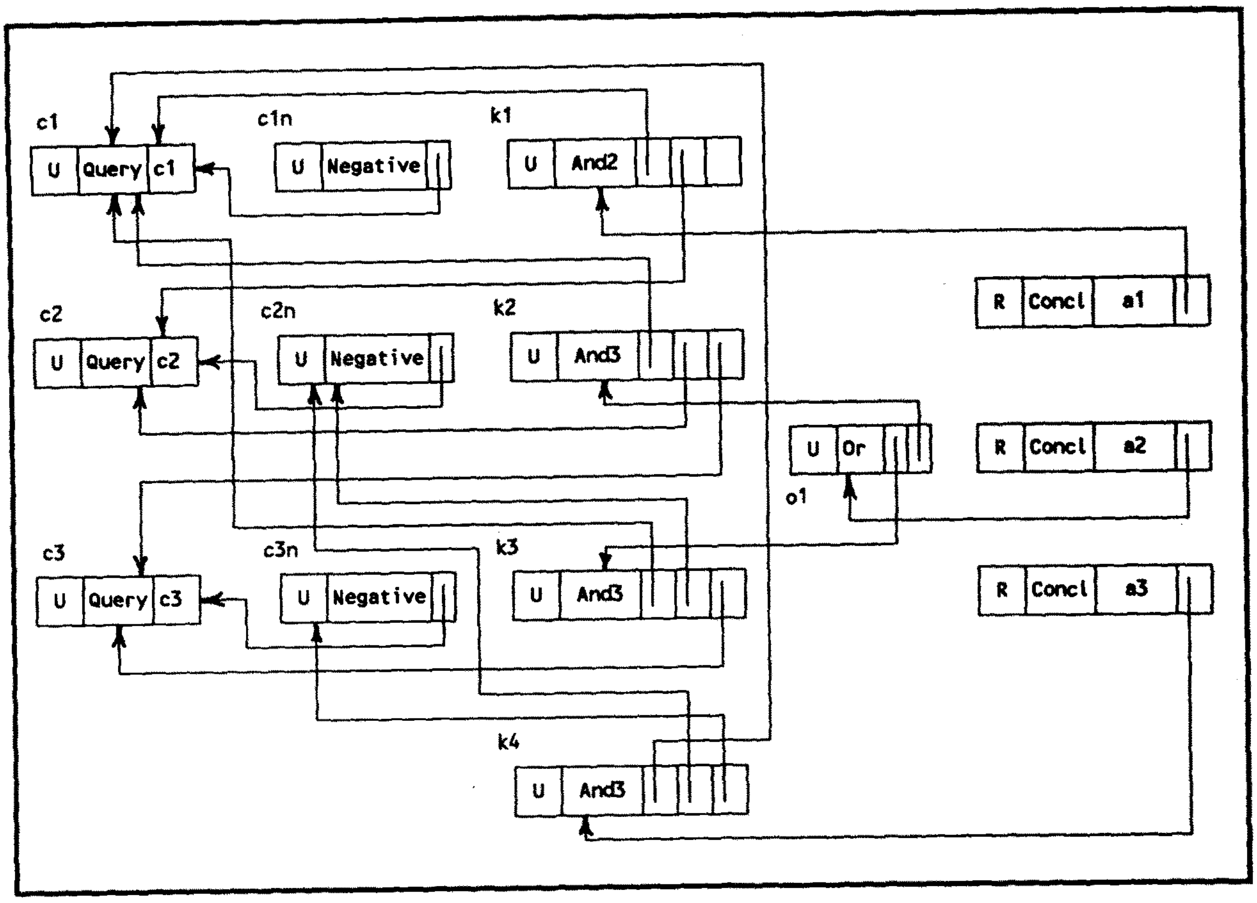


Figure 8 Complete graph

the same manner as the automobile fault diagnosis process described above, and it allows dialogues of the following form to be achieved:

> Query *c*1: Evidence? y
> Query *c*2: Disturbance? y
> Query *c*3: Other symptoms? n
> Query *c*4: Impairment? y
>> Conclusion *a*3: Dementia
> Query *c*5: Memory? y
> Query *c*6: Change? n
>> Conclusion *a*1: Delirium

It should be noted that it is not the aim of this paper to discuss the implications of the use of knowledge-based systems in the medical domain. This is an area of active research and it is beyond the scope of this paper. However the reader is directed to Reference 17 for a comprehensive treatment of this topic.

EXTENSIONS TO GRAPH REDUCTION KNOWLEDGE-BASE DEVELOPMENT

Partitions

The system so far described ensures that no unnecessary questions are asked of the user as a byproduct of backward chaining. No packets (and, in particular, no

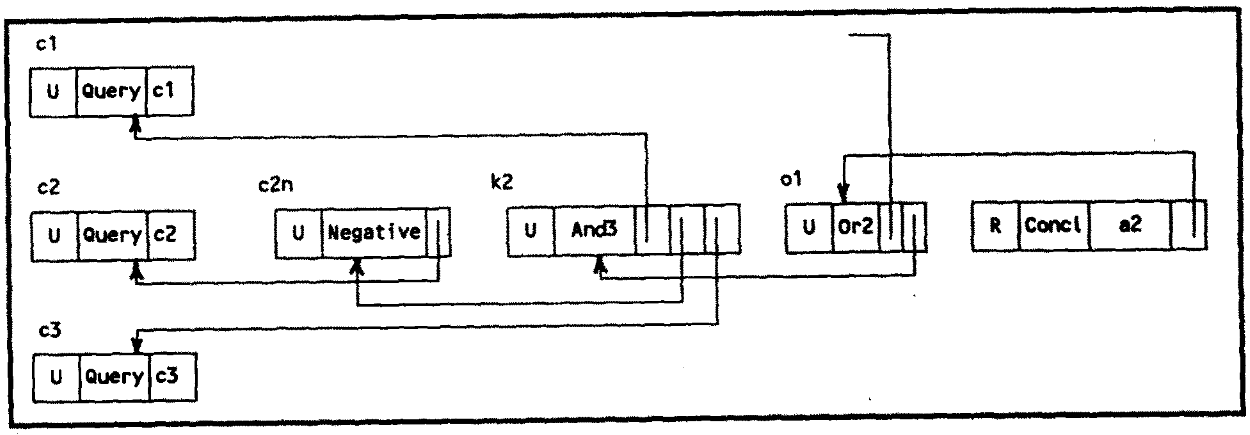


Figure 9 Subtree for rule $C1 \wedge C2 \wedge \neg C3 \rightarrow A2$

	a	b	c	d	e	f	g	h	i	j
c1: Evidence	y	y	y	y	y	y	y	y	y	y
c2: Disturbance	y	y	y	y	y	y	n	n	n	n
c3: Other Symptoms	y	n	n	n	n	n				
c4: Impairment		y	n	n	n	n	y	n	n	n
c5: Memory			y	y	n	n		y	y	n
c6: Change			y	n	y	n		y	n	y
a1: Delirium	x	x	x	x	x	x				
a2: Dementia		x					x			
a3: Amnestic Syndrome			x	x				x	x	
a4: Organic Personality Syndrome			x		x			x		x

Figure 10 Organic psychiatric disorders decision table

packets representing questions) may be reduced until they have a status of 'ready'. Initially, only the packets representing conclusions, or the actions on the right-hand sides of rules, are ready, and the ready status is only propagated to those subcomputations ('and's and 'or's etc.) that are actually needed.

This does not represent any real saving, as questions that contribute to no conclusions at all are unlikely to remain in the rule base of a system once its design is complete. However, the same effect may be used to great advantage if only a subset of all possible conclusions are of interest, and if there is some way of ensuring that only interesting conclusions have their status initially set to 'ready'. In such a case, only those questions that directly contribute to an interesting conclusion would ever be asked.

Fortunately, this is a very simple modification to make to the system. First, the knowledge engineer divides the conclusions up into a number of categories (which may be as broad or narrow, and as meaningful or arbitrary, as desired), and a "category" packet is generated for each:

```
new id=category   st=ready
   arg1="name of category"   arg2=...
```

If there is only one conclusion in a category, the second argument of the corresponding packet points directly to that conclusion packet. If there is more than one, it should point to a tree of 'propagate' packets, as described below.

The category packets are the only ones in the whole system that are created with a status of 'ready'. When one is reduced, it prints the question 'are you interested in [first argument string]?', and waits for input. If the input is 'yes', then it simply activates its second argument, thus propagating the 'ready' status toward the relevant conclusions. If the answer is 'no', nothing is done, and those conclusions remain 'unready'.

The 'propagate' packets perform no computation at all, and are only used to spread the status of 'ready' throughout a larger number of conclusions. When one is reduced, it simply activates all of its own argument packets.

These category packets may also be used in a slightly more general way; related members of the first level of categories may be grouped into supercategories, and so on, producing a tree-like hierarchy of categories, giving the user complete control over which parts of the rule base are activated, and which are not.

In most situations, the user of a rule-based expert system can be expected to have some idea of what kind of problem is to be solved. The ability to leave the majority of conclusions inactive can speed up computation enormously. Of course, this only prevents certain conclusions from being activated (and consequently prevents questions that only lead to those conclusions from being asked); it does not alter the logic behind the conclusions at all.

In fact, there is no reason why the hierarchy of categories should not be arranged to give control over the activation of individual conclusions, so that the rule base may be used to answer one specific question, without any unnecessary work.

Timing

In the worst case (an uncontrolled use of the rule base in which all conclusions are activated, and all queries are asked of the user), every packet is picked for reduction twice; once before its argument packets have been reduced, when its job is to propagate the 'ready' status throughout the system, and then a second time, when it is actually reduced to either TRUE or FALSE. Thus, the time required (for a single processor) is directly proportional to the number of packets in the initial configuration, which is itself proportional to the size of the original decision table. (The initial configuration consists mainly of one tree of "and" packets for every column in the decision table, the size of that tree being given by the number of conditions contributing to that column, which is in turn bounded by the total number of conditions). Thus, the worst case time for deductions is linear in the problem size.

When partitioning is used to isolate individual conclusions, the best-case time may be achieved. When only one conclusion is active, and there are enough processors in the system to perform all of the appropriate reductions concurrently, the time required is proportional to the depth of the AND–OR tree controlling the active conclusion. Because each packet in the tree can reference three other packets, that depth is approximately the base-3 logarithm of the number of conditions in each AND tree, plus the base-3 logarithm of the number of AND trees in the OR tree. Thus the best-case time for a deduction is logarithmic in the problem size.

Non-Boolean inputs

In many systems, the responses to queries are not all either TRUE or FALSE, but are drawn from a larger set, such as the integers. In such cases, an extra layer of packets is required. The input from the user is requested by a modified form of "query" packet, which rewrites itself as an integer data packet. The following are examples, with the first being before reduction, and the second after reduction:

```
id=QueryInt   st=ready   arg1="How many eggs"

id=Integer   st=Data   arg1=6
```

The extra layer is composed of relational packets (corresponding to the relational operators =, < , > etc.) which have two arguments; when both the arguments have been reduced to integer data, they themselves are rewritten as either TRUE or FALSE Boolean data packets, depending upon the result of the comparison. These relational packets take the places of the original query packets.

COMMENTS AND CONCLUSIONS

This paper has demonstrated that graph reduction can be successfully applied to the implementation of production-rule knowledge-based systems. We have shown an approach to the mapping of decision tables to a form suitable for parallel processing.

The graph-reduction approach has several advantages.

The ALICE machine treats each packet as a completely independent entity which is reduced in isolation. This vastly simplifies the allocation and sharing of resources, which is often a significant overhead in other forms of parallel processing.

The approach we have used also has a best case logarithmic time for its computations which is out of the question for sequential systems. It gains some extra efficiency from ALICE's selective reduction policy.

Owing to the existence of complete formal specifications for the ALICE systems[19], and the simple nature of the input decision tables, validation and verification can become a realistic goal[20].

We feel that this approach will be of lasting value, because it only requires very simple input, and it makes use of an emerging technology[21].

REFERENCES

1 Darlington, J & Reeve, M (1981) 'ALICE: multiprocessor reduction machine for the parallel evaluation of applicative languages' *ACM/MIT Conf. Functional Programming Languages and Computer Architecture* NH, USA

2 Reeve, M & Zenith, S E (Eds.) (1989) *Parallel Processing and Artificial Intelligence* John Wiley, UK

3 Peyton-Jones, S L, Clack, C and Salkild, J (1989) 'High performance parallel graph reduction' *Proc. PARLE '89 Parallel Architectures and Languages Europe. Vol 1: Parallel Architectures* Springer-Verlag, Germany

4 (1984) *OCCAM Programming Manual* Prentice Hall

5 Jones, G (1987) *Progamming in Occam* Prentice Hall, USA

6 Hains, G & Todd, B S (1988) 'The parallel implementation of a medical diagnostic model' *Proc. Third International Conference on Supercomputing – Vol 1* pp 222–229

7 Seitz, C (1985) 'The cosmic cube' *Communications ACM* Vol 28 No 1 pp 22–33

8 Metzner, J R & Barnes, B (1977) *Decision Table Languages and Systems* Academic Press, USA

9 Cohen, P R & Feigenbaum, E A (1982) *The Handbook of Artificial Intelligence – Vol 3* Pitman

10 Davis, R & King, J (1976) 'An overview of production systems' *in* Elcock, E W & Michie, D (Eds.) *Machine Intelligence – Vol 8* John Wiley, USA, pp 300–332

11 Murrell, S & Plant, R (1994) 'Decision tables: formalisation, validation and verification' *Report 93–0602* Department of CIS, University of Miami, USA

12 O'Leary, D E (Ed.) (1994) *Collected Papers of AAAI Workshops on Validation and Verification 1988–92* John Wiley, USA

13 Preece, A D, Shinghal, R & Batarekh, A (1992) 'Verifying expert systems: a logical framework and a practical tool' *Expert Systems with Applications* Vol 5 pp 421–436

14 Rushby, J (1988) 'Quality measures and assurance for AI software' *Contractor Report 4187* NASA, USA

15 Culbert, C (1990) 'Verification and validation of knowledge-based systems' *Expert Systems with Applications* Vol 1 No 3

16 Moreno, H R & Plant, R T (1993) 'A prototype decision support system for differential diagnosis of psychotic, mood, and organic mental disorders' *Medical Decision Making* Vol 13 pp 43–48

17 Jakab, I (1992) 'Artificial intelligence in medicine and psychiatry: new developments in the 1990s' *Proc. Third Annual Symposium Int. Association of Knowledge Engineers* Washington DC, USA, pp 241–261

18 Servan-Schreiber, D (1986) 'Artificial intelligence and psychiatry' *Journal of Nervous and Mental Disease* Vol 174 No 4 pp 191–202

19 Murrell, S (1988) 'State transition specifications for abstract machines' *DPhil Thesis* Computing Laboratory, University of Oxford, UK

20 Murrell, S & Plant, R T (1993) 'Validation and verification of graph-reduction knowledge-based systems' *Working Paper* Dep. Computer Science, University of Miami, USA

21 Townsend, P (1987) 'Flagship hardware and implementation' *ICI Technical Journal* Vol 5 No 3 pp 575–594

APPENDIX 1

```
To reduce initial:
  new(c1) id=query  st=unready
    arg1= "c1: Dim Headlights"
  new(c2) id=query  st=unready
    arg1="c2: Battery Terminals Corroded"
  new(c3) id=query  st=unready
    arg1="c3: Battery Terminals Loose"
  new(c1n) id=negative  st=unready  arg1=c1
  new(c2n) id=negative  st=unready  arg1=c2
  new(c3n) id=negative  st=unready  arg1=c3
  new(k1) id=and2  st=unready  arg1=c1  arg2=c2
  new(k2) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3
  new(k3) id=and3  st=unready  arg1=c1  arg2=c2n
    arg3=c3
  new(k4) id=and3  st=unready  arg1=c1  arg2=c2n
    arg3=c3n
  new(o2) id=or2  st=unready  arg1=k2  arg2=k3
  new id=conclude  st=ready  arg1=K1
    arg2="a1: Clean Battery Terminals"rc=1
  new id=conclude  st=ready  arg1=r2
    arg2="a2: Tighten Battery Terminals"rc=1
  new id=conclude  st=ready  arg1=K4
    arg2="a3: Recharge Battery"rc=1
  rewrite id=null  st=data  rc=0

to reduce query:
  write string "Query: C"
  write integer arg1
  write string "?"
  read char x
  read char y
  k=0
  if (x=89) or (x=121)
    k=1
    rewrite id=true  st=data  arg1=arg1
  if (x78) or (x110)
    k=1
    rewrite id=false  st=data  arg1=arg1
  if k=0
    rewrite id=query  st=ready  arg1=arg1

to reduce positive:
  require arg1
  if (arg1.id)=true
    rewrite id=true  st=data  arg1=arg1.arg1
  else
    rewrite id=false  st=data  arg1=arg1.arg1

to reduce negative:
  require arg1
  if (arg1.id)=true
    rewrite id=false  st=data  arg1=arg1.arg1
  else
    rewrite id=true  st=data  arg1=arg1.arg1

to reduce and2:
  require arg1
  require arg2
  if ((arg1.id)=true) and ((arg2.id)=true)
    rewrite id=true  st=data  arg1=arg1  arg2=arg2
  else
    rewrite id=false  st=data  arg1=arg1  arg2=arg2

to reduce and3:
  require arg1
  require arg2
  require arg3
  if ((arg1.id)=true) and ((arg2.id)=true) and
     ((arg3.id)=true)
    rewrite id=true  st=data  arg1=arg1  arg2=arg2
      arg3=arg3
  else
    rewrite id=false  st=data  arg1=arg1  arg2=arg2
      arg3=arg3

to reduce or2:
  require arg1
  require arg2
  if ((arg1.id)=true) or ((arg2.id)=true)
    rewrite id=true  st=data  arg1=arg1  arg2=arg2
  else
    rewrite id=false  st=data  arg1=arg1  arg2=arg2

to reduce or3:
  require arg1
  require arg2
  require arg3
  if ((arg1.id)=true) or ((arg2.id)=true) or
     ((arg3.id_=true)
    rewrite id=true  st=data  arg1=arg1  arg2=arg2
      arg3=arg3
  else
    rewrite id=false  st=data  arg1=arg1  arg2=arg2
      arg3=arg3

to reduce conclude:
  require arg2
  if ((arg2.id)=true)
    write string "Conclusion:A"
    write integer arg1
    write char 10
    rewrite id=true  st=data  arg1=arg1
  else
    rewrite id=false  st=data  arg1=arg1
```

APPENDIX 2

```
To reduce initial:
  new(c1) id=query  st=unready
    arg1="c1: Evidence"
  new(c2) id=query  st=unready
    arg1="c2: Disturbance"
  new(c3) id=query  st=unready
    arg1="c3: Other Symptoms"
  new(c4) id=query  st=unready
    arg1="c4: Impairment"
  new(c5) id=query  st=unready
    arg1="c5: Memory"
  new(c6) id=query  st=unready
    arg1="c6: Change"
  new(c1n) id=negative  st=unready  arg1=c1
  new(c2n) id=negative  st=unready  arg1=c2
  new(c3n) id=negative  st=unready  arg1=c3
  new(c4n) id=negative  st=unready  arg1=c4
  new(c5n) id=negative  st=unready  arg1=c5
  new(c6n) id=negative  st=unready  arg1=c6
  new(k1) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3
  new(x1) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3n
  new(k2) id=and2  st=unready  arg1=x1  arg2=c4
  new(x2) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3n
  new(x3) id=and3  st=unready  arg1=c4n  arg2=c5
    arg3=c6
  new(k3) id=and2  st=unready  arg1=x2  arg2=x3
  new(k4) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3n
  new(x5) id=and3  st=unready  arg1=c4n  arg2=c5
    arg3=c6n
  new(k4) id=and2  st=unready  arg1=x4  arg2=x5
  new(x6) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3n
  new(x7) id=and3  st=unready  arg1=c4n
    arg2=c5n  arg3=c6
  new(k5) id=and2  st=unready  arg1=x6  arg2=x7
  new(k8) id=and3  st=unready  arg1=c1  arg2=c2
    arg3=c3n
  new(x9) id=and3  st=unready  arg1=c4n
    arg2=c5n  arg3=c6n
  new(k6) id=and2  st=unready  arg1=x8  arg2=x9
  new(k7) id=and3  st=unready  arg1=c1  arg2=c2n
    arg3=c4
  new(x10) id=and3  st=unready  arg1=c1
    arg2=c2n  arg3=c4n
  new(x11) id=and2  st=unready  arg1=c5  arg2=c6
  new(k8) id=and2  st=unready  arg1=x10
    arg2=x11
  new(x12) id=and3  st=unready  arg1=c1
    arg2=c2n  arg3=c4n
```

```
new(x13) id=and2   st=unready   arg1=c5
  arg2=c6n
new(k9) id=and2   st=unready   arg1=x12
  arg2=x13
new(x14) id=and3   st=unready   arg1=c1
  arg2=c2n  arg3=c4n
new(x15) id=and2   st=unready   arg1=c5n
  arg2=c6
new(k10) id=and2   st=unready   arg1=x14
  arg2=x15
new(x16) id=and3   st=unready   arg1=c1
  arg2=c2n  arg3=c4n
new(x17) id=and2   st=unready   arg1=c5n
  arg2=c6n
new(k11) id=and2   st=unready   arg1=x16
  arg2=x17
new(x18) id=or3   st=unready   arg1=k1   arg2=k2
  arg3=k3
new(x19) id=or3   st=unready   arg1=k4   arg2=k5
  arg3=k6
new(r1) id=or2   st=unready   arg1=x18   arg2=x19
new(o1) id=conclude   st=unready   arg1=r1
  arg2="a1: Delirium"
new(r3) id=or2   st=unready   arg1=k2   arg2=k7
new(o3) id=conclude   st=unready   arg1=r3
  arg2="a3: Dementia"
new(x20) id=or3   st=unready   arg1=k3
  arg2=k4   arg3=k8
new(r4) id=or2   st=unready   arg1=x20   arg2=k9
new(o4) id=conclude   st=unready   arg1=r4
  arg2= "a4: Amnestic Syndrome"
new(x21) id=or3   st=unready   arg1=k3   arg2=k5
  arg3=k8
new(r5) id=or2   st=unready   arg1=x21   arg2=k10
new(o5) id=conclude   st=unready   arg1=r5
  arg2= "a5: Organic Person... Syndrome"
new(i1) id=iswanted   st=unready
  arg1= "g1:Dementia" arg2=o3
new(g2) id=propagate3   st=unready   arg1=o1
  arg2=o4   arg3=o5
new(i2) id=iswanted   st=unready
  arg1="g2: All Other Diagnoses"arg2=g2
new(g0) id=propagate2   st=unready   arg1=i1
  arg2=i2
rewrite id=iswanted   arg1="starting"   arg2=g0

to reduce iswanted:
  write string "Are you interested in "
  write string arg1
  write string "?"
  read char x
  read char y
  if (x=89) or (x=121)
    rewrite id=wakeup   arg1=arg2
  else
    if (x=78) or (x=110)
      rewrite id=useless   st=data   rc=0
    else
      rewrite id=iswanted   arg1=arg1   arg2=arg2

to reduce wakeup:
  require arg1
  rewrite id=useless   st=data   rc=0

to reduce propagate 2:
  require arg1
  require arg2
  rewrite id=useless   st=data   rc=0

to reduce propagate3:
  require arg1
  require arg2
  require arg3
  rewrite id=useless   st=data   rc=0
```