Functions in Context Data Base

Austin Tate

AIAI-TR-1

Paper published in the Proceedings of the Second Workshop on Architectures for Large Knowledge Bases, sponsored by the U.K. Alvey Directorate at Manchester University, U.K. on 9th-11th July 1984, Published on behalf of the Alvey Directorate by the Institute of Electrical Engineers, London.

Appendix published in the Proceedings of the First Workshop on Architectures for Large Knowledge Bases, sponsored by the U.K. Alvey Directorate at Manchester University, U.K. on 22nd-24th May 1984, Published on behalf of the Alvey Directorate by the Institute of Electrical Engineers, London.

Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

Functions in Context Data Base

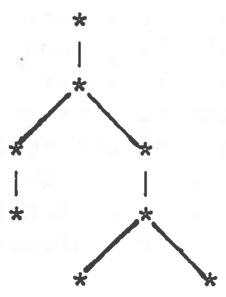
Austin Tate
Dept. of Artificial Intelligence
University of Edinburgh
Hope Park Square
Edinburgh EH8 9NW

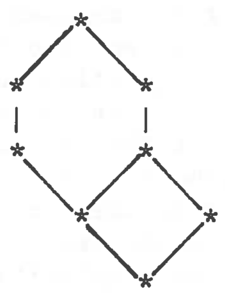
21-Jun-84

Introduction

Several Artificial Intelligence Data Base Support Systems provide a "context" mechanism. Examples are CONNIVER (McDermott and Sussman, 1972), QA4/QLISP (Sacerdoti et al, 1976), PEARL (Deering et al, 1981) and HBASE (Barrow, 1975). These provide facilities for efficiently storing a changing data base by remembering the alterations made to an earlier state of the data base. They provide facilities for the storage of data with respect to a fully ordered sequence of such layered changes. Hence, a tree of layers is formed in which the current "context" is defined by a path of layers back up through the tree. "Partitioned Semantic Networks" (eg, Hendrix, 1975) also provide similar features.

Some AI systems (such as planners able to reason with partially ordered action sequences - the so-called "non-linear" planners) have a requirement to be able to store and retrieve data in a partially ordered network of such layers. The incremental nature of the process by which partially ordered plans are built makes it very desirable that we store only the changes made to the data base at any point (rather than trying to maintain a complete copy of the data base as new ordering constraints or extra information at nodes is added).


A tree of layers


A partially ordered network of layers

In a tree of layers, there is a strict overriding order on changes to data. However, in the partially ordered network of layers, an answer to any query will depend on changes made at points in parallel with the point where a query is asked as well as the answers available by retracing to earlier points where data was stored or altered.

Functional Statements

In order to provide a readily understood basis for the storage of data which changes in different "contexts", a functional form of statement is used

    function (argument 1, argument 2, ...) = value in which the value for a given function name and arguments is unique in any one context but may change between contexts. Several AI researchers have exploited the uniqueness of a function´s value to provide added control information in search processes (e.g., Bundy, 1977).

The Meaning of "Context"

The interpretations of each node in a context, the meaning of layered changes that go into defining the value that can be retrieved for a given function name and arguments, the meaning of the ordering (perhaps only partial) between the nodes, and the pupose of the whole collection of nodes and links being handled at any particular time (called a configuration) are all user defined. In an AI planning system, the nodes may be interpreted as actions, the layered changes to statements at a node being used to represent alternatives being considered or different levels of specification, and the partial order between them may define a time ordering. In a software tools database, nodes could be interpreted as software modules, the layered changes to statements at a node could be used to represent the different properties of the various versions of the module, and the relationships defined between nodes may be used to represent compilation orders for systems. Other uses may include the specification of a class/sub-class/instance property inheritance system where multiple super-classes were allowed and, perhaps, different levels of abstraction were present.

Previous Work

Tate (1976) describes a question answering system implemented in the NONLIN planner which can store and retrieve functional statements made with respect to points in a partially ordered network of layered changes to a data base. However, the algorithms used assumed that only a single network structure was being handled. It remained the responsibility of the planner to establish (to the data base and question answering system) the particular network configuration to be used at any stage. Hence, the planner used a tree of layered changes to the various structures it employed and separately maintained information to give to the data base and question answering system which could then be used for storage and retrieval in a "current" partially ordered network being handled. This use of different representations introduced its own complexities.

Motivation

The efficiency of the data base, alternatives management and question answering system are critical to the overall performance of the planner. Hence, the overall requirements have been restated in the "Functions in Context" data model and interface specification in order that a single system can be provided to combine data storage and retrieval in a partially ordered network of nodes with management of incremental changes to any of those nodes.

One major motivation for this was the desire to improve the internal management of the necessary net marking schemes without altering a simple functional interface to the planner. Another, was the growing realisation that hardware schemes proposed for context addressable memory often included many of the basic mechanisms necessary for the efficient implementation of the required operations and could more efficiently manage some of the net marking operations. Hence, the hope is that the statement of the requirement in a form that is independent of the planner′s use of the facilities will enable workers on content addresšable memories and new data base systems to consider their relavance for the highly dynamic data storage system envisaged here.

References

Barrow, H,G. (1975) HBASE: a fast clean efficient data base system, POP-2 Program Library Documentation, Edinburgh University.

Bundy, A. (1977) Exploiting the Properties of Functions to Control Search, DAI Research Report No. 45, Edinburgh University.

Deering, M, Faletti, J. and Wilensky, R (1981) PEARL - a package for efficient access to representations in LISP, IJCAI-81 pp 930-932, Vancouver, Canada

Hendrix, G.G. (1977) Expanding the Utility of Semantic Networks through Partitioning, IJCAI-75, Tbilisi, USSR.

McDermott, D.V. and Sussman, G.J. (1972), The CONNIVER Reference Manual, MIT AI Lab Memo No 259

Sacerdoti, E.D., Fikes, R.E., Reboh, R., Sagalowicz, D. and Waldinger, R.J. (1976) QLISP: a language for the interactive development of complex systems, SRI Tech Note, SRI International, Stanford, Ca.

Tate, A. (1976) Project Planning using a Hierarchic Non-linear Planner DAI Research Report No. 25, Edinburgh University

Statement of the Requirements of the "Functions in Context" Data Model

1. Statements

1.1 Statements are of the form: f(arg1,arg2,...)=value at node

1.2 For any fixed f, arg1, arg2, etc the value is unique at a given node

1.3 The value can vary at different nodes

1.4 The special value "undef" is used to indicate that the value for the given f, arg1, arg2, etc is not set

1.5 Notionally, the value in all possible statements at all possible nodes is initially undef

1.6 The function name must be a string (called a "simple" identifier)

1.7 The arguments may be a string or number ("simple" identifiers) or "compound" identifiers of the form f(arg1,arg2,...)

2. Nodes

2.1 A new node may be created at any time

2.2 Optionally, the new node may be indicated as a new version of some existing node

2.3 A new derived node will inherit all statements given to the old version of such a node

2.4 The inheritance may be static (the node becomes an independent copy of the present state of the original node) or may be dynamic (changes to the original node or its own dynamic earlier versions may be reflected at the new node unless explicitly overridden)

3. <u>Links between Nodes</u>

3.1 Nodes can be related via a directed link (hence they can participate in tree or graph ordered relationships)

3.2 It is an error to attempt to introduce a cyclic link. Such a link will not be stored. Redundant links will be removed by the system

3.3 The meaning of this link can be attributed by the user (eg, "after" for a temporal link, "is-a" for a class/subclass or instance link, etc). Note that due to the restriction in 3.2, interpretations of the link apparently requiring cyclic graphs must be represented at a level that does not involve cycles

3.4 Values for any given f, arg1, arg2, etc at later nodes override those at earlier nodes

3.5 Different values for any given f, arg1, arg2, etc at nodes not ordered with respect to one another will result in multiple possible answers on retrieval (see 6.7b)

4. <u>Islands</u>

4.1 The set of nodes connected together by directed links are said to be in a single island.

4.2 It is undecided if an island has any real function in the data model (such as retrieval being relative to a "current" island in the "current" configuration) and if an explicit recognition of the nodes not participating in any directed link should be distinguished as in a "none" island

5. <u>Configurations</u>

5.1 Several individual nodes, trees of nodes or graphs of nodes as specified by the directed links between them may co-exist in a single configuration (ie, there may be several islands in a configuration)

5.2 The meaning of the directed link attributed by the user may vary between the different "islands" of nodes

5.3 It is possible to alter the "current" configuration being handled by the system by stating the nodes that participate in the new configuration and the directed links that hold between them

5.4 All statements made for a node or retrieval requests are with respect to the current configuration

5.5 A new configuration may be created at any time

5.6 Optionally, the new configuration may be indicated as a new version of some existing configuration

5.7 A new derived configuration will inherit all nodes, directed links and name associations given to the old version of such a configuration

5.8 The inheritance may be static (the configuration becomes an independent copy of the present state of the original configuration) or may be dynamic (changes to the original configuration or its own dynamic earlier versions may be reflected at the new configuration unless explicily overridden)

5.9 To give access to meta-data stored by the system itself (such as support_statements) and to ease the use of the system in a simple non-context mode, there will be a predefined node in every configuration which will be made available when a configuration is opened. This will be known as the GLOBAL node for the configuration

6. Retrieval

6.1 Retrieval of statements is via a partial specification of the form of the statement required

6.2 The node at which the retrieval request is made is restricted to being fully defined. It is specified as a node number allocated by the data base system when the relevant node was created

6.3 The function name and arguments in a retrieval request can be defined by retrieval specifications. There should be some method of giving a retrieval specification that means "match all possible function and argument combinations of any arity" (?? on its own - 6.4)

6.4 Allowable retrieval specifications must include the following:

```
a) ?? to match anything

b) ?not(<further specification>)

c) ?or(<further specification 1>,<further specification 2>,...)

d) ?and(<further specification 1>,<further specification 2>,...)

e) ?name  (a "logical" variable whose value is restricted or set
          during matching but which is reset for different matches)

f) ?included_in(<argument place>,
                <specification of identifier which includes the
                 one currently being matched in the indicated place>,
                <specification of value of this super-statement>)
```

eg, ?included_in(1,colour(??),red) would match any identifier, say X, that participated in a statement of the form colour(X)=red

Others may be provided for comparison of identifiers, etc

6.5 Retrieval results should be returned via a "generator" token that can be used with a "try_next" mechanism to generate answers one at a time. It is permissable for all result generator tokens to become invalid whenever a new configuration is opened (11.1)

6.6 Retrieval operators ignore statements whose value is "undef"

6.7 Retrieval must be able to return two classes of result

a) statements with the required value at the given node or statements with the required value at some node which participates in a directed link relationship with the given node and whose value is not overridden at the given node

b) a set of directed links which would need to be included in the current configuration in order to cause a statement with the required value to be available at the given node

6.8 In order to accommodate both types of result and the different uses to which the results may be put, the try-next mechanism from the results generator token will return

a) the instantiated f(arg1,arg2,...) form

b) the instantiated value

c) the "contributing" node (the node from which the value was made available)

d) a set of directed links which need to be added to the current configuration in order to make this result valid (this will be the empty set for results where no links are necessary)

6.9 The request for a retrieval (to get a generator token) may specify that only results where no extra links are necessary are required

7. <u>Explicit Deletions</u>

7.1 Setting f(arg1,arg2,..)=undef at node resets the value to "not set" which is ignored by retrieval operations (hence, the statement may be considered as a deletion of an explicit statement if desired)

7.2 The ability to explicitly throw away nodes, links, intermediate versions of nodes or configurations and/or generators may be desirable

8. <u>Statement Support Maintenance</u>

The user of the system will often make a statement at some node based on a computation or inference made from a set of statements retrieved from other contributing nodes in the current configuration. One common requirement of such use is to ensure that the support for such computations or inferences continues to be present

8.1 Allow support statements of the form:
f(arg1,arg2,...)= <value> from <contributing node> to <node>
to be stated. Any interaction with some existing statement should be reported as an error. A user defined annotation (any string) should be allowed to be associated with any support statement

8.2 When a statement is made at a node, a check should be performed against all existing support statements. If there are any conflicts, the set of support statements invalidated should be removed from the data base and returned to the user (hence keeping the data base valid) along with any associated annotation on them

8.3 It should be possible to perform a check on the support statements that would be invalidated if a statement was stored without actually storing it

8.4 The support statements should all be accesible as normal statements in the form:

```
support_statement(<annotation>,f(arg1,arg2,...),<value>,<to node>) =
                                                  <contributing node>
```

at a GLOBAL node which is predefined in each configuration (5.9)

9. Name Associations

9.1 For any configuration it is possible to associate a user provided name or tag (any identifier) with some value by storing e.g.
assoc(<name>) = <value> at GLOBAL.

9.2 For such a purpose, the value must be able to represent a node, and to allow for the naming of configurations, it must also be able to represent a configuration.

9.3 To help with configuration name associations, there will be a predefined configuration which which is made available when the data base system is initialised

9.4 Given a name or tag , it is possible to look up the associated value and treat it as a configuration, node or any user defined value

9.5 To ease the use of this facility for simple identifier associations with respect to a configuration, the storage and retrieval of statements of the form   assoc(<simple identifier>) = <value> at GLOBAL
for a fixed <name> could be provided through a simple interface.

10. Program Services

10.1 A user provided annotation (any string) can be associated with a node. This will be with respect to the current configuration

10.2 The list of nodes in the current configuration can be obtained, together with any user provided annotation associated with the nodes

10.3 The set of immediately linked successor nodes of any node can be obtained for any configuration

10.4 The set of immediately linked predecessor nodes of any node can be obtained for any configuration

10.5 Two nodes can be checked to see whether they are ordered with respect to one another for any configuration

11. Transactions

11.1 A configuration may be opened as the current configuration, any currently open configuration is aborted (11.4)

11.2 The GLOBAL node for the configuration (5.9) is made available when a configuration is opened

11.3 A commit on the current configuration may be performed to make permanent all changes to the nodes in the configuration, the directed links introduced between them, the statements made at nodes, node annotations and name associations. This also closes the configuration

11.4 An abort on the current configuration resets the configuration to its state at the corresponding open. This also closes the configuration

11.5 A common operation is to commit the current configuration, create a new configuration with the committed one as dynamic parent and then open this new configuration. This operation should be provided in a packaged way

11.6 A common operation is to commit the current configuration, create, open and use several new configurations with the committed one as dynamic parent. One of the children will then be chosen for further effort. This type of operation should be supported efficiently

11.7 When the data base system is initialised, the predefined configuration used to aid in name association (9.3) or simple non-context use of the data base is pre-opened

12. Desirable Properties of an Implementation

12.1 Creation of a new version of a node which dynamically inherits the statements in an old node may be delayed until changes are actually made to the new version (this could improve the efficiency of making a new version of a configuration in which little change is expected). This feature may be allied to the transaction mechanism

12.2 Garbage collection of unreachable information, node versions or configuration versions

12.3 The retrieval generator should be lazy-evaluated

12.4 Multiple disjoint "contributing" nodes for a retrieval result could be returned to reduce the alternatives that need to be considered by a user. This would need to be allied to multiple contributing nodes in support statements (8.1) and an automatic system to allow the number of disjoint contributors to be reduced to one without disallowing a statement to be stored

12.5 Overlay of a property inheritance scheme to allow statements of the form is-a(<class>=<subclass or instance> to be mapped to a form that provides automatic property inheritance to all nodes in a configuration

12.6 Declaration of the form of meta-data statements stored in the GLOBAL node of each configuration in order that these can be accessed by user programs. Only the support_statements must be accessible in this way. However, other program services such as getting the links, predecessor nodes, successor nodes, annotations, etc could all be provided by this method if desired

12.7 It has been found useful to allow a particular argument of all statements with a given compound identifier to be altered. This can be used to refer cyclic structures into the data base, etc. The values of all statements involved are not altered

12.8 If the implementation maintains indexes to look up from a given function name or arguments, it would be useful to allow a user to indicate when an index for some identifier used as a function name or as an argument in some given place should not be kept

13. <u>Performance and Size Targets</u> (based on use for a particular AI planner)

13.1 It should be possible to store statements with more than 5 arguments and a "compound" argument should be able to have a depth of 3 or more

13.2 It should be possible to maintain more than 1000 nodes in a single configuration

13.3 It should be possible to maintain an average of 5 to 10 layers in a dynamically inherited version of a node. However, on one or two nodes, more than 10,000 layers (representing over 10,000 major choice points to the planner) will be needed

13.4 It should be possible to maintain more than 20 name associations in a configuration

13.5 It should be possible to maintain more than 1000 configurations (a new planner design should reduce the number of configurations needed to between 10 and 100)

13.6 It should be possible to maintain more than 100 retrieval result generators concurrently

13.7 It should be possible to maintain node and support statement annotations of at least 255 characters

Specification of a particular interface

Tokens manipulated by the interfaces

<identifier>	is "simple" (string or number) or "compound" of form f(arg1,arg2,...) for 0 to n arguments
<value>	is undef or <user interpreted data>
<data base item token>	
<statement token>	
<support statement token>	is a special form of <statement token> in that its identifier, value and contributing node (GLOBAL) have predefined formats
<node token>	
<global node token>	is a special <node token> returned when a configuration is opened
<config token>	
<global config token>	is a special <config token> returned when the data base system is initialsed
<generator token>	undef for no possible answer(s)
<retrieval result token>	termin for no (more) result(s)

Useful Constants for the various sections

```
1.  stored_ok          = null list
    undef              = not set value

2.  dynamic            = true
    static             = false

3.  linked_ok          = true
    not_linked         = false

4.  no_parent_node     = predefined <node token>

5.  no_parent_config   = predefined <config token>

6.  with_links         = true
    without_links      = false
    simple_id          = -1

7.  none.

8.  no_annotation = ´support´
    conflict      = true
    no_conflict   = false
    global_node   = special <node token> for the predefined node in each config

9. to 11. none

12. as_function_name = 0
```

Interface functions for the various sections

```
0.  initialise() -> <global config token>

    terminate()

1.  store(<compound identifier>,<value>,<at node token>) ->
          list of <support statement tokens> removed to keep data base valid

2.  new_node(<parent node token>,<dynamic flag>) -> <new node token>

                     <parent node token>    = no_parent_node for no parent
                                              or <node token>
                     <dynamic flag>         = dynamic for dynamic inheritance
                                            = static for static inheritance

3.  link_nodes(<from node token>,<to node token>) -> <inserted flag>

                     <inserted flag>        = linked_ok if the link is added
                                            = not_linked if it would have
                                              introduced a cycle

4.  none

5.  new_config(<parent config token>,<dynamic flag>) -> <new config token>

                     <parent config token> = no_parent_config for no parent
                                              or <config token>
                     <dynamic flag>         = dynamic for dynamic inheritance
                                            = static for static inheritance

6.  get_all(<identifier specification>,<value specification>,
            <node token>,<link flag>) -> <generator token>

                     <link flag>            = with_links if results should
                                              include those where directed
                                              link additions would be
                                              necessary to support the
                                              answer
                                            = without_links otherwise

    try_next(<generator token>) -> <retrieval result token>

                     <retrieval result token> could be termin if no result

    identifier(<retrieval result token>) -> <data base item token>

    value(<retrieval result token>) -> <user interpreted data>

    contrib_nodes(<retrieval result token>) -> list of <node tokens>

    added_links(<retrieval result token>) -> list of pairs of <node tokens>
    arity(<data base item token>) -> <arity>

                          <arity>             = 0 to n for a compound
                                                identifier
                                              = simple_id for a simple
                                                identifier

    instantiation(<data base item token>) -> <external form of identifier>

                           <external form of identifier> is a string, number,
                                          or a defined f(arg1,arg2,...) form

    identifier_components(<compound data base item token>)
                           -> list of <data base item tokens>

                           result has function name first followed by arguments

7.  delete_node(<node token>)

    delete_link(<from node token>,<to node token>)

    delete_generator(<generator token>)

8.  store_support(<annotation>,<compound identifier>,<value>,<at node token>,
                  list of contributing <node tokens>) -> <stored flag>

                          <annotation>         = no_annotation or
                                                 <user interpreted data>
                          <stored flag>        = no_conflict if no conflict
                                                 occurs with existing
                                                 statements and hence
                                                 this can be stored
                                               = conflict  otherwise

    invalidated_support_if(<compound identifier>,<value>,<at node token>)
                            -> list of <support statement tokens>

9.  store_assoc(<name>,<assoc value>)

                          <name>               = <simple identifier>
                          <assoc value>        = <user interpreted data>
                                                 or <config token>
                                                 or <node token>
    get_assoc(<name>) -> <assoc value>

10. store_node_annotation(<node token>,<annotation>)

                          <annotation>         = <user interpreted data>

    get_node_annotation(<node token>) -> <annotation>

    nodes_in_config() -> list of <node tokens>

    prenodes(<node token>) -> list of <node tokens>
    succnodes(<Node token>) -> list of <node tokens>

    before(<earlier node token>,<later node token>) -> <boolean>

    after(<earlier node token>,<later node token>) -> <boolean>

    in_parallel(<node token>,<node token>) -> <boolean>

11. open_config(<config token>) -> <global node token>

    commit_config

    abort_config

    close_and_open_derived_config() -> <config token> -> <global node token>

12. data_base_item(<identifier>) -> <data base item token>

    set_arguments(<data base item token>,<place>,<new argument identifier>)

    no_index_for(<retrieval specification>,<place>)

                           <place>         = as_function_name  for function name
                                           = 1 to n for an argument place

                           if <retrieval specification> is "match all function
                           name and argument combinations of any arity" then
                           indexes will not be kept for any identifiers in the
                           specified place (??)
```

<u>Implementation Notes</u>

1. { f arg1 arg2 ... } is used to represent f(arg1,arg2,...)

2. A <data base item token> can be used interchangeably with the unique identifier for the represented data base item (ie, a data_base_item is performed as needed on an identifier and an instantiation is performed as needed on a data base item token).

3. A software implementation of the data model may introduce a structure called a "layer". Each node in each configuration has an associated layer. a layer tree is built up from a root layer normally using layer tokens which are integers that increase monatonically along any branch. Statements are stored with respect to the current associated layer of the node being referred to. A new transaction notionally creates a new associated layer for each node in the configuartion with the old associated layers as their dynamic parents. It is only on a commit that these new associated layers are genuinely associated with the nodes of the configuration. It is possible to only create new layers for nodes where statement values are actually changed. Layer to value association pairs are stored with the relevant identifiers in such a way that rapid access to the relevant layer to value association can be obtained when the associated layer of a node is known. This involves knowing the parent of any layer.

4. Consideration of the implementation of the functions in context data model on a generic associative store (such as FACT at Strathclyde University) has been made. Generic associative stores allow set oriented operations on trees of objects such as configuration and node layers. In additions, once a particular graph of nodes is being operated on, the generic stores can be used to perform rapid operations on ONE such graph. These operations can be overlayed with content addressing of the tuples and values of interest to the query routines.

5. The operations performed to support queries in the functions in context data model require fast operations on complex graphs. A dedicated graph processor or use of a generic associative store to implement the following functions would be of assistance.

   5.1 load a graph of nodes and links between pairs of nodes

   5.2 add or delete nodes or links removing redundant links in the process

   5.3 set a node as the "focus"

   5.4 mark a set of nodes as "special" (defined by some external system)

   5.5 queries:

      5.5.1 is a node before, after or in parallel with the "focus"

      5.5.2 get set of "special" nodes in parallel with the "focus"

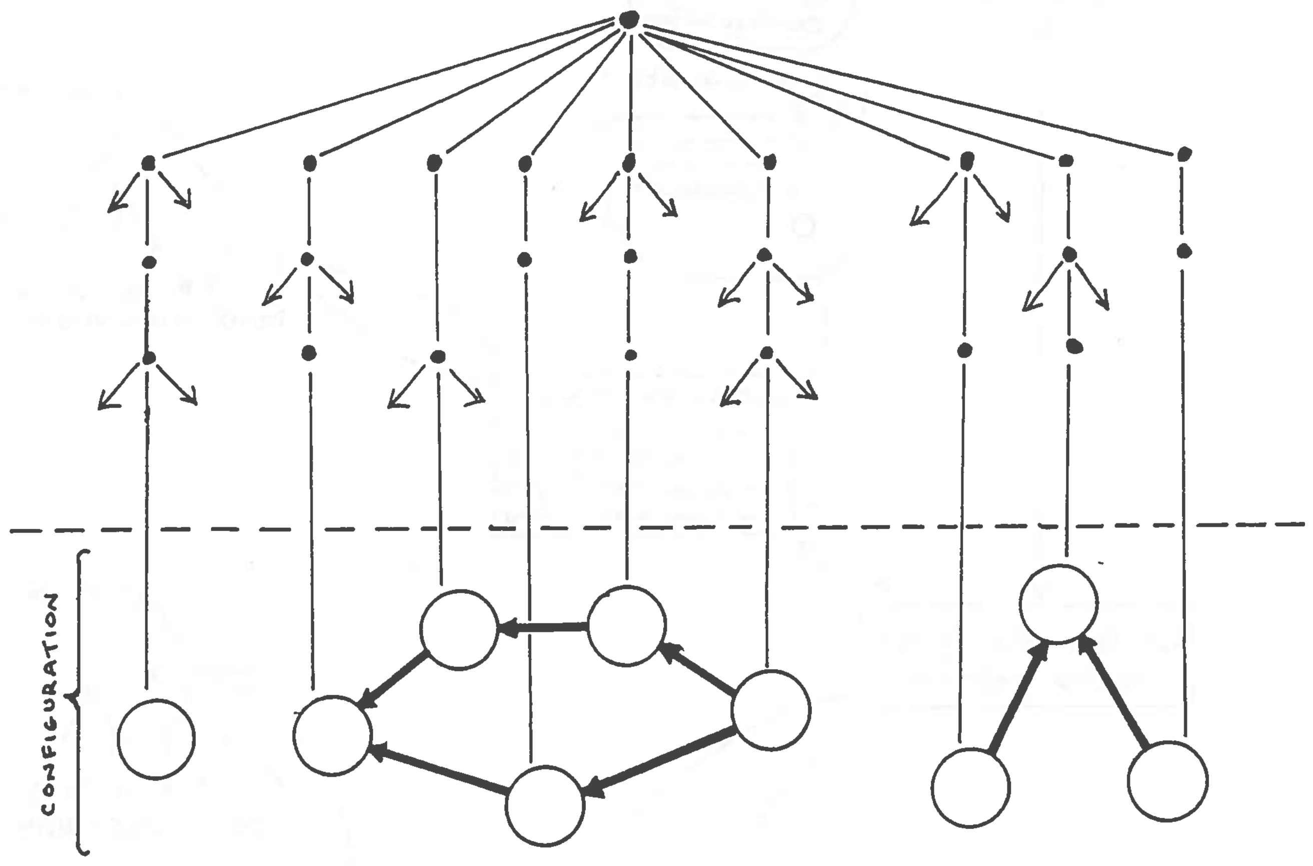
      5.5.3 get set of "special" nodes before "focus" which are not themselves before another "special" node itself before the "focus" (i.e. get last incoming "special" node on any branch before the "focus")

   5.6 ability to read out the node and links lists

Acknowledgements

This work was supported by an IT Fellowship from Edinburgh University and a Fellowship from Systems Designers.

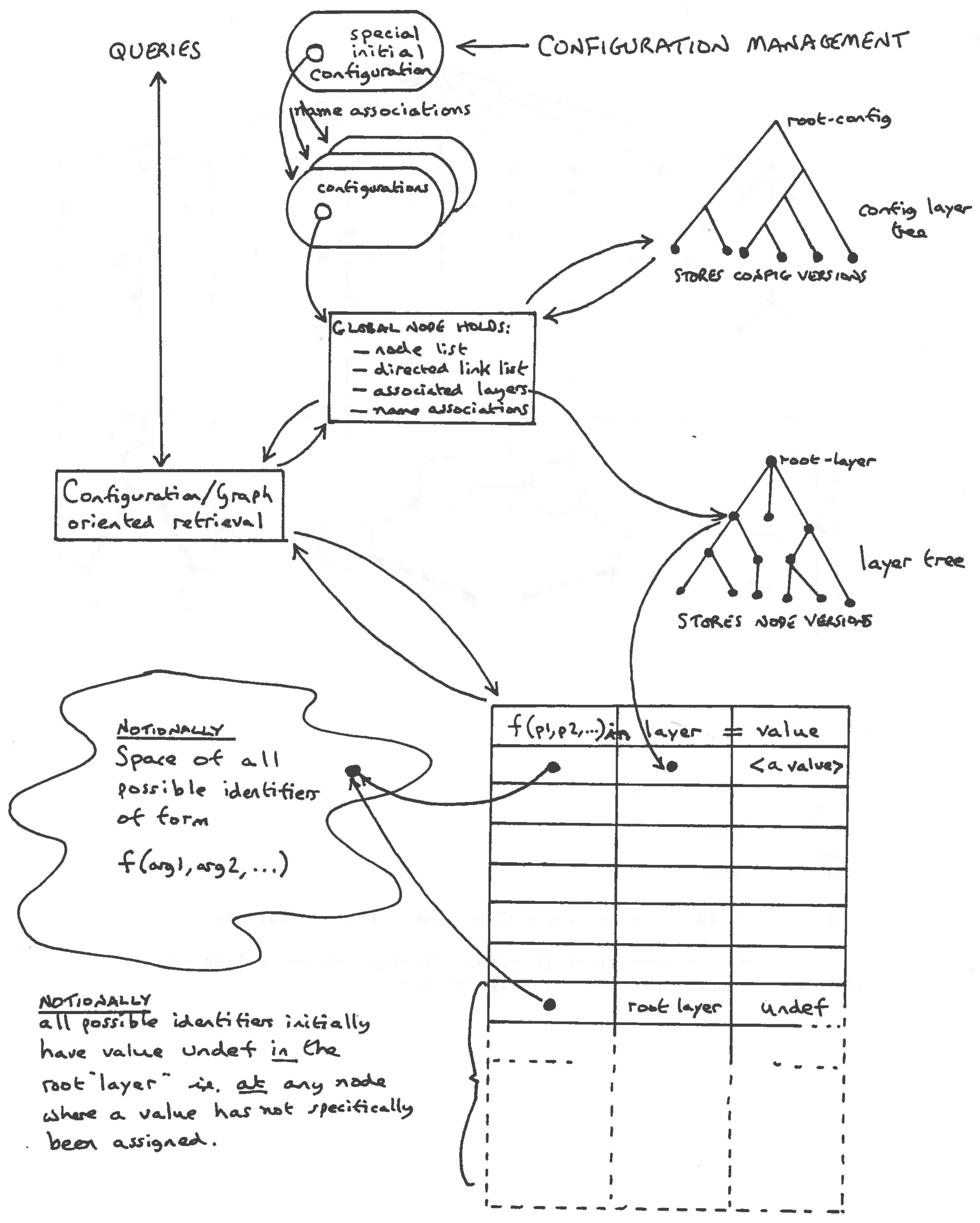
Diagrammatic Representation of a Configuration



Key

←	directed link
○	node
●	layer
|	child to parent layer link (new version creation)
↙ ↘	alternative links to other children layers to lead to other alternative configurations

A possible software implementation structure



NOTIONALLY
all possible identifiers initially have value undef in the root "layer" ie. at any node where a value has not specifically been assigned.

APPENDIX

A Partially-ordered Context Layered Knowledge Base for AI Planning

A position paper to the First Alvey Workshop on Architectures for Large Knowledge Bases. Manchester University, U.K. 22nd - 24th May 1984.

A Partially-ordered Context Layered Knowledge Base for AI Planning

Austin Tate
Department of Artificial Intelligence
Hope Park Square
Edinburgh EH8 9NW

I am concerned with the production of an AI planner based on a hierarchic representation of a domain. The planner generates a partially ordered network of tasks that can satisfy stated objectives in the application domain. It thus generates a PERT-like project network. Typical domains include civil engineering, maintenance, command and control, spacecraft sequencing, robotic device control, etc.

As part of this work, I am concerned with the following activities:-

- the design of a flexible planner prototype
- the design of a knowledge based interface for description of the application domain to the planner and for the presentation of planning results to the user
- the integration of the (re-startable) planner with run-time condition monitoring and re-planning on failures
- the use of the planning system on a large, realistic demonstration application.

The project is based on earlier work on the NONLIN planner (Tate, 1976), "Goal Structure" information to capture the "intent" of plans (Tate, 1975) and the Task Formalism domain description language (Tate, 1976).

In order to construct the planner envisaged, a support knowledge base is required which can cope with a great deal of dymamic activity in terms of information that is related to time and to search alternatives. A simple system able to cope with the required operations has been in use in the existing NONLIN planner for some time. Improvements to the "Question Answering in a Partially-ordered Network of Nodes" algorithms used have been proposed (Daniel and Tate, 1982).

The closeness of the required operations of marker propogation on graphs to those becoming possible on advanced hardware data bases such as NETL (Fahlman, 1979) and FACT (McGregor and Malone, 1982) and the slowness of serial computer implementations of the inherently parallel algorithms involved have led to the proposal to abstract out the required functionality. This will allow a simple slow implementation at present (to enable the AI planning research to get on with its own job) and at the same time provide a statement of requirement for those concerned with large, flexible knowledge bases for Intelligent Knowledge Based Systems.

Functions in Context Data Model

The development of an entity/relationship data base system with context varying values is an essential component of the planning system envisaged. The functions to be performed by this system are presently implemented in NONLIN by a simple tuple store (HBASE) with incremental fully ordered changes of values (rather like the CONNIVER tree-structured context mechanism). On top of this, code integrated into the planner models a partially ordered set of changes of value. This scheme is a known performance bottleneck of the existing NONLIN planner.

We wish to abstract out the full data model which is necessary to support the planner and to provide a clean interface to this as a separate system. The data model will store clauses of the form:

f(p1,p2, ...) = value <u>in</u> context

and allows for retrieval of partially specified matching items (via a "lazy-evaluated" "generator" and "try-next" mechanism). The system will allow the specification and retention of a context as a point in a partially ordered set of changes to the values of entities (which may be relationships or tuples of other entities). Consideration must be given to the highly dynamic nature of the definition of the current context (as a point which is built up as as layer upon layer of changes at various points in a graph of nodes that is itself being constantly reconfigured). There will be a search space containing many alternative graph arrangements concurrently being explored. Garbage collection of unusable information would be critical.

The "Functions in Context" data model has very general applicability and could be used in areas such as software engineering tool communication, multiple inheritance knowledge representations, etc. as well as in planning. Although it can support the storage and retrieval of multi-valued logical formalisms with special treatment of "contexts", it can also be used to support more traditional boolean logics or logic programming unit clauses.

The work to be performed during the planning research will provide a software implementation of the data model for a limited size data base sufficient to support the needs of the planner and other demonstration components needing it. This will use earlier work on HBASE, RBASE and NONLIN's Question Answering System as starting points.

However, the design and implementation will take into account the requirement for efficiency and size at a later date by ensuring that interfaces exist at which hardware tuple stores and concurrent access persistent stores could be used to implement the "Functions in Context" data model. Property inheritance schemes to exploit generic associative semantic stores will also be considered (to provide support for a multi-valued many-sorted logic formalism).

References

Barrow, H.G. (1975) "HBASE: a fast clean efficient data base system" D.A.I. POP-2 library documentation. Edinburgh University.

Daniel, L. and Tate, A. (1982) "A retrospective on the ´Planning: a joint AI/OR approach´ project" D.A.I. working paper no.125. Edinburgh University.

Fahlman, S.E. (1979) "NETL: a System for Representing Real World Knowledge" MIT Press, Cambridge, Mass. USA.

McDermott, D.V. and Sussman, G.J. (1972) "The CONNIVER Reference Manual" M.I.T. AI Lab. Memo no.259.

McGregor, D.R. and Malone, J.R. (1981). The FACT Database: A System using Generic Associative Networks. Research Report No. 2/80. Department of Computer Science, Univ. of Strathclyde.

Tate, A. (1975) "Using Goal Structure to Direct Search in a Problem Solver" Ph.D. Thesis, Edinburgh University.

Tate, A. (1976) "Project Planning Using a Hierarchic Non-linear Planner" Department of Artificial Intelligence Report 25, Edinburgh University.

Tate, A. (1981) "RBASE - a Relational Data Base System on EMAS" ERCC, Edinburgh University, Technical Monograph No. 1.