I

Whereas Sommerville and Wood present an alternative to conventional classification, Coelho describes a new procedure for applying a hierarchical classification scheme. Nevertheless, one common feature is the use of the logic programming language PROLOG, which has been found to be eminently suitable for creating and interrogating databases. Computer scientists can treat this chapter as a case study in the development of a natural language interface TUGA, which has subsequently been used in other areas beside classification (Coelho 1985). As most librarians are much less familiar with such interfaces than with menu selection and command languages, the account of TUGA's capabilities should prove informative. The technique of classification by means of comparison with materials already in the database also deserves consideration. Although in cases of doubt librarians often check the shelves or catalogue, classification is usually taught as if it should be done purely by reference to an abstract scheme. Coelho's method would help to ensure consistency.

The structure of the frames in the first chapter and the nature of the logical assertions in the second dictate the type of questions that can be answered. The traditional author/title catalogue also has a structure designed for the purpose of enabling the answering of certain types of question. This is discussed in the first part of the chapter by Davies. Cataloguing is a process governed by a large body of rules concerned to a large extent with onomatological relations (i.e. those involving variant forms of personal, corporate and geographic names) and textual relations (e.g. between different editions, or between original works and translations, adaptations, concordances and commentaries). The original task of codification can be compared with knowledge engineering in other domains in terms of difficulty. Panizzi wrote:

> When we drew up these rules, easy as it may seem, my assistants and myself worked all the day long for weeks; we never went out of the library from morning to night. We worked the whole day, and at night too, and on Sundays besides . . . (quoted in Cowtan 1872, p. 282)

This chapter concentrates on the issue of the extent to which cataloguing is a suitable domain for an expert system and takes a brief look at some Swedish work on improving the codification of the rules by using techniques of knowledge engineering.

REFERENCES

Coelho, H. (1985) The paradigm of logic programming in a civil engineering environment. *Computers and Artificial Intelligence* **4** (2) 115–124.

Cowtan, R. (1872) *Memories of the British Museum*. London, Bentley.

Minsky, M. (1975) A framework for representing knowledge. In: Winston, P. H. (ed.), *Psychology of computer vision*. New York, McGraw-Hill.

Vallée, J. (1982) *The network revolution*, Harmondsworth, Penguin.

In. "Intelligent Information Systems: progress and prospects", ed. R. Davies. Ellis Horwood, 1986. 1986a

1

A software components catalogue

Ian Sommerville, Department of Computing, University of Lancaster, UK, and
Murray Wood, Department of Computer Science, University of Strathclyde, UK

1. INTRODUCTION

The costs of developing an maintaining large and complex computer software systems are immense and make up an increasing part of the Gross National Product of the countries of North America and Western Europe. Although precise figures are difficult to come by, Lehman (1980) suggested that software costs made up about 3% of the US GNP in 1977, and it is reasonable to surmise that this figure has increased significantly since then. Indeed, Boehm (1979) points out that the growth in expenditure in software considerably exceeds overall economic growth, with the result that by 1990 software costs would constitute between 7% and 10% of the GNP of advanced nations.

Software engineering is the application of computer science and associated topics in the development of software systems, and it is now clear that effective software engineering practices are of critical importance if software is to be developed on time, within budget and to specification. At the time of writing, the USA, Japan and most West European nations have established special research programmes in software engineering. The aim of these programmes is to develop techniques which will increase the productivity of software developers and decrease the costs of software development and maintenance.

In the UK, the Alvey report (1982) identified the need for a programme of software engineering research, and the main thrust of this programme is the development of so-called Integrated Project Support Environments (IPSE). An IPSE is an integrated set of software tools (programs which assist software development) which support the development and maintenance of computer systems from initial conception through to operation and maintenance. They differ from programming support environments like Unix™† (Ritchie & Thomson 1978) in that they support development activities such as requirements specification and design as well as implementation activities such as programming and system testing.

† Unix is a trademark of Bell Laboratories.

The work described in this chapter is being carried out in the context of the ECLIPSE IPSE (Alderson *et al.* 1985) which is being developed by a consortium of UK companies and universities. Part of the ECLIPSE project is explicitly concerned with developing methods and tools to support the reuse of existing software, and our work on the development of a software components catalogue is part of this project. This catalogue is not simply a keyword-based system but incorporates knowledge-based techniques which we believe enhance its utility and will improve its performance in practical use.

In the remainder of the chapter, we discuss software reuse in general and identify different classes of reusable software components. We then go on to discuss the requirements for our system, identify problems which arise with keyword-based approaches, describe the notion of conceptual dependency and then describe the prototype implementation of our components catalogue. Finally, we identify further developments in this area which we intend to pursue in the near future.

2. SOFTWARE REUSE

The reuse of software components which have been developed in previous projects is one of those motherhood principles which is generally espoused but little practised, except in very specific areas of software development. There are powerful economic arguments for reusing an existing component rather than reinventing that component anew, and Boehm *et al.* (1984) have suggested that we will only see very significant improvements in software productivity when software reuse is widely practised. Not only does reusing an existing component reduce the development cost of a software system, it should also reduce the testing costs as, presumbly, that component has already been tested in some other system.

Given, therefore, that reuse is widely accepted as a 'good thing' and that it has a clear economic benefit, why then is it not more widely preactised, particularly in the development of large and complex software systems? The reasons why software reuse is the exception rather than the rule are partly technical, partly human and partly a consequence of the way in which software is produced. Our work is aimed at some of the technical problems and space does not permit a full discussion of the non-technical factors which influence the reuse or otherwise of software. In summary, the human problems are a result of the fact that software development is seen as a highly skilled activity and reuse implies some kind of deskilling; the economic problems are a result of the difficulties of deciding who owns an intangible object like a softwarer component when the component is developed by X for Y.

Leaving aside these problems which present difficulties all of their own, the technical problems of reusing software may be summarized as follows:

(1) It is more expensive and difficult to develop a generalized component for potential reuse than it is to develop a specific component for a specific system. Indeed, as relatively little software reuse is practised, we are not really at all sure what characteristics of a software component lead to or militate against future reuse of that component.

(2) There are no effective catalogues of existing software components apart from ad hoc lists provided with specific systems such as Unix. Where such lists exist, only very simple keyword-based retrieval systems are provided to assist the user in finding the required component.

(3) The nature of software components and system requirements is such that in many cases a software component is almost, but not ideally, suited for a particular task. Whether or not that component can be reused depends largely on the difficulty of modifying that component.

The ECLIPSE reuse project is tackling all of these technical problems associated with software reuse, and we are particularly concerned with the development of a software components catalogue. This is not simply a classified list of software components; it is an integrated component classification and retrieval system which is based on the semantics of a natural language description of the function of software components.

3. REUSABLE SOFTWARE COMPONENTS

The first thing that we have to do in setting up a software components catalogue is to decide what we mean by a reusable component. It seems to us that there are three classes of software component which might usefully appear in such a catalogue:

(a) General-purpose software systems which might be used, *without change*, in a variety of different applications. Typically, such components are large software systems in their own right — a database management system is perhaps an archetypal example of such a component. Other examples are word processing systems, editors, etc. It is useful to have such components in binary form only, and it is useful to have entries for such components in the catalogue even if the component itself is not held in the component database.

(b) Primitive software components such as functions or abstract data types (an abstract data type is a data type where the implementation details are concealed and access is limited to a number of pre-defined functions). Again these would probably be incorporated in a software system without change although, on some occasions, it may be useful to have these in source code form so that they could be adapted and optimized for use in a particular software system. Examples of such components include mathematical functions such as those provided in the NAG (Numerical Algorithms Group) library and some (but not all) Unix command processes (deroff, which removes text formatting commands, for example). The characteristic of components in this class is that they do one thing and one thing only and that it is possible to define what the component does in a precise and (perhaps) in a formal way.

(c) Software sub-systems which are essentially large chunks of code which carry out a set of related tasks. In general, these will only be useful if they are available in source code form, as it is unlikely that they will do exactly what is required for a particular software system. They provide a general capability, but it may be necessary to adjust their interface in order to make use of them is particular circumstances. Examples of such components might be the lexical analysis part of a compiler (a generalized component which produces tokens from an input stream of characters), a navigation sub-system, and a graphics sub-system to provide graphical I/O on a bit-mapped workstation.

Components from class (a) above are generally very large and software systems tend to be developed around them rather than simply to include them. We have not considered the problems of including such components in our catalogue but, intuitively, it seems that a simple retrieval mechanism is all that is required to find such components.

Initially at least, we have concentrated on building a catalogue of primitive components, class (b), and on establishing a retrieval mechanism for such components. Although the economic benefits of reusing larger-scale components such as those from class (c) are obvious, there are great problems in classifying and retrieving such components.

For example, say a user requests a 'tokenizer', ideally the components catalogue would know that this is more or less what the lexical analysis phase of a compiler does and that such a component would fulfil the user's requirement. Thus, the components catalogue must incorporate some semantic knowledge about how a component is used and about the component's logical function.

Furthermore, even if components in class (c) are not directly applicable to a particular requirement, it may be the case that within that component there are single functions or sets of functions which might be reused. These may not be primitive components in their own right as they may depend on particular sub-system data structures, but they may be sufficiently adaptable to be modified; or it may be possible to reuse the data structure. It would be very useful indeed if (somehow) the components catalogue could incorporate the knowledge that a particular sub-system made use of other operations and that these *semi-primitive* functions are potentially reusable.

4. REQUIREMENTS FOR A COMPONENTS CATALOGUE

The primary requirement of the component catalogue is the ability to match users' requests for software components onto descriptions of software components which satisfy these requests. Since software components have the property that they are flexible (their code or designs may be adjusted) the system must also have the capacity to match requests onto components which only partially satisfy the requirements ('fuzzy matching') and which, with alteration, may be of use. In order to perform such matching, the system is dependent on a representation scheme for component requests

and component descriptions which captures the relevant information from both, yet is simple enough to be manipulated by computer.

A second requirement is that the system should be usable with the minimum of training effort by any user/developer of computer software. If a large amount of effort is spent in training then the benefits from reuse are reduced. This requirement rules out formal specification as a method of software component description and request specification. (Formal specification also seems inappropriate for performing fuzzy matching). Ideally the system would have natural language interfaces whereby software component descriptions or requests for software are analysed, producing corresponding internal representations. Since developing such an interface is a research project in itself, we have limited our requirements for interfaces so that the analysis of component descriptions is done by an expert human cataloguer and the request interface is a standard form which takes keyword or simple phrase responses to system prompts.

A third requirement of the system is the ability to build knowledge of software components into the matching process. There are a number of areas where knowledge is required. Knowledge is required to recognize that, although requests/descriptions of software in natural language are syntactically different, they are semantically similar. Knowledge is required to determine when a component, although not wholly satisfying a request, is close enough to be of interest. Also, it may be useful to utlize knowledge that components are contructed from sub-components which perform distinct tasks, therefore extending the range of requirements that the 'super' component may satisfy. Most of software component knowledge does not consist of concrete rules; rather it is of the form of heuristics.

The final requirement of the component catalogue is that the system should be extensible. Since the software development process covers a very wide range of application areas, it is impossible to construct a demonstrable system that utilizes software components from all areas. The methods used should be such that they can be applied to software components regardless of the application type.

5. KEYWORD CLASSIFICATION APPROACH

In this section we briefly decribe an initial approach to the problem and assess how it meets our requirements. The approach was based around a classification of software and an association of keywords with each of the recognized classes. Using the software components available with the Unix operating system as our source of components, we constructed a hierarchical classification of Unix software. Each classification was analysed and a representative set of keywords for each class selected. The classification was based on a one or two line summary of the component extracted from the Unix operating system manual. The software components were manually classified into the respective classes, no limitation being placed on the number of classes a component could belong to. Retrieval consisted of

requesting keywords from the user and returning the components in the class(es) associated with the keyword. If the class was considered too large to return for user analysis, then a straight keyword search would be applied to the descriptions in the class and only those containing the keyword were returned. A thesaurus facility was incorporated into the system by including not only keywords but words similar in meaning and semantic derivatives as representatives of each class.

Although this approach is quite straightforward and performed reasonably well for the small library of software components we had (~700 components), we identified a number of weaknesses. Classes of components tended to be too large to form manageable responses to the user. Even in our small catalogue, the 'communication' class of software component had almost 100 components. This could be partially solved by splitting the class into sub-classes, but it then becomes difficult to identify appropriate keywords for each class. When a request results in a large number of components, applying a keyword search is unsatisfactory, as keywords alone do not adequately describe components. This causes problems ranging from null repsonse, because the user specifies an unused keyword, to an overwhelming response because the keyword is too general. These problems are caused by attempting to describe software components in terms of single keywords and they arise in most keyword-based retrieval systems. What is required is a description that is more representative of software components.

A second problem of this approach is that the classification scheme is too rigid. It is constructed by analysing the subject area and deriving classes that seem appropriate. Therefore, components either fit into a class or do not. If a component does not fit a class, it is not possible to denote this and there is no easy way of adding new classes. In classification terminology, a scheme that uses predefined classes is termed enumerative, and there are acknowledged weaknesses in their use. Prieto-Diaz (1985) concludes in a survey of classification methods that in areas where flexibility is required, 'An enumerative scheme is not practical beyond small collections . . . Enumerative schemes are usually avoided for document retrieval, index construction or classification of document abstracts.' He argues for the use of faceted classification. Such as classification allows the creation or synthesis of specific categories by combining terms from different classes or 'facets'. This approach is more flexible than an enumerative scheme but is still dependent on keywords.

Finally, the approach partially satisfies our requirement to build knowledge into the system, in the sense that the classification approach using a thesaurus does cope with semantically similar keywords, but we argue that keywords do not adequately describe the semantics of software components and so this is of little consequence. Building knowledge about software components into the system is also difficult since knowledge has to be associated with a complete class of software components or with a single keyword.

In conclusion we felt that keywords in themselves do not sufficiently

describe software components nor requests for software components and that an enumerative type of classification scheme does not provide the required flexibility.

6. CONCEPTUAL DEPENDENCY

We felt that is was necessary to look for an internal system represenation for software components other than that of keywords. Natural language understanding/question-answering systems provide a mechanism for representation of natural language concepts, known as conceptual dependency. As we will show, is seems that the basic idea behind conceptual dependency may be applied in a simplified way to the representation of software component descriptions and software component requests.

'Conceptual Dependency is a representational system that encodes the meaning of sentences by decomposition into a small set of primitive actions' (Lehnert 1978). The core of such an approach is a number of fundamental **concepts** which are sufficient to capture the semantics of any domain of interest. Schank (1972) lists three types of concept — nominals, actions, and modifiers. Nominals are considered to be things that can be thought of by themselves without the need of relating them to some other concept, e.g. duck, book, New York, file, bit, etc. An action is that which a nominal can be said to be doing. There are certain basic actions which are the core of most verbs in a language. A modifier is a descriptor of the nominal or action to which it relates, and serves to specify an attribute of that nominal or action, e.g. adverbs and adjectives. Each of these conceptual categories can relate in specified ways to each other. These relations are called **dependencies**.

These ideas are implemented in a natural language database system (Waltz *et al.* 1977) in a form known as Concept case Frames. Each concept case frame consists of the **act** (typically related to the verb) and a list of noun phrases which can meaningfully occur with the act. Each act covers a number of related verbs. These frames are used as tools to build a query to the database by filling in the slots of the template with information extracted from the current request, earlier requests, world knowledge and default values.

A practical realization of this approach involves an analysis of the problem domain, resulting in the distinct conceptual actions that can occur, the distinct conceptual objects that are acted on and the modifiers which describe these actions and objects. These are then related in the form of conceptual case frames for each action, which have slots which specify objects that are meaningfully manipulated by the action in the domain of interest. A natural language parser maps the verbs in textual input onto the conceptual actions, and nouns onto the object slots.

An example is the conceptual action PTRANS which represents the transfer of physical location. PTRANS has object slots requiring an actor, an object, an origin and a destination. The sentence 'John gave Mary the

book' would map onto a concept case frame representing the PTRANS action as follows:

```
                John
                  | source
John — PTRANS — Book
actor     | destination    object
                Mary
```

This would form the system's internal representation of the sentence, capturing the semantics of the act to give.

As we stated in the assessment of the keyword classification approach, keywords alone do not satisfactorily describe software components. It seems that a system which is based around the concepts of software components, relating these concepts in the form of frames, provides a more complete picture of a software component.

Within the restricted natural language domain of software component descriptions there are only a limited number of basic concepts. Following Schank we can separate three fundamental types of concept — action, nominal and modifier. Actions correspond to the basic, fundamental **functions** that software components perform. Nominals correspond to the objects that perform the function (the software component itself), objects that the function manipulates and objects produced as a result of the function. Modifiers describe actions and nominals. Corresponding to dependency relations or conceptual case frames we have our own frames, 'Software Function Frames'. There is a software function frame for each basic function that software performs, based around the action, with slots for the objects manipulated by the component. Since, in general, software components perform a function and furthermore it is the function which characterises the software component, a representation scheme which captures the function provides a sound foundation for the description of software components, the description of software component requirements and the matching of the two.

This is confirmed by the work of Prieto-Diaz (1985) in his classification scheme for software. When considering descriptors for software components after a study of software descriptions, he concludes, 'Program listings are characterized by describing the function performed by the program . . .'. He then goes on to use the function and the objects that are manipulated by the function as a basis for a software classification scheme.

A component catalogue based on these ideas requires an analysis of the software component domain resulting in a set of basic functions for software. There is a direct relationship between verbs in software descriptions and the basic functions of software components. There should be one basic function for each 'classification' of conceptually similar verbs, that is, verbs that describe semantically similar software functions. An example of conceptually similar verbs might be search, look and find. Also required is a classification of the objects manipulated by software components into classes or 'nominals' that represent conceptually similar objects. An example of a nonimal might be 'file_bit', that is, objects that are parts of a file; typical members of this nominal classification are line, word, pattern, etc.

Having decided on the basic functions of the software component domain it is necessary to develop a set of software function frames. For each recognised basic function of software there is such a frame which relates the objects to the function. Each frame has a variable number of slots, the number being dependent on the meaning of the function. In our current domain the number of slots ranges from one to three. An example of each is:

— control function has one slot, for the object that is controlled.
— print function has two slots, the object that is printed and the object that is the destination for the print (e.g. a terminal or a line printer).
— the communication function has three slots, the object that is communicated, an object that is the source of what is being communicated and an object that is the destination for what is being communicated.

For any software component function all, some or none of the slots may be filled. Currently these slots are unrestricted, that is, any object can fill the slot. In later versions we shall consider restricting the objects that can fill a slot to those that can be meaningfully associated with an action. Rules defining such restrictions would be useful in automatic analysis of limited natural language requests or descriptions.

As an example of a software function frame, the conceptual action of 'searching' might have three slots — one for the actor carrying out the search, the software component itself, one for the object that is being searched and one for any object that is being searched for. A completed software function frame that captures the function of the Unix software component 'grep', which can be described by the phrase 'searches a file for a specified pattern', might be:

```
                pattern
                  | object that is searched for
'grep' — search — file
actor        object that is searched
```

Thus the actor is the software component called 'grep', the object being searched is a 'file', and the object being searched for is a 'pattern'.

While there are other properties of software components such as operating system, implementation language, host machine, etc. which require consideration before a software component may be reused, we feel that they offer little as far as retrieval is concerned. If there are a number of components which perform the same function on the same objects then

these environmental considerations can be used either by the user or the system to distinguish them.

In conclusion we would argue that software components, unlike more conventional objects of information retrieval, have a special characteristic which allows them to be described in more detail than is possible with independent keywords. This special characteristic is the function that the software component performs. We suggest that a suitable representation for software component descriptions and software component requests is a software function frame which relates the software function to the objects.

7. INITIAL RESULTS

We have implemented a prototype component catalogue based on the ideas outlined above, using the programming language Prolog and a set of software components from the Unix Operating System. There are four main areas to the work: the construction of a dictionary of software component terminology; construction of a library of software components represented by software function frames; construction of a sub-system that interacts with the user building a software function frame representing the request; and construction of a matching sub-system which attempts to find a best match for the user's request within the library of software component representations.

Through experimentation with different versions of our ideas we have found that the classification of any items, be they verbs as conceptual actions or nouns as nominals, or the construction of software function frames to represent components should be as wide-ranging as possible. That is, if an item could possibly belong to a multiplicity of classes then it should be entered in all those classes rather than try and tie items down to one specific class. This approach copes to some extent with the problem of different classifiers having different views of how a software component should be described. Thereafter the matching process is strictly constrained, relaxing gradually if no match is found. This rule resulted from classifying words in one particular class or components as being of a particular function. In experimentation with users we discovered that they legitimately viewed components as being described by a different function which, because of the strict classification, caused a failure to match. Initiating the search process by first looking for strict matches and gradually relaxing the criteria for a match until one is made, the system funds the components(s) that best fit the users requirements.

We now give a brief overview of the constituent sub-systems.

7.1 Construction of the dictionary

From previous work on the analysis of software component descriptions we had built a dictionary of about 300 verbs and nouns relevant to the function of Unix software components. (This dictionary is continually updated as the system registers unknown words which are then added to the dictionary by the human cataloguer). The verbs and nouns were manually classified into

classes which were thought to represent the basic actions and basic objects of software components. This was an iterative process — as the system was developed and tested and new words were discovered, classes were merged and split until it reached its current state. Some examples of dictionary entries are:

```
verb(locate,action_search).
verb(terminate,action_control).
verb(display,[action,_print,action_inform]).
noun(directory,nom_file).
noun(expression,[nom_maths,nom_filebit]).
```

These are in the form of Prolog facts. A fact has the form

```
functor(Arg1,Arg2,...).
```

In the example dictionary entries above, there are only two arguments. The functor or name of the clause is either verb or noun, depending on whether the word is a verb or noun. The arguments are the word itself (1st arg.) and its conceptual classification (2nd arg.). If the word could belong to a number fo classes then the classification is a list (denoted by the '[]' brackets in Prolog) of alternatives. Therefore the above facts denote that:

— locate is a verb classified as an action of type *search*.
— terminate is a verb classified as an action of type *control*.
— display is a verb classified as an action of type *print* and an action of type *inform*.
— directory is a noun classified as a nominal of type *file*.
— expression is a noun classified as a nominal of type *maths* and a nominal of type *filebit*.

Obviously many of the words do not strictly fit into one class only. Display, for example, can mean 'print' or it can mean 'provide information'. In Keeping with the 'loose classification' philosophy such words are entered in all the classes they could meaningfully occur in.

In some contexts, actions such as 'provide information' and 'print' are closely related. For example, the action associated with the phrase 'report an error' could belong to either class. It is often worthwhile, if a search is exhausted using one action, to search using an alternative, closely related action. To provide this facility we have a second classification of actions whereby a limited number of actions are denoted as being possible alternatives for each other — a thesaurus of actions.

7.2 Construction of the library of components

As we have stated earlier, we used part of the library of components available with the Unix operating system as our source of software components. Although Unix software components are intended for software

development they are a realistic example of reusable software components. It was one of our original aims that the system should analyse natural language descriptions components, itself building the corresponding 'software function frame'. Since the aim of this prototype version was to test the capability of our representation as the basis for the matching of requests against descriptions, the natural language analysis part of the system (which is ambitious anyway) was carried out manually. The Unix manual entries for the 300 or so components we selected to use were studied, extracting the information which described the function of the component. For each function a software function frame was constructed. Again, the maxim of 'loose classification' was applied, resulting in components being described under all appropriate functions.

7.3 The user interface

Our requirements were that the system should be usable without special training effort or learning of query languages. In this prototype system the user's input takes the form of keyword responses to prompts from the system. The system builds a software function frame representing the user's request by prompting the user, either for a verb describing the action the component performs, or a noun representing objects manipulated by the component. If the user inputs a verb, the system finds a skeleton frame corresponding to the action which conceptualizes the verb. For each frame slot there is a corresponding prompt which the system displays to the user in search of objects that are manipulated by the component. If the request is initiated with an object, the user is then prompted for a verb which describes the way in which the object is manipulated. During the request-building process the system can be asked for help regarding appropriate responses. The system uses partially completed frames to search the database, collecting together all the known values for 'slots' yet to be filled. These are then displayed to the user for use in responding to the remaining prompts.

An example of the request-building process plus responses is shown in Fig. 1. The components retrieved are from the utility program section of the Unix manual. The emphasis in this prototype system has been in developing a representation for component descriptions and therefore the interface has been kept relatively simple. In later version we aim to make use of the windowing facilities of the SUN workstation to improve the user interface to the components catalogue. (In the example, the system prompts are in normal typeface, user responses in bold typeface and author's comments in italics.)

7.4 Matching of requests and descriptions

Since requests and decriptions have the same internal representation, namely software function frames, using Prolog's built-in control and search mechanisms (see later example) means the matching process is only depen-

Input a verb describing the ACTION the component performs: **compile**
Verb: compile
compile is an action of type compile
By specifying objects the search is restricted — only complete relevant fields
Type ? for typical examples

Is thre a source language that is compiled: **pascal**
Is there a target language/machine: *(non specified)*

PC is a pascal compiler *(component satisfying request)*

Do you wish to continue search ? **y**

(more components satisfying request)
PX is a pascal interpreter
PIX is a pascal interpreter and executor
PMERGE assembles separate pascal programs into a single program
(The search has been relaxed by replacing the verb 'compile' by the conceptual action 'compile'. Included in this classification of action are the verbs 'interpret' and 'assemble'. Therefore any components which 'interpret pascal' or 'assemble pascal' are found.)

Do you wish to continue search ? **n**

Fig. 1. — Request formulation and system response.

dent on rules for the best match of the two. An exact match is required if possible. This is implemented by looking for components that are described using the same verb and nouns as the user specifies, thereafter a match is sought of actions instead of verbs, then nominals instead of nouns, then if multiple object slots are filled single object matches are sought and finally just matches of action or object alone.

As in all information retrieval systems, the notions of recall (the proportion of relevant material retrieved) and precision (the proportion of retrieved material that is relevant) are important considerations. By completely relaxing the search we can maximize recall but at the cost of poor precision. Crucial to the component catalogue is that recall should be maximized so that any relevant components are retrieved. At the same time there is no point in returning a large portion of irrelevant components to make sure all relevant components are returned. This is one of our criticisms of a keyword-based system. We argue that since our representation is more representative of a software component we have increased both recall and precision.

8. SYSTEM IMPLEMENTATION

In choosing an implementation language for a system based on these ideas, there are a number of criteria to be considered. We have to be able to represent and manipulate large numbers of component representations. We have to be able to search through these representations matching on all, some or one of the fields in the representation. We would like be able to add 'heuristic' rules which aid the 'fuzzy' matching process and also, eventually, rules representing knowledge about software component functions.

As well as having the general properties of being easy to read, understand and being more 'high–level' than imperative programming languages such as Pascal or C, Prolog goes some way towards satisfying the above requirements. Firstly, since Prolog is designed to solve problems involving objects and the relationships between objects, it provides a natural representation for our software function frames. For example, the search function described earlier is represented by the Prolog clause:

```
search(Component,Where,What).
```

which is interpreted as 'Component' performs the 'search' function on the object 'Where' searching for objects of type 'What'. A realisation of this for the software component called 'grep' and described earlier as 'grep searches files for specified patterns' would be:

```
search(grep,file,pattern).
```

Central to our catalogue is a Prolog consisting of such 'facts' represeting the library of software components.

A second feature of Prolog is the built-in control and pattern-matching capabilities designed for search space traversing. These facilities are necessary when searching a large database of component representations and are highlighted in the following example based on the clauses shown in Fig. 2.

Clause (i) is a Prolog 'fact'. As we have said above, this represents the component 'grep' which 'searches files for specified patterns'. There would be at least one of such facts for each component in the database.

Clause (ii) is A Prolog 'rule'. This states that

IF
component 'Comp' performs a search action on objects of type 'Comp_Where' for objects of type 'What'
AND
'Require_Where' is a noun of classification 'Nominal'
AND
'Comp_Where' is also a noun of classification 'Nominal'
THEN


```
(i)   search(grep,file,pattern).
(ii)  search(Comp,Require_Where,What):-
          search(Comp,Comp_Where,What),
          noun(Require_Where,Nominal),
          noun(Comp_Where,Nominal).
(iii) search(Comp,Where,Require_What):-
          search(Comp,Where,Comp_What),
          noun(Require_What,Nominal),
          noun(Comp_What,Nominal).
(iv)  noun(word,file_bit).
(v)   noun(pattern,file_bit).
```

Notes:

1. Clauses (ii) and (iii) are presented in a simplified form. Since they are defined recursively the could in fact cause an infinite loop. The loop-checking mechanism has been omitted for the sake of clarity.
2. In Prolog, arguments starting with capital letters are variables, lower case constants.
3. Commas between clauses denote 'and'.
4. A match occurs in Prolog if two clauses have the same functor and for each argument constants match constants and variables match anything. If a variable matches a constant it becomes 'instantiated' to that constant.

Fig. 2 — A snapshot of a component catalogue on Prolog.

the component 'Comp' will also search objects of type 'Require_Where' for objects of type 'What'.

The rule basically says that, if a component searches for objects 'What' in objects of the class of nomional 'Comp_Where', and furthermore 'Require Where' and 'Comp Where' are classified as the same type of nominal, then a request for components searching through objects of type 'Require_Where' may be satisfied by a component that searches through objects of type 'Comp_Where'.

This rule is one of the ways in which the criteria for a match are relaxed. It replaces the requirement that requests should be matched against descriptions with objects matching objects by the less strict requirement that only the nominal classification of the objects need match.

Clause (iii) is similar to (ii) except that it is the objects that are searched for that are in the same nominal classification.

Clauses (iv) and (v) are facts stating that the nouns 'word' and 'pattern' belong to the nominal classification 'file_bit'.

Now consider a request for a component that 'finds particular words in a specified file'. Ideally this would be converted to a Prolog goal of the form:

```
search(Comp,file,word).
```

Using its built-in search mechanism and pattern-matching facilities, Prolog tries to prove this true in its database of known information. To do this, the

goal must either match (Note 4 above) against a fact or match against the head (left-hand side of the ':–' symbol) of a rule in which case the body (right-hand side) becomes the new goal(s) which has to be proven.

A match is sought in a top-down manner through the database: the relative order of facts and rules is therefore important to the meaning of a Prolog program. Searching for a match for the above goal fails with clause (i) since 'word' fails to match 'pattern'. Assuming there are no other component facts which match the goal, the search reaches clause (ii). The goal matches the head of the clause with the variables 'Require_Where' and 'What' being instantiated to the constants 'file' and 'word' respectively in the process. The criteria for a match have been relaxed by replacing the requirement that the component searches files by the requirement that the component searches objects that are in the same nominal classification as file.

We now have a conjunction of goals, to satisfy, namely:

```
search(Comp,Comp_Where,word),
noun(file,Nominal),
noun(Comp_Where,Nominal).
```

The first goal fails since there is no component that searches anything for words, i.e. there are no facts with 'word' as their third argument. This path of the search has been exhausted and therefore Prolog employs its built-in backtracking mechanism, undoing all instantiations of variables until an alternative path in the search for a 'proof' is found.

This is found by matching the original goal with clause (iii) instantiating 'Where' to 'file' and 'Require_What' to 'word'. In this case the search has been relaxed by replacing the object that is searched for by its nominal classification. This time the new goals are:

```
search(Comp,file,Comp_What),
noun(word,Nominal),
noun(Comp_What,Nominal).
```

Prolog now proceeds from the start of the database, searching for a proof for each of these goals in succession. search(Comp,file,Comp_What) matches clause (i) with 'Comp' becoming instantiated to 'grep' and 'Comp_What' to 'pattern'. The goal is therefore proven and removed from the goal list. Since the instantiations are perpetuated through the conjunction of goals the third goal becomes noun(pattern,Nominal). The second goal, noun(word,Nominal), is next to be proven, matching against clause (iv) with the result that 'Nominal' is instantiated to 'file_bit'. This leaves the third goal, now noun(pattern,file_bit) which matches clause (v). The body of clause (iii) has been proven true therefore the head is deduced to be true. Since 'Comp' was instantiated to 'grep' in the process, the original goal succeeds as:

```
search(grep,file,word).
```


In this way the system finds the component 'grep' which may satisfy the user's requirement.

A simple example of how knowledge may be built into the Prolog database in the form of a rule might be:

```
search(Comp,Where,What):– edit(Comp,Where).
```

This rule states that if a component that searches objects of type 'Where' is required, then a component that edits objects of type 'Where' may be of use. This reflects the general ability of editors to search for specified patterns. This rule would be positioned in the Prolog database so that it was used after all components that perform a search action had been sought.

9. PROBLEMS IN DEVELOPING PROTOTYPE SYSTEM

Any classification scheme, be it based on keywords or action/object relationships, is somewhat arbitrary and dependent on the classifiers' experience and understanding. This can be partially overcome by discussion and experimentation with a range of interested parties in an attempt to reach consensus. Our approach to classification, whereby items are classified in as many classes as possible, overcomes this to some degree. Contextual ambiguity is also handled by classifying items in all the classes in which the could occur.

A fundamental problem in the development of any component catalogue is the testing of the system. Testing should be done by non-developers. Since it is only feasible to provide a limited catalogue of components, the tester has to be guided to what type of component exist. This guidance has to be some sort of description of the software components available, this inadvertently providing the tester with a form of request for the components. Related to this problem is the general lack of reusable software components with which to build a components catalogue. Ideally the system would be used by an organisation such as a software house developing its own software so that it could be reused in future, storing it in the catalogue for later reference. Development of a general library of software components to test the system is infeasible, so we are dependent on showing that the system performs adequately with the Unix components and that the methods used are general enough to be applied in other areas of software use.

A potential weakness of the developed system is that software components are not described in enough detail. It would be a simple task to extend the component representation so that components could be distinguished by environment features as well as their function. Similarly, the software function frame could be extended to include modifier slots in order that actions and objects are described in more detail. This is not necessary in our limited library of components, but in the interests of generality they should be added. An example of how modifier slots might be added to the software function frame is:

```
comp(Name,verb(Verb,Verb_Mod,Action),
  noun(Noun,Noun_Mod,Nominal),...).
```

Thus verbs and nouns would be represented by clauses which grouped together the item, its modifier and its classification. A completed frame with modifiers for the component 'cc' which 'compiles C programs' would look like:

```
component(cc,verb(compile,_,compile),
   noun(program,c,nom_prog),Dest_Lang).
```

The verb has no modifier (it is ignored using'_') but the object that is compiled, program, is described by the modifier 'c'.

10. CONCLUSIONS

We have presented an overview of a prototype implementation of our ideas. The aim has been to develop a representation for software component descriptions and software component requests which captures their *meaning*. We described an approach used in natural language understanding to represent the semantics of situations known as conceptual dependency. Following this method we recognised the main concepts of the software component domain, namely the function performed and the objects manipulated by the function. The relationship between the function and objects is represented by what we have termed software function frames. This representation if particularly suitable for information retrieval in the software component domain for three reasons. Firstly, compared to other areas of information retrieval, the software component domain has only a limited dictionary of terminology. Secondly, this domain has clearly recognizable concepts that characterize software components. Thirdly, these concepts have clearly defined relationships, that is, functions operate on objects producing objects. Whether this approach would be suitable for other areas of information retrieval is dependent on the satisfaction of these three criteria.

We argue that, since a system based around these ideas attempts to capture the meaning of descriptions and requests, it has a better capability for matching the two than that provided by independent keywords. In fact, in the worst case, the system behaves as if it were keyword-based. We have shown how it is possible to build knowledge about software components into the system. Although the interface is simplisitic, it meets our initial requirement in that it is straighforward to use. The system is also flexible and extensible. It is a simple matter to add new conceptual actions or conceptual nominals — there is no predefined classification scheme into which new classes have to be accommodated.

Although we have argued that by representing software component concepts the developed system should result in better recall and precision

than that provided by a keyword system, we recognise the importance of a systematic evaluation to confirm this. As stated above, evaluation is not straightforward. As well as the problem of not being able to provide users with a comprehensive catalogue and therefore having to describe the available components in some way, there is the added problem of what capabilities a 'keyword system' should have in order to compare the two. Currently we are developing a keyword system with a range of capabilities and see evaluation taking into account user reaction to system usability as well as recall and precision.

Experimentation and evaluation have produced a number of ideas for development in future versions. Firstly, we would like to extend capabilities of the user interface to the system. One way this will be done is to allow users to describe their requirements in the form of simple sentences or phrases, the system questioning the user over any ambiguities of other problems.Secondly, the interface will be extended to make full use of the window-based graphics of the SUN workstation.

We intend extending the descriptive capability of the software function frame by storing modifiers with their related nouns and verbs. We have shown how this might be done above with the example of a 'C' compiler. Another example might be the type of sort a sort component performs, e.g. quicksort, heapsort, etc. Furthermore, we aim to increase the 'knowledge' built into the system by storing sub-functions of typical software components as well as their primary function. Examples might be the editors have search and substitution capabilities, navigation systems have sub-systems for calculating latitude and longitude, and so on.

Another possibility is to add learning capabilities to the system. Learning capabilities would make use of feedback information from the user regarding the success of failure of the cataloque to satisfy his/her requirements. This would involve the introduction of a weighting system where components were attributed a weight and a function, the weight indicating the degree to which the component was described by a function. On a failure to satisfy a user, this weighting would be reduced; on success, it would be incremented. Furthermore, if a user specifies an orginal query which is unsuccessful and then a further query retrieves interesting components, a relationship could be set up between the original query and the components that were eventually retrieved. This relationship would have a low weighting which would be increased if the process was repeated by another user.

ACKNOWLEDGEMENTS

The work described here was funded by the Alvey Directorate, UK. Thanks are due to our collaborators in the ECLIPSE project, namely Software Sciences Ltd, CAP Industry Ltd, Learmonth and Burchett Management Systems Ltd, The University of Lancaster and the University College of Wales at Aberystwyth. Thanks also to Roy Davies and Professor John Campbell for their constructive comments on the first version of this chapter.

REFERENCES

Alderson, A., Falla, M. E. & Bott, F. (1985) An overview of ECLIPSE. In: McDermid, J. (ed) *Intgrated Project Support Environments*. London, Peter Peregrinus.

The Alvey Report (1982) A programme for advanced information technology. The Report of the Alvey Committee. London, HMSO.

Boehm, B. W. (1979) Software engineering. R. & D trends and defense needs. In: Wegner, P. (ed) *Research directions in software technology*. Cambridge (Mass.), MIT Press.

Boehm, B., Penedo, M. H., Stuckle, E. D., Williams, R. D. & Pyster, A. B. (1984) A software development environment for improving productivity. *Computer* **17** (6), 30–42.

Lehman, M. M. (1980) POrograms, life cycles and the laws of software evolution. *Proc IEEE* 68 (9) 1060–1076.

Lehnert, W. G. (1978). *The process of question understanding*. Lawrence Erlbaum Associates, Inc.

Prieto–Diaz, R. (1983) Knowledge representation applied to library cataloging. In: Perrotti, S. (ed), *Anais do XVI Congresso National de Informatica, SUCESU, Sao Paulo, Brazil, October 1983*, pp. 42–47.

Prieto-Diaz, R. (1985) A software classification scheme. Ph.D. thesis, University of California, Irvine, USA.

Ritchie, D. M. & Thomson, K. (1978) The Unix time-sharing system. *Bell Sys. Tech., J.* **57** (6) 1991–2020.

Schank, R. C. (1972) Conceptual dependency: a theory of natural language understanding. *Cognitive Psychology* **3** 552–631.

Waltz, D. L. & Goodman, B. A. (1977) Writing a natural language data base system. In: *5th International Joint Conference on Artificial Intelligence*, pp. 144–150.

2

Library manager: A case study in knowledge engineering

Helder Coelho, Centro de Informatica, Laboratorio Nacional de Engenharia Civil 101, Av. do Brazil, 1799 Lisboa, Portugal

1. INTRODUCTION

Early computers had fairly limited power and were difficult to program. Since that time, hardware technology advances have increased the computational power of the typical computer by several orders of magnitude. Some of this additional computing power has been used to make computers easier to program by developing assemblers, compilers, operating systems, and so on. Meanwhile, effective Artificial Intelligence (AI) techniques have been developed, mainly in the last 20 years. These techniques work in practice; but the resources required, particularly processing time and memory, were until recently prohibitive for many real-world applications (Duda *et al.* 1980). With the falling cost of processing power and memory capacity, this is no longer the case. The R&D programmes on fifth generation computers, under progress from 1982, also opened new ways of knowledge processing.

Moreover, recent advances in many fields suggest a new role for AI, quite different from the traditional view. Two key notions of this new role are the conceptual interface between programs and human users and the idea of knowledge refinement. Programs have been developed that can apply specialist knowledge at least as well as the human expert.

This chapter will examine the actual relevance of knowledge engineering for the future of informatics, by illustrating it with a case study drawn from the work of the Computational Logic Group at LNEC.

2. ADVANCED INFORMATION PROCESSING: KNOWLEDGE ENGINEERING

Answering questions about a specific knowledge domain is the first step taken by a human being learning to become an expert in the art of problem-solving in that domain. The same method is chosen when designing and building systems that enhance such abilities and become helpful problem-solvers and consultants.