CODIL: The Architecture of an Information Language

C. F. REYNOLDS*
Department of Computer Science, Brunel University, Uxbridge UB8 3PH

This paper describes the CODIL information language in terms of an infinitely recursive model for poorly structured information. This approach is significantly different from the established approaches to programming language and database theory. The paper then shows how the model has been adapted to work efficiently in the existing CODIL and MicroCODIL interpreters.

Received March 1987, revised December 1987

1. INTRODUCTION

1.1 General

Twenty years ago an unusual ad hoc approach was proposed to tackle the problems of poorly structured information. It was called CODIL, which stood for COntext Dependant Information Language.[2,4,5] Such a move was contrary to the trends of the time, which was to move from the badly structured primitive data bases of the 1960s towards highly formalised and well-structured models, such as those proposed by Codd[1] (1970). As a result the original CODIL research team was disbanded. Since then the work has continued on a small scale, with a complete software package being written and tested on a wide range of applications.[7,8,11,12]

The recent upsurge in interest in knowledge bases and expert systems reflects a realisation among the computing fraternity that there are many problems which cannot be adequately described in terms of well-structured rules and data. During a period of convalescence the author took the opportunity to see how the original CODIL ideas might be updated, in order to present a range of modern research ideas in a context suitable for use in the school classroom. The result is a package, now being marketed, called MicroCODIL.[13-17]

The choice of computer for implementing Micro-CODIL was dictated by the very large numbers of BBC microcomputers in British schools. The need to use a small computer (for example 25K bytes of working memory) led to a complete rethinking of the design. This has led to a better understanding of the theoretical model underlying the original CODIL proposals. This model, and the way in which it has been adapted to produce operational interpreters, forms the subject of this paper. In some cases certain of the new concepts (in particular 'memories') have not been implemented because of the restrictive nature of the computer used.

1.2 The underlying philosophy

It is important to realise that the CODIL approach is philosophically very different from that underlying procedural, functional or logic languages. In CODIL all the complexity is in the 'data' being processed by a single very simple but very general 'decision making' routine. This contrasts with approaches which attempt to localise the complexity in a set of 'rules' (of some kind or another) which operate on some kind of ordered atomic 'data' structure.

Of course, when it comes to actual operational tasks this philosophical distinction may not be immediately apparent. For instance it may sometimes be useful to think about CODIL as a production rule based system[9] or alternatively to use it as if it were a relational data base.[3] These similarities simply reflect a convergence caused by the nature of the task being performed. The number of such operational similarities is legion and, unless the nature of the convergence is discussed in detail, it is easy to mislead by a casual reference. On the other hand there are other areas where the differences become particularly noticeable. For example the idea of having a function with parameters identified by order is totally alien to CODIL, which makes no assumptions about the order of variables, or even whether they have null or multiple values.

To allow the paper to concentrate on the structure of CODIL, discussion of other approaches to computer languages, etc., is only included when it will make a positive contribution to understanding. For the same reason detailed reference to earlier work on CODIL has been kept to a minimum. For instance the implications of a highly recursive approach were briefly examined some time ago[16] but were not followed up at the time due to lack of resources. In particular this paper only briefly mentions human factors or psychological modelling[6,17] despite the fact that much of the motivation for the work has been linked to observations on how non-computer people think about information. These matters are more appropriately covered in a separate paper.

It should be noted that all examples are written to conform to the MicroCODIL syntax and do not necessarily apply to CODIL. This is done for convenience of the reader, who need not be aware of the changes that have happened to the language over the years. The only exception is in the use of ';' and ':' as item separators to allow multiple items to be shown compactly on a single line – which is not done in MicroCODIL.

2. THE COMPLETELY GENERAL MODEL

The starting point for understanding CODIL is a highly abstract but very simple model. It can be described in terms of three definitions given below:

(1) *An item is a logical collection of items.*

This definition is infinitely recursive, and in any actual implementation it must be terminated by introducing

* Address for correspondence: CODIL Language Systems, 33 Buckingham Road, Tring, Herts, HP23 4HG.

some kind of atomic structure. There is no restriction on what items an item may contain, and in particular an item may contain itself. An item may consist of items of a similar degree of complexity as itself or it may only be decomposable into simpler (or more complex) items.

It should be noted that this approach has much in common with a natural-language dictionary, where each word is defined in terms of others with no escape! If a word is found which is not so defined this is due to the incompleteness of the dictionary, rather than evidence of 'atomic' words.

(2) *A decison-making unit selects an item, and either compares it with another item, or adds it to the latter item.*
The decision-making process involves the stepwise decomposition of an item, into its components, the comparing of items, and the transfer of items. This may be going on in parallel, but for any particular activation there will be a single item whose component items describe the current information context. Such an active item is referred to as a memory. Each memory item will itself be a component in a higher-level memory. It is clear that the decision-making process must reflect the recursive nature of items and memories.

(3) *Anything that is valid anywhere, is valid everywhere.*
This is really a consequence of the first definition. The only objects in the system are items, and it is sensible to treat items in a uniform way. At the same time it must be realised that an item, or a group of items, must be able to be used in ways which provide facilities which are conventionally provided by components of very different types. For instance a CODIL item may have to act as a boolean function, program, subroutine, data value, label, etc., depending on the context. In functional programming terms CODIL items are very heavily overloaded.

Because of Godel's theorem it should be realised that there will be items which will not have an unambiguous interpretation everywhere, and so the above statement is also idealistic. A good example is the self-referential item, x: = x + 1, which is an assignment in command context, always false as a condition, and infinitely recursive as data.

3. RESOLVING THE ABSTRACT MODEL

In translating the abstract model into a practical working system the following factors need to be considered.

Resolving the infinite recursions. It is impossible to build a working system with the totally recursive definition of an item. An 'atomic structure' has to be introduced below which further decomposition is not necessary. For the same reason there will be some highest level memory which encompasses the whole system.

Functional capability. The abstract model contains no facilities for input or output, arithmetic, etc. Such facilities will need to be provided.

Psychological orientation. The system is designed for human beings, rather than for electronic machines or mathematically precise automata. As a result simplicity and ease of understanding must be considered. These factors have been discussed elsewhere.[16]

Optimisation for poorly structured information. The highly recursive structure makes this approach particularly suitable for handling poorly structured information. (It should be noted that any well structured problem can be considered as a subset of a more general poorly structured problem which has been selected because it conforms to certain convenient predefined rules. There is therefore no reason why an information system designed to handle a given class of poorly structured problems should not be able to handle related well structured problems. The reverse is demonstrably not true.)

Modular construction. To maximise the ease of implementation, understanding and flexibility a highly modular structure is required. In particular there should be as clean a separation as possible between the representation of simple items, the structures within which they are held, the decision-making procedures and the functional packages. One of the major advantages of this approach is that this can be done remarkably easily.

Efficiency. In any practical software gratuitous recursion is to be avoided as this would significantly degrade the overall performance. This means that, for example, arithmetic is treated as a black box facility rather than attempting to model it within the design architecture.

4. TERMINOLOGY

This section defines the basic information structuring terms and, where appropriate, relates them to conventional computing technology, or to psychological modelling as appropriate.

4.1 Domain name

All information in CODIL is identified by a named domain and domain names are the only kind of name allowed in the language.
In CODIL a name is simply the means of identifying information, and the mechanism is independent of the type of information being addressed. Syntactically *all* names are equivalent and there are no formal distinctions equivalent to those found in more conventional languages between, for example, procedure names, file names and variable names.

4.2 Items of information

An item of information is a representation of either:
(1) *A domain*
(2) *A subset of a domain*
(3) *A property of a domain*
(4) *An operation on a domain*

Items are the logical 'words' of the CODIL system, and *all information* is stored as lists of items. Items correspond to the logical 'chunks of information' of psychological models.

Each item is identified with a domain name, a representation of a member (or property) of that domain, and some additional information on how the representation relates to the domain. The typical item corresponds to a variable and its value. For instance the item UNIVERSITY = Brunel indicates that one is within the domain UNIVERSITY and has selected a subset containing a single member (or item value), Brunel. Such items will be referred to as 'simple items'. Another item might be COUNTER (TYPE) = N6 which is used in MicroCODIL to define a property of COUNTER, which is that its values should be integer numbers of no more than six digits. The representation of items involving more general subsets, properties and operations will be discussed later.

4.3 Statement

A statement is an unbranched list of one or more items which describes a real or hypothetical situation.

If a statement consists only of simple items it can be considered as being equivalent to a record consisting of correspondingly named fields. Alternatively a statement can be considered as being equivalent to a production rule, in which all non-terminal items represent the conditions for which the terminal item is to be applied.

There are no system-imposed constraints on the order of the items in a statement or the domains from which they are drawn. However there may be advantages to the user and to the system in favouring certain task-specific orderings. For this reason the ordering of items is usually maintained, and items are explicitly numbered.

4.4 Memory

A memory is a statement which is currently being examined by the decision-making process.

The main memory is referred to as 'the facts' and is intended to correspond to the human short-term memory. (In making this analogy it is important to realise that the aim is to produce a memory structure which the user can identify with. An accurate psychological model would be unsatisfactory in operational terms because, for example, it would sometimes lose information during interruptions!) In computing terms a CODIL memory can be considered as an associatively addressed cache memory.

4.5 Construct

A construct is a list of zero or more statements which jointly define a named domain.

A construct is a named list of lists of items – and as long as this is how it appears to the rest of the CODIL system its internal organisation is irrelevant. This is an extremely important feature as it means that the way in which the information is stored (both physically and logically) is of no concern to the system as a whole. In MicroCODIL there is a useful 'shorthand' for compressing constructs by losing the duplicated leading items from adjacent statements. This aids user comprehension while minimising processing and storage demands.

The name of a construct is a domain name and may be used as part of an item. This means that a construct may contain items which represent constructs which may contain items.... It is even possible for a construct to contain items which recursively refer to itself.

Depending on the use to which it is put, a construct may be considered to be equivalent to a file of data, a procedure or set of production rules, or a function returning a probability (usually as a simple 'true' or 'false'). However these distinctions are purely semantic.

4.6 Knowledge base

The knowledge base is the collection of constructs which make up the task universe.

The knowledge base is equivalent to human 'long-term memory' and contains all the information available to the system. It can be considered as a high level memory, in which the component items are the names of the constructs.

5. THE CODIL ITEM STRUCTURE

5.1 Introduction

Before specifying all the elements of a CODIL item it is useful to look at a simple example. The item UNIVERSITY = Brunel behaves as if it consisted of a lower level memory containing:

DOMAIN = UNIVERSITY
VALUE = Brunel
SELECTOR = Set membership
LENGTH = 6
TYPE = Alphabetic
ISA = ORGANISATION

where SELECTOR represents the '=' symbol in the original item, LENGTH represents number of bytes in the value, and TYPE reflects the type of bytes which compose the value. The ISA *item is a property associated with* DOMAIN = UNIVERSITY which indicates that all items in the UNIVERSITY domain are also in the ORGANISATION domain.

While such an expansion conforms to the theoretical model it is undesirable, from a practical point of view, to carry it out in such an explicit manner. In psychological terms such activities are somewhat similar to the processing of information in the sensory short-term memories, where processing is carried out subconsciously. There is good reason for thinking that, for the user's own good, such detailed processing should be hidden. From the efficiency point of view introducing additional levels of decomposition means significantly more recursion, which it would be desirable to avoid.

It was stated earlier that it is essential to bottom out the recursive item definition at some point, and the boundary described above seems to be the most convenient level from both the user and the system point of view. Thus items such as UNIVERSITY = Brunel would be in the user-visible domain, while items such as LENGTH = 6 are internal variables from a notional low-level memory within the supporting software's 'black box'.

In practice there must be methods of crossing this boundary when required. The user needs indirect access to domains such as ISA or LENGTH and such an interface is provided by making the domain names known to the system. Thus the length of a UNIVERSITY item can be probed by UNIVERSITY (LENGTH) = 6? while the relationship between the UNIVERSITY and ORGANISATION domains can be defined by UNIVERSITY (ISA) = ORGANISATION.

Domain names, such as LENGTH, and ISA, which have special meanings to the system software, are known as system names. Items involving such names will, in appropriate circumstances, be 'expanded' into a self-contained module of code within the software, rather than into collections of items.

5.2 The component parts of an item

In this section the component parts of an item are discussed in comparatively general terms, and for arbitrary implementation details, such as the maximum number of characters in a domain name, the reader is referred to the appropriate reference documentation. It should be noted that there are no context restrictions on any of the features. In addition there is a series of library routines which carry out basic functions on items, when provided with the appropriate item addresses. This is an essential part of the modular approach and means, for example, that a high level module can ask if two items are identical without even having to know the data types supported by the software!

Item level. Items usually have a *level number* associated with them. This relates to their position in a higher level information structure, and is used for control purposes.

Item name. All items must have a domain name, and this is the only explicitly required component of an item. This name may be the name of a construct or a system function.

Qualifier name. Any item may be qualified by another domain name, which is written in brackets after the item name. This indicates that the processing of the item must be modified by applying an item from the named domain. If an item is not explicitly qualified, a user-defined default qualifier may be applied. It should be noted that there are no type restrictions on qualifiers. This means that a qualifier might be a numeric subscript, a probability, a system name, a construct or be undefined.

Qualifier number. An explicit bracketed numeric subscript may be associated with an item when it is desired to select a single item from a statement by its relative position. In addition a value may be dynamically assigned by the system if automatic indexing is being used.

Probability. All items have a probability value associated with them. This is explicitly shown if it has a value between zero and unity. By default all items with no explicit probability assume the probability of unity, and any item which acquires a probability of zero 'vanishes' from the system.

Automatic indexing indicators. The symbols £ and @ are used to indicate that automatic indexing is to be applied. When used they cause a suitable value to be placed in the 'qualifier number'. (Automatic indexing is not discussed further in this paper.)

Expression indicator. A ':' is used to indicate that the item value is an expression to be dynamically evaluated rather than a literal value. This will often involve arithmetic but may be used as a pointer. For example OWNER:= OCCUPIER indicates that whenever an attempt is made to access OWNER the appropriate properties of OCCUPIER should be used.

Item logic. Combinations of the symbols '<', '=', and '>' are used to relate the item value with the parent domain. On its own a '=' symbol indicates that the value identifies a member of the named domain. The '<' and '>' symbols are used to delimit ranges.

Item value. In the current implementations item values are limited to ASCII character strings, which can be interpreted as numbers if their form allows.

Terminal punctuation. An item will normally have a final ',' or '.' associated with it. A full stop indicates that the item is at the end of the statement in which it is held.

5.3 The alternative approaches

It is important to realise that there are some arbitrary decisions in the above structure. The choices taken have been shown to work but it is useful to look at some of the alternatives.

The most fundamental relates to whether a particular facility is to be treated as a primitive or not. For example, should an arithmetic expression, currently represented by a colon, be replaced by a named qualifier, such as (EXPRESSION)? In the same way, should the (APPROX) facility be given a shorthand such as '~'?

Because of the highly modular approach it should also be comparatively easy to extend the range of data types CODIL can handle. In particular there is no conceptual reason why other types of information, for example pictures or sound, should not be included as long as an appropriate type tag is included in the item, and appropriate input, output and processing modules are added to the system library.

6. CONSTRUCTS

Constructs are the means of holding information within the system. In logical terms each construct consists of zero or more statements, and each statement contains one or more items. In practice there are no constraints on how the information is actually stored as long as it presents the correct interface. Any information structure can be used as long as a suitable demon is constructed. A suitable demon should be able to respond to the following requests:

(*a*) Provide the next item from the current statement.
(*b*) Provide the first item from the next statement.
(*c*) Provide the next statement in toto.
(*d*) Indicate the end-of-construct condition as appropriate.
(*e*) Accept a complete statement and incorporate it into the existing statements on the construct.

In addition the demon may have access to information in the currently active memory, which can allow additional information to be communicated.

These rules provide a highly modular interface. The decision-making process, and other support routines, do not know, or need to know the actual logical or physical structures needed to store the information. When an item or statement is requested it does not matter whether it comes from a 'codil' file, the keyboard, an array, a relational data base file, a semantic network, or a function which computes appropriate values and returns them in item format. If the construct is logically indexed with a 'key' the effect is to operate on the subset of the construct relevant to the key. This may be done by searching the construct, using indexes, or by using the key as a parameter in a function call. When a statement is written to a construct the demon may append it, insert it in a logically suitable place, ignore it as a duplicate of information it already has, increase a frequency count, or even initiate a restructuring process. In the other direction the demon does not need to know why it has been asked to provide information or which routine has requested it.

The important thing to realise is that there is a complete separation of responsibility between the main processing routines and tne construct demons. This approach is essential when one is handling open-ended problems, where the structure of the information to be manipulated is not known in advance. This is in marked contrast with conventional programming languages, where much of the 'action' assumes a knowledge of the structure of the information being processed.

A variety of construct types has been used in CODIL and/or MicroCODIL, from data compressed formats in the computer memory up to physically indexed clear-text files. The order of the statements may dynamically change using an ordering algorithm in the construct demon, and some interesting learning models have been tried. In each case the demon protects the decision-making unit from 'seeing' the storage structure directly.

7. MEMORIES

Because the current implementations do not support parallel processing, the hierarchy of memories of the theoretical model becomes, for all practical purposes, a stack. This can be considered as a zoned 'last in first out' list of items. The physical organisation of the stack is a matter for the memories demon. If a request for matching items is not satisfied in the current memory zone it can continue the search in the next higher zone. When items are found it returns a list of pointers to those items. From the point of view of the user the main zones are as follows.

System functions. If all else fails, and the domain name corresponds to a system function, the logical address of that function is returned. The available functions are predefined when the software is loaded.

Constructs. The highest level dynamic memory consists of all the constructs within the knowledge base. When this memory is searched the type and logical address of the construct is returned.

Item properties. These items represent the properties associated with a domain name. For instance items such as UNIVERSITY (ISA) ORGANISATION would be held at this level.

Global items. These are a list of user defined items which are permanent in that, once defined, they cannot be deleted.

The facts. This is the main working memory, and is notionally equivalent to human short-term memory. It is organised as an associatively addressed list of items which are 'garbage collected' on a last in – first out basis. By subtle modification of the addressing rules one can get a range of sophisticated behaviours, including its use as a recursive stack with local and global variables as required.

Work item. In describing the item structure (Section 5.2) it was indicated that it was occasionally necessary to look into a conceptually lower level to get information on, for example, the length of an item value. There is therefore a 'dummy item' available at the bottom of the facts which is used to hold system generated items.

In addition to the main memory stack, described above, there are several other memories which carry out specific roles. The most important of these is represented by the 'TRACE' window in MicroCODIL. This contains copies of the items that have been recursively examined to reach the current context. This window provides an answer to the question 'How did I get here'. Another example of a supplementary memory would be the file format descriptions associated with reading conventionally formatted BASIC files into MicroCODIL.

8. THE DECISION-MAKING UNIT (DMU)

8.1 The basic approach compared

As seen by the user, the Decision Making Unit is deliberately very simple. The basic process is to take two lists of items and compare/combine them as far as this is possible. If the system compares

```
1 PRODUCTCODE = 253,
2   QUANTITY > = 1000,
3     QUANTITY <5000,
4       UNITPRICE = 45.
```

as 'criteria' with

```
1 CUSTOMERNO = 12345,
2   PRODUCTCODE = 253,
3     QUANTITY = 2500.
```

in the 'facts' memory it needs little imagination to see that the facts will be changed to:

```
1 CUSTOMERNO = 12345,
2   PRODUCTCODE = 253,
3     QUANTITY = 2500,
4       UNITPRICE = 45.
```

This operation can be compared with a simple production rule:

```
PRODUCTCODE = 253;    QUANTITY > = 1000;
  QUANTITY < 5000;  ——→  UNITPRICE = 45
```

or a COBOL statement of the kind:

```
IF PRODUCTCODE = 253 AND QUANTITY > =
  1000 AND QUANTITY < 5000 MOVE 45 TO
  UNITPRICE.
```

If we ignore the limitations of the Relational Data Base in handling ranges the operation can also be considered as a natural join between tuples selected from two relations with the structures:

```
PRICELIST (PRODUCTCODE, QUANTITY, UNIT-
  PRICE) 253  1000–5000  45
ORDER   (CUSTOMERNO,   PRODUCTCODE,
  QUANTITY) 12345  253  2500
```

Such analogies must be handled with care. For instance, a COBOL programmer would rarely write a statement along the lines of the example given above. Instead he would write a more general statement such as:

```
IF PRODCODE = PRODUCTCODE AND QUAN-
  TITY > = MINQUANT AND QUANTITY < MAX-
  QUANT  MOVE  UPRICE  TO  UNIT-
  PRICE.
```

CODIL can be used in the same way, but could be significantly slower. This is because procedural languages are optimised towards doing repetitive tasks, while CODIL is optimised towards dynamic flexibility.

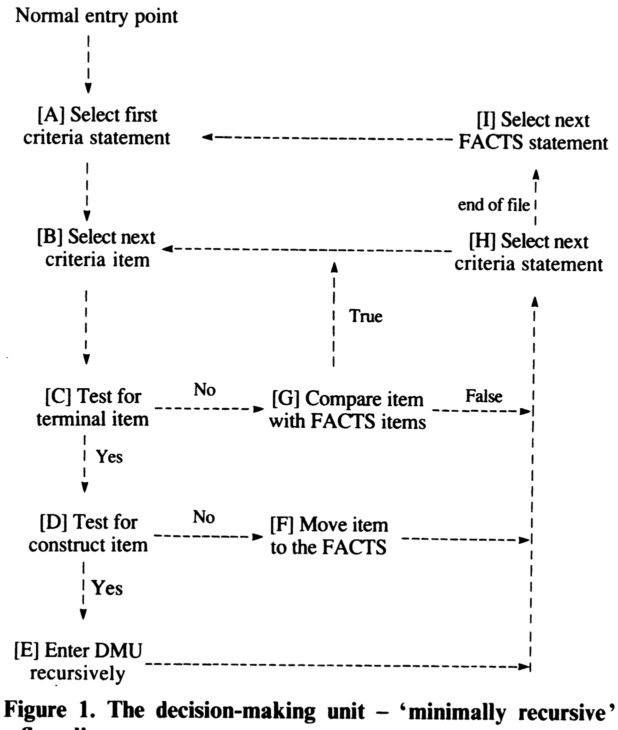
From these simple examples it is possible to make two important observations, as follows.

In CODIL there are no syntax distinctions between the two statements being compared. In this, it is similar to the relational data base, and more general than production rules or procedural languages such as COBOL. The CODIL user is, of course, free to use some files only as 'program' and some only as 'data' if this is appropriate to his application.

In CODIL there is no requirement for the statements to correspond to any semantic predefinitions. In this it is more general than the relational data base or procedural language approach. The CODIL user is, of course, free to create constructs which conform to predefined semantic rules if this is appropriate to his application.

8.2 A 'minimally recursive' DMU

At the heart of any CODIL interpreter is a 'Decision Making Unit' or DMU which controls the comparison of lists of items. This reads 'criteria' items from the keyboard or the knowledge base, compares them with the current contents of the facts, and initiates actions as a result. The overall process is very simple and a non-recursive flow diagram is shown in Fig. 1.



Normal entry point
[A] Select first criteria statement
[I] Select next FACTS statement
end of file
[B] Select next criteria item
[H] Select next criteria statement
True
[C] Test for terminal item
No
[G] Compare item with FACTS items
False
Yes
[D] Test for construct item
No
[F] Move item to the FACTS
Yes
[E] Enter DMU recursively

Figure 1. The decision-making unit – 'minimally recursive' flow diagram.

The DMU is normally entered at the point [A] with items coming from the keyboard or a construct in the knowledge base. The first statement is selected, and then the first item within that statement [B]. This item is expanded into its component parts, inserting default values as appropriate. If the item is at the end of a statement it is deemed to be true by definition [C]. If the item is a construct [D] the DMU is re-entered recursively [E] using the construct as a source of further criteria items [A]. Otherwise the item is added to the Facts [F]. In either case the next criteria item comes from the next statement [H].

For a non-terminal item a comparison is made with the items in the Facts [G]. If the comparison is true the next criteria item is selected [B]. Otherwise the first criteria item from the next statement is selected [H].

When the end of the criteria construct is reached the DMU simply returns back to a higher level at [A]. However if the entry item at [A] involved a pair of constructs in the form 'construct1 = construct2' the DMU was entered at [I] and in this case the facts are updated with the next statement from construct2, and the DMU is re-entered using construct1.

If the item being compared at [G] is a construct (i.e. 'construct?' or 'construct1 = construct2?') the DMU is re-entered but processing takes a slightly different route. All items selected at [B] are compared at [G], and if a terminal item is found to be true the process terminates and a 'True' is passed back to the higher level.

The above description is quite deliberately vague on a number of points. For instance [H] selects the next criteria statement, but it is not explained how this is done. This ambiguity is quite deliberate for two reasons. First of all the 'next' statement on the construct being used is decided by the appropriate construct demon – and hence it is not the concern of the decision-making unit. Secondly the decision-making unit may impose additional constraints. It is not proposed to discuss these in detail in this paper but three main approaches have been tried.

(*a*) The next statement is the one which sequentially follows the previous one if you list the construct. This gives a 'rule' order equivalent to that used in procedural languages.

(*b*) Only one 'rule' has been selected successfully. Once this has happened the logical end of the construct is reached.

(*c*) The first 'new rule' is selected. The search for the next 'new rule' starts at the beginning of the construct. The process continues until there are no new rules. This gives a 'rule' order equivalent to that used in production rule based systems.

This means the DMU can operate either as if it were processing a sequential set of rules, or a set of production rules, or various combinations of the two.

8.3 The item comparison process

When a criteria item is selected the memory demons are asked for a list of items drawn from the same domain. At the lowest level the value of each item in the list is compared with the criteria item, while at a higher level the results of the comparisons are correlated and a final probability is returned. It is easiest to discuss the comparison process in 'bottom up' order.

Matching an item list. The comparison process starts with the DMU asking the memory demon for a list of items which match the item name of the current criteria item, either directly or through the ISA hierarchy. Given this list the DMU first checks to see if there is a relevant qualifier, for example (NUMBER), which can be satisfied immediately. (In this case the NUMBER function replaces the list with a single entry referring to a temporary item containing the number of items in the original list.)

The DMU then scans the list comparing each item in turn with the criteria item, using the process described above. If any comparison is true the DMU treats the criteria item as true, and in the same way one false item on the list makes the criteria item false. If the end of the list is reached and a true? item has been found the criteria item is treated as true. This would happen, for example, if you compared AGE = 30 as criteria with a range of AGE > = 20; AGE < 40 in the facts memory. If the best result is 'undefined' the criteria item is false.

If the item list is empty individual item comparisons cannot take place, and the result is false, unless the criteria item explicitly refers to the null set.

Comparing individual items. In MicroCODIL all item values are held as character strings, but can be treated as either real numbers or integers if they correspond to an appropriate format. When comparing values (after evaluating expressions if relevant) a numeric comparison is performed whenever meaningful. Case distinctions are ignored. Additional comparison functions are provided for partial string matching and approximate matching (near miss numbers or mis-spellings). Further such facilities can be added in a modular fashion, and might, for example, carry out phonetic matching, compare dates, or map words such as 'seven' with numbers in digital form.

When comparing items it should be realised that a comparison between AGE > 30 and AGE < 20 should be treated as false because there is no possible overlapping value. Using a simple set-theoretic approach it is possible to define a simple table look-up routine to handle all possible combinations.

Conditional constructs and the fuzzy interface. If a direct comparison between the criteria item and the contents of the current memory is false, the DMU can make two further attempts to try and get a true response. If the criteria item corresponds to a construct the item is recursively passed to the DMU in a conditional mode, returning true if any single statement is true. Alternatively an item qualified by a construct is processed in a similar manner – with the construct being indexed by the criteria item value. As this approach is recursive it provides a flexible fuzzy interface.

Probabilities. The above discussion assumes that probabilities are not being used (or more correctly that all probabilities are unity). If probabilities are used the DMU looks for the most probable result. If the found probability exceeds a given threshold the item is true (with the found probability being carried forward), otherwise it is false.

8.4 Applying items to the facts

In Fig. 1 items are shown as being 'applied' to the Facts. This term is used because items will fall into three groups. The first group will change the contents of the Facts Memory. The second group will change the contents of other memories (or the names table, see later) and the third group will be involved in output, which has no direct effect on the main memory structure. Only the first situation will be described here.

When an item is moved to a memory, and there is only a single item matching by name, the existing item is overwritten. This is equivalent to the action taken in language systems that only accept single-value items. In practice it is very important to keep this as the default, and the system will only use multiple-item values if they are specifically requested. If there are no matching items, or multiple values are being used, the item is appended to the end of the memory.

8.5 The fully recursive structure

In reality the power of CODIL is a direct result of the way in which the modules which make up the DMU call each other in a highly recursive network. These modules are all no more than a few lines long and fall into the following categories.

Basic sequence control. This module is concerned with requesting the next item from the appropriate demon and deciding which other modules to call. The module either iterates when the item is false (for instance when getting the next item from the keyboard), calls itself recursively if the item is true, or calls other control modules (which may again call it back at a deeper level of recursion).

Processing. One module controls the 'comparison' process and another controls 'actions'. Normally their activities are self-contained, but they can recursively re-enter the DMU if the items being processed have the form *construct*, *construct1* = *construct2* or *item* (*construct*)...

Construct selection and repeating processes. Further modules are used to select constructs and to control any repeating processes. (This is equivalent to looping and reading the next record in a conventional procedural language.)

System functions. One module controls all calls to system functions, wherever they originate. Some of these functions, such as those concerned with back-tracking, will re-enter the DMU at a deeper level.

9. THE SYSTEM FUNCTIONS

An item name can correspond to a predefined system function. Normally an item with a system name will be treated in an identical manner to any other item, but in certain contexts, defined below, it will be used to trigger a call to a precoded function. Such 'system items' can be said to expand into some kind of 'activity' rather than into smaller items. The relevant contexts are identified as follows.

The value context (*val*). This is the simplest context in that it is activated whenever the system requests a current value for a named item. If the memory demon does not find a matching item the relevant 'val' system function is used to generate a temporary item with an appropriate value.

The command context (*cmd*). This context is used to modify system parameters, and also to select display windows. The only normal system word that corresponds to a conventional explicit command in a procedural language is EXIT.

The compare context (*cmp*). This context is entered when two item values are being compared, and the criteria item has a named qualifier such as (APPROX) or (LENGTH). Such system functions carry out non-standard comparisons and indicate whether the values are to be considered equal, and with what probability.

The parameter context (*par*). This is also activated when items are compared, but before they are fully decomposed into their component parts. Thus (NUMBER) is used to count the number of items with matching names, while (LAST) restricts the system to looking at the last item (if any) which matches the current items name.

The expression context (*exp*). This is similar to the value context except that it returns a property associated with the item name.

The modifier context (*mod*). The kinds of activities triggered by such items are very varied and a few examples are appropriate. The first group correspond to the 'exp' functions and are used to define properties such as ISA. Two are extremely important. These are (ACTION) and (CONDITION) and they are used to define default qualifiers. For instance the item MELTINGPOINT (CONDITION) = APPROX ensures that all unqualified comparisons involving MELTINGPOINT are carried out using the approximate mapping function identified by (APPROX). Such (ACTION) and (CONDITION) properties are transmitted along the ISA hierarchical structure, and so can be made to apply globally.

Other 'mod' functions superficially resemble conventional procedural commands, and it is instructive to see why an apparently unusual format is used. For example CURRENTORDERS (LIST) gives the construct the property of being visible on the display unit or printer, while CURRENTORDERS (DELETE) will give the construct CURRENTORDERS the property of being deleted! The explanation is that in CODIL the emphasis is on the information being processed (which happens to have properties) while in command oriented languages the emphasis is on operations (which happen to require something to operate on).

The after context (*aft*). The after context occurs after an item has been processed, but before the next item is selected. It is particularly relevant for diagnostic purposes, fuzzy matching and processing demons.

Further contexts are possible – particularly relating to input and output but have not been implemented in either CODIL or MicroCODIL.

10. THE NAMES TABLE

An important implementation feature of both CODIL and MicroCODIL is the name table. This can be considered as a kind of data dictionary which contains all the domain names known to the system, together with any associated properties and it is organised so that the software can rapidly access the information it contains. For system names the table contains information on the contexts in which the name is active; for constructs it contains information about where and how it is stored; for other items it may contain information on properties and defaults associated with the domain name. The ISA pointers are extremely important as the software automatically follows up the ISA chains in order to find the current default properties for the current item.

Conceptually this information can be considered to be held in high level memories, and is so described earlier in this paper. However, the 'data dictionary' approach means that all properties associated with domain names are held globally. In MicroCODIL the advantages of compactness and speed greatly outweigh any disadvantage from not having 'local' property definitions.

For future systems, implemented on more powerful machines, there seems to be no reason why properties should not be held in any memory, and be subject to a 'last in first out' regime.

11. CONCLUSIONS

This paper has described the conceptual background to CODIL in information processing terms, and has shown how the ideas have been implemented in CODIL and MicroCODIL. In doing so, many novel or unusual features of the approach have had to be passed over, and it is proposed to describe them in later papers. For instance there is much more that can be said about the handling of constructs and the construct demons; the subtleties of sequence control within the decision-making routines have only been hinted at; and the 'global/local' aspects of the CODIL memories exhibit some very distinctive features. In addition a paper is planned which looks at CODIL in human factor terms, and more are planned describing a wide range of applications.

More work is necessary on the theoretical aspects of CODIL. At the most general level there are no constraints on the logical consistency of the information being processed. This 'freedom' to be ambiguous or inconsistent is essential if the user is to model his incomplete view of poorly structured information. On the other hand the user may impose logical constraints at a level appropriate to his task, and, given appropriate demons, select storage structures which provide the appropriate support. It is quite clear that the recursive item model cannot be adequately described by the more normal theoretical approaches to programming languages and data bases, and it is believed that some new mathematical tools may have to be developed.

REFERENCES

1. E. F. Codd, A relational model of data for large shared data banks. *Comm. ACM* **13**, 377–387 (1970).
2. International Computers Ltd. UK Patent 1,265,006 (1968).
3. D. Omrani, Some studies of the relational data base and the CODIL language. *Ph.D. Thesis*, Brunel University (1979).
4. C. F. Reynolds, CODIL: The importance of flexibility. *The Computer Journal* **14**, 217–220 (1971*a*).
5. C. F. Reynolds, CODIL: The CODIL language and its interpreter. *The Computer Journal* **14**, 327–332 (1971*b*).
6. C. F. Reynolds, An evolutionary approach to artificial intelligence. *Proceedings, Datafair '73* 314–320 (1973).

7. C. F. Reynolds, A data base system for the individual research worker. *Proceedings, International Symposium, Technology of Selective Dissemination of Information* 1–8 (1976).
8. C. F. Reynolds, G. Sutton and M. Shackel, Using CODIL to handle poorly structured information. *Proceedings, Medical Informatics Europe*, 465–474 (1978*a*).
9. C. F. Reynolds, *The Design and Use of a Computer Language based on Production System Principles.* Brunel University, Technical Report CSTR/15 (1978*b*).
10. C. F. Reynolds, A psychological approach to language design. *Proceedings, Workshop on Computer Skills and Adaptive Systems*, 77–87 (1978*c*).
11. C. F. Reynolds, CODIL as an information processing language for university use. *IUCC Bulletin* **3**, 56–58 (1981).
12. C. F. Reynolds, A software package for electronic journals. *7th International Online Information Meeting*, 111–118 (1983).
13. C. F. Reynolds, MicroCODIL as an information technology teaching tool. *University Computing* **6**, 71–75 (1984).
14. C. F. Reynolds, A microcomputer package for demonstrating information processing concepts. *Journal of Microcomputer Applications* **8**, 1–14 (1985).
15. C. F. Reynolds, *MicroCODIL Manual* (and software). CODIL Language Systems, 33 Buckingham Road, Tring, Herts, UK (1986).
16. C. F. Reynolds, Human factors in systems design: a case study. In *People and Computers III*, edited D. Diaper and R. Winder. Cambridge University Press (1987).
17. C. F. Reynolds, Introducing expert systems to pupils. *Journal of Computer Assisted Learning* (in press) (1988).

Wilkes Award

The Society is pleased to announce that the Wilkes Award for 1988 has been won by William Roberts for his paper 'A formal specification of the QMC message system' which was published in the August 1988 issue (Vol. 31, No. 4) of *The Computer Journal*. Each year an Award Panel considers all those papers, published in the previous year, where one or more of the authors was under thirty years of age at the time of submission. The award is then made to the author(s), provided they satisfy the age criterion, of the best paper in that category.

The paper addresses two problems of communication, that of the human communicating with the computer and that of two (or more) humans using a computer as the medium through which to communicate with each other. The problem with the first kind of communication is that a computer will not perform any task unless the task has first been analysed thoroughly and described clearly and then the description is communicated unambiguously to the computer. The problem with the second kind of communication is that, having created a physical link, the user must be provided with a facility whereby he can understand how to use the link without having to search a plethora of incomprehensible manuals. This is called generating a 'user friendly' interface.

The objective of the paper is to create an algebra for describing a computer system formally and then to use this algebra to describe the 'Message System' (a computer-based mail and notice board facility) developed at Queen Mary College, London. From the paper a reader can learn how to specify formally a computer system and, at the same time, how to design an easily used local 'Message' facility.

From an early start in computing at age eleven, William Roberts graduated in Mathematics from Exeter College, Oxford in 1982. At the same time as studying for his degree, he was also deeply involved with a consultancy who were developing commercial microcomputer applications. In 1984 he moved to Queen Mary College, London where he first gained an MSc (with distinction) in Computer Science and then joined a small, Alvey-funded research group. This research led to the award-winning paper, and at the end of funding he stayed at QMC with responsibility for developing the Computer Science Network.

Mr Roberts was one of twenty seven authors who were considered for the award. His paper was chosen because of the way in which it shows so clearly how a new and developing technique from computer science can be used in a new and developing computer application. The Wilkes Award, which consists of a silver gilt medallion, was instituted by The British Computer Society to mark the retirement of Professor M. V. Wilkes as Professor of Computing Technology at the University of Cambridge, in recognition of his pioneering work in both computer hardware and software and his unstinting efforts on behalf of the Society.

Announcement

23–25 July 1990

UNIVERSITY OF LEICESTER, UK

International Workshop on Semantics for Concurrency

The International BCS-FACS Workshop on Semantics for Concurrency will take place on 23–25 July at the University of Leicester, Leicester, UK, in the week following ICALP 90 to be held at the University of Warwick, Coventry, UK. Leicester is conveniently positioned in central England, and is within easy reach of Coventry. Those wishing to stay on after ICALP may obtain accommodation for the days preceding the workshop at very reasonable prices. During this time, there will be an opportunity to join in a programme of entertainment. The preliminary announcement for the workshop has met with a very encouraging response.

Semantics of concurrent systems is one of the most vigorous areas of theoretical computer science, but suffers from disagreement due to different, and often incompatible, attitudes towards abstracting non-sequential behaviour. When confronted with process algebras, which give rise to very elegant, highly abstract and compositional models, traditionally based on the interleaving abstraction, some argue that the wealth of contribution they have made is partially offset by the difficulty in dealing with topics such as fairness. On the other hand, the non-interleaving approaches, based on causality, although easing problems such as fairness and confusion, still lack structure, compositionality, and the elegance of their interleaving counterparts. Since both these approaches have undoubtedly provided important contributions towards understanding concurrent systems, the workshop will concentrate on what they have in common, rather than the way they differ.

The workshop will incorporate a number of tutorials, devoted to invited talks concentrating on giving an overview of major approaches to concurrency. The invited speakers will include: Prof. Robin Milner (Edinburgh), Prof. Eike Best (Hildesheim), Prof. Antoni Mazurkiewicz (PAS).

Topics will include (the list is not exhaustive): mathematical models and notations for concurrency including categorical and topological methods; non-interleaving and partial-order semantics; distributed computation; process calculae; behavioural equivalences; behavioural properties of concurrent systems including fairness; logics for concurrency; real-time systems.

Organising Committee: Marta Kwiatkowska, Mike Shields, Rick Thomas.

Address for information:

Dr. Marta Kwiatkowska, Workshop on Semantics for Concurrency, Department of Computing Studies, University of Leicester, Leicester LE1 7RH, UK. Tel: +44-533-523603. e-mailJANET: mzk@uk.ac.le.