A Data Dictionary for Learning Data Analysis

R.N. MADDISON* AND A.J. GAWRONSKI**

*Mathematics Faculty and **Academic Computing Service, The Open University, Walton Hall, Milton Keynes, MK7 6AA*

A data dictionary system, DADICS, was conceived and developed as an online activity for students of an OU data analysis course. The system holds a highly structured knowledge base of facts about a number of conceptual data models, corresponding modelling rules, and the user's session state. These all form a semantic network of sentences, each with a subject, verb, and one or several objects. By simple commands the user may: choose a model representation – eg entities, attributes and relationships; select a local or global model; and list as sentences the relevant facts about any element. Alternatively he may update a model, insert facts as discovered, and command the system to highlight inconsistencies. The implementation is a relatively short general-purpose Pascal program that interprets the stored facts to determine what is required.

1. STUDENTS' AIMS AND OBJECTIVES

Many people and organisations have information that they may wish to hold, manage and manipulate using a computer system. Some people in industry, commerce and government need education and training in the techniques of data analysis developed in recent years. So, as part of its Continuing Education programme, The Open University decided to create a 40-50 hour self-study course on Data Analysis For Information System Design. It will be available from about May 1983 to anyone over 21 living in the UK who wishes to study it.

The course is intended for those who want a grounding in the principles and skills of data analysis; those who wish to learn to analyse, model and manage data; and those managers who may move into areas where the techniques are relevant and who need to appreciate the problems, either to manage users or creators of database systems or to specify application requirements.

Students will learn:

- why and how to analyse the structure of information and thus develop local and global conceptual data models as the first stage of design; various strategies combining data and functional analysis are covered;
- why and how to normalize;
- various ways of representing information structure – e.g. as entities, attributes and relationships or as normalized aggregates;
- how to transform between the various model representations;
- how to choose appropriately between the different techniques, including considering factors affecting practical applications and the use of a data dictionary.

After the course students should be able to apply, analyse, evaluate and synthesise these generalisations and principles.

The course materials include:

- Course Text, Activity Booklet, Standard Telephones and Cables (STC) case study, and Study Guide;
- two 24-minute video cassettes on loan for three months;
- an audio cassette;
- online access for up to five hours to any of the main computer systems of the OU Academic Computing Service.

There are two activities in which students will learn by doing online practical work. The first is solely concerned with normalization. For the second, which is far bigger, we have provided a new data dictionary system, DADICS. This holds the data models for the STC case study and two of the other contexts that students will have met in their earlier study of the course. The structured exercises using DADICS bring in all the main concepts in the course. In their first few exercises students will be guided through retrievals and the various model representations. Then they will progress to more open-ended exercises until eventually they should be able to analyse stated requirements, create their own model, store it in a dictionary starting from scratch, and check its consistency.

The main concepts include: functional dependencies, local and global conceptual data models, entities, attributes, relationships, aggregates, pure entity types in 3NF, identifiers, candidate identifiers, synonyms, homonyms, normalization, transforming between model representations, various types of retrieval from and the updating of a data dictionary, consistency and completeness checks, and various ways in which all these can be combined.

2. OUR OBJECTIVES

Thus in summary our task was to design and implement a system to provide online practical work for each student to reinforce understanding of all these concepts and techniques.

No software package that could form the basis of satisfying these requirements existed. It was necessary to start from scratch. To keep down the cost of course development the DADICS design and development costs had to be kept to a minimum, ie within three months work by each of two people. This constrained the design and features, yet essentially was achieved.

After itemizing in far greater detail than above both our and the students' objectives we had three main tasks to construct the new practical activity:

- to create the rules and details of the main example contexts round which the students' exercises would later be formed;
- to specify, design and implement the software;
- to create and test the students' exercises and write the descriptive text.

Our main purpose here is to describe the software. In practice we iteratively refined the software design and development as the three tasks interacted.

3. NON-DBMS METHOD

To base the design around an existing database management system, eg the Codasyl-based DBMS or the relational (RDBMS) software available on the Open University Academic Computing Service DEC system computers initially seemed 'obvious'. But it was quickly realised that any such data dictionary would very quickly become highly inflexible, would not meet the detailed teaching objectives, would take too long to implement and cost too much.

For example in detail as shown in Figure 1 we wanted a student to be able to choose an example context, eg STC, being one of several with which he or she was already familiar, and change easily between viewing its global conceptual data model and any local model, eg Bill of materials. We wanted students to view any such model as any one of:

- entity, attribute and relationship types (EAR) – without duplicated attributes;
- pure entity types (PET), ie in third normal form – where attributes may be duplicated in different entity types and no relationship types exist;
- aggregates – where attributes are grouped according to what they are fully functionally dependent on, identifying attributes may be duplicated, and then relationship types are added.

```
READY!
dadics


Welcome to DADICS 1B(1)-5,


Please choose context from:
   Nil,
   STC,
   Tv-Rental,
   University,
   Erroneous Tv-Rental,
by typing N, S, T, U or E > STC chosen.



> Select: Bill-of-materials


> Model: EAR


> List: entity-type
Material-item
Method
Component


> List: Material-item
Material-item  has-unrel-ident Product-code
               has-unrel-fun-d Stocking-unit, (in-Bill-of-m), (in-Routing-TD)
                               Description, (in-Bill-of-m), (in-Routing-TD)
                               Status
                               Product-group
               belongs-to      Bill-of-materials
                               Routing
                               Routing-Transmission-Div
                               Routing-TXE4
                               Costing
               has-descript-at Description, (in-Bill-of-m), (in-Routing-TD)
               has-cardinality 120000
               indicates       a product used or made
               was-chckd-by-on B.McAlister, 11Feb82
How-procured   has-owner       Material-item
Where-used     has-owner       Material-item


> Help: Stocking-unit
Sto  abbreviates  Stocking-unit
Stocking-unit  is-of-type  attribute-type
Stocking-unit  indicates  e.g. kilograms metres hundreds


> EXIT
Do you want to finish off (Y/N): Yes
If you've made any changes or defined your own dictionary,
would you like to retain this for another terminal session (Y/N): No
```

Figure 1. Brief example student online dialogue

We also wanted a student to be able to store in any order all the items of information that an analyst would have, like the order that an analyst might receive or deduce them; and be able to command the system to list and explain every inconsistency. Such inconsistencies and incompleteness should indicate areas of the conceptual data model needing further work. It was relatively easy, though time consuming, to define the various rules describing the information content as this is highly structured.

We did not expect students to have met Codasyl or relational database manipulations, or Codasyl schemas; and we did not want them to be sidetracked by or have to learn any such details. So if such features were present underlying the implementation, for ease of use such features would have to be hidden completely from the student users.

We felt that to meet the requirements by developing an application program that used an existing database package would be impossible within the person months available for analysis and programming.

4. SEMANTIC NETWORK

Instead we developed a relatively short Pascal program that does not use an existing DBMS. It has a collection of procedures that input, display and interpret the user's commands. These call other procedures that manipulate a semantic network of stored facts. The database of facts include the details of the local and global conceptual data models for the chosen context, eg STC. The facts also include the rules of the model representations, eg rules about entities, attributes, relationships, consistency and completeness. The facts also hold the user's session state, eg that the latest Model command requested EAR

Program procedures (in Pascal) – each calls lower procedures.	**Data structure envisaged**
Execute a command, eg Select, List.	Semantic network of logical sentences, with subject, verb, objects.
Perform a procedure on sentences; eg for all sentences matching a high level pattern do a procedure.	Semantic network of short sentences, with subject, verb, object.
Perform a procedure on a short sentence; eg insert, delete, check if present.	Record structure of elements, with pointer chains, forming the dictionary.
Perform a low level procedure on an element, eg insert, lookup, match an incomplete word.	Element with token of characters, being a word, term or name.

Figure 2. Programmer's view during development of DADICS

representation, and that the latest Select command selected Bill of materials. The facts are to be used in many different ways.

All this knowledge is held in the same form, as sentences with a simple syntax. Each sentence has one subject, one verb, and one or more objects. Each subject, verb and object is an element, ie a token which is a string of characters not including space or comma and representing a word, term or name. The elements on their own each have no meaning. Example tokens are: STC, EAR, entity-type, Select, Bill-of-materials, Material-item, Bom, et, Mati. We shall describe the semantic network and the commands in turn. Figure 2 shows the structure of the entire system.

The semantic network enables facts to be viewed in a natural and straightforward way, as sentences. To the student user each sentence consists of subject, verb and one or more objects. Each subject, verb and object is a word, term or name. In this logical view each sentence represents a fact. Facts can be inserted and removed in any order. Consistency and completeness are not required at each update. Indeed an analyst may work with local models that seem inconsistent. Although each sentence individually makes sense, part of the meaning and structure arises because the same word, term or name occurs in several different sentences.

A command, Describe, is available to retrieve all sentences containing a given word. More useful, however, is the List command, which uses other sentences as rules to decide whether or not a particular sentence should be displayed, and if so then how. The output from the simple command 'List Material-item' thus depends through the semantics of related sentences on the model representation – eg EAR, and the selected local model – eg Bill-of-materials. This makes the list command easy to understand and use by the student. Behind the scenes, however, it needs efficient pattern matching procedures.

The *physical storage structure* is chosen for efficiency. This structure is *not* seen by students. The words, terms and names collectively form the elements. Each record represents one element. Its data is: a token which is the lexical character string for printing that element; together with the pointers to link that node to others so as to represent the semantic network.

The semantic network chains represent directly all sentences that are subject, verb and single object, called 'short' sentences. Most logical sentences are short. Logical sentences with two or more objects are represented similarly. The higher level pattern matching procedures deal with long to short sentence conversion on the fly. Retrieval of all short sentences that involve a given element in a given way, eg as subject, involves only direct pointers, with no searching or testing. The pointer chasing turns out to be very efficient. Most of these short sentences will be a full sentence. Each other such short sentence found will correspond to one long sentence that can be reconstructed similarly. In practice the programming requirement is not so often to retrieve each sentence meeting a criterion, but more frequently to execute a named procedure on each such sentence, and the middle level procedures provide this directly. All elements and all sentences are distinct. This is ensured by the insertion procedures so it can be assumed by the retrieval procedures.

5. THE COMMAND INTERPRETER

As each student works alone, and may be an inexperienced or casual user of computer systems, the system must be friendly, helpful and easy to use. We wanted minimum student time spent on preparatory reading before the first online session. So the system is command driven, a technique we have found very successful for many other student packages. Each keyed command gives an immediate response. The commands are simple to learn, powerful and versatile. Several kinds of help are available, so that the documentation is hardly needed.

Key strokes are minimised. For example each command is invoked by a single key stroke for its initial letter. For all commands the system fills out the rest of the command word. It also prompts for a further word or sentence if the command needs it.

When keying as a word any element that exists already in the dictionary the student needs only to key enough characters to distinguish it – usually only two or three characters. The student may make the system attempt to recognise and complete the word by keying Escape.

All common words have abbreviations, usually of two to four letters – eg 'et' abbreviates entity-type. Students need these since some pairs of elements, eg Customer-no and Customer-name, have too many matching leading letters for quick easy distinct keying. 'Cuno' and 'Cunm' are easier.

At any time, ie when stuck, the student may key a question mark. This can be when a command is expected, or between words, or within a word. The system's response gives all options available at the current level; then repeats the incomplete line and allows the student to continue input.

The syntax is simple: a command word followed where appropriate by one or several elements. The only separators are space, comma and return. Any invalid character produces an audible bleep, it is not displayed, and the student can continue the line. Syntax is checked as the command line is keyed, not at the end of the line – except that word spelling is only checked at the end of the word or when Escape is keyed.

The combination of these features facilitates the implementation of a very simple command parser. Free format is not a problem and error recovery is quite straight forward.

Several forms of help are available. As stated above, keying a question mark will produce the available options, eg provide a list of the elements from the dictionary that match the current incomplete word.

A powerful Help command is provided. The H for Help may be used alone (followed by return) or followed by any element, eg by the name of any command. Help alone produces a list of the commands and control character functions. Help followed by a command name produces an explanation of that command. Help Model and Help Select also state the current model representation and what is currently selected, ie session state information.

Help followed by any other element produces text stating its abbreviated and full forms; its type; and free text indicating its meaning – provided such descriptions exist in the knowledge base of stored facts. If the word is undefined, ie not in the dictionary, then that is stated. If the word is a synonym, or a synonym for the reverse of some other word, then that is stated – eg owns is the reverse of has-owner. A chain of synonyms is followed recursively.

Using the simple List command, consisting of keying just L (for List:) followed by one element, many kinds of lists are available. These include, for example, the following, where the bold characters are keyed by the user.

- List the available conceptual data models (List: **cdm**).
- List details of a named conceptual data model (eg List: **Bom**).
- List the attribute types of the selected model (List: **at**).
- List its entity types (List: **at**).
- List its relationship types (List: **rt**).
- List functional dependencies, ie which attribute is functionally dependent on which others (List: **fd**).
- List synonyms (List: **syn**).
- List identifiers of each entity, or candidate identifiers of each, or functionally dependent attributes of each.
- List details of a named entity type, eg Material item (List: **Mati**).
- List details of a name relationship type.
- List details of a named attribute type.

Where appropriate, many of the above commands will produce different lists as output depending on:

- the context previously chosen, eg STC;
- the current model representation, eg EAR or PET or Aggregates; from the latest Model command;
- the element(s) currently selected. This can be any element, eg the name of a local model; or any combination of elements; or 'all'; from the latest Select command.

For example with model EAR and with a selected local or global conceptual data model such as Bill-of-materials in STC, when listing a named entity type, such as Material-item, the following types of information will appear:

- the identifying attributes (where these are unrelated to any other entity);
- functionally dependent attributes;
- descriptive attributes;
- optional attributes;
- which conceptual data model the entity belongs to;
- cardinality;
- free text description;
- the analyst who was responsible for the definition;
- the date of the analyst's definition;
- relationships that this entity is owner of and those it is a member of.

The execution is accomplished by interpreting appropriate stored facts to decide which others to display.

6. METALEVELS OF FACTS

These many kinds of lists and the consistency checking of incomplete stored models are all implemented by interpreting certain stored facts. The interpretation controls procedures that act on other stored facts. We designed the system so that we, as implementors, but not the students, think of the stored facts as forming several metalevels. For simple ease of use it is probably better that the student is unaware of the underlying structure, so the users can be completely unaware of what follows.

Each metalevel involves elements and facts that represent rules that describe classes of allowable elements and facts at the next lower level. Metalevel 0 means the representation of the real world universe of discourse and equivalent database item occurrences.

The student learning data analysis works at metalevel 1. For examples: Material-item is the name of an entity type; it has full identifier Part #, Issue # and Location #; these three are named attribute types.

Metalevel 2 includes facts such as: the term entity-type is-of-type modelling-concept; at metalevel 1 zero or more elements may be defined to be an entity-type – eg Material-item above. Metalevel 2 includes similar facts about other modelling concepts, eg the term attribute-type. It includes that the term has-full-identifier is a modelling verb; and that each entity type may have a full identifier consisting of one or more attribute types. Metalevel 2 includes many tens of similar facts. Each describes a valid way of use at metalevel 1 of one particular modelling verb, with appropriate particular modelling concepts. It also includes facts stating which modelling verbs belong to each model representation.

The metalevel 2 facts are sentences that can be both interpreted by the programmed procedures and used without change in help messages, explanatory error messages, and warning messages to student users.

Our research showed that whether or not a particular metalevel 1 fact belonged to a particular model representation could be arranged to depend only on the modelling verb – ie on the second word in the metalevel 1 sentence. The non-dependence on subject and on the objects gives an enormously advantageous simplification.

Metalevel 3 has just a few facts representing in exactly the same way rules that cover all metalevel 2 facts, and so on.

This structure was chosen for several reasons. First it allows many kinds of list to be produced from a short general Pascal procedure. Essentially it interprets facts at metalevels 2 and 3 to cause printing of the appropriate metalevel 1 facts in the appropriate sequence. Elements such as attribute-type and has-full-identifier do not appear in the procedure coding. Appendix 1 gives an example. Appendix 2 gives the formal rules.

Second it allows users to insert and to remove facts in any order. Most editor systems require the user either to key commands to indicate the place eg where an insertion is to be made, or to handle line numbers similarly. Since in DADICS no information is carried by the sequencing of (metalevel 1) facts, students do not have to learn any syntax or semantics of editing operations.

Although retrievals and insertions seem quite natural, there are associated problems. A user can key in nonsense such as contradictory facts. This was a deliberate design decision. For example while an analyst is creating a conceptual data model a name that starts as an attribute type may later become an entity type or a relationship type. The fact that it was an attribute type name can be removed as easily as it was inserted originally. This would not be possible with a data dictionary that insisted that at all times the incomplete stored facts were entirely consistent. To help maintain consistency of the data dictionary contents we have stored the meta rules and the meta meta rules. These can be used to validate the consistency of the contents. A command is available to apply the consistency rules either to a named element or to all. The checks include:

- that an element is a synonym or abbreviation for at most one other;
- that elements used in facts other than abbreviations or synonyms are defined to be of an appropriate type;
- that every fact conforms exactly to a meta-rule allowing it to exist, with every element in it being of an appropriate type;
- that in an EAR model no attribute appears in more than one entity.

Third, it allows consistency checking to be done fast. The command 'Apply rules to: all' checks each element in the dictionary in turn. While dealing with a particular element, all the facts with that element as subject are checked. Each fact is thus checked exactly once – when its subject element is checked. If the element is the subject in any sentence other than a synonym or abbreviation then it must be defined to be of some type. Each sentence in which that element is the subject must have its subject, verb and objects matching by types those of a higher metalevel rule that allows that sentence, else that sentence is erroneous.

This same checking algorithm applies to all elements and facts; it does not depend on the metalevel. When the student commands a consistency check of a single named element, its abbreviation (if any) is found, and synonyms (if any) are followed through to find the preferred terse form. The checking is then as above. If this element is the subject in any sentence then it must be defined to be of some type. Other sentences involving the element must conform to rules.

For most elements the first facts inserted with that element as subject give the abbreviation and type. Hence these frequently needed facts are quickly found, later insertions having been made at the end of the semantic network chains.

Hence the entire dictionary can be checked with an acceptable response time. When the OU ACS computer is heavily loaded the response is about ten seconds for checking all the STC information, which has about 1200 facts involving 500 elements. During development of DADICS we used this procedure with considerable success both to check our example context models and to check our higher metalevel rules.

Fourth, the procedure during a list command is simple, fast and straightforward. The procedure is essentially to find and list the relevant appropriate metalevel 1 facts that fit higher metalevel rules. Our algorithms and semantic network pointers avoid any searching.

Fifth, if during the execution of a List command some appropriate allowable facts are not found then the student's model may be incomplete. This gives quite a different kind of check from the Apply rules command. Whereas the apply command looked at each metalevel 1 sentence and checked whether it fitted a metalevel 2 rule; the list command takes each metalevel 2 rule and finds the metalevel 1 sentence(s) that fit it. For each metalevel 2 rule that says something should have or may have corresponding metalevel 1 facts, and for which no such facts exist, the student's model should have or may have further facts added. This procedure can thus produce warnings of all the possible types of missing yet allowable metalevel 1 facts. Each omission can optionally produce a warning message, part of whose text will come from the relevant higher metalevel fact. Using the Warnings command the student can opt for such warnings to be either suppressed or printed. Data dictionaries usually hold details such as analysts' names, dates and information sources. We did not want warnings to students about omissions of such facts, since they are not relevant to the teaching objectives and might be frustrating. So we have arranged that unimportant warnings are always suppressed. This is achieved by the interpretation of further stored facts – eg 'information-source-is belongs-to no-warning-verb'.

Sixth, having all the metalevel facts stored in the same way made our testing and development easier. We could insert and remove facts representing rules easily during a test run. Source coding changes and recompiling were rare. As a consequence the project never became bogged down in program or data testing or system development because the system could tell us immediately of any gaps or inconsistencies.

The student is completely unaware of the existence of meta-facts and rules and of any implementation details. The sentences do not form a natural language or very high level data description language, though at first sight they may appear to do so. The only structure embedded in the Pascal program is that a sentence consists of elements as subject, verb and objects. There are only a small number of element names referred to in the source coding of the Pascal program. These include the names of the commands and a few elements such as modelling-concept, may-have, is-of-type, belongs-to. The program is the coding of the procedures that give the real meaning of the sentences involving these reserved element terms. This gives extensive flexibility from an operational point of view. For example, a new kind of conceptual data model representation could be added without alteration to the Pascal program.

Each fact does not make reference to any other fact. Each fact may be thought of as a meaningful association between particular elements – the class of meaning being carried by the verb element. This essentially allows the user to store almost anything.

Since the rules are logically separate from the program, the system is general purpose – suitable for a number of different application areas. In principle it can be extended to cover schemas, conceptual process models, programs and the interrelationships of all these. In principle each collection of sentences that has a particular verb can be thought of as a relation with the subject, verb and object as items which together form the useful identifier.

7. QUALIFIERS

In STC there are several local data models. These are not fully consistent with each other. For example in Bill-of-materials the entity type Material-item is identified by Product-code. In TXE4-routing there are two entity types, Part-issue and Material-item. Part-issue has identifier Part-no and Issue-no; and amongst its other attributes some of the dependent attributes of the Material-item of Bill-of-materials. The TXE4-routing Material-item has as its identifier Part-no, Issue-no and Location-no; and amongst its other attributes some of the other attributes of the Bill-of-materials Material-item.

Some of the differences can be resolved because they are synonyms. But others cannot. In developing a global conceptual data model some discrepancies can be resolved by adopting a more general structure. But to correctly store all the STC local views needs qualifiers on certain facts, such as on what is the identifier of Material-item. Qualifiers are our way of representing time, manner and place phrases of ordinary english sentences.

A *qualifier* is an element for which there is a stored fact that says it is of type qualifier-type. Any fact that includes a qualifier among its objects is regarded as only true within the situation in which that qualifier is true. For example the qualifier '(in-Bom)' is arranged to be true within Bill-of-materials, by the existence of the fact '(in-Bom) belongs-to Bill-of-materials'.

8. SELECTION

The student can select a particular local conceptual data model – eg Bill-of-materials. Alternatively he can select any main element, e.g Material-item, which is an entity type, or Part-no which is an attribute type. He can also select a list of elements, but rarely needs to do so. Select 'all' is available.

It is possible to test by various simple algorithms whether or not any particular fact is currently selected. The command Select Bill-of-materials essentially only stores the fact that 'Select has-currently Bill-of-materials', to remember the current state. During the execution of a command such as List Material-item a number of stored facts are retrieved internally. Each of these is a potential fact for printing. Each such potential fact is subjected to a test-select procedure. If the stored facts include 'select includes all' then the potential fact will indeed be printed. Otherwise whether or not it is printed depends on:

- whether or not any of its elements are among the list of selected elements;
- whether or not any of its elements are related to any selected element through a fact using belongs-to as the verb;
- whether or not the potential fact has got qualifiers and whether or not the qualifiers are so related;
- whether or not the potential fact involves a modelling verb which is only appropriate to a particular model representation; for example model EAR does not include candidate identifiers;
- whether the subject of the potential fact is among the selected elements or belongs to a selected element, in the case where the verb belongs to a particular class defined by a higher metalevel statement; this ensures appropriate selection of facts such as that a relationship has a particular entity as owner, since the relationship need not belong to a local model even if the owner entity does;
- whether or not at least one of the objects of the potential fact is similarly related. Thus for example a potential fact that a particular entity has a particular dependent attribute will only be selected if the attribute belongs to the currently selected local model.

9. NORMALIZATION

The Normalize command is used to synthesise Third Normal Form structures from the functional dependencies that involve the selected attribute(s). Attributes can alternatively be selected by selecting the local model(s) that they belong to. Thus for example new local models that are combinations of existing ones can be synthesised.

The command to normalize the functional dependencies uses the same test-select procedure to retrieve those facts about functional dependencies that involve any attribute that is currently selected. These attributes are then listed in alphabetical order. The selected functionally dependent facts are then listed. The attributes are then grouped according to what they are functionally dependent on. All those in a particular group are functionally dependent on the same collection of attributes. That collection becomes the identifier for an entity type in third normal form. There are special procedures for dealing with and showing candidate identifiers.

10. OTHER COMMANDS

The Create command allows the user to define a new element and optionally to give it an abbreviation. Behind the scenes this creates two new elements linked by a fact such as 'Bom abbreviates Bill-of-materials'.

The Describe command prints all known facts that involve a particular element. This works irrespective of whether or not the facts are consistent.

The Help command gives help such as a description of any other command, or help with any named element. For an element it gives the abbreviation, what it is a synonym for, that the element is of a particular type, and any free text description indicating the meaning of the element. Commands such as 'Help model' and 'Help select' also give the current state.

The 'Insert fact' command does just that. But if the user types an element that does not already exist then he is asked whether it is spelt correctly. If yes then Create is invoked – giving an opportunity to create an abbreviation, then the original insert fact command is continued. With over 1000 words in the dictionary this gives an automatic spelling checker, since an attempt to insert a fact using a non-existent word produces a prompt asking if it is spelt correctly.

The Remove fact command removes an existing fact.

The commands Terse and Verbose control the use of abbreviations in output.

The Overlay command allows the fast insertion of a collection of facts held one per line in a named file.

The Exit command asks the user for confirmation – in case the user keyed an E by accident, and gives opportunity to store the current data dictionary contents as a file to allow continuation later.

The Warnings command allows the user optionally to switch on or off the printing of warnings about missing facts during execution of a List command. However, even with warnings switched on, the suppression of distracting classes of warnings such as concerning information sources or analyst's names can be automatically arranged by the presence of suitable stored metalevel facts.

Each command is programmed as a Pascal procedure – in some cases less than twenty statements. These call other high level procedures, eg to enable the user to input the rest of the command line – being a fact or an element or a list of elements; and appropriate processing and output. The high level procedures can call other procedures – eg one of which will cause a named procedure to be carried out for each sentence that has a particular element as a verb.

11. FURTHER DEVELOPMENT

After the student version was completely developed the use of DADICS for research was considered. A facility has been added whereby the user may insert facts that define new commands and procedures in terms of existing commands and the 30 or so low level procedures and functions that are explicitly programmed in Pascal. A new named procedure is defined by inserting one or more sentences with the name as subject and 'means-do' as verb. The rest of the sentence looks like a statement or block in Pascal. Available constructs include: repeat ... until ...; if ... then ... else ...; begin ... end; boolean expressions involving parentheses, and, or, not; and unlimited nesting of all kinds. Since facts can be inserted and removed in any order, the definition of inner procedures can be inserted or altered after the outer ones. They can be missing when execution of an outer procedure is attempted, producing an execution error. In general whenever an error occurs during the execution of a user-defined procedure a trace is automatically invoked. This produces details of the interpreted execution of the command that caused the error, including details of all calls of inner procedures. During the testing of new procedures the tracing can be controlled, for example to suppress the trace of some inner construct that is known to be correct, by inserting or removing metalevel facts involving the element 'want-trace'.

To cut development time and cost, in a few places our existing Pascal coding explicitly represents some rule. These places could be reprogrammed in a more general way to interpret new stored facts representing those rules. Thus one could give further generality and remove from the Pascal coding all knowledge of the rules of the data dictionary, its model representations, and its user's procedures. The explicit coding of most of the existing 15 commands of Table 1 as blocks of statements that call lower level procedures could be replaced by stored facts which could be interpreted to have the same effects. These procedures could then easily be further enhanced if required.

Table 1

The following commands are available.

Apply consistency rules	Create element	Describe
Exit	Help	Insert fact
List	Model	Normalise
Overlay	Remove fact	Select
Terse mode	Verbose mode	Warning

It might be possible to formalise the current system and make it more general purpose than a data dictionary. It could provide a means of defining rules of a high level language and the types of data structures involved and testing out examples. It could also be arranged to behave more as an intelligent knowledge-based system with a general structure of facts and rules about the facts and rules of procedures.

12. CONCLUSIONS

Dadics is a general purpose data dictionary system mainly driven by interpretation of stored facts. This gave easy and cheap implementation. Students have been able to learn to synthesise data analysis concepts by using it.

Great flexibility results from having the metalevel facts stored and manipulated just like the ordinary data dictionary contents. Rules of model representations can be created and changed online without reprogramming or recompiling. Powerful new procedures using combinations of existing ones can similarly be created and interpretively executed.

There is scope for further work to store further procedures as interpretable data. This would facilitate easy creation of further even higher level procedures and model representations.

Table 2

User dialogue	*Programmed procedure*	*Stored facts*
> Select: Bill – of – materials		
	→ Deal with abbreviation.	← Bom abbreviates Bill-of-materials
	Insert selection.	→ Select has-currently Bom
> List: entity-type	→ Deal with abbreviation.	← et abbreviates entity-type
	Check for et is-of-type ?	← et is-of-type mc
		mc abbreviates modelling-concept
	Check for mc is-of-type ? (not found)	
	If absent:	
	for each case of ?S is-of-type et:	← Mati is-of-type et
	if selected, e.g.	← Select has-currently Bom
	Mati belongs to Bom	← Mati belongs-to Bom
Material-item	← then print S (verbose by default).	← Mati abbreviates Material-item
...	(Repetitions omitted.)	...
> Model: EAR	→ Check valid.	← EAR is-of-type Model-representation-type
	Insert model.	→ Model has-currently EAR
> List: Material-item	→ Deal with abbreviation.	← Mati abbreviates Material-item
	Check for Mati is-of-type ?	← Mati is-of-type et
	Check for et is-of-type ?	← et is-of-type mc
	If present and = mc:	mc abbreviates modelling-concept
	for each metarule, e.g.	
	et may-have ?V ?O-type	← et may-have hui at
		at abbreviates attribute-type
	where V is-of-type modelling verb	← hui is-of-type mv
		mv abbreviates modelling-verb
	and belongs to current model	← Model has-currently EAR
		← hui belongs-to EAR
	do: for each Mati V ?O:	← Mati hui Prcd
	check it fits metarule, e.g.	
	O is-of-type O-type	← Prcd is-of-type at
	If selected, e.g.	← Select has-currently Bom
	O belongs to Bom	← Prcd belongs-to Bom
	← then print sentence Mati V O.	← hui abbreviates has-unrelated-identifier
Material-item has unrel – ident Product – code		
	If no fit found and warnings on then print warning.	
...	(Repetitions omitted.)	...
> Apply rules to: Mati	→ If to all then repeat as below for all elements.	
	Check synonyms, abbreviations.	← Mati abbreviates Material-item
	Check only one Mati is-of-type ?	← Mati is-of-type et
	If present: for each Mati ?V ?O:	← Mati hui Prcd
	find V is-of-type ?V-type	← hui is-of-type mv
	find O is-of-type ?O-type	← Prcd is-of-type at
	Check for metarule(s), e.g.:	
	et may/should-have V O-type	← et may-have hui at
	Attributes such as Prcd should appear only once in the entities of an EAR model, i.e. using unrelated verbs.	
	If metarule not found or other error then print explanation.	
Consistency check completed.		

Table 3. Contents of a 'Nil' data dictionary

ab ab abbreviates
it ab is-of-type
ow ab owns-relationship
ho ab has-owner
id ab is-domain-of
hm ab has-member
mc ab modelling-concept
mv ab modelling-verb
et ab entity-type
rt ab relationship-type
at ab attribute-type
degt ab degree-type
domt ab domain-type
cdm ab conceptual-data-model
prt ab property-type
rolet ab role-type
afa ab arises-from-at
afr ab arises-frm-role
hca ab has-cardinality
hfd ab has-funct-dep
hufd ab has-unrel-fun-d
acty ab activity
ev ab event
sch ab schema
ssch ab subschema
agg ab aggregate-model
ear ab ent-att-rel-model
pet ab pure-ent-type-model
hmv ab has-modelling-verb
model ab model-representation
isf ab is-synonym-for
sh ab should-have
mh ab may-have
cr ab consistency-rules
be ab belongs-to
cbib ab can-be-ident-by
hfi ab has-full-ident
hci ab has-cand-ident
hui ab has-unrel-ident
hupi ab has-unr-part-id
isi ab info-source-is
iro ab is-reverse-of
cont ab contains
syn ab synonym
fd ab functional-dependency
hda ab has-descript-at
ind ab indicates
hdeg ab has-degree
hdom ab has-domain
hoa ab has-optional-at
hfor ab has-format
hpr ab has-property
ifdo ab is-funct-dep-on
hexc ab has-exist-condn
exct ab exist-condn-type
0or1to0or1 ab none-or-one-to-none-or-one
0or1to1 ab none-or-one-to-one
0or1tom ab none-or-one-to-many
1to0or1 ab one-to-none-or-one
1to1 ab one-to-one
1tom ab one-to-many
mto0or1 ab many-to-none-or-one
mto1 ab many-to-one
mtom ab many-to-many
wcbo ab was-chckd-by-on
el ab element
maho ab member-may-have-related-owner
muho ab member-must-have-related-owner
oahm ab owner-may-have-related-members
ouhm ab owner-must-have-related-members
ouh1m ab owner-must-have-one-related-member
mnco ab member-must-not-change-owner
maco ab member-may-change-owner
Int ab Integer-domain
Stlg ab Sterling
Nam ab Name-domain
qt ab qualifier-type
nwv ab no-warning-verb
mvea ab m-verb-ent-att
mvre ab m-verb-rel-ent
acty it mc
at it mc
cdm it mc
degt it mc
domt it mc
et it mc
ev it mc
exct it mc
prt it mc
qt it mc
rolet it mc
rt it mc
sch it mc
ssch it mc
mv it mc
ab it mv
afa it mv
afr it mv
be it mv
hci it mv
hda it mv
hdeg it mv
hdom it mv
hca it mv
hexc it mv
hfd it mv
hfi it mv
hfor it mv
hm it mv
hmv it mv
ho it mv
hoa it mv
hpr it mv
hui it mv
hufd it mv
hupi it mv
ifdo it mv
ind it mv
isi it mv
iro it mv
isf it mv
it it mv
mh it mv
sh it mv
wcbo it mv
at mh ifdo at
et mh hui at
et mh hupi at

Table 3 (cont)

et sh hfi at
et mh hci at
et mh hufd at
et mh hfd at
rt sh ho et
rt sh hm et
rt sh hdeg degt
rt mh hexc exct
rt mh afa at
rt mh afr rolet
et mh be cdm
rt mh be cdm
at mh be cdm
et mh hda at
et mh hoa at
et mh hpr prt
at mh hpr prt
at mh hdom domt
et mh hca el
cdm mh ind el
et mh ind el
rt mh ind el
at mh ind el
domt mh hfor el
domt mh ind el
rolet mh ind el
qt mh be el
qt mh ind el
cdm mh isi el
et mh isi el
at mh isi el
cdm mh wcbo el
et mh wcbo el
at mh wcbo el
rt mh wcbo el
acty mh wcbo el
ev mh wcbo el
domt mh wcbo el
rolet mh wcbo el
prt mh wcbo el
mveat it mc
mvea it mveat
mv mh be mveat
mv mh ind el
nwv it mveat
cr it mveat
hci be mvea
hfi be mvea
hui be mvea
hupi be mvea
hfd be mvea
hufd be mvea
hda be mvea
hoa be mvea
mvret it mc
mvre it mvret
mv mh be mvret
hca be mvre
ho be mvre
hm be mvre
ind be mvre
isi be mvre
select it mc
model it mc
model mh hmv mv
ear it model
pet it model
agg it model
tra it model
ear hmv be
ear hmv hui
ear hmv hupi
ear hmv hufd
ear hmv hda
ear hmv hoa
ear hmv ho
ear hmv hm
ear hmv hdeg
ear hmv ifdo
ear hmv hdom
ear hmv hfor
ear hmv hexc
ear hmv hca
ear hmv ind
ear hmv isi
ear hmv wcbo
pet hmv hfi
pet hmv hci
pet hmv hfd
pet hmv hda
pet hmv ifdo
pet hmv hdom
pet hmv hfor
pet hmv ind
pet hmv isi
pet hmv wcbo
agg hmv be
agg hmv hfi
agg hmv hfd
agg hmv hda
agg hmv hoa
agg hmv ho
agg hmv hm
agg hmv afa
agg hmv hdeg
agg hmv hexc
agg hmv ifdo
agg hmv hdom
agg hmv hfor
agg hmv ind
agg hmv isi
agg hmv wcbo
tra hmv afa
tra hmv ho
tra hmv hm
tra hmv afr
tra hmv hdeg
tra hmv hexc
tra hmv be
sh be cr
mh be cr
ind be nwv
isi be nwv
hpr be nwv
hfor be nwv
hmv be nwv

Table 3 (cont)

mh be nwv
sh be nwv
ab be nwv
isf be nwv
iro be nwv
wcbo be nwv
maho it exct
muho it exct
oahm it exct
ouhm it exct
ouh1m it exct
mnco it exct
maco it exct
0or1to0or1 it degt
0or1to1 it degt
0or1tom it degt
1to0or1 it degt
1to1 it degt
1tom it degt
mto0or1 it degt
mto1 it degt
mtom it degt
Code it prt
Stlg it domt
Int it domt
Nam it domt
Yrmthda it domt
ow iro ho
cont iro be
syn isf isf
fd isf ifdo
cbib isf hci
Stlg hfor signed-integer-and-2-decimals
Nam hfor up-to-55 characters
Yrmthda ind Year Month Day
Int hfor digits
afa ind rt arises from at
afr ind rt arises from role

APPENDIX 1 PATTERN MATCHING

The semantic network is interrogated by pattern matching. Some of these features have been copied and enhanced from SOLO, a programming language developed for the Open University's third level course: D303, Cognitive Psychology. Some of the DADICS procedures and functions coded in Pascal are as follows, where * denotes a known element as a parameter and ? denotes an element to be found.

check for *S *V *O; if present: ... if absent: ...
check for *S *V ?O; if present: ... if absent: ...
check for ?S *V *O; if present: ... if absent: ...
check for one & only one object *S *V ?O; if present: ... if absent: ...
check for one & only one subject ?S *V *O; if present: ... if absent: ...
for each case of ?S *V *O do: ...
for each case of *S *V ?O do: ...
for each case of *S ?V ?O do: ...
for each case of ?S *V ?O do: ...
for each case of *S ?V ?O do: ...
for each *element {among objects} do: ...
if elements *E1 = *E2 then ... else ...
if descriptions *D1 = *D2 then ... else ...

The Table 2 example is rather simple, but is quite adequate to demonstrate a number of features of the system.To help you follow the logic: arrows denote flow of data, parameters and variables in the procedure description have been replaced by their actual values, some abbreviations have been expanded, and substantial further generality has been omitted. In general Select and Model just update the current state, List takes metalevel 2 rules and prints metalevel 1 sentences that fit, and Apply consistency rules takes each sentence with a particular subject and checks that higher metalevel rules justify its existence.

APPENDIX 2 META MODELS

The referee and the editor have asked us to include an appendix giving formal details of the meta models at both our meta levels 2 and 3.

We have a file of about 400 facts held as text, one fact per line. Table 3 shows most of it, for brevity omitting free text descriptive facts such as 'has-member indicates relationship-type has entity-type as member', which

- about 100 abbreviations, each gives a terse and a verbose element;
- 19 elements where each is-of-type modelling-concept;
- 30 elements where each is-of-type modelling-verb;
- about 50 rules such as x may-have/should-have y z, described below;
- the 8 modelling verbs that belong-to mvea; these may appear in sentences such as 'Material-item has-unrelated-identifier Product-code', which is selected if its object belongs-to those selected;
- the 5 modelling verbs that belong-to mvre; these may appear in sentences such as 'Where-used has-owner Material-item', which is selected if its subject belongs-to those selected;
- ear, pet, agg, tra each is-of-type model;
- 17 verbs for which ear has-modelling-verb; these can appear in the output from a List command with Model ear;
- 10 that Model pet has, similarly;
- 16 that Model agg has, similarly;
- 7 that Model tra has, similarly;
- 11 verbs for which no warning is given if no suitable facts exists when listing;
- various existence conditions for relationships;
- examples of property-type and domain-type;
- synonyms. A 'Nil' dictionary for a user starts with these 400 facts from the file, together with a few inserted during program initialisation. These few are 15 such as 'List is-a command' covering Table 1, and the default 'select includes all'. The meat is the rules of the form x may-have/should-have y z. This

means one or more facts such as s v o... may/should occur; where the subject s is-of-type x, the verb v is-of-type y, and each object o... either is-of-type z or is-of-type qualifier-type. If z is 'element' then every element is allowed among the objects.Conversely, for self-consistency, almost every fact must conform to a rule of that form. Here 'almost' covers the exceptions: the verb being 'abbreviates', 'is-synonym-for', 'is-reverse-of', 'is-of-type', or 'includes'; the verb and object being 'is-a command'; and where the subject s is-of-type st, and either st is-of-type stt and stt is not modelling-concept or st is-of-type stt is not found. Thus both the few facts inserted during initialisation are self-consistent, having no rules; and a 'Nil' dictionary from the file is self-consistent.

REFERENCES

1. PM681 *Data Analysis For Information System Design.* Activity Book. The Open University (1983) SUP 09646 1.
2. M352 *Computer-based Information Systems: Case Study: Standard Telephones Cables.* Open University (1980) ISBN 0 335 14005 X.
3. C. Beeri, & P.A. Bernstein, Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Transactions on Database Systems.* Vol. 4, No. 1 (March 1979) 30-59.