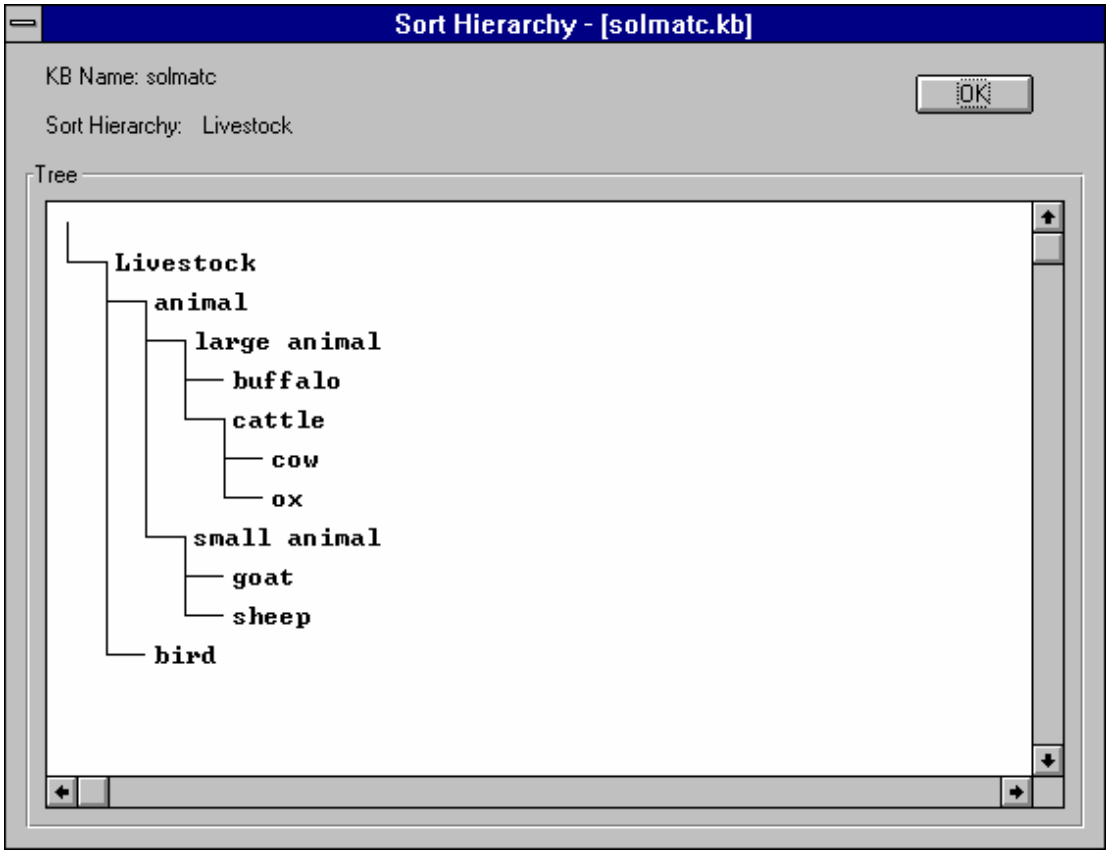
WinAKT Version 1.00

software description and user notes



David A. Randell and Fergus L. Sinclair

School of Agricultural and Forest Sciences
University of Wales, Bangor

Project R6322. Department For International Development. Forestry Research Programme

WinAKT Version 1.00

software description and user notes

David A. Randell and Fergus L. Sinclair

School of Agricultural and Forest Sciences
University of Wales, Bangor

Project R6322. Department For International Development.
Forestry Research Programme

Contents

March 1997

WinAKT v1.00

This document is a description of WinAKTv1.00 - the **A**groforestry **K**nowledge **T**oolkit for Windows as it stands at the present time. The software is still under active development and so this should be viewed as a draft. The operation of the program is described here, for information on how to use it to acquire knowledge, please refer to the first six parts of the manual for AKT1 and AKT2 - the methodological guidelines for the initial Apple Macintosh version of the software that was a precursor of the present program (Walker et al, 1994).

WinAKT is software for creating knowledge-based systems (KBS). Knowledge collected from people and documents (sources) about a specified subject (the domain) is represented as a set of statements. A series of hierarchies that describe relationships amongst terms used in the statements is also generated. A task language allows the user to manipulate the information stored in the knowledge base using automated reasoning procedures. The result is a powerful tool for acquiring and using qualitative knowledge about agroforestry. A brief overview of the functionality of WinAKT is given in Appendix B.

1 Introduction

WinAKT v1.00 is implemented in LPA WinPROLOG v3.30 and will run under either Windows 3.x, Windows 95, or Windows NT. In terms of general performance this program was developed to run on portable computers with at least a 486-DX processor and 16MB RAM. Further details of machine settings and the installation and running of WinAKT are given in Appendix A. New users should be able to find their way around the program simply by using the supplied knowledge base and systematically going through each section below. A general overview of the history, rationale and methodology used in the design of the program appears in Appendix B.

2. Preliminaries

In this document knowledge base program files are identified by the file type: *.kb. Menu items that appear in WinAKT are indicated by the use of italicised and underlined text, thus, *File* denotes the file menu on the main menu bar on the top-level window in WinAKT. Similarly, paths identifying sub-menus and sub sub-menus, are indicated by the use of the forward slash symbol "/", thus e.g. *File/Open KB* points to the sub-menu *Open KB* under *File*. Explicit references to buttons used in WinAKT's suite of windows are indicated in the text by enclosing the button name in paired-square parentheses, e.g. '[OK]' and '[Cancel]'. It is assumed that all the program files supplied with the WinAKT installation disc have been correctly written to the HD of your machine (see Appendix A for information on installation).

3. WinAKT

On boot up, the welcome window for WinAKT is displayed; if the user selects [OK], the program continues to load; eventually displaying the main window. If on the other hand, the user selects [Cancel] the program is immediately aborted.

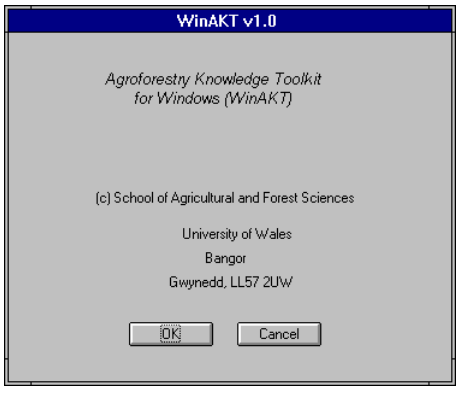


Figure 1: WinAKT's welcome window

On selecting [OK] the welcome window is replaced with the main window - this window remains as the top level window until the user exits from the program.

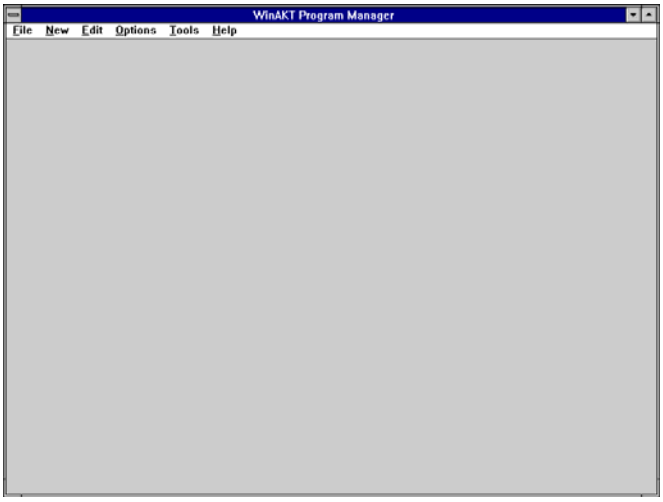


Figure 2: WinAKT's main window and menu options

The main menu of WinAKT's top-level window breaks down into six parts: File; New; Edit/View; Options; Tools and Help; with each part further subdivided again. The top-level rationale underlying this functionality is as follows:

File performs top level operations on individual KBs themselves, e.g. opening, re-naming and closing KBs. *New* deals with the direct editing of selected functions on an opened KB e.g. creating a new statement, or sort hierarchy. The same functionality is also available under *Edit*, but with the reduction of steps needed via this route. *Edit* deals with operations on selected KBs, e.g. viewing and/or editing sets of statements, sources, and other parts that together form a KB. *Options* provide functions that apply to specified KBs, e.g. changing global defaults for different search modes available to the user. *Tools* again apply to individual KBs; in this case running a general query on the contents of the KB, searching on a particular formal term or source used in the KB, or running specific macros on selected KBs. Finally, *Help* provides the usual on-line help utilities found with Windows programs, including the formal grammar used in WinAKT's formalised knowledge representation language and listings of the pre-defined macros available to the user.

3.1 The File menu

The *File* Menu breaks down into 13 separate items. Each menu item has the following functionality:

New KB	create new KB
Open KB	open existing KB file from HD or floppy
Save KB	save currently selected KB under existing name
Save KB as	save currently selected KB under another name

Loaded KBs	displays loaded KBs
Pending KBs	displays new or loaded KBs that have been changed since loading
---	
Select KB	select new or loaded KB
Current KB	display currently selected KB

Page Set Up	
Print Review	
Print	print the content of any WinAKT text/graphics win
Write	writes the content of any WinAKT text/graphics window to a *.doc file
---	
Exit	Closes down WinAKT

Most of these functions should be reasonably intuitive to understand and use in concept and implementation. The only distinctions that should be emphasised here are between: 'Close KB', 'Save KB' and 'Save KB as'.

New KB prompts the user to name a new KB. If a KB already exists using the same name, the user is prompted to re-name the new KB with a unique name. A maximum of eight characters are allowed for the filename prefix, while two characters (in this case the suffix 'kb') is reserved for the file suffix name.

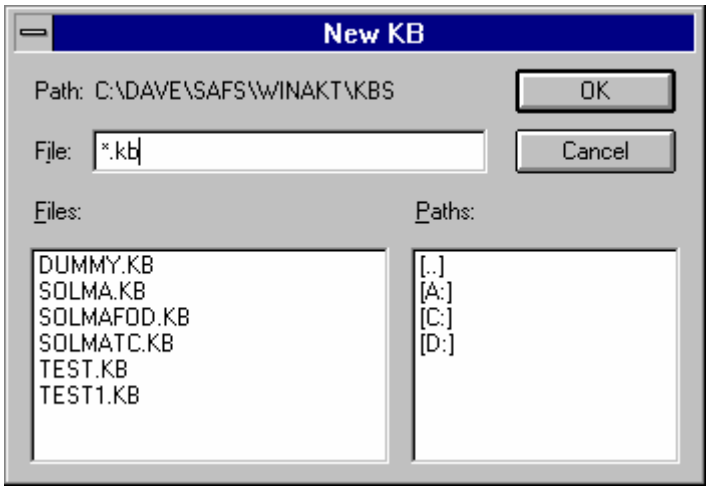


Figure 3: Creating a new KB

Open KB opens an existing KB from the HD or floppy.

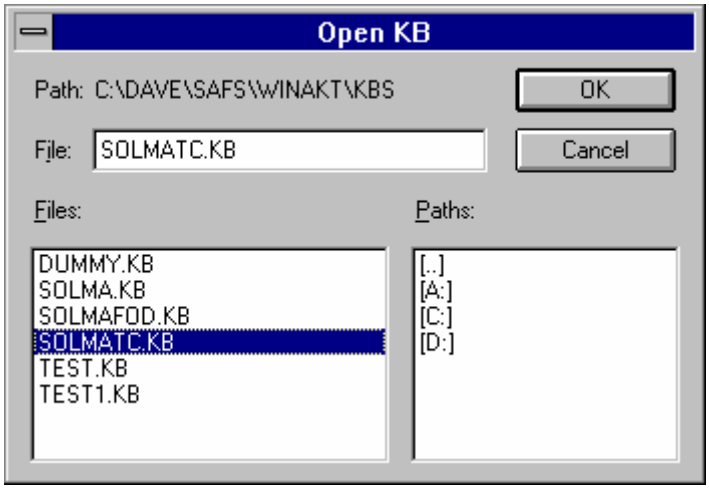


Figure 4: Opening a KB - in this case the KB solmatc has been selected to be opened

An existing KB is opened via *File/Open KB*. The user is prompted with a window that lists the *.kb files in the current directory. A double click on an entry will expand the directory in the normal manner. Selected KB's are displayed in the edit field, and a click on [OK] with a named KB displayed here will load the KB into WinAKT. On loading a window is displayed indicating that the loading is underway. On completion, this window is then replaced by a new Topic Hierarchies window (shown below) that provides the user with knowledge about the content of the knowledge base. In the event that the user has not saved information about specific topics in the KB, no topics are shown - as in the example window shown in Figure 5. If on the other hand, topics have been created and saved with the KB, these are duly listed. The user then has the option of either looking at the information relating to these

topics, or to bypass this and go directly to the main menu proper by selecting [Close], whereupon this window will disappear.

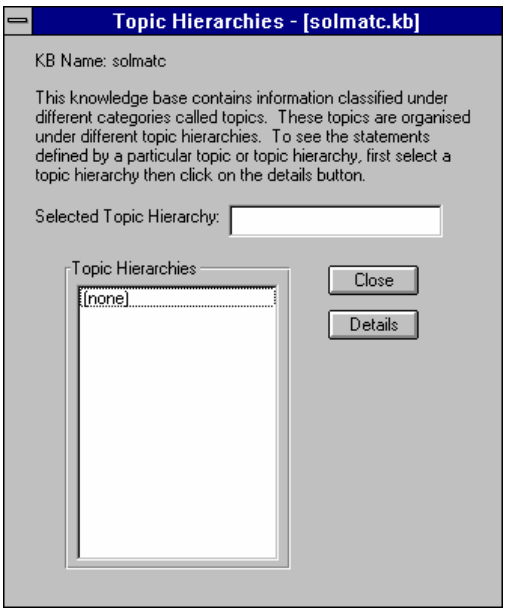


Figure 5: Topic Hierarches displayed after loading a KB

After closing this window, any subsequent exploration via topics is done via the main menu under Edit/Topic Hierarchies ...

Close KB closes an opened KB. That is to say the file will be saved to the HD or floppy; prompting the user where necessary when edits have been made and have not been saved. The user is given the option to close without saving; in that case the close down operation saves the file as it was up to the last save operation made by the user.

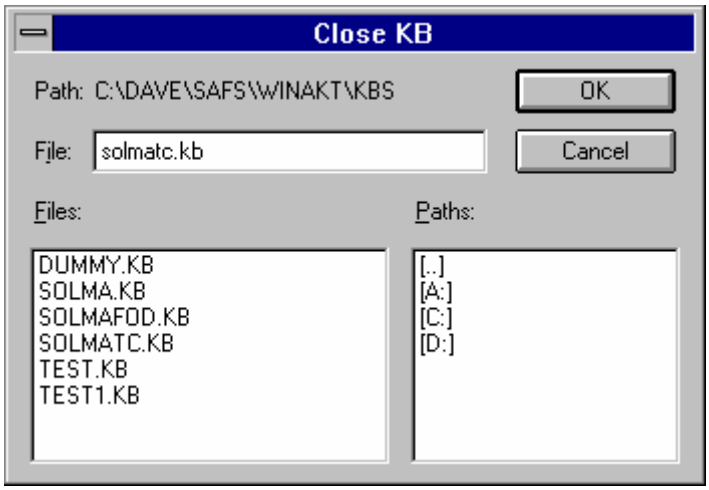


Figure 6: Closing a KB - with the knowledge base ‘solmatc’, selected

Save KB saves the currently selected KB under its existing name. *Save* updates the selected KB and overwrites the extant file.

Save KB as in contrast to *Save* saves any selected KB under another name supplied by the user. WinAKT has been designed to deal with more than one KB at a time. This means that the action of taking any selected KB and saving it under another name than the original

name, simply copies all the extant facts of the original KB to the new file, except that each of the new facts are now indexed to the name of the new rather than the old KB name.

WinAKT provides options on the type of save operation that can be done, e.g. whether or not any notes or memos associated with individual statements that go to make up the KB, are copied across in the process - as in some cases these may not be required. The options are dealt with in more detail below; however the important point is that by default any *Save As* operation copies all the contents of the selected KB to another file name, while keeping the extant file intact.

As the user opens KBs, these are stored in the Prolog database and their names stored in a list. The last opened KB dictates what the currently selected KB is. At any time this is immediately obvious to the user as WinAKT appends the currently selected filename to each specific window that the user opens. The same applies to all the windows that are subsequently opened while the selected KB is the current KB. Similarly, load and switch to another KB, and that new KB will dictate the changed window suffix, and any child windows thereof. Closing down opened KBs, sees a similar behaviour. Where other opened KBs exist the last opened KB will be the new currently selected KB, and so on, until no KBs remain loaded. In that case the main window returns to the title displayed on boot up. Having each window so named, ensures that the user at any time edits/saves any changes to the intended KB. It also avoids ambiguous named windows of the same type where, for example, direct comparisons are being made by the user.

3.2 The New menu

The *New* menu breaks down into nine items:

Statement
Source
Sort
Sort Hierarchy
Topic
Topic Hierarchy
Formal term
Diagram
Macro

The operations launched by each of these are replicated in functions defined under the *Edit* menu and are not discussed further here. The main point to bear in mind is that all these operations are operations on the currently selected KB. To edit a KB other than the one set as the current KB, the user must first select that other KB, then use these options. The New menu is provided as a short cut for the user, which is particularly useful when creating the KB during the initial knowledge acquisition process.

3.3 The Edit menu

The *Edit* menu breaks down into 19 items:

Undo

Cut
Copy
Paste
Clear
Select all

Find ...
Replace ...

Statements ...	displays all the statements in the currently selected KB
Sources ...	as for the sources
Sorts ...	as for the sorts

Sort Hierarchies ...	as for the sort hierarchies
Formal terms ...	as for the formal terms
Synonyms ...	as for the synonyms
Topics ...	as for the topics
Topic Hierarchies ...	as for the topic hierarchies
Diagrams ...	as for the diagrams

Current Source	displays the current source

Statistics	displays basic statistical information of the currently opened KB

Statements ... displays all the individual statements in an opened KB. A listing of the natural language translations of the formalised statements are shown, together with their internal numerical indexes. Above the window listing the statements are two windows that jointly displays both the natural language and formalised representations of the selected statement. From this window the user can pull up further details for any selected statement.

Immediately below we show a statements listing window of the *same type* that the user has called up via the Boolean Search tool (Figure 7).

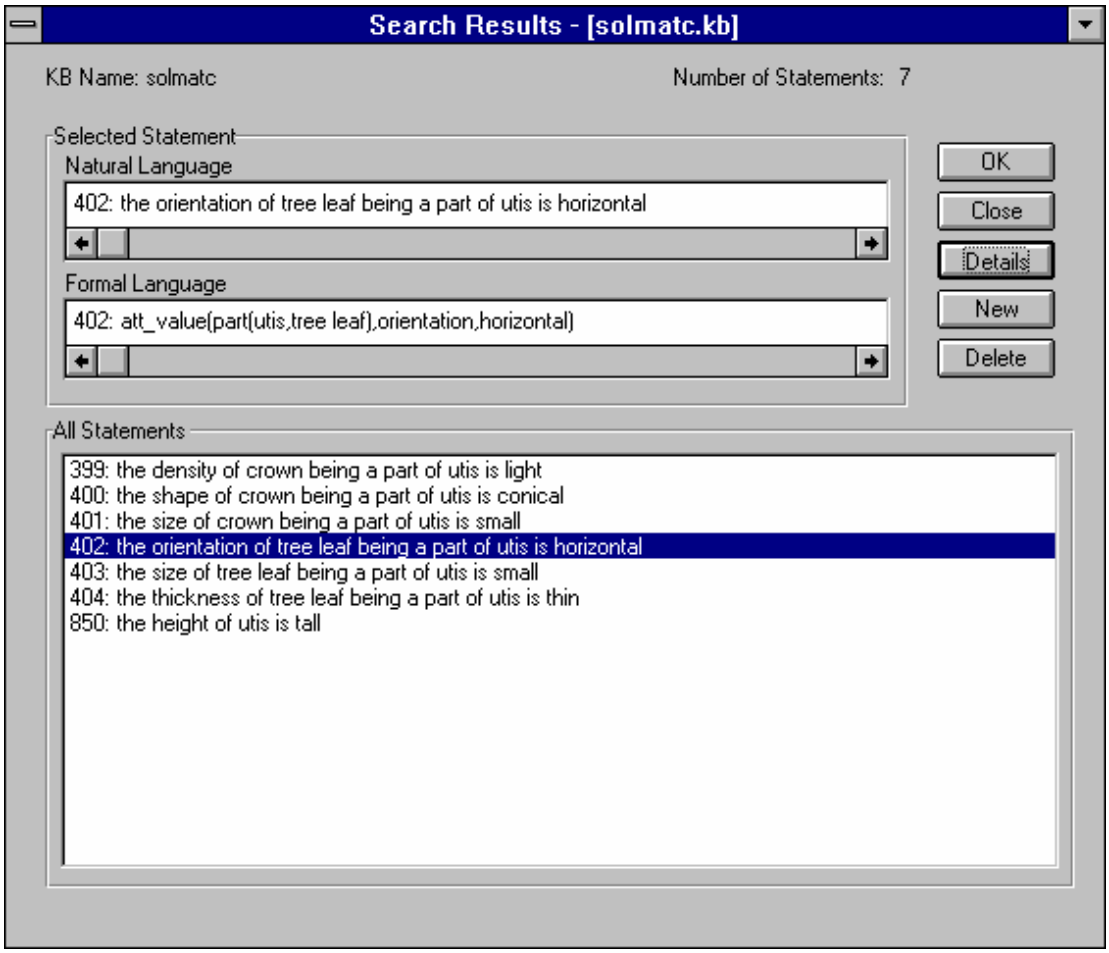


Figure 7: Displaying subsets of statements in a KB - with one selected

On selecting a particular statement, details can be called up via this window's [Details] button (Figure 8).

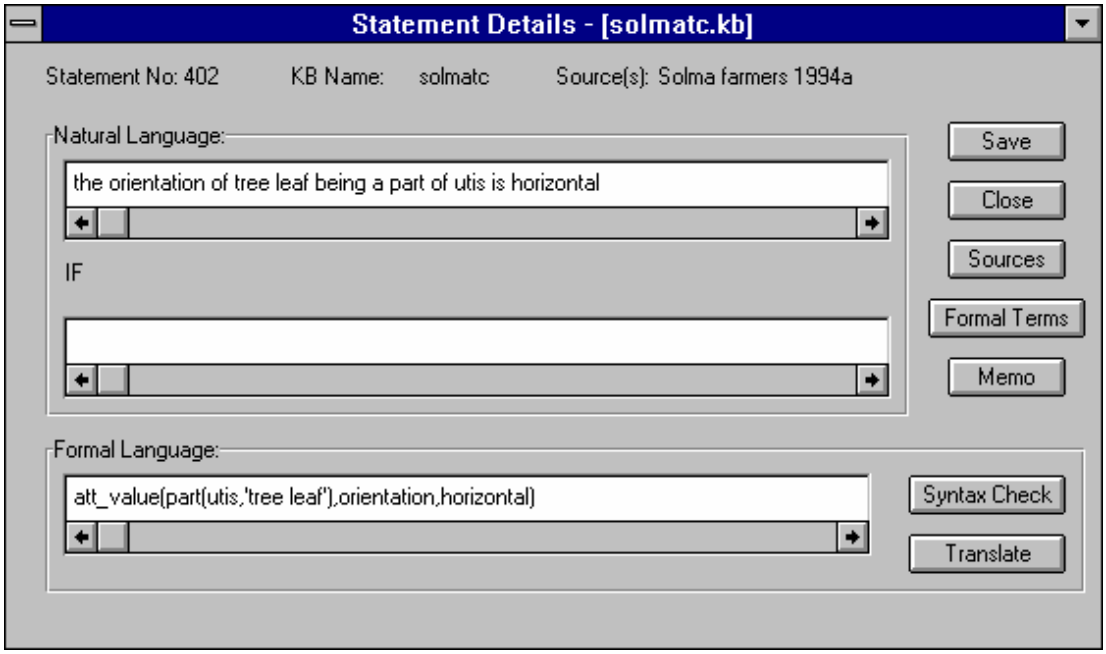


Figure 8: Displaying statement details - in this case a statement contained in the KB about the tree species utis - the one shown selected in Figure 7.

Sources ... displays the sources in the currently selected KB.

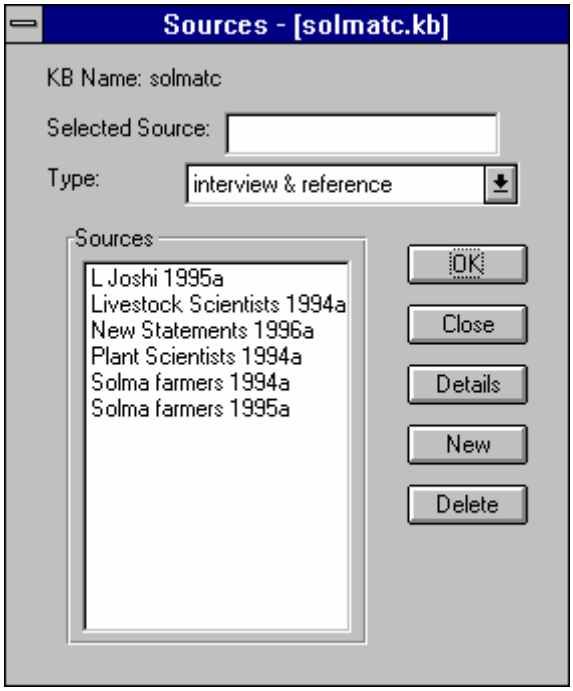


Figure 9: Displaying the sources

Two types of source are currently allowed for: interview and reference. The former is used to collate material typically garnered during an informal or formal interview session; the latter for collating information from documented sources. Each type of source maps to a different template with pre-defined edit fields. Sources may be edited either by creating a new template and filling in the edit fields, or by using [Details] to get the appropriate source details window, and then modifying it accordingly.

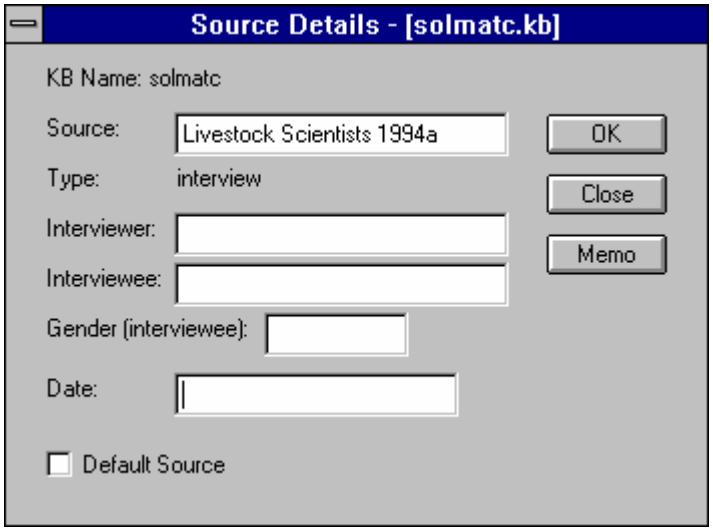


Figure 10: The interview source window

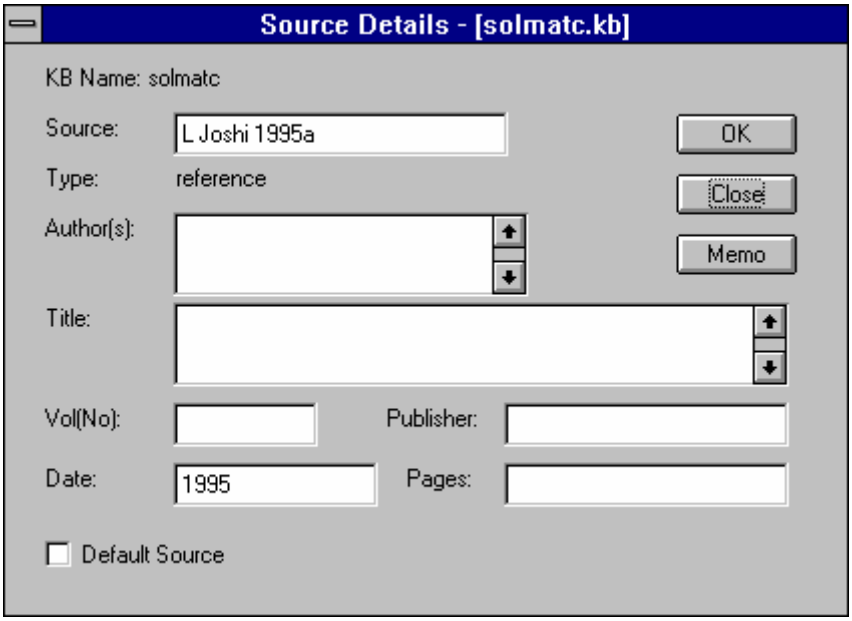


Figure 11: The reference source template

Sorts ... displays all the defined sorts in the KB. A sort is a formal term which represents objects (such as trees, oak trees or soil) in the domain. These sorts can be embedded in a tree structure called a sort hierarchy, which allows local taxonomic information to be represented (so that, for example, the fact that an oak 'is a type' of tree, is recognised). These hierarchies are treated individually and accessed under Sort Hierarchies ... The sorts window is functionally identical to that under Edit/Formal Terms ... and where the user selects the term type, sorts.

Sort Hierarchies displays the list of the set of named sort hierarchies of the currently selected KB. These sort hierarchy names are in turn the names of individual sorts, and as such appear in any listing of sorts for that KB. As named sort hierarchies, these special sorts are the most general sorts in the set of discrete sort hierarchies that are defined in the KB, i.e. each named sort hierarchy functions as the most general sort of some particular sort structure within the KB.

It is important to bear in mind that not every sort that is defined in the KB will appear in the sort hierarchies for that KB. Only sorts that are put into a supersort/sort/subsort relationship will appear in the list of hierarchy sorts. The important distinction here is made clear by distinguishing between the role of a sort that is a term in the formal language, and functioning of the term as a sort in an explicitly defined sort hierarchy. Any sort in a sort hierarchy is a term in the formal language, but not necessarily vice-versa.

Sorts displays the list of named sorts within the KB. These named sorts include any special sorts that are the named hierarchies - see Sort Hierarchies above. The list of sorts cover all the formal terms defined as sorts within the KB. Sorts displays the same information as that provided by formal terms and where sorts are specifically selected under the type 'sort'; given the special use to which sorts assume in WinAKT, these are given their own edit/view window.

When editing sorts in a sort hierarchy, sorts are neither created nor deleted from the KB but are simply appended or detached from that hierarchy. For this reason a clear distinction is made in WinAKT between the operations defined by the window buttons for [New] and [Delete], and [Append] and [Detach] as applied to sorts. New and Delete either adds or removes formal terms (sorts) from the KB; while Append and Detach operates on the specified sort hierarchy only, and leaves the sorts defined as formal terms in the KB. A similar rationale applies to sources within WinAKT, where named sources as formal terms are separated out from those that are used in a statement, with those that are not - see Sources. Sorts (formal terms) cannot be deleted from a KB if they are used in a statement within the KB, or appear in a defined sort hierarchy; if the user wishes to delete a sort from the KB, all the statements and sort hierarchies containing those terms must be edited first so that no use of that formal term appears in the KB.

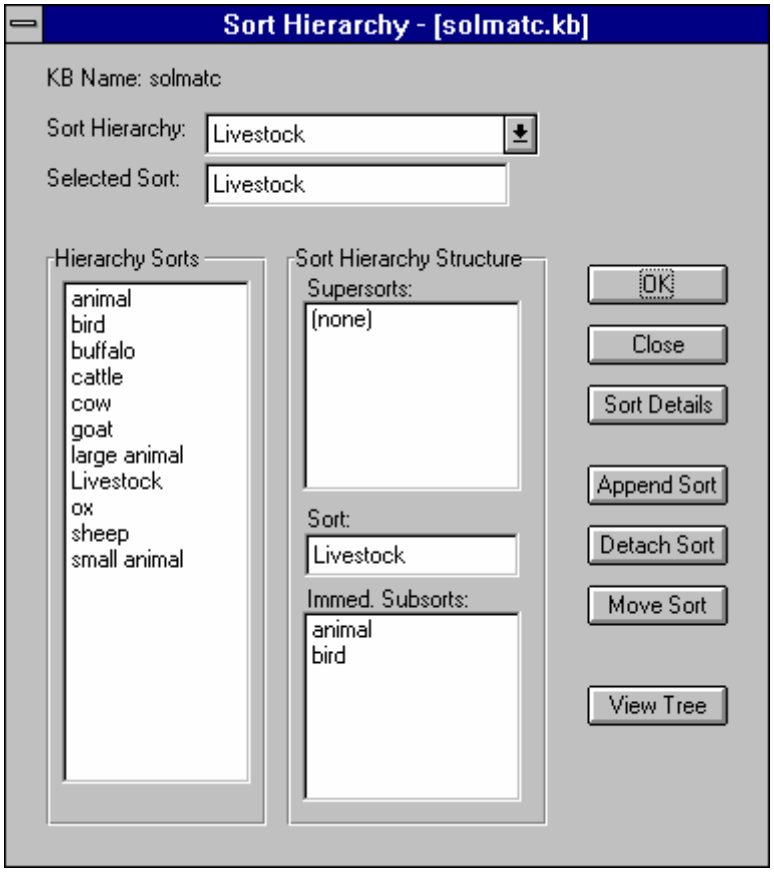


Figure 12: Displaying details for the selected sort hierarchy: Livestock

The user can get specific sort details for the selected sort by using the [Details] button, can navigate through the sort hierarchy by clicking on the chosen subsort or supersort in the Sort

Hierarchy Structure window, or alternatively, display the tree structure by clicking on [View Tree].

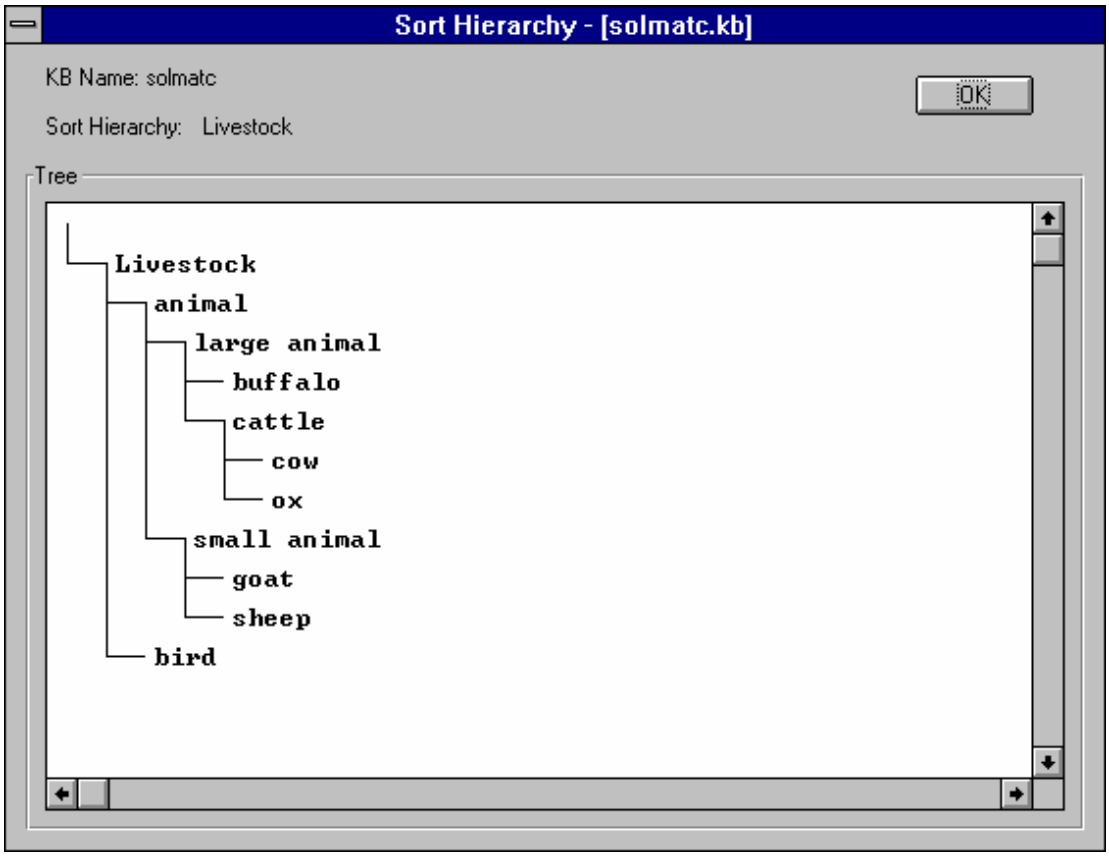


Figure 13: displaying the sort hierarchy as a tree structure.

Editing the sort hierarchy is also launched from the Sort Hierarchy window; pre-defined sorts can be appended, detached and moved within a specified hierarchy, by respectively clicking on the [Append Sort], [Detach Sort] and [Move Sort] buttons. In all these operations the user defines the new sort/subsort relation and the KB is updated accordingly. Options for editing existing sort hierarchies, i.e. moving or deleting either whole branches or singleton sorts, are provided.

Formal Terms This function lists all the formal terms defined by the KB. These terms subdivide into different types that can be selected by use of the drop-down list box. For example, by selecting; 'attributes' all and only the attributes used in the KB will be displayed. From this window the user can select a formal term and call up its details for viewing or editing as the case may be.

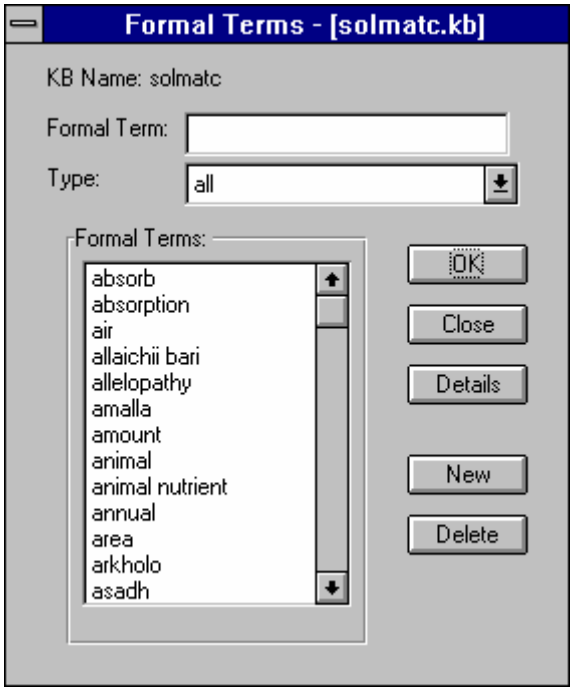


Figure 14: displaying the formal terms

Synonyms Synonyms are a special case of formal terms. Strictly speaking these are not generally used in the explicitly defined formalised sentences used to define the KB; rather they are generated from the formal term details window where a special edit widow is reserved for their use. Synonyms are most frequently used for mapping from scientific to local names of plants and animals, though synonyms can be generated for any formal term in a KB.

Figure 15: Displaying synonyms

Topics Topics are a way in which the knowledge base can be partitioned. A topic is simply a name attached to a search string (an 'alias'). When a topic is selected the search string that it is an alias for is used to select the set of statements defined by the string. The topics may

relate to subject matter (for example, all the statements in the KB relating to the 'feeding value of tree fodder' as opposed to all those relating to 'pests and diseases') or they may relate to different sources (for example 'farmers knowledge' as opposed to 'scientists knowledge') or to a combination of subject matter and source (such as, all the statements by women about tree-crop interactions). Topics are created via the *Tools/Boolean Search* menu function (described below) - where a search on a set of formal terms is defined, then assigned to an alias. These aliases are then picked up and re-worked as topics, and displayed accordingly.

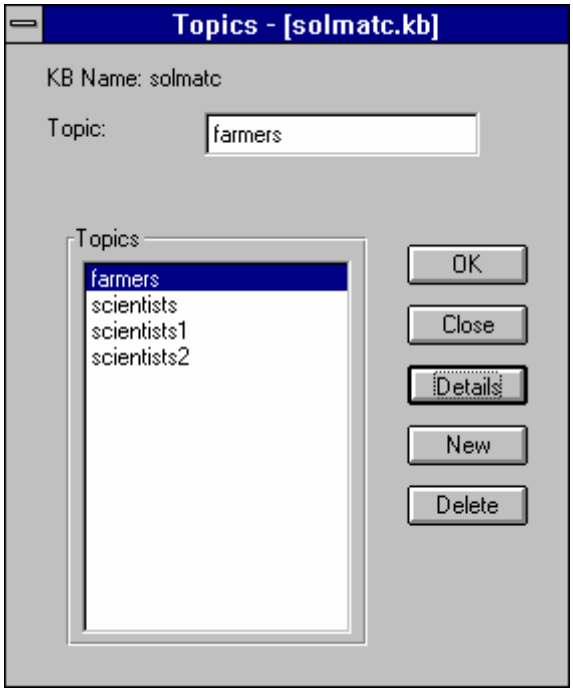


Figure 16: Displaying the topics - with the topic ‘farmers’ selected.

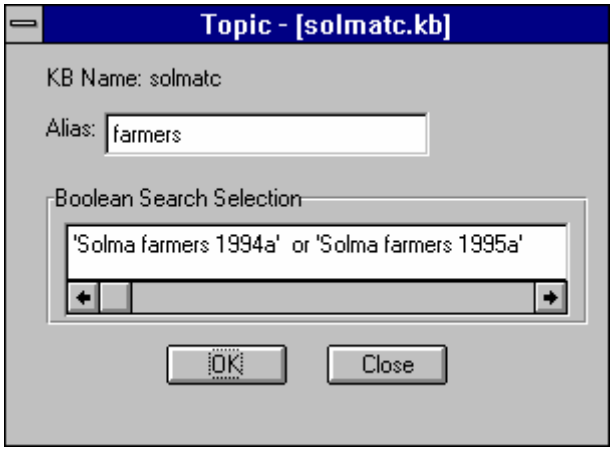


Figure 17: Displaying topic details - in this case showing the defined search string for the topic ‘farmers’.

Topic Hierarchies This menu function lists all the topic hierarchies defined in the KB. From here the user can in turn create new topic hierarchies or edit existing hierarchies, much in the same way as sort hierarchies are created and modified (see above).

3.4 The Options menu

Search Defaults This function lets the user determine the depth of search when collecting lists of statements in the KB. Three modes are available: (i) sort; (ii) sort and subsorts, and (iii) supersorts, sort and subsorts.

For a given target sort: a search on sorts will return all and only those statements with some specified sort in the KB; while sorts and subsorts will return any additional statements that use any subsorts of the target sort. The last case supersorts, sort and subsorts returns in addition, any statements that have sorts higher up the sort hierarchy for that specified. Where no supersorts or subsorts have been defined the returned set for either case is empty.

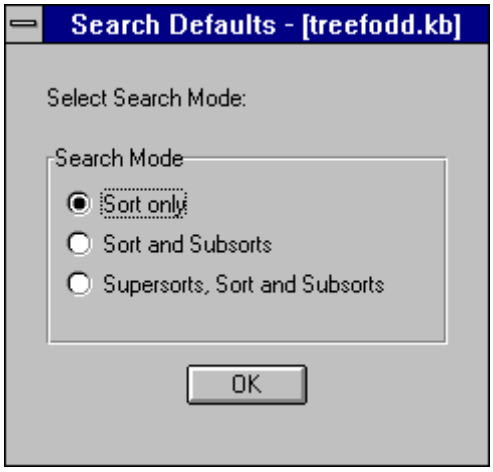


Figure 18: Setting the search default

3.5 The Tools menu

The Tools menu has three items:

Query	invokes the query tool
Macro	launches the macros/tools defined for the current KB

Boolean Search	Boolean (and/or) search on the formal terms of the current KB

Boolean Search The Boolean Search function allows the user to select individual formal terms and collect the set of sentences explicitly defined in the KB that use these.

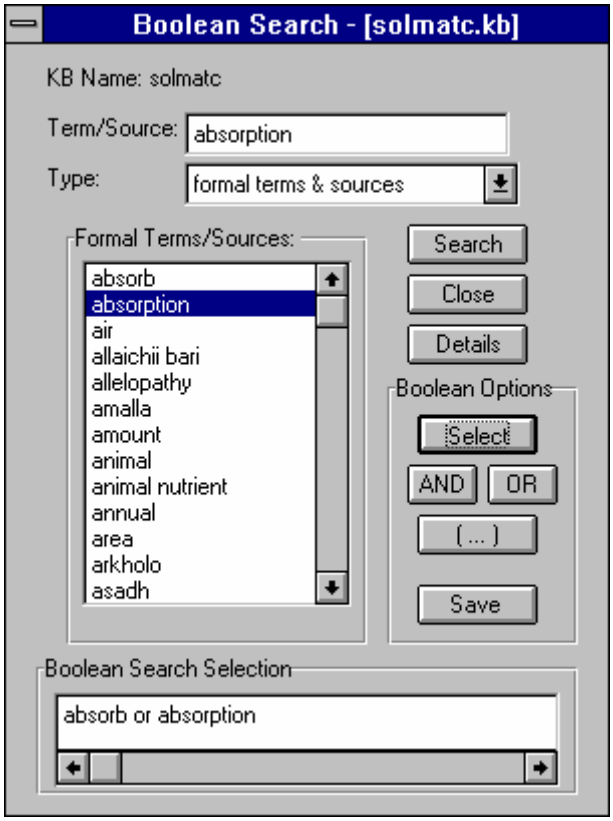


Figure 19: The Boolean Search window - in this case the user has chosen to search for all the statements in KB that either contain the term ‘absorb’ or ‘absorption (or both)

Boolean combinations of these terms can be constructed using Boolean and/or operations; where ‘or’ is defined as inclusive; meaning either or both. For example, the search string ‘tree or plant’ collects together all and only those sentences that use either the term ‘tree’ or ‘plant’ (or both); while ‘tree and plant’ collects all and only those individual sentences which use both terms. The user can create search strings of any complexity. The infix notation used assumes association of terms to the right hand side unless parentheses are used to force a different interpretation. For example, the logical meaning in English of ‘tree and plant or animal’, is ambiguous; since this could mean either (a) ‘(tree and plant) or animal’, or (b) ‘tree and (plant or animal)’. Since both interpretations are possible, and both apply to the same string, in the absence of brackets WinAKT allows only one of these potential meanings, that is (b). In practice then, within WinAKT the string ‘tree and plant or animal’ would be internally parsed as ‘tree and (plant or animal)’. If however, the alternative meaning is intended as in (a), the explicit use of paired parentheses is required in the search string itself by typing in the string ‘(tree and plant) or animal’.

As a guide, if only one type of Boolean connective appears in a search string - either only a number of 'and’ connectives or a number of ‘or’ connectives, then no explicit bracketing is required (since the logical meaning is unambiguous); but if these types of connectives are mixed in the same string, then for every pair of terms straddling either side of each connective, it is recommended that explicit bracketing is used to make the logical meaning clear.

After defining the search string this can be saved by selecting the [Save] button. A second window then appears which allows the user to assign an alias name to the search string.

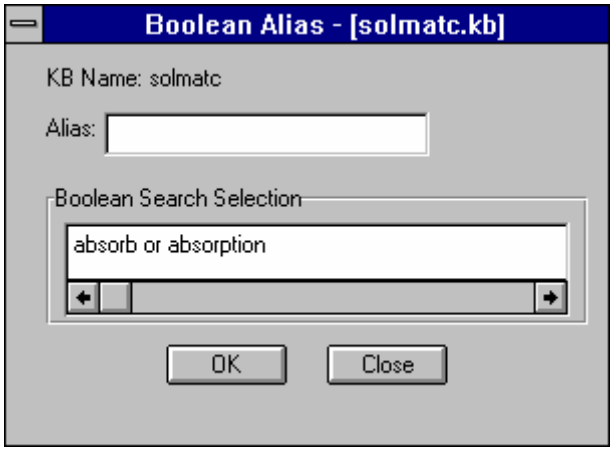


Figure 20: Saving the Boolean search string under an alias

After the alias is saved, aliases can picked up by selecting the alias option under the type drop-down listbox in the main window. Aliases can either used individually or functioning as parts of other defined search strings. Note however, that in both cases the selected alias is displayed in the Boolean Search Selection text window as the full Boolean search string, and not as the user defined name. This is intentional, and is used to remind the user exactly what that alias defines in terms of the KB.

Query the query tool is used to identify formal statements in the currently opened KB of a given type. A simple pattern matching operation is used to determine the degree of generality in the list of statements returned. For example, running a goal after typing 'X if Y' in the edit field of the main window, will match all the conditional statements in the KB of this type. In this case, the variables are defined by the use of upper-case letters, e.g. 'X' and 'Y'.

For any given query string, three solution options are available: [First] returns the first match in the KB; [Next] incrementally matches the next and subsequent entries in the KB - one per subsequent click of this button; and [All] generates all the solutions to the query. Where no solutions exist or no more solutions are returned for an explicit query, this is written to the output window.

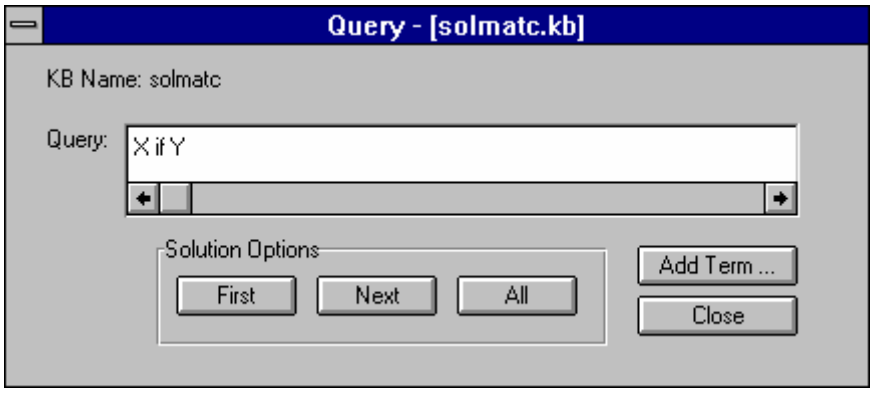


Figure 21: The Query window - here with a query about what conditional statements are contained in the KB, the answer appears in Figure 22 below.

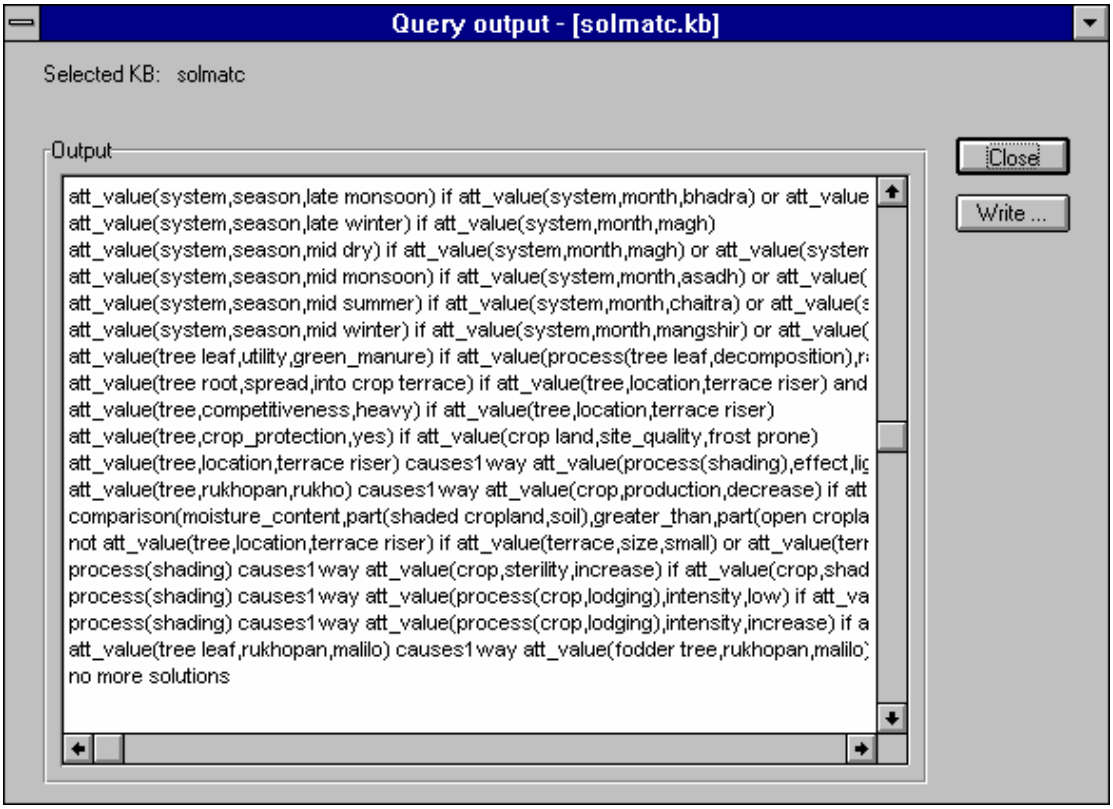


Figure 22: The query output window - displaying all solutions to the query ‘X if Y’

To get more specific instances from a given query goal, one systematically replaces variables with appropriate constants or functions as defined by the formal grammar. For example: typing in the goal string: ‘process(shading) causes1way Y if Z’ would collect all the statements shown in Figure 20 beginning with: “process(shading) causes1way ...” while typing in the string: ‘process(shading) causes1way att_value(X,Y,increase) if Z’ would only pick up the first of those statements.

Macro Macros are customised programs or 'tools' that use a set of pre-defined primitive functions and control structures (the task language) to make it easy for users to develop automated reasoning procedures for use with their knowledge base. The primitives and control structures are used to define the macros which perform useful tasks on a compiled KB. Some macros are supplied with WinAKT and these may be used as they stand or be modified by the user, who may also create entirely new macros. Allowing the user to define their own set of macros greatly increases the flexibility of WinAKT so that the program can be tailor-made to specific user requirements which may not be envisaged by the program developer.

There are three parts to the task language:

(i) primitive functions,
(ii) control structures, and
(iii) macros.

Some macros that perform some basic tasks that are frequently required by users are supplied with WinAKT. The user may define other new macros but only the software developer, having access to the original source code, is in a position to define further primitive macros and control structures.

Defining Macros

Macros are defined via the Macro interface. The user types in the name of the macro, the inputs and outputs and the head (or defining term) and body (the definition itself) of the formal definition. The macro is tested, then saved. Once saved the defined macro becomes immediately available for use, either on its own or appearing in the body of the definition of another macro.

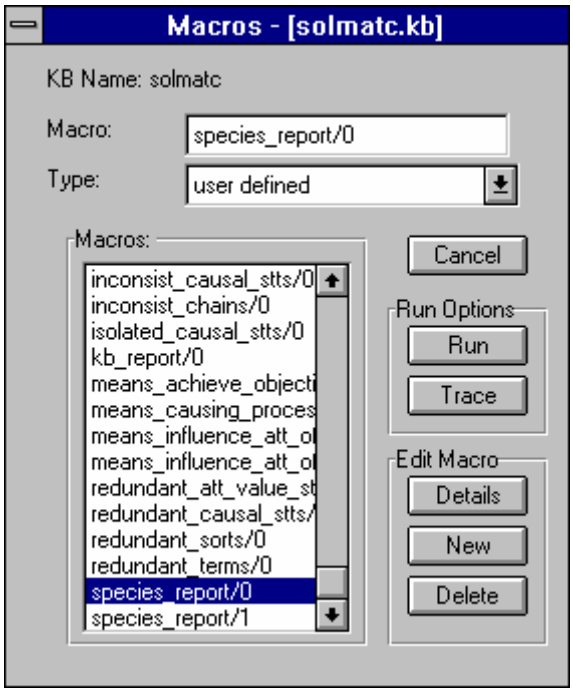


Figure 23: Displaying (user-defined) macros - with the macro called 'species report/1' selected

The writing of macros is relatively straightforward; the only part that the user must attend to with some care is the writing of the definition of the macro. For this a simple formal language has been specified in which the definition is written.

Grammar and syntax for the Macro Language

The formal language used to write definitions uses a finite vocabulary of characters derived from the English alphabet {a,b,c, ...,z,A,B,C,...,Z}, the set of positive integers {0,1,2,3,...,N-1,N) together with punctuation markers and other special characters. The latter includes paired rounded parentheses '(' and ')'; the squared parentheses '[' and ']'; the comma ',' and the period mark '.', while the special characters include the underscore symbol "_" and the hyphen "-". Other special characters derived from the standard ascii set of characters can also be used, but typically these will only be used in quoted strings - see below.

Terms are defined to be concatenated sets of upper/lower case letters, with or without underscores. For example 'a', 'string' , 'String' and 'A_string'. are terms. Terms are either constants or variables. Arbitrary sets of concatenated characters defined above are called strings. Thus any term is a string, but not vice-versa.

Lists are defined to be any sequence of terms separated by commas enclosed within paired square brackets; thus e.g. '[]', '[a]', '[a,b,c]' are lists with respectively zero, one and three elements.

Variables are terms whose first character is an upper-case character or the underscore symbol. Terms beginning with a lower-case letter are constants. Variables are place holders for variables or constants.

Macros have a predicate name and a set of arguments - defined below. Arguments can be macros, constants, lists or variables.

A macro is written as a function as follows: Name(Arg1,Arg2,...,ArgN-1, ArgN), where Arg1,Arg2,...,ArgN-1, ArgN are the arguments and where N is a non-negative integer. In the case where a macro has zero arguments (i.e. no explicitly defined inputs and outputs to the function) the macro collapses to a constant. Thus e.g. 'macro', 'macro(X)', 'macro(X,Y)' are respectively declared macros of zero, one and two arguments - with the first named macro here functioning as a constant. For brevity these are formally defined as follows: 'macro/0', 'macro/1' and 'macro/2', using the Name/Arity convention - where the arity referred to here is simply the number of arguments for a specified function. This Name/Arity convention is used throughout this manual and is used as the name of macros used in WinAKT.

One other point of nomenclature: when a variable is substituted either from within the program, or by the user by a different term, we say the variable has been "instantiated". So, for example, given the macro ask/2: ask(Prompt,String); if the user typed in the string :"Enter the name of the KB that you want to inspect." for the variable Prompt, this would be represented in the program as: ask(`Enter the name of the KB that you want to inspect`,String) with variable Prompt now instantiated. As the macro is run, variables are automatically instantiated according to the nature of the inputs and outputs defined for that macro.

Writing the definition to the Macro editor window

Each line of code in the definition must be terminated with a comma (",") except the last line of code which is terminated with a period mark ("."). After typing in the complete definition, type <Enter> (thereby placing the text cursor on a new line) before saving the macro.

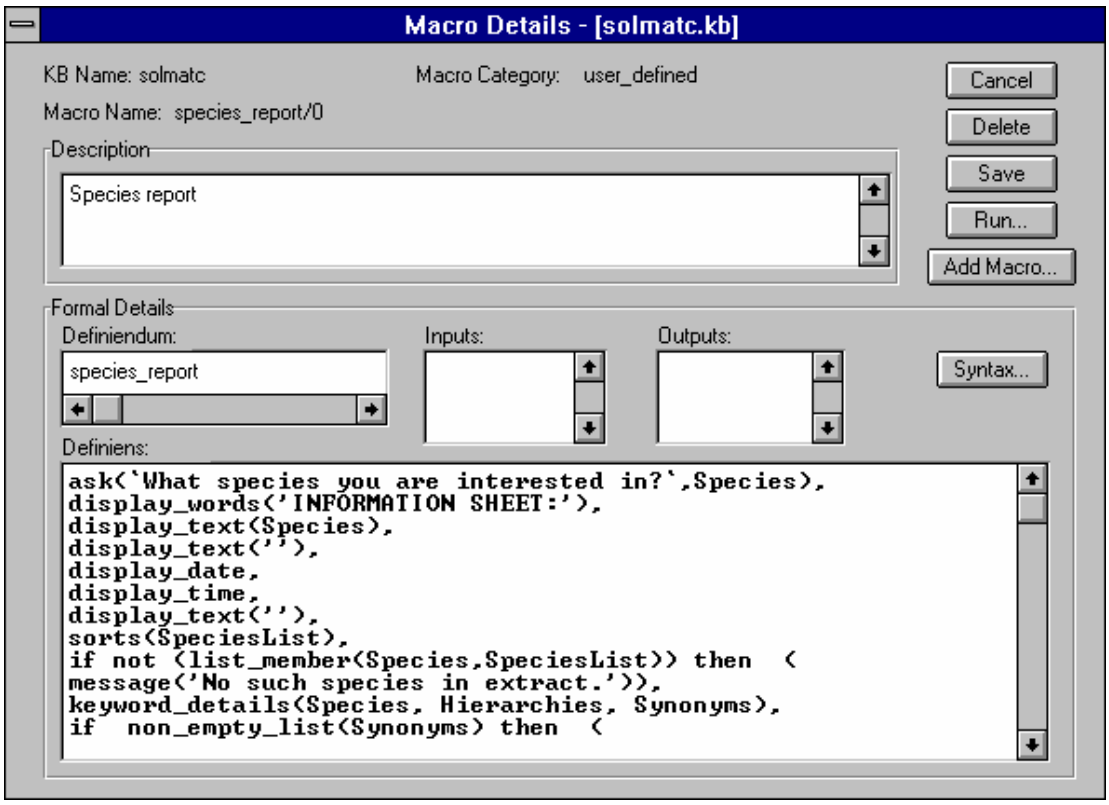


Figure 24: The macro details window - for the selected macro *species_report/0*

Code comments ("/* ...*/") can be added to the definition either immediately following individual lines, or between lines of code; though NOT within a defined function. Comments are inserted between the two paired symbols "/*" and "*/" written in this sequence. These

characters and any string appearing between them is ignored by the program, though the comments themselves are internally stored with the macro definition for future inspection and will re-appear when the user calls up the Macro Details window for that macro. An additional comment marker is the single character “%”. Appearing in a line in the definition, it has the effect of ignoring any characters on that line after (and including) the symbol - see the example definition shown immediately below.

The following shows an example of a macro definition - in this case the complete definition for *species_report/0* - cf Figures 23 and 24.

Definition	Explanation
ask(`What species you are interested in?`,Species),	% prompt user for a species
display_words('INFORMATION SHEET:'),	% print sheet header
display_text(Species),	% print species
display_text(''),	
display_date,	
display_time,	
display_text(''),	
sorts(SpeciesList),	% get list of sorts
if not (list_member(Species,SpeciesList)) then (	% if sort is not in list then
message('No such species in extract.')),	% print message to this effect, o’wise
keyword_details(Species, Hierarchies, Synonyms),	% get the sorts hierarchies and synonyms
if non_empty_list(Synonyms) then (	
display_words('Synonyms :'),	
display_text(Synonyms),	% print synonyms (if any)
display_text('')),	
foreach Hierarchy in Hierarchies	% for each hierarchy
do (	%
path(Hierarchy, Species, Path),	% get the complete path (branch) to the sort
foreach Level in Path	% then for each sort in a branch
do (	%
display_text(Level)),	% print sort
display_text('')),	
statements_about1(Species,all,Species_att_values,all_relevant),	% get all statements about the species
translate_statements(Species_att_values, English_att_values),	% translate formal to natural language
display_list('', English_att_values),	% ouput translated statements
list_length(English_att_values,Number),	% calc number of statements
display_text(''),	
display_words('Number of statements about '),	% print summary
display_words(Species),	
display_words(' in this knowledge base '),	
display_words(': '),	
display_words(Number).	

Notice the use of single line code comments shown here. One can also use indenting associated with paired parentheses "(" and ")" to enable the formal definition to be read and hence understood more easily.

Consistency and clarity in writing definitions is strongly recommended, not only to help other potential users who may which to understand any macros defined, but also for the person writing the macro to keep a clear record of what they are intending. It is vitally important that a top-level informal but clear natural language description of the macro is contained in the macro definition. This is necessary if the macro needs to be de-bugged or modified at any point in time. Passage of time will invariably result in the meaning of inadequately documented macros being forgotten making the macro difficult or impossible to use.

Testing Macros

Once defined, several checks must be done before the macro can be added to the set of extant macros and used correctly.

Syntax errors

The first check is to check the macro's syntax or grammar. This check simply makes sure that one has the necessary information input regarding the macro name, specified inputs and outputs, and that the text entered forming the combined head and body of the defined macro as a valid program is recognised as such. However, it is important to realise that that is all

that the syntax check does. If the syntax check succeeds, it does not follow that the macro when run or that it will will successfully terminate. For example, the macro may call a function that calls another function (not explicitly appearing in the macro's formal definition) that is not defined. In this case the syntax check would succeed, but the macro when run would fail with a corresponding error message saying that an undefined function had been detected.

Run time errors - "u" and "f" - the trace tool

To identify where problems other than syntax errors occur in a defined macro, a trace tool is made available to the user and is used for diagnostic purposes. As the name suggests the trace tool simply traces through the macro's definition line by line. In so doing the macro is unpacked and displayed as a series of indented lines of code in the output window. When a macros calls another macro, the output displays the depth of the calls being made by indenting to the right in a tree-like structure that develops in a top/down and left to right pattern.

Undefined macros

Any undefined function detected by the trace is marked with a "u" appearing on the same line where the undefined function is first called by the program. The user can then see where this function is called, and identify the macro that contains that undefined function.

Normally undefined macros will not be encountered by the user, since macros used will be selected from the defined set that has been loaded into the application. However, undefined macros can arise if, for example:

(i) the user mistypes the macro name, or;
(ii) fails to identify the macro from the loaded set when typing in a new or modified macro definition, or;
(ii) the user preferring a top-down approach to writing macros, decides to write undefined macros in the body of the definition first, with the intention of fleshing out any undefined macros later.

In the case where a top-level call to a macro fails with that macro registered as being undefined, the name/arity of the macro is not written out to the output window. With all other cases the name/arity of the macro is written out together with the trace tree.

Failed computation

Alternatively, a macro can fail to terminate successfully because of some error in the type of information being passed between functions when the macro is being run. In this case the intended computation will fail, but whereas before the trace displayed a "u" to indicate the presence an undefined function, the trace displays an "f" indicating the point where the failure occurs.

In this particular case the macro has been identified, and hence is defined, but a mismatch between the definition of the macro and the information being input into that macro has arisen during its running. As before, the user can then identify the macro where the problem lies, only in this case the user will need to inspect any instantiated variables in the trace, and compare these to the defined macro in order to identify the nature of the error.

Running Macros

Macros can be run from the main macro listing window (Figure 23) by selecting a macro then clicking the [Run] button; similarly, to trace through a macro one uses the [Trace] button.

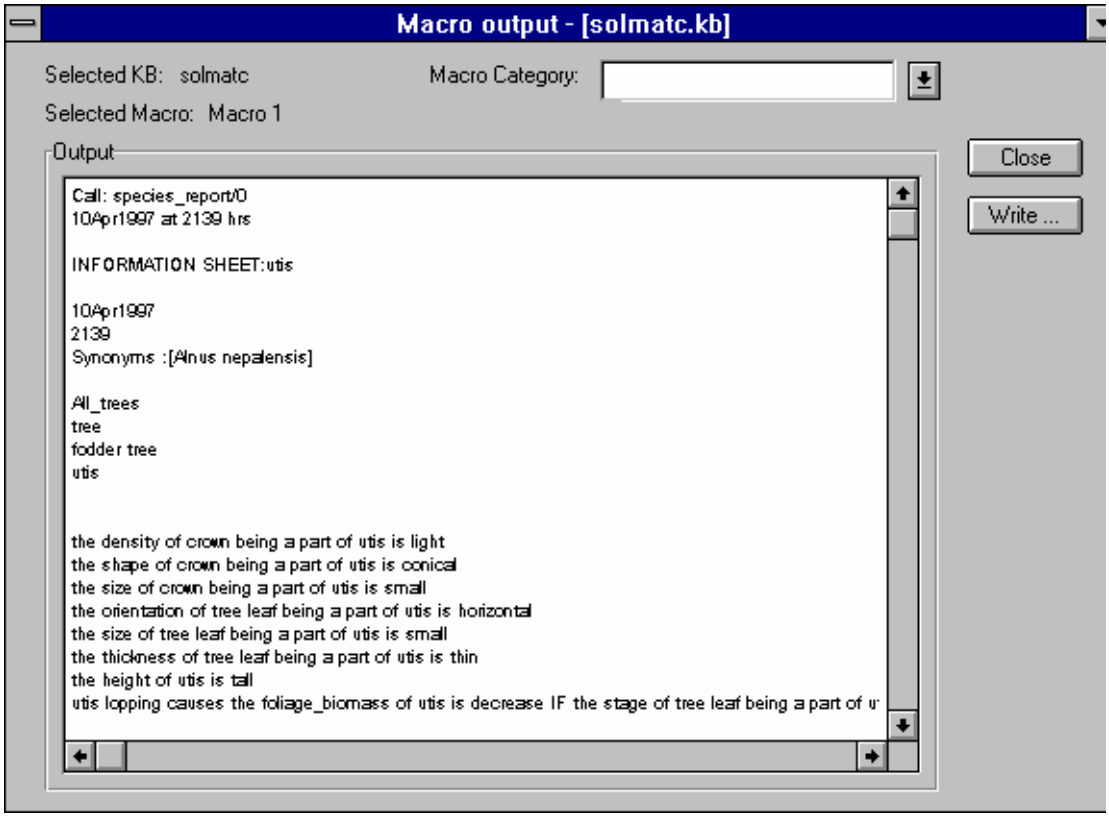


Figure 25: The macro output window - running the macro species_report/0 for the tree species: utis

Saving Macros

Displayed macros are saved by clicking the [Save] button in the Macro Details window. While it is recommended that each macro when saved is checked for run-time errors first, the save operation does not enforce this condition. The only check enforced in that the definition passes the syntax check. However, in the case where a macro is being defined and the user returns a syntax error that cannot be attended to immediately, the macro can still be saved by placing the entire definition of the macro within the commented out characters, e.g.

```
/*
ask(`What species you are interested in?`,Species),          % prompt user for a species
display_words('INFORMATION SHEET:'),                         % print sheet header
display_text(Species),                                       % print species
display_text(''),
display_date,
display_time,
display_text(''),
sorts(SpeciesList),                                          % get list of sorts
if not (list_member(Species,SpeciesList)) then  (            % if sort is not in list then
message('No such species in extract.')),                     % print message to this effect, o'wise
keyword_details(Species, Hierarchies, Synonyms),             % get the sorts hierarchies and synonyms
if  non_empty_list(Synonyms) then  (
*/
```

Note the nested commented out characters in the above example. In this case the whole of the partly defined macro is commented out. Obviously the macro cannot be run, but can be subsequently called up by the user, the main commenting out characters removed, and the definition completed.

For the save operation to succeed, the user must have correctly specified the name, the inputs/outputs, the description of the macro, and the definition itself. In the event that any of these conditions are not satisfied, the user is prompted accordingly.

Saved macros are added to the set of defined macros and are automatically appended to the list of macros displayed in the Macros window.

Relation to Prolog

The macro programming language sits on top of the Prolog language used to implement WinAKT. As such certain conventions common to Prolog track across to the macro programming language itself and will be immediately recognisable to any user familiar with this particular developmental language. However, there are some clear differences: the main being the procedural non-recursive nature of the macro programming language adopted.

3.6 The Help menu

The Help menu follows the general format for Windows help files and is therefore not discussed further here.

4. Reference

Walker D H; Sinclair F L; Kendon G; Robertson D; Muetzelfeldt R I; Haggith M and Turner G S (1994): *Agroforestry Knowledge Toolkit: methodological guidelines, computer software and manual for AKT1 and AKT2, supporting the use of a knowledge-based systems approach in agroforestry research and extension*, School of Agricultural and Forest Sciences, University of Wales, Bangor.

4 Appendices

Appendix A: Installation and general machine recommendations

As WinAKT is written in Prolog, the usual recommendations for optimum performance for this developmental language applies. In this case machine specifications should favour larger quotas of RAM over the type and speed of the processor, rather than vice-versa.

Installation
WinAKT comes with the following compressed program files:

(i) winakt.exe
(ii) winakt.exp
(iii) winakt.ovl
(iv) winakt.ini
(v) winmem32.dll

(vi) ctl3d.dll

(vii) solma.kb

(viii) winakt.doc - the user manual - this document.

Of these (i) to (v) are the program files for WinAKT itself; (vii) is a sample demonstration KB; while (vi) is a Windows dll file that replaces older versions of this dll file known to create problems when running WinAKT - see section 1.3: ctl3d.dll - below. You will need to copy (i) to (v) to your hard disc (HD); these must be kept in the same directory. The sample KB can be added in that same directory, or preferably in a separate directory.

To boot up WinAKT, either double click on the *.exe file, or create your own program group - see your Windows documentation for further details.

Switch settings
The recommended memory settings for the running of WinAKT are: /V1/P4000/h1000/t5000. This string is assigned to the *winakt.exe* file in the set of programs supplied in the normal manner. The assignment is done via the “Properties” setting (Windows 3.x) and via a “Shortcut” (Windows ‘95) - see your Windows documentation for further details on how to assign these strings to the relevant executable file.

These settings rely to some extent on the specification of the machine used, also on the size and/or number of knowledge bases (KBs) that are open and/or under development. Given there are no apriori methods available to the program developer to determine minimal settings required for the running of arbitrary tasks within WinAKT, if the program fails to complete a given task, then the memory settings must be increased in a piecemeal manner. This can be avoided to a large extent by minimising computational overheads by: (a) keeping the maximum number of opened KBs to a minimum; and (b) reducing the number of programs running in the background and loaded into machine memory to a minimum too. If however, in running WinAKT you hit this particular problem; try proportionately increasing the settings - or follow the suggestions indicated by the error messages generated, if any.

ctl3d.dll
Included with WinAKT is a *.dll file - ctl3d.dll. This dll file is used by Windows to display the 3D look of its Windows. Older versions of this dll file are known to cause problems with WinPROLOG, so you are recommended to check whether your version of Windows includes a dll that is no older than the date given with the supplied dll. If the corresponding dll file in Windows is *older* than that supplied with WinAKT, you are advised to replace the dll used by Windows with the supplied dll. Failure to comply with this recommendation may result in the occurrence of system General Protection Faults (GPFs) or the program failing on boot up.

Appendix B: WinAKT: a brief description of functionality and rationale

WinAKT is a Windows re-implementation of AKT that was originally developed for the Macintosh platform[1].

WinAKT is software for creating knowledge-based systems (KBS). Knowledge collected from people and documents (sources) about a specified subject (the domain) is represented as a set of statements. A series of hierarchies that describe relationships amongst terms used in the statements is also generated. This allows general purpose algorithms to be used to manipulate the information stored in the knowledge base which in turn allows the computer to reason automatically with the information mirroring, for example, modes of reasoning commonly associated with human intelligence and problem solving. The result is a powerful tool for acquiring and using qualitative knowledge.

A central feature of the software is a definite clause grammar. This grammar defines formal sentences of varying degrees of syntactic complexity that may be represented. The grammar facilitates explicit identification of named actions, processes, and objects; their attributes and values for those attributes. The grammar was designed to represent ecological knowledge about agroforestry; but might also be used to handle ecological information about other domains.

WinAKT exploits qualitative as opposed to quantitative information because this is generally the sort of knowledge held and used by farmers, for which the software was designed. It is this qualitative information that readily lends itself to automated inference and logical deduction.

Just as with natural languages, the formal language used in WinAKT has a grammar or syntax; but in the case of a formal language the syntax is rigorously defined, and not all statements expressible in a natural language such as English, French or Nepali, can be formalised in the restricted language. Using a formal language forces the user to clearly represent what was actually articulated by local people. Our experience has shown that this apparent restriction actually facilitates the collection of useful knowledge about the chosen domain by forcing the user to concentrate on, abstract and then explicitly represent information.

Information is entered into the KB in WinAKT as textual statements written in the formal grammar. Typically the user types in a formal statement, which if syntactically correct, is parsed and back-translated by the software as a stylised natural language statement (allowing the user to check the sense), then saved to the KB. The source of the statement is recorded and other information about it can also be added.

In the development of a KB, the user creates object or sort hierarchies. These are generalised taxonomic hierarchies. A KB can contain more than one hierarchy. These explicitly represented hierarchies are then used by the program to simultaneously reduce the number of statements to a minimal set, but also by explicitly representing this information, facilitate general inference within the program.

In addition to the sort hierarchies, WinAKT allows topics to be created. These topics map to user-defined strings of Boolean (and/or) combinations of formal terms - which are the simplest elements of the formal language used. Each topic applied to the KB partitions the KB into discrete and meaningful sub-sets of statements. By creating topics, the user can in turn create hierarchies of topics, and thus provide knowledge (i.e. metaknowledge) about the content and interconnectedness of the concepts encapsulated in the KB. This function not only provides the KB developer with a quick way to assess and explore a KB, it also provides other users with a guide to what is in the knowledge base.

[1] Walker D H; Sinclair F L; Kendon G; Robertson D; Muetzelfeldt R I; Haggith M and Turner G S (1994): *Agroforestry Knowledge Toolkit: methodological guidelines, computer software and manual for AKT1 and AKT2, supporting the use of a knowledge-based systems approach in agroforestry research and extension*, School of Agricultural and Forest Sciences, University of Wales, Bangor.

Once the KB is sufficiently well-developed, the user can interrogate the KB in one of several ways. The first is with the use of a Query tool that enables the user to call up individual statements within the KB according to type; or in the use of a set of supplied and user defined tools or macros. These tools are written in a simple programming language which is made available to the user, and allows the user to define their own tools according to their own particular needs.

The whole process of KB development is an iterative one; with the user employing various pre-defined functions and user-defined tools to create a KB that is both broad and dense. By 'broad' we mean that the concepts used are sufficient to describe the intended domain, and by 'dense', that the concepts expressed in the set of formal statements and rules of inference applied to them, form a tightly interconnected inferential web of deducible consequences.

Contents