EPILOG: The Computational System for Episodic Logic

PROGRAMMER'S GUIDE

Stephanie Schaeffer

Chung Hee Hwang

John de Haan

Lenhart K. Schubert

August 1993

Revised September 2000

Prepared for Boeing Computer Services, Seattle, Washington

Under Purchase Contract W-278258

Contents

Chapter 1

Introduction

This manual is intended for readers who are going to maintain or modify the **EPILOG** system, or who are interested in the implementation details. For those who just want to know what the system does or how to use it, the User's Guide would probably be more appropriate. Some prior knowledge of the system would be helpful before reading this manual.

Currently, the **EPILOG** system is around 35,000 lines of (documented) lisp code. To make such a large system managable, the system is divided into several packages and directories which will be described shortly. Any new pieces added to the system should follow the same pattern.

As well, there are some guidelines we have tried to adhere to. To make the source code more readable, the common lisp constructs *if* , *do* , etc have been used, although this makes the system less portable than if we had used only basic lisp structures. No lisp construct with "invisible" side effects is used - such as *mapcan , mapcon* , *rplacd* , etc. Although these work quickly and make neat (albeit unreadable) code, they can have side effects which are difficult to debug and fix, and usually don't turn up until months after the initial implementation using the construct. Speed has been sacrificed for safety in these cases. Global variables (including tweakable parameters) are all of the form *...* (just like common lisp global variables).

Figure 1 shows a general outline of the system parts and their interactions. The large circle called "EPILOG core" contains the main storage and inference facilities. The smaller circles surrounding it are specialists, which can assist during the inference process. Some also maintain their own representation of input facts. The specialists closest to the EPILOG core are automatically loaded by the system. Although theoretically EPILOG should be able to run with no specialists, to make the system operate reasonably quickly, and to make life easier for the user, some are considered essential. The circles slightly further out are optional specialists. The square object is the response generator, which can optionally say the answer to a question, or repeat input and inferred formulas.

Dashed lines in the figure show that the component being pointed to can have its internal information altered by the other component. For example, EPILOG formulas are used to create and modify timegraphs in the time specialist. The formulas are also used to maintain information in the set specialist, which can then make inferences back to EPILOG. The EPILOG command **add-hier** changes the type hierarchies used by the type specialist. Solid lines show transfer of information only. For example, EPILOG can only send requests for evaluation to the string specialist, and the string specialist can only return answers to these requests. The interactions of parts of the system will be described in more detail later.

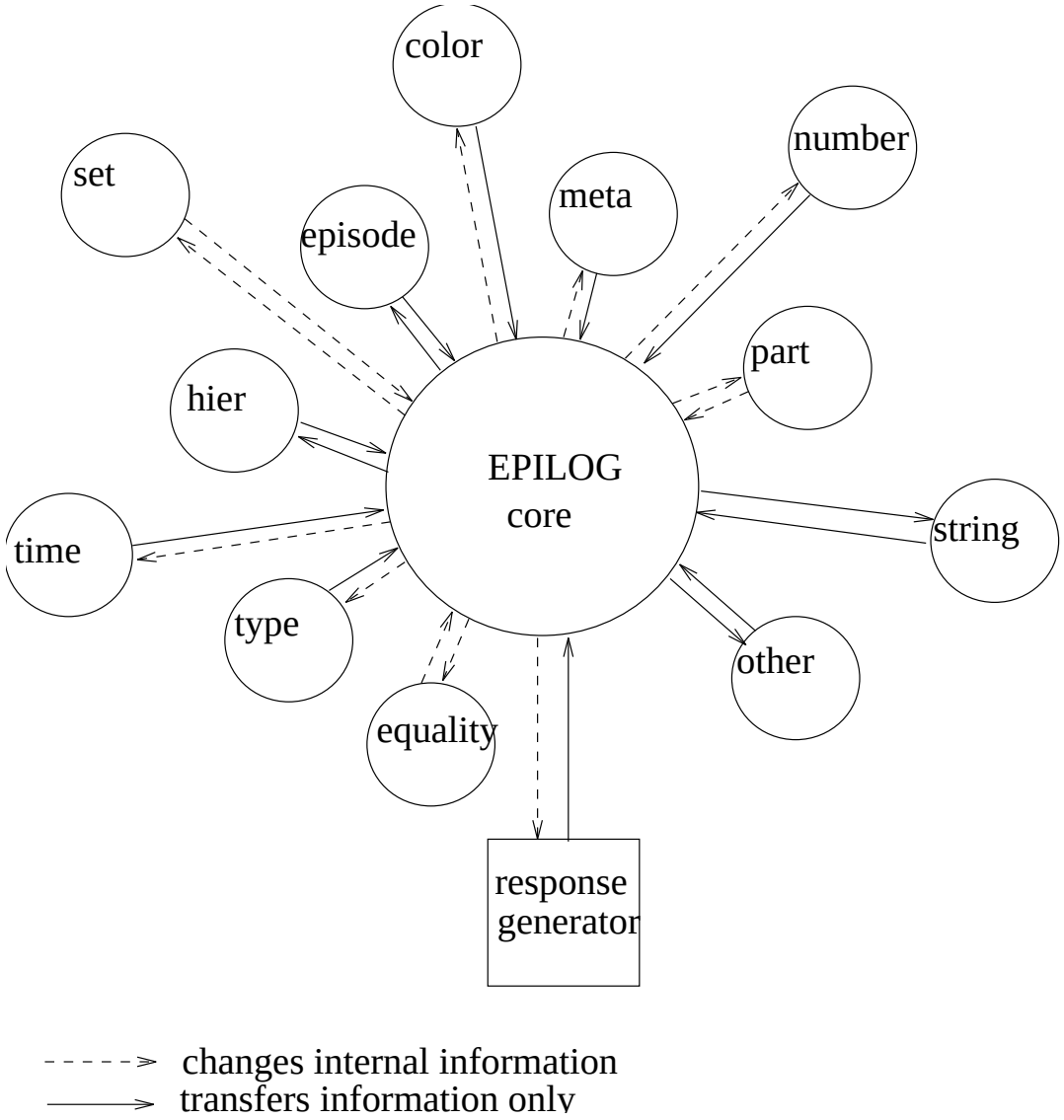


Figure 1: The EPILOG system

1.1 Packages and Directories

The main directory of the distribution is named *EPILOG* . Within this directory, the system source and compiled files reside in the sub-directory *epi* . In *epi* , there are several system files, as well as a number of sub-directories. The system is divided into several component parts - each of which resides in a separate lisp package, and is located in a different sub-directory. The EPILOG core shown earlier actually consists of several packages: general lisp routines which could be useful for any application (*lib*), lower level routines used for EPILOG and the response generator (*epilib*), a special interface for using the specialists and for use by the specialists (*spec*), the main entry and inference part of EPILOG (*story*), alternate user interfaces in *eshell* and *epiuser* , and low level set handling routines for use by both the equality and set specialists in *set-equal* . The response generator is in a package called *response* , and each specialist is in its own separate package (*time-specialist, number-specialist, ...*).

The packages and directories are:

System part	**Package (nicknames)**	**Directory** *epi/*
core of **EPILOG**	*epilog (epi)*	*story*
general routines	*lib*	*lib*
low level EPILOG routines	*epilib*	*epilib*
specialist interface routines	*specialist-interface (spec)*	*spec*
response generator	*response*	*response*
epi shell	*eshell*	*eshell*
user interface	*epiuser*	*epiuser*
low level set routines	*set-equal-stuff (set-equal)*	*set-equal*

Specialist for	**Package (nicknames)**	**Directory** *epi/*
temporal relations	*time-specialist (time)*	*time*
arithmetic relations	*number-specialist (number)*	*number*
set membership	*set-specialist (set)*	*set*
equality	*equality-specialist (equality equal)*	*equal*
colors	*color-specialist (color)*	*color*
types	*type-specialist (type)*	*type*
non-type predicates, arranged hierarchically	*hier-specialist (hier)*	*hier*
parts	*part-specialist (part)*	*part*
episodic relations	*episode-specialist (episode)*	*episode*
string relations	*string-specialist (string)*	*string*
others' beliefs	*belief-specialist (belief)*	*belief*
meta	*meta-specialist (meta)*	*meta*
routines for connecting to external routines	*other-specialist (other)*	*other*

Figure 2 shows the package interactions. Solid lines indicate that the package being pointed to "uses" the other package. Dashed lines indicate that the package being pointed to knows about the other package, but that its routines are not imported. There are some additional packages defined and used within the epi-shell. These will be described in the section on that component.

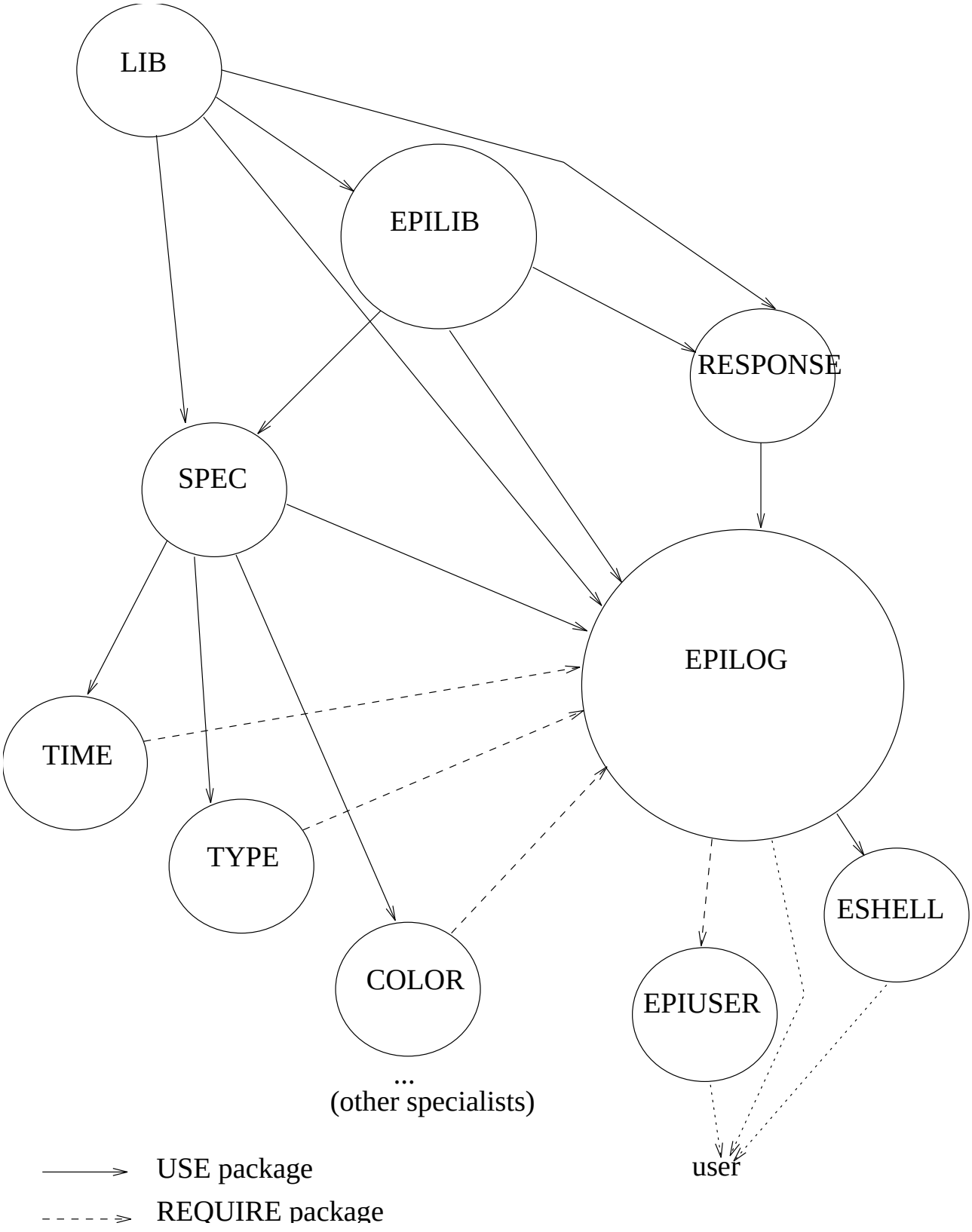


Figure 2: Packages in EPILOG

The user has the option of "using" the *epiuser* package, which exports ONLY the user routines, or just loading it and prefixing *epiuser:* in front of commands. Routines in this package intern all their arguments into the *epilog* package, and then call the corresponding *epilog* routine. Alternatively, one can go into the *epilog* package, and simply call the routines from there. Because all the symbols are interned into *epilog* , a user should be able to flip back and forth between *epilog* and the original package that *epiuser* was loaded into.

In addition to the source directories in *epi* is a directory called *help* . This contains all the interactively available help, in individual files called *subject* **.hlp** , where *subject* is the name of a command or topic that help is available for.

The remaining items in *epi* are as follows:

File	**Purpose**
epi.lisp	Alternative name for startup
compile-all.lisp	Contains loads for all compiles
specialists	Initial definitions of available specialists
start-up-msg	Contents displayed when EPILOG started up
epi-hier	Minimal hierarchy required for system operation
epi-indicate	Minimal indicators required for system operation
epi-init.lisp	Site initialization file loaded whenever system starts

Whenever a new specialist is added, the file *specialists* should be updated. *epi-hier* and *epi-indicate* should not be changed, although they may be added to if a "site minimum hierarchy" is desired. The message in file *start-up-msg* may be changed if desired - it is a simple text file which is read in and displayed.

At the same level as the *epi* directory is a directory called *test-files* . This contains sample hierarchies, lexical and translation information, and test files that show what the system can do.

Chapter 2

EPILOG Core

2.1 Environments

The state of the system's knowledge is stored in a structure called an *environment*, whose various fields will be described below. When the belief specialist is not active, there is only a single environment, which is stored in the special variable ***environment***. When the belief specialist is active, in addition to the system's main reasoning environment, there may be other environments that store beliefs that the system attributes to other agents. By binding ***environment*** to a different environment, the system can temporarily relinquish its own beliefs and assume those of another agent, for the purposes of reasoning by simulation about that agent. In addition, the belief specialist creates an environment for storing knowledge which is shared by all agents, including the system itself. This shared knowledge base is stored in ***shared-kb***. Simulative inference is covered in more depth in the section on the belief specialist.

The environment structure is defined to have the following fields:

name: a symbol that identifies the environment. Used only to facilitate debugging.

input-array: an array containing all of the wffs that have been entered into this environment.

input-index: an integer, the index of the element in **input-array** where the next wff will go. The number of elements stored in **input-array** is **input-index** + 1.

main-net: a hash table that stores the wffs believed by the system, indexed by main classification (see page 17).

topic-net: a hash table that indexes the same wffs, but by topical classification (see page 17).

equivalence table: a structure that keeps track of classes of terms that the system believes to be equal.

element-table: a hash table for finding the names of formulas and subformulas that have been seen before (see page 12).

specialist-info: a place for specialists to add their own information to the system's knowledge state. We use a hash table indexed by specialist name, rather than a separate field for each specialist, so that specialists can be added dynamically without redefining the environment structure.

use-environments: a list of other environments that are accessible whenever this environment is accessible. If the belief specialist is not loaded, the value of this field is always nil. See Section 3.2 for an explanation of how this field is used when the belief specialist is active.

2.2 Formula Entry

Input formulas, whether user-input or inferences made by the system, go through the following processes:

normalization - distribution of negations, standardization of variable names, making the list form into a symbol form which allows sharing of subformulas

skolemization - creation of skolem constants for top level existentially quantified formulas

splitting of top level conjuncts - top level conjuncts, and conjuncts within * and ** constructions are split into several formulas

application of simplification schemas - replacement of the original formula by an inference made using simplification schemas. These act as transformation rules.

consistency checking - the new formula is evaluated to see if it already exists, or contradicts previous knowledge.

classification and storage - classification is used both to decide where to store the new formula as well as a starting point to look up formulas which can be applied to it

input-driven inference - determines consequences of the input formula

2.2.1 Normalization

During normalization, the original list form given is transformed into a symbol, which contains properties indicating the various parts and properties of the formula. This symbol, and any sub-symbols created during the process, are stored in a hash-table in the **element-table** field of ***environment***, based on the entire list form they represent, and whenever another list form identical to that one is found within a formula, the same symbol is used. This allows sharing of subformulas, which saves space, as well as ensuring that we only have to parse each formula once to know all its properties, saving computation time later.

Sometimes we need to temporarily normalize something during input driven inferencing to look up matches for partially instantiated keys or to verify them. These we do not want kept in the hash table if the attempt fails, but we may want them if it succeeds. A separate hash table is kept for this purpose. When finding the atom for a particular subpart during one of these "temporary" normalizations, we first look in the main hash table, and if we find the item, we use that. If not, a new one is created in the temporary table (***temp-table***). If later, we find that this attempt succeeded, the new item will be created in the main table.

The normalization procedure is as follows: First some predicates and operators are standardized (in *pre-norm*): *not* replaces ~ , *implies* replaces => and -> , *less-than* replaces < , *lt=* replaces <= , *greater-than* replaces > , *gt=* replaces >= , *equal* replaces = , *and* replaces & , *or* replaces | , and *qquote* replaces *qq* . The user may input whichever form is more convenient, but the system will change it to use the one it prefers.

Next negations are distributed (in *distr-neg*) so that they have minimal scope. This involves "passing" *not* through *and, or, not, implies, poss, nec* , and quantifiers. Quantifiers and some predicates and

operators contain information under the property *negation* telling how to distribute negations involving them. Then nested *and* 's and *or* 's are collapsed into one level (*collapse-connectives*). Then all occurrences of operators marked *distributive* are distributed over conjunctions in the formula (in *distr-ops*), including *nec* , propositional attitudes, and *implies* - although this is really only working and useful for *nec* . For propositional attitudes, the right hand argument is no longer a simple formula (it usually has operator *that* or is a propositional term used earlier in a _ construction).

Then the main body of the normalization process starts. This is a recursive process which takes the list form of a formula and makes it into a symbol, with properties indicating the type of formula, and various parts of the formula (such as arguments, predicate, quantifier, etc). Each resulting symbol will have a property called *entity-type* , which will contain one of the following: *wff, term, pred, op, quantifier* . In addition, a property called *print* on each symbol will contain the expanded version of the symbol. When finished, the resulting formula symbol will have under property *wff-type* one of the following:

quantified - formulas involving a quantifier, variable, optional restriction clause, and a main clause. These are contained under the properties *quantifier* , *variable* , *restrict* , and *main* . If this is a top level formula, a property called *types* contains a list of all internal variables with their types or sorts.

logical - formulas involving a logical operator, such as *and* , *or* , *implies* , <=> , or a number predicate. The properties involved with this type of formula are *pred* and *arguments* . If this is a top level formula, and is a conditional or equivalence, it will also contain the *types* property described above.

modified - formulas involving a sentence level operator, and arguments (usually just one, and it is a formula). The properties involved with this type of formula are *operator* and *arguments* .

episodic - formulas involving one of the episodic operators ***, *, @* , or _ . The first argument is a formula, and the second an episodic or propositional term. The properties involved with this type of formula are *pred* and *arguments* .

causal - formulas with a causal connective as predicate (e.g. *because*) and formulas as arguments. The properties involved with this type of formula are *pred* and *arguments* .

prefix - formulas involving any other type of predicate and arguments The properties involved with this type of formula are *pred* and *arguments* .

quasi-quoted-expression - the wff is a quasi-quoted expression. The normalized formula will be stored under property *quote-wff* .

constant - a named formula

variable - a variable representing a formula

A similar process is used on the predicates and arguments of the formula so that everything becomes a symbol with all properties mapped out. Complex predicate symbols are named *pred1* , *pred2* , etc. Similarly, complex term symbols are named *term1* , *term2* , etc; operators *op1* , *op2* , etc; and quantifiers *quantifier1* , *quantifier2* , etc. The user may attach a name him/herself using the name operator **!** , in which case the user assigned name will be used for all instances of that term, predicate, wff, etc. Simple predicates, terms, etc which are already symbols will not be renamed.

A predicate in the form *(pred terms)* will be lambdified. The shortened form is allowed for ease of entry for the user, but is automatically converted by the system. Predicates have one of the following under *pred-type* :

modified - the predicate is a predicate-forming operator, acting on another object (or objects) (a term, predicate, wff, etc). Properties to look under are *operator* and *arguments* .

lambda - the predicate is a lambda expression *(L x (x ...))* . It also contains properties as if it were a quantified wff.

constant - the predicate is a simple constant (like *red* , or *girl*).

variable - the predicate is a variable representing a predicate

Terms have one of the following under *term-type* :

modified - a term-forming operator is acting on other object(s) Properties to look under are *operator* and *arguments* .

function - the term is a function acting on other terms Properties to look under are *operator* and *arguments* .

record - the term is a record term ($ 'sort ...) The record itself will be under the *print* property.

quasi-quoted-expression - the term is a quasi-quoted expression. Substitution and some functional evaluation may occur within such a list. If the quasi-quoted-expression is also a wff, it will contain property *quote-wff* . Otherwise the quasi-quoted expression itself can be found under the *print* property.

quoted-expression - the term is a quoted expression of the form '(....), where no modification happens to the contents of the list. The quoted expression itself can be found under the *print* property.

wff - the term is a formula. Properties applicable to wffs can be found here.

constant - the term is a simple constant (like *lrrh* , or *c1*).

variable - the term is a variable representing a term

Operators have one of the following under *op-type* :

modified - the operator is an operator-forming operator, acting on another object (or objects) (a term, predicate, wff, etc). Properties to look under are *operator* and *arguments* .

constant - the operator is a simple constant (like *very* , or *nec*).

variable - the operator is a variable representing an operator

Quantifiers have one of the following under *quant-type* :

modified - the quantifier is a quantifier-forming operator, acting on another object (or objects) (usually another quantifier -e.g. (nearly A)). Properties to look under are *operator* and *arguments* .

constant - the quantifier is a simple constant (like *A* , or *most*).

variable - the quantifier is a variable representing a quantifier

During this process, variable names are standardized. Matchable variables (variables which can be unified with a constant, leading to an input driven inference or goal reduction) are standardized to *X, Y, Z, X1, Y1, Z1* , etc; non-matchables to *U, V, W, U1, V1, W1* , etc. If a sort is involved, it precedes the variable name, for example *TIME-X, EP-U* , and if the variable is a lambda variable, it is preceded by

L- (e.g. *L-U, L-EP-W*). Sub-literals in a formula are picked out and classified as to whether they occur positively or negatively (generally the negative literals occur in the antecedent of a rule, positive literals in the consequent of a rule or a story fact).

Some special catches are put in so that an input formula of the form *((qq (...)) true)* is automatically normalized to (...). Lambda reduction is also done automatically if applicable (so *(John (L x (x kiss Mary)))* would be reduced to *(John kiss Mary)*), and predicates are expanded to lambda expressions where these are preferred (so *((adv-e (during yest)) (john kiss mary))* would be normalized to *((adv-e (L x (x during yest))) (john kiss mary))*).

The normalization process also includes error checking to ensure that the input given is correct episodic logic. If an error is found, the user will be notified and the normalization will stop immediately. Warnings are less severe, and the process will continue. Errors checked for include the following: the appropriate number of arguments are involved in quantified or logical expressions, simplification schemas must be equivalences, meaning postulates may only have meta-level variables or sorted variables (outside quasi-quoted expressions), predicates cannot be operators or quantifiers, variables (in rules other than schemas, and in quasi-quoted expressions) cannot be predicates, operators, or quantifiers, etc. The checking is by no means exhaustive, but does catch most typical errors, especially those caused by improper bracketing. The lisp "catch" and "throw" constructs are used to quickly exit the normalization process if something goes wrong.

2.2.1.1 Known Problems and Possible Improvements

Splitting of top level conjuncts and skolemization are checked for and occur BEFORE normalization (to save time - there is no point normalizing a formula which will be immediately replaced anyway). Special checks are put in to try to ensure correct input, but as they don't go through the entire process, sometimes invalid input can be missed.

2.2.2 Skolemization

Top level input formulas which are quantified with *E, E!* , or *the* have a skolem constant created for the existential variable. This constant is substituted for the variable throughout the formula, and the new conjunction replaces the original formula. The constant is kept on the property list of the variable so that future references to that variable can be replaced by the constant within that formula and even in future formulas until there is another existential quantification using that variable. For *E!* and *the* , a uniqueness formula is added to the conjunction which further states that any other constant which conforms to the conditions set by the formula must also be equal to this skolem constant.

Skolemization does not take place inside modally embedded formulas.

2.2.2.1 Known Problems and Possible Improvements

Currently the existentially quantified variable will "remember" the constant it was skolemized to, but if the variable has been changed from its original input form (because a schema applied and the result was normalized), the original variable is "forgotten".

2.2.3 Application of Simplification Schemas

Simplification schemas are just meaning postulate axiom schemas which apply BEFORE a formula is stored, and whose inferences REPLACE the formula rather than add to the knowledge base. Otherwise, the inference procedure is identical to that described in the input-driven inference section. The simplification schemas must be equivalences (otherwise normalization will complain), but when applied, they are treated as implications (i.e. the schema only fires in one direction - left to right, rather than both). Only one simplification schema can operate on a formula - after that the formula has changed!

The replacement formula then goes through the entire normalization, splitting, application of simplification schemas, etc, process.

Note: the splitting of top level conjuncts acts like a built in simplification schema by replacing the existing formula with several. If desired, episodic formulas can be prevented from having their conjuncts split by setting **episodic-split** to nil - a meaning postulate can be added to do this easily so that both formulas are kept (*(A x_wff (A y_wff (A z_ep ((qq ((x and y) * z)) true) ((qq ((x * z) and (y * z))) true))))*).

2.2.3.1 Known Problems and Possible Improvements

There is no control on which schema applies first - they are applied as they are found by the classifications involved, and in the order they occur within that classification. Since a formula can only ever have one schema applied (after that it becomes a new formula, with a different appearance), this could potentially be a problem if one is not careful in designing the schemas.

2.2.4 Consistency Checking

The system check new input (both user and inferences) to see if it is consistent with previous knowledge. How hard it tries to determine consistency is user-controllable through the parameter ***consistency-effort*** :

0 - no consistency checking (not recommended unless you are sure there are no inconsistencies)
1 - lookup only (checks existing formulas to see if identical one present)
2 - verification with probabilities of 1 only (including specialist evaluation)
3 - standard verification (including formulas with probabilities < 1)
7 - question (does full blown qa attempt)

The default is 3 (i.e. simple lookup and specialist evaluation).

If an inconsistency is found, i.e. the new input contradicts previous input, another user-controllable parameter ***consistency-action*** is used to tell the system what to do about it.

0 - enter the conflicting information anyway (without warning user)
1 - warn the user, but enter the formula anyway
2 - don't enter the information - throw it away (without warning user)
3 - warn the user, and then discard the formula
4 - try to combine the new formula with previous information

An existentially quantified inference will also be checked to see if an entity with those qualifications exists already - if so, the inference will not be loaded. This prevents generating extra formulas and skolem constants if the same rule fires more than once on the same data.

As part of the verification process, a formula may be simplified. The simplified formula will then be entered rather than the original. The new formula will be put through the simplification schemas, as well as the splitting and skolemizing process.

2.2.4.1 Combining Contradictory or Supporting Evidence

If a new formula is supported by previous formulas, their probabilities are combined so that the net effect is a higher probability than any of the supporting formulas. If the new formula contradicts previous formulas, both the new and old formulas have their probabilities reduced. On any formula, a property called *support* contains probabilities of supporting evidence, and a property called *contradict* contains probabilities of contradictory evidence. Supporting evidence probabilities are combined together, and contradictory together, and then the two resulting probabilities are used to determine the final probability (this is because the method used for combining contradictory evidence is order dependent), so we do all the formulas that support each other first (that is order independent), and do the problem step last.

When combining two pieces of supporting evidence, the following formula is used for the new probability:

$$p_1 \ = \ p_2 \ = \ (1 \ - \ ((1 \ - \ p_1) \ * \ (1 \ - \ p_2)))$$

where p_1 and p_2 are the probabilities of the two pieces of evidence.

When combining two pieces of evidence which contradict each other, the following formula is used (again p_1 and p_2 are the probabilities involved):

$$p_1 \ = \ ((p_1 \ - \ p_1 \ * \ p_2) \ / \ (1 \ - \ p_1 \ * \ p_2))$$

$$p_2 \ = \ ((p_2 \ - \ p_1 \ * \ p_2) \ / \ (1 \ - \ p_1 \ * \ p_2))$$

Future inferences with formulas where evidence has been combined may have inaccurate probabilities, as it is difficult to tell which formulas have been involved in a path leading up to that point. Currently we are attempting to find a better method for combining evidence. Also, better checking is required to ensure that we don't combine supporting evidence where one is more specific than the other, or subsumes it.

2.2.5 Classification and Storage

The classification routines determine classification lists which are then used as hash keys for storing formulas. There are 3 kinds of classification - the main, predicate based classification, topical classification, and a special classification for meaning postulates. In addition, the main and topical classification can be a list of classification lists - indicating nested subnets for formulas involving belief and other mental attitudes. The main classification consists of lists of the following forms:

(concept predicate role)

or

(concept specialist-name)

where the second form is used if the predicate involved has specialists which are interested in it. The topical classification consists of lists of the form:

(concept topic)

Meaning postulate classification consists of lists of the form:

(second-level-operator top-level-operator)

Main subnet classifications are of the form:

((concept predicate role) (concept predicate role) ... (concept predicate role))
where all the predicates except the last are mental attitude predicates. Topical subnet classifications are of the form:

((concept **tp.mental-attitude** *) (concept* **tp.mental-attitude** *) ... (concept topic))*

The main and topical classifications are both derived from an intermediate classification of the formula. This classification is based on the trigger keys involved in a formula. For a conditional formula, this will be a subset of the literals in the formula. For other formulas, the trigger key will be the formula itself. Each trigger key is separated into component parts - in particular the predicate and arguments. An intermediate classification is made which consists of lists of the form:

(type-predicate) (for type predications)

(concept predicate role)
This classification is used directly for the main classification, unless a specialist is associated with *predicate* in the second form. In that case, *(concept specialist-name)* is used. For topical classification, *(type-predicate)* is expanded to *(type-predicate* **tp.spec** *)* , and the combination of the *predicate* and *role* in the second part is used to determine the *topic* , resulting in classification *(concept topic)* .

Number constants are not included in the classification, nor are simple strings. If we are trying to determine the relation of something to the number 3, finding information on how other objects relate to that specific number will not be helpful - only information about the original object and other objects of that type will be. Using classifications involving such constants will only waste space and time needlessly.

If an argument is not a simple constant, the classification routines must do a little work to try to figure out which part(s) of the complex term to use. If the term is a functional term, the function itself is ignored, and all its arguments are used as individual arguments in the classification. For example,

((date year1 04 day1 00 00 00) before e3)
is classified under *(year1 time-specialist), (day1 time-specialist)* , and *(e3 time-specialist)* . If the term is modified by an operator, in most cases the operator is ignored, and the arguments to it used (like a function), but in some cases the operator is important (decided by whether or not they have a topic indicator list associated with them). For example,

((K wolf) fierce)
is classified under *(wolf fierce subject)* and

((To (L x (x scare Mary))) fun)
is classified under *(Mary scare object)* . (This should probably also be stored under things that are *fun* as well, but currently isn't).

Predicate operators are also ignored unless they are "important" (as decided for operators on terms). In this case, the operator is treated like a predicate, and the arguments to it combined with the arguments to the entire predicate. For example,

(John (make juice))
is classified under *(John make subject)* and *(juice make object)* , while

(Mary (very pretty))
is only classified under *(Mary pretty subject)* .

Another possibility for classification of such objects is to simply use the modified construct as the concept or predicate in the classification because then every time that classification is reached (through

forward or goal-directed inference), the classification must be taken apart and the modified construct examined to see what the "next level up" (or down) should be. It is better to do such determinations once, when the classification is first calculated, than every time such a classification is used. For example, something involving the term

(K (L x ((x wolf) and (x fierce))))

would have to be re-examined to decide that *wolf* should be used to determine the next level up - *four-legged-quadruped* .

When a formula is classified for input-driven or goal-driven inference, a special temporary property is placed on it called *lit-classes* - and the value of this property is a list, each element of which is a key literal in the formula, followed by the classifications determined by that literal (all of this done after the type literals have been examined so that variable's types are included). Both the input-driven inferencer and access actions in question answering then take this list, and look up items to compare with each literal based on its classifications. The other method was to completely classify the whole formula, and then compare each key literal with everything obtained by the classifications - however, this leads to many fruitless comparisons. For example, if one were trying to answer

*(E x_ep (x before yesterday) ((John kiss Mary) * x))*

the classifications are *(John kiss subject)* , *(Mary kiss object)* , and *(yesterday time-specialist)* . It would be fruitless to compare literal *(x before yesterday)* with anything retrieved from the classification *(John kiss subject)* .

Classification also keeps track of modal embeddings and computes equivalent classifications as well. So if *c1* and *John* are equal, and the formula being classified is *(c1 happy)* , its classifications are *(c1 happy subject)* and *(john happy subject)* .

2.3 Inference Requirements for Input-Driven and Goal-Directed Inference

The following system operations are used in both input-driven and goal-driven inference.

2.3.1 Comparison of Formulas

Formulas are compared for compatibility or for incompatibility. At the lowest level, unification is done on the variables involved. Specialists can help by doing the comparison - this is discussed in the chapter on specialists. This section will concentrate on the "ordinary" comparisons.

Formula comparison is done by a recursive routine called *compare-wffs* . For most comparisons, both formulas must be of the same type *prefix, episodic* , etc except that *prefix* formulas with variable predicates (from axiom schemas) may also match against *episodic* and *causal* formulas. The operator *nec* is ignored during formula comparison. Variable formulas of course may be compared against any other formula.

Corresponding pieces of each formula are then compared to see if the formulas are (in)compatible - predicates, arguments, quantifiers, etc. Specialists may also be used to help compare predicates (rather than doing the entire comparison).

During comparison, any formula can be considered to be equivalent to *((qq formula) true)* . In addition, formulas can be "lambdified" for comparison if they are being compared with a formula with only one argument and a variable predicate (e.g., from an axiom schema).

All comparisons, whether the "ordinary" type or specialist variety return a list of "comparison" records. Each of these structures (**comparison-info**) contains the following fields:

wff1 - the first formula being compared to see if it is (in)compatible with

wff2 - the second formula

lit1 - the literal from the first formula which was actually compared

lit2 - the literal from the second formula which was actually compared

subs1 - substitutions necessary in *wff1*

subs2 - substitutions necessary in *wff2*

residue1 - a residue (considered part of *wff1*)

residue2 - a residue (considered part of *wff2*)

Note that *residue1* and *residue2* will always be nil in the case of "ordinary" comparison - only specialists add these. Each subgoal action contains a pointer to a comparison-info record, and they are used immediately when calculated during input-driven inference.

The details of formula comparison are as follows:

compare-wffs *wff1 wff2 compatible &key prefer cleanup etc*
either formula is a variable - use unification (mp's only)
wff1 has a variable predicate and only 1 argument; *wff2* has more than one argument (mp's only)
call *compare-wffs* with *wff1* and *wff2* after doing reverse lambda conversion *on* *wff2*
wff2 has a variable predicate and only 1 argument; *wff1* has more than one argument (mp's only)
call *compare-wffs* with *wff1* and *wff2* after doing reverse lambda conversion *on* *wff1*
wff1 has predicate *true* and *wff2* does not
call *compare-wffs* with first argument of *wff1* and *wff2*
wff2 has predicate *true* and *wff1* does not
call *compare-wffs* *wff1* and first argument of *wff2*
both formulas have the same wff type - comparison is based on that type
constant - compare to see if equivalent
episodic
if last arguments (the episodes) are compatible
if predicates and negations are (in)compatible
- use *compare-wffs* to see if first arguments are compatible
if *wff1* is negated, and predicates and negation of *wff2* are (in)compatible
- use *compare-wffs* to see if first arguments are not (in)compatible
if neither formula is negated, and predicates are compatible
- use *compare-wffs* to see if first arguments are (in)compatible
prefix or *causal*
if predicates and negations are (in)compatible
compare argument lists of *wff1* and *wff2* to see if they are compatible
also try comparing using specialists
modified

if negations can determine (in)compatibility and the operators are compatible
compare arguments lists to see if each consecutive pair is compatible
logical (embedded only, and never negated)
if predicates are equal, and both formulas have the same number of arguments
compare arguments from the two formulas for (in)compatibility using *compare-wffs*
quantified (embedded only)
if quantifiers are compatible, and variables are compatible
if both formulas have a restriction
use *compare-wffs* to compare the restrictions for compatibility
else if we are comparing with an axiom schema, not comparable, else keep going
use *compare-wffs* to compare the main clauses for each wff for (in)compatibility
wff1 is *quantified* , and *wff2* is not *quantified* or *logical* (used with mp's)
use *compare-wffs* to compare the main clause of *wff1* with *wff2* for (in)compatibility
wff2 is *quantified* , and *wff1* is not *quantified* or *logical* (used with mp's)
use *compare-wffs* to compare *wff1* with the main clause of *wff2* for (in)compatibility
wff1 is modified with operator *nec* (ignore the *nec*)
- use *compare-wffs* to compare the first argument of *wff1* with *wff2* for (in)compatibility
wff2 is modified with operator *nec* (ignore the *nec*)
- use *compare-wffs* to compare *wff1* with the first argument of *wff2* for (in)compatibility

When comparing predicates:

compare-predicates *neg1 pred1 neg2 pred2 compatible*
if either predicate is a variable
- use unification to compare
if predicates are equal
- use *neg1* and *neg2* to determine (in)compatibility
if specialists can determine relationship between *pred1* and *pred2*
- use that relationship with negations to determine (in)compatibility
if predicates have same pred type - compare based on that type
modified
if negations can determine (in)compatibility and the operators are compatible
compare arguments lists to see if they are compatible
lambda
- use *compare-wffs* to compare the predicates
if one of the predicates is a *lambda* predicate
try reducing the lambda predicate, and then comparing

When comparing operators, the only interesting case if modified operators, which are compared just like modified predicates (operators compared, and argument lists compared). Similarly for terms which are functions or modified terms.

2.3.1.1 Unification

At the lowest level of formula comparison is unification - variable with variable or constant, and constant with constant. The unification routines are used both by the comparison just described, as well as by specialists which are trying to compare two formulas. Equivalence classes are taken into consideration when comparing constants. Specialists may provide their own "constant-comparing" routines, which do nothing more than indicate that the specialist is willing to work with the literals even if they don't match perfectly in every respect.

The union-find algorithm from "The Design and Analysis of Computer Algorithms", by Aho, et al, is used. A tree-like data structure is used to keep track of which argument is currently substituting for other arguments. The root of the tree (or the 'name' of the set which the tree represents) substitutes all other arguments in the tree (or members in the set). Access to this structure consists entirely of union and find operations. Using a union-find algorithm with near linear time complexity allows the unification algorithm also to be near linear.

Sorts may optionally be taken into consideration when unifying (the default is to check them), and no sort is considered to match any sort (actually, this really shouldn't be done that way, but we have to allow for the possibility that something is being compared before it has its sort assigned, or the user forgot or neglected to attach a sort).

A "prefer" flag tells which variable to choose when unifying two variables. Normally the one that creates the most optimal tree would be chosen, but because of the way subgoal creation and rule instantiation work, it is often preferable to have the variable in the set-of-support wff be substituted for the variable in the retrieved wff, rather than vice versa.

2.3.2 Substitution

Substitution is a non-destructive process which recursively creates a new list formula, based on a normalized formula and a list of substitutions. Whenever an item matching an item to be substituted for is found, it is replaced in the output list with the thing to be substituted. No substitution occurs inside a quoted expression, but does inside a quasi-quoted expression.

2.3.3 Formula Verification and Simplification

When verifying a formula (during consistency testing, subgoal creation, or input-driven inference), it is possible that part of the verification process may simplify it - either by having specialists evaluate functional terms, or by evaluating a literal in a complex formula to YES or NO. The result of the verification will either be YES, NO, UNKNOWN, or a simplified version of the original formula.

2.3.3.1 Simple Lookup

First the formula is examined to see if it has a probability attached - if so, it has been asserted and is true, with that probability. There is a possibility that its negation is also stored, and has a higher probability, so it must be checked for as well before answering YES.

The formula to be evaluated is classified, and the resulting classifications sorted. Only the first classification (and any that are equivalent to it) are actually looked under (if the formula is there, it should be under all those classifications). Each formula found under that classification is compared against the

given formula, taking equivalence classes into consideration when comparing terms, and specialists may help in comparing predicates. The result is either YES, NO, or UNKNOWN. No simplification is possible here.

2.3.3.2 Using Specialists

Specialists can be used to evaluate literals. First any functional terms are evaluated (e.g. *(max-of 4 5 9)* could be evaluated to *9*). Even if the resulting literal cannot be evaluated to YES or NO, this simplified version will be returned. All interested specialists are invoked to evaluate the formula until one answers with a YES or NO. Details on this are provided in the chapter on specialists.

2.3.3.3 Complex Formulas

For complex formulas, such as quantified formulas, conjunctions, disjunctions, and implications, the key literals (trigger and non-trigger) are obtained and evaluated using one of the above two methods. The results of those evaluations are substituted for the literals in the actual formula, and this is then simplified. It may simplify to YES or NO, or may just become a shorter formula because some literals have been dropped. For example, if *(lrrh girl)* is known, *((lrrh human) implies (lrrh happy))* can be simplified to *(lrrh happy)* , and *((lrrh girl) or (lrrh boy))* can be evaluated to YES.

2.4 Input-Driven Inference

Input-driven inference is inference done on newly input formulas using either story knowledge or meaning postulate axiom schemas. The same process with minor changes is used by both. Inference with story rules is easier to understand and so is described first.

2.4.1 Inference with Knowledge Rules

When the new formula is classified, a list consisting of its trigger keys and their classifications is placed on the formula. This is then used to start the inference process. For each trigger key, its classifications are used to find other formulas which then have their trigger keys compared to the input formula's key. If there is a successful match (when comparing against a negative key, the keys should be compatible; against a positive key, incompatible), the substitutions determined are made in the two formulas, and in the remaining keys of the retrieved formula. These remaining keys are then weeded out, removing any which are made up entirely of constants, and ordering the rest of them so that those with the least variables come first. Formulas are obtained to try to satisfy these keys. Substitutions resulting are again made in the formula and remaining keys. When all keys that can be satisfied are, the process stops. The resulting formula is then verified and entered into the system as an inference.

This process is called "rule instantiation". If all keys are matched (i.e. all variables are matched), this is *total* rule instantiation; otherwise *partial* rule instantiation. The following describes the process at a lower level. It is slightly more complicated than this, but this gives a general outline.

make-input-driven-inferences *(item)*
get list of trigger keys for *item* and their classifications
for each key *k*

```
    for each classification  c  of  k  (that we haven't looked at for this key)
      for each formula  f  under  c
        if we haven't looked at it already,
          and
        it hasn't been used in this inference chain yet
        then
          ; try contrapositive - except for mp's and if the input was a type lit
          for each +trigger key  k2  in retrieved formula
            unless  k2  is a type literal and  k  is in the consequent of a conditional
              compare the two keys for incompatibility
              if successful, (try-with-rest-of-keys item f k2 ...)
          ; regular
          for each -trigger  k2  in retrieved formula
            compare the two keys for compatibility
            if successful, (try-with-rest-of-keys item f k2 ...)
      end for formula
    add super classifications of this class to list of classes to examine
    end for class
  end for keys
end of make-input-driven-inferences
```

try-with-rest-of-keys *(item wff key trigger-keys other-keys comparison-info subs1 subs2)*

```
  if there was a successful comparison (  comparison-info  )
  then
    do substitutions throughout  item  ,  wff  , and lists of keys
    add to  subs1  and  subs2
  remove any  trigger-keys  which consist of constants only
  if no  trigger-keys  left
     (finish-inference  item wff trigger-keys other-keys  )
  order  trigger-keys  so those with fewest variables are first
  classify first key  k in  trigger-keys  , and lookup formulas to compare against it
  for each formula  f  found
    if  f  is a simple formula (no disjunctions or conditionals)
      compare with  k  for compatibility (if  k  context is - then compare for incompatibility)
    try with rest of keys (whether successful comparison or not)
  end for formula
end of try-with-rest-of-keys
```

finish inference *(item wff trigger-keys other-keys subs1 subs2)*

```
  verify each key in other-keys
  if any variables were not matched, and we want total rule instantiation
```

return with no inference
if *wff* is an mp and not all meta-level variables have been matched
return with no inference
do substitutions throughout *item* and *wff* , also subbing in evaluation values for verified *other-keys*
enter new formula (the normalization and consistency checking will simplify it)
end of finish-inference

2.4.1.1 Input-Driven Inference Termination Criteria

Forward inference continues to try to infer new formulas from the input formula and the resulting inferred formulas until an inference is made which is not very "interesting". This calculation is based on the combination of probability and an "interestingness" value assigned to the formula (probability and interestingness are multiplied and compared against ***interest-threshold***).

2.4.1.1.1 Interestingness

Interestingness, in this system, is a numeric value given to a concept or a formula which indicates in a quantitative way how "interesting" or "valuable" a piece of information is. This is used to decide what facts are worth following up and which aren't. For example, *there is a crack in a wing* is far more interesting than *there is a crack in a seat* , and we want to follow up the consequences of the first fact, but not necessarily the second. In EPILOG, the interestingness is currently based on the "interestingness" of the arguments in a formula (at the time the formula is entered), and the "interestingness" of the predicate involved. Interestingness of individuals is based on what has already been predicated about them. Interestingness of predicates is based on the topics they indicate, inherited from the parent if the predicate is in a hierarchy,or may be user specified. In addition, a formula may "inherit" some interest from the formula which it was inferred from. This "inherited interest" can overcome some dips in the interestingness in chains of input-driven inferencing.

2.4.1.1.1.1 Backing Up Interest Along Causal Chains

A part of the system which is under investigation backs up high interest levels from interesting consequents to their causal antecedents. This results in high interest levels for things which cause interesting things. This is a rather difficult process, because it involves defining a "causal chain". Currently the predicates *involve* , *because* , and *cause-of* indicate a causal connection, and modified formulas (with *nec* , etc) which contain those predicates do as well. This puts a great burden on the user however, to ensure that there are rules to maintain the causal connections. Since it has limited usefulness without complete causal chains, and takes up time looking for them, the system has this option turned off initially, but it can be turned on by tweaking ***back-up-interest*** .

2.4.1.2 Known Problems and Possible Improvements

Currently inference involving either two conditionals (formulas with either a conditional predicate - a number, *implies* , or <=> - or those quantified with a quantifier other than *E,* or *WH*), or a conditional

and a disjunction are NOT attempted. A method of disjunction management is needed for this, as well as better indications of what a "useful" inference is. More work with termination criteria and interestingness are required as well.

2.4.2 Meaning Postulate Inference

The inference process is almost identical to that described above, except that all "meta" variables MUST be matched, and an mp may be applied whenever it can be - where the input-driven inference allows a rule to apply only once in an inference chain. All input formulas have meaning postulates applied, regardless of interest level (except other mp's and schemas). Meaning postulate inference happens only when story facts or rules are input, not when an mp is input (contrast this to regular knowledge rules which can be applied after the fact).

2.4.2.1 Known Problems and Possible Improvements

Currently meaning postulate axiom schemas are input as rules which have variables over predicates, formulas, quantifiers, etc, and are carefully controlled to prevent explosions. However, this representation is not adequate for all the types of schemas we need to be able to use.

2.5 Question Answering

EPILOG can answer YES/NO questions posed in episodic logic, as well as some wh-questions. A wh-question (to this system) is a quantified formula whose top level quantifier is *WH* . (Actually, a WH quantifier at any level will make it a wh-question, to EPILOG, although it makes no semantic sense to ask questions containing a WH quantifier with some other quantifier at the top level). The YES/NO question answering mechanism described below is used to answer the question, with one difference - it doesn't stop after the first answer is obtained - it continues looking for more YES answers. The user may specify that all answers are to be found, or only n answers (this is specified when the question is asked, as a parameter to the **question** routine).

When queried, the system first tries the fastest, easiest method available - a simple lookup, or specialist evaluation. If this fails, a user-set effort level (***question-effort*** , or given as a parameter to the **question** routine) governs how hard the system will work to try to answer the question. The effort levels are as follows:

0) Lookup/verify only - the system does a simple lookup of the question, or may use a specialist to evaluate it (using *verify*). No variable matching is done, so the only questions that can be answered contain specific individuals (*?(lrrh pretty) (wolf1 like-to-eat lrrh)*), and equivalent formulas must exist in the knowledge base or specialists' domains.

1) Allow natural-deduction-like breakdown of subgoals. This also allows the use of the agenda. For questions which consist of several parts, or with variables to be matched, this is the minimum level required (especially for wh-questions). No inference is actually done, just retrieval of facts and unification between the facts and the question. Conjunctions and disjunctions can be handled by the natural-deduction-like breakdown of subgoals, but conditional questions (and some disjunctions) require assumptions and cannot be handled with this effort level.

2) Allow inference. Not only facts are allowed to be used, but also conditional statements (rules and to a limited extent, meaning postulates). Goal reduction is used to make inferences with the question and knowledge base rules.

3) Allow assumptions to be made. For some kinds of subgoals, the natural way to solve them is to assume one part, and try to prove the rest (for conditionals and disjunctions). These assumptions are actually entered (temporarily) into the knowledge base, and input-driven inferencing is optionally attempted on them. This is the most expensive operation because the assumptions must be retracted when the question is finished.

Besides the effort level, there are numerous controls set up so that the user can control the question answerer if he so desires. Some of these control agenda positioning, and some control access actions.

To answer a question, the system takes the question and its negation as the *proof* and *disproof* attempts, respectively, and then tries to prove both simultaneously - if the proof subgoal can be proven, the answer to the question is YES; if the disproof subgoal is, the answer is NO. Each subgoal (top level or result of an inference) is put through a natural-deduction-like splitting process, where it is split into simpler subgoals. Formulas from the knowledge base are retrieved and applied to the lowest level subgoal to produce new subgoals.

2.5.1 The Question Answering Process

An agenda of actions is used to keep track of what can be done to try to answer the question. There are two types of actions on the agenda: *access* actions, and *subgoal* actions. When a new subgoal is added, a natural-deduction-like splitting process recursively splits the subgoal into simpler subgoals, building a subgoal tree, until a "leaf" subgoal is reached (non-splittable), and access actions are added for that subgoal. A "difficulty" measure is added to each subgoal depending on the split done.

Access actions use the classifications of a set-of-support *proof* or *disproof* subgoal to look for more formulas to compare against the subgoal. If there are many trigger keys in a subgoal formula, there will be a separate access action for each one. "Super" and "sub" (using parent types and instances) classifications are accessed as well to find formulas to apply. Both regular backward chaining and contrapositive (forward) inference are supported here. If a successful comparison is found, a *subgoal* action is added for it.

Subgoal actions prepare and add a new subgoal using "goal reduction" or "goal chaining" - the dual of "rule instantiation". The new subgoal is calculated from the two formulas and the comparison information (substitutions, residues, etc) from the *access* action that generated it. Answers (YES or NO) are propagated back up the subgoal tree, combining with other subgoals if necessary to determine the answer to the parent subgoal. If an answer is achieved at the root of a subgoal tree, the question is answered. Otherwise the new subgoal is added to the subgoal tree, and goes through the same natural-deduction-like splitting the original subgoals went through.

The agenda, at any time, contains actions for both the *proof* and *disproof* attempts, ordered by type of action (subgoal preferred to access, regular preferred to contrapositive), depth of search so far (deeper preferred to more shallow, up to a certain point, and then it works the other way), a combination of interest and complexity (the interest is divided by the complexity to "normalize" it, and a high value of this ratio is preferred to a low value), the "difficulty" value determined by subgoal splitting (low preferred to high), and probability (high preferred to low). The agenda ordering parameters will be discussed in more detail shortly.

The system takes one action from the agenda at a time, and does it. If a subgoal action result is YES, the answer is backed up to the parent subgoal. This may or may not solve the parent subgoal as well - if so, the answer is backed up again. If the top level *proof* subgoal answer is YES, the answer to the question is YES; if the top level *disproof* subgoal answer is YES, the answer to the question is NO. We stop going through the agenda if any one of the following stopping conditions are met:

1) There are no more items on the agenda. There is not enough information to answer the question.

2) The maximum number of iterations ***qa-iterations*** has been reached. The question cannot be answered in the limits set by the user.

3) An answer has been found, and it is of sufficient probability. A threshold (***question-threshold***) allows us to ignore answers that are not very likely, and to keep searching for others. If the minimum effort (***minimum-effort***) has still not yet been expended, the system will continue going through the agenda and not stop here. It is possible that there are several paths leading to (possibly different) answers, which must then be combined. The ***minimum-effort*** parameter tells the system it should try at least that many agenda items, even if it finds an answer before using that many, just in case there are more answers. Answer combinations will be described shortly.

4) This is a wh-question, and the current depth is deeper than that required for any of the other answers to the question. For wh-questions, often all the answers are obtainable with the same amount of work each - i.e. the same depth. The user can decide how much of a difference is allowed here (***max-wh-difference***). If this parameter has been set, the maximum depth is set when the first answer to a wh-question is found (the depth needed for that answer + the maximum difference allowed); the agenda is reordered so that depth is more important, and then the depth of each agenda item is compared to the calculated maximum. If it is exceeded, the system stops. The reordering is necessary to ensure that we don't stop prematurely, and has the effect of changing the search from a depth-first oriented to a breadth-first oriented search.

If the system stops, invoking the routine **question** (or **q**) with no arguments continues the previous attempt.

2.5.2 Subgoal Splitting

Subgoals are put through the splitting process whenever added, whether the subgoal is a top level subgoal (the question or its negation), the result of a previous split, or a new subgoal resulting from an inference. A difficulty measure is added during the splitting process - splits which requiring assumptions have the highest difficulty; those requiring no splitting have the lowest difficulty. This difficulty measure is also cumulative - the difficulty of a particular subgoal is the sum of the difficulty measure obtained in each split required up to that subgoal. The subgoal tree is built by recursively splitting subgoals using the following:

1) For a conditional subgoal, the antecedent is assumed, and a subgoal is created to prove the consequent. The consequent subgoal also goes through the splitting process. A high difficulty level is assigned to these subgoals - ***quantified-difficulty*** (20) for quantified subgoals, ***conditional-difficulty*** (30) for implications.

2) If the effort level is not high enough to support assumptions, this type of subgoal will not even be attempted. For a conjunction where all the parts are "independent" (i.e. none of them has

an unquantified variable from another part), each part becomes a new subgoal, and the "child" subgoals' answers will be combined with *and* to get the parent subgoal's answer. Each child subgoal also goes through the splitting process. A difficulty level of ***split-difficulty*** (10) is assigned to this subgoal.

3) For a disjunction (again with "independent" parts), the negation of all except the first part are assumed, and a new subgoal is added to prove the first part. A difficulty level of ***assume-difficulty*** (30) is assigned to these subgoals. If the effort level does not support assumptions, the disjunction is split into separate subgoals for each part, and the child subgoals' answers will be combine with *or* to get the parent subgoal's answer. In this case, the difficulty level is set to ***split-difficulty*** , just as for conjunctive subgoals. Each child subgoal also goes through the splitting process.

4) All other subgoals are considered non-splittable "leaf" subgoals, with difficulty 0. Access actions are added for these to the agenda.

Assumptions are actually entered into the knowledge base during this process. They are flagged as assumptions, and may only be used in verification or inference by the subgoal resulting from the split where the assumption was made, and any descendants of that subgoal. Input-driven inference may optionally be attempted on the assumption. The assumptions and their inferences are all retracted when the question is finished.

The resulting subgoal tree is a tree of **subgoal** records, each of which contains the following fields:

parent - a pointer to the parent subgoal record. If this is null, this particular subgoal is either the top level *proof* or *disproof* subgoal.

proof-prop - either *proof* or *disproof* , depending on which attempt this subgoal belongs to.

subgoal - the normalized formula this subgoal is set up for. If this is the top level *proof* subgoal, the formula is the normalized question; if the top level *disproof* subgoal, it is the negation of the question; and otherwise it is the subgoal to be solved at this level.

complexity - the complexity of the subgoal (regular wff complexity). This is used in positioning agenda items for this subgoal.

interest - the "interestingness" value associated with this subgoal. This is used in positioning agenda items for this subgoal.

inherit-interest - the interest inherited from parent subgoals. This is used in positioning agenda items for this subgoal.

prob - the support set used to determine probability for this subgoal. This is used both to position agenda items for the subgoal, and in the final probability calculation of the answer.

c-prob - the calculated number probability for *prob* . This is faster to use during positioning comparisons than recalculating for *prob* each time.

assumptions - a list of assumption formulas (and any inferences from them) which are legal for use in solving this subgoal (i.e. the assumptions were made by a parent subgoal).

answer - the answer for this subgoal, if known (*yes, no*). When sibling subgoals are solved, the parent subgoal combines all these answers to get its own answer.

combo - this indicates how child subgoal answers should be combined. Legal values are *and* and *or* . It is not necessary that all subgoals be solved to answer the parent, and whenever a child subgoal is

solved, its answer, along with the *combo* value, may indicate an answer for the parent immediately without waiting for other subgoals to finish.

subgoals - a list of pointers to the child subgoal records to be combined with the *combo* value.

depth - the depth of search this subgoal is at. Any inference adds another level. This is used in positioning agenda items for this subgoal, and also for stopping wh-question answering if it is desirable to control it so that it only gets answers that take about the same amount of work.

difficulty - the cumulative difficulty measure given to this subgoal by the subgoal splitting process. This is used in positioning agenda items for this subgoal.

rules - a list of the formulas which have been applied in a chain of reasoning from the top level *proof-prop* subgoal to get to this one. Each element of the list consists of the formula used, the key used in it, and the context of that key. This is used by access actions for the subgoal to prevent cycling back and forth, doing forward and backward chaining with the same rule.

disallowed-rules - list of rules which were used and require that a non-conditional be used before they may be applied again (prevents digging into a hole with transitive rules).

wffs - a simple list of the formulas involved in getting to this subgoal, including those used in subgoal reduction as well as those used in simplifying subgoals.

subs - a list of substitutions made for wh and existential variables. This is maintained for wh-questions only, where the answer to the question is the list of wh substitutions, not a YES or NO.

previous - a list of all previous subgoals (parents, grandparents, etc) in the inference chain leading to this subgoal. This is used as a fast method for testing that we aren't going in circles.

useful - a property given to a subgoal if its answer was actually used in solving its parent.

access-items - a list of the agenda access items for this subgoal. This is maintained so that after a subgoal has been solved, we can lower the positioning of those items to prevent repeatedly solving the same subgoal, rather than continuing with the rest of the question. This doesn't apply to wh-questions, however, where we do want to repeatedly answer the same question or portion thereof, finding different values for the wh-variables.

2.5.3 The Agenda

The agenda is simply a list of actions to be done to try to solve the question. Each action came from a particular subgoal in either the *proof* or *disproof* subgoal tree. The actions are ordered in the list based on a number of criteria to be discussed shortly, and may be rearranged during question answering as well. The question answering mechanism always takes the first item in the list to do though.

The agenda actions are **agenda-item** records, containing the following fields:

action-type - *access* or *subgoal* . This indicates what type of action is to be done, and thus which fields of this record will be used.

subgoal-parent - a pointer back to the subgoal record in either the *proof or disproof* tree which this action is attempting to help solve.

position - the position (number) calculated for this action, using the criteria given below.

subgoal-action - if this is a *subgoal* action, this will point to the **subgoal-info** record for the action (to be discussed shortly).

access-action - if this is an *access* action, this will point to the **access-info** record for the action (to be discussed shortly).

complexity - the complexity of this action. For access actions this is simply the subgoal complexity (copied from the parent subgoal). For subgoal actions, this is a combination of the complexity of the subgoal, retrieved formula, the keys which were compared, and residues, if they are involved.

interest - the subgoal "interestingness", copied from the parent subgoal.

inherit-interest - the subgoal's inherited interest , copied from the parent subgoal.

prob - the probability support set. For an access action, it is simply copied from the parent subgoal, but for a subgoal action, it also takes into account the formula compared against to get this action.

c-prob - the calculated value of *prob* .

rules - the list of rules and keys used in them from the subgoal parent. For a subgoal action, the new rule just compared against is also included.

disallowed-rules - the list of temporarily disallowed rules from the subgoal parent. For a subgoal action, the new rule just compared against may also be included.

2.5.3.1 Agenda Positioning

A large number of factors come into play in trying to decide which of a multitude of possible actions will lead to an answer the fastest. This is a very difficult problem to solve, and by using a number of different parameters we hope to at least approximate good behavior here. The agenda position is a calculated number, and agenda items are arranged with low numbers high in the agenda, and high numbers at the bottom. The parameters are:

Probability - The component of the agenda position contributed by the probability is the product of the current calculated probability of the subgoal and the ***prob-importance*** parameter (100). Naturally high probabilities are preferred to low - which would mean that if there were several possible paths leading to answers we'd like the one with the highest probability. This doesn't always work out in practice, although paths of the same length may work in the desired manner. Initially the probability is not known. Previously 1 was used here, but it was discovered that this made the system "prefer" some of the old initial subgoals to the more promising new subgoals, just because the new ones had incorporated actual knowledge base facts with probabilities less than 1. To compensate for this, the probability used for initial subgoals in the agenda positioning is ***intial-prob*** , which is initially set to .75. The negative value of this component is actually used in the agenda position number, since high probabilities are good and low ones bad.

Depth/Rank - The component of the agenda position contributed by the depth/rank is the product of the depth (i.e. number of inferences used) of the subgoal and the ***rank-importance*** parameter (100). Higher rank is preferred over lower, i.e. we'd like the system to continue work on a promising inference chain. Up to a point, anyway. If the question answerer goes off on a tangent, it may continue working on a fruitless inference chain and go off into never never land wasting resources. Since most of the inference chains required really aren't that long, this can be controlled by penalizing the depths beyond a certain limit, rather than rewarding them. The parameter ***qa-depth*** contains a "maximum" depth. When a subgoal gets beyond half this depth, its depth is subtracted from the maximum, and that number is used in the calculation, rather than the depth itself. This has the effect of tapering off the depth contribution after the midpoint until when the depth reaches the maximum it does not contribute at all. This will position it lower in the agenda, allowing other subgoals to be tried. If this is the only path available, it

will still be followed, so nothing will be lost. The "maximum" is initially set to 20, which means that subgoals with depths deeper than 10 will start to feel the pinch. The negative value of this component is actually used in the agenda position number, since (initially) high ranks are good.

There is an additional factor which may be involved in the depth calculation - for wh-questions only. If ***max-wh-difference*** has been specified, then when the first answer is obtained for a wh-question, a maximum is set using the depth that answer required and the maximum difference. The agenda is then reordered, with items with a depth higher than that maximum highly penalized (by subtracting a large number from their depth before using it in the calculation - this is either the maximum depth ***qa-depth*** if it is specified, or 10). This creates a discontinuity into the positioning function, which causes the items with depths lower rise to the top of the agenda, regardless of other considerations, making the search more "breadth-first". All subsequent agenda calculations for that question answering attempt use the new depth calculation.

Complexity and **Interest** - The complexity of the subgoal (based on the type of formula, number of variables involved, etc) and the interestingness of the subgoal (based on how "interesting" that combination of predicate and argument are) were initially treated separately in the agenda position calculation. Low complexity items were preferred, and high interest ones preferred. However, often the ones with high interest also had high complexity, and this biased the agenda the wrong way. Instead, the ratio of interest to complexity is now calculated - "normalizing" the interest, if you like. This ratio is multiplied by the weight ***interest-importance*** (10) (and as in the previous two components, the negative value is actually used, since a high ratio is preferred to a low one). In addition, an inherited interest value may be used (if ***use-inherit*** is t). The maximum of the agenda items's interrest and its inherited interest is used in the calculation described above. This means that very interesting subgoals who then spawn a not very interesting child subgoal can still be solved, as their interest will be inherited by the child subgoal and it will still be considered interesting. The value in ***qa-inherit-amount*** is used to degrade the inherited interest as it passes down a long chain (this amount is subtracted at each step).

Difficulty - When subgoals are split into smaller subgoals to make answering them easier, some of the resulting subgoals will be easier to solve than others. The subgoals are each given a difficulty value, which is cumulative, so that numberous splits can lead to high difficulty values. This difficulty value is multiplied with the weight ***difficulty-importance*** (50) to give the agenda position component. This value itself is used, as low difficulty is preferred to high. Simple subgoals, which involve no quantification, disjunctions, conjunctions or conditionals have a difficulty level of 0. Quantified subgoals which are not conditionals (e.g. existentially quantified) have a difficulty level of ***quantified-difficulty*** - set at 20. These subgoals are considered harder to solve because they require more lookup to find constants which will match the variables. There is also more danger of going off on a tangent with them. Conditional subgoals which are solved with the "assume antecedent - prove consequent" method have a difficulty level of ***conditional-difficulty*** - set at 30. While this method is the only one that will solve a subgoal in some cases, it does require more effort, and the simpler subgoals (for the opposite answer for example) may be quicker to solve. Disjunctive subgoals which are solved with the "assume the negation of one of the disjuncts, prove the others" method have a difficulty level of ***assume-difficulty*** - set at 30. This method only gives an answer for some rather esoteric questions, so other subgoals are preferable. It must be available for some of the questions though. Simple independent subgoal splitting which requires no assumptions gives a difficulty value of ***split-difficulty*** to each sub-subgoal - this is set to 10. These subgoals require a little more work than a simple subgoal, because their results need to be combined, but they are certainly much easier than the other types of subgoal splitting.

Subgoal vs Access - If a match has been found during an access, it seems only logical to follow it up as quickly as possible and determine the resulting subgoal - after all, the answer might be right there! So

subgoal actions are preferred over access actions. Access actions are penalized by adding ***qa-access-weight*** to them (40). In addition, if two agenda items have the same calculated position, and one is a subgoal and the other an access, the subgoal will be placed higher. Also, when an access action has "had its turn", its position number is increased by adding a new component - which is the product of half the access penalty (***qa-access-weight***), and the subgoal's probability, and the access action is repositioned in the agenda.

Contrapositive vs Regular Backchaining - In most cases of question answering, especially with the subgoal splitting and handling capabilities the system has, only simple backchaining is needed. However, on occasion, a contrapositive inference may be necessary (a "forward" inference). Since these are not usually helpful, they are penalized by adding ***contra-weight*** to lower them in the agenda. In an access action, if the literal to be compared occurs is negated, this will likely be the case, and for a subgoal action, if the literal involved in the rule the subgoal was compared against occurs in a negative context we know this to be the case. The value of this additional weight is initially set to 20.

Proof vs Disproof - normally there is no bias for the proof or disproof attempt. However, if half of ***qa-iterations*** have been spent on only one attempt, to the complete exclusion of the other, the system will readjust the agenda to try to give the other attempt a try. It finds the top item in the agenda for that other attempt, and then "primes" it with an inherited interest of ***favor-interest*** , and gives it an agenda position ***favor-position*** above the current top item. This gives the other attempt a boost.

Subgoal comparison contents - if a subgoal action includes a residue in its comparison information, it is penalized by a large amount - ***residue-penalty** (10000). The action is still on the agenda if required, but is low enough that it will not interfere if other (less expensive) actions are available. Also, if the comparison substitutions show that only variables are substituted for variables, the penalty ***var-for-var-penalty*** (2000) is added. Often these actions are of the "embedded existential" variety, or applying a rule to a rule question, and there are usually easier ways to solve the question.

Meaning postulate vs Regular Access - access actions involving meaning postulate classifications are also added to the agenda for some questions. It is possible that a menaing postulate will have to apply before the question can be answered. Since this is the minority of cases, and we want to control the action of axiom schemas as much as possible, such actions are penalized heavily with ***mp-weight*** (10000). They are available if nothing else will solve the question, but won't interfere otherwise.

Solution vs no solution - if the subgoal an access action was added for is solved, other access actions for that subgoal (except for wh-questions) are moved down in the agenda by ***finished-weight*** (8000), since generally only one solution for each subgoal is required to solve the question. This prevents duplication of effort to solve the same (interesting) subgoal more than once, neglecting the rest of the question.

2.5.4 Access Actions

An access action looks up formulas to apply to a trigger key for a subgoal, and compares the keys of the formula against that trigger key. It starts initially with the classifications for the trigger key of the subgoal, and may also look at "sub" classes (instances and sub-types), and "super" classes (parents and individial types). Looking up a formula and comparing it are comparatively inexpensive operations, at least comapred to subgoal actions. To "equalize" the actions more, each access action may look at a number of formulas under a classification, and even a number of classifications. When a classification is exhausted, its super and sub classifications are added (for the first classifications, both are added; after that only super accesses are added to super ones, sub for sub so that we don't run in circles). Then the next classification in the list is used to retrieve formulas, and compare them to the trigger key of

the subgoal. When a successful comparison is found, or ***max-wffs*** formulas have been looked at, or ***max-class*** classifications have been seen, the access action will bestopped, and placed back on the agenda in a lower position. The next time this action is chosen, it will start where it left off so that no formulas are missed. If there was a successful comparison, a subgoal action for that comparison is added to the agenda.

Rules are prevented from firing in both the forward and backward directions, which could cause the system to oscillate forever. This is done by remembering the trigger key from the rule that compared successfully earlier in the inference path. If the rule is to be applied again, the same key must be used.

Also, care is taken with transitive rules to prevent recursively digging into a hole. A non-conditional wff must have been applied between successive applications of the transitive rule.

Access action information is kept in an **access-info** record, which is available to the agenda item the action belongs to. Fields of this record are as follows:

literal - the literal from the subgoal to try to compare with

context - + or -. This indicates whether the literal occurred positively (e.g. consequent) or negatively (e.g. antecedent) in the subgoal, which is then used to help determine whether comparisons should be for compatibility or incompatibility.

classes - the list of classifications yet to look under for this literal. This is initially set to the classifications for that literal within the subgoal formula, and is expanded to also include super and sub classes during the access action.

interest - the "interestingness" of the literal (this accompanies the literal, context and classifications in the 'lit-classes property that classification adds to a formula).

inherit-interest - interest inherited from subgoal parent

current-class - the current classification being examined. If this is set, it means that this access action was previously started, and got part way through the formulas retrieved for this classification when it was stopped (either by a successful comparison, or reaching one of the maximums).

wff-list - the list of formulas retrieved under *current-class* that have yet to be compared with *literal* .

classes-tried - list of the classifications that have already been looked under by this access action. It is possible when getting "super" classes that several classifications might have the same "super" classification, so this prevents duplication of effort, and wasting of resources.

wffs-tried - list of the formulas that have already been compared against by this access action. The same formula may be accessible from several different classifications, and this prevents duplication of effort.

flags - a list of flags to be used in determining which classifications to add. The flags include *-sub* , which says that sub classifications (including those using subtypes as well as instances) should be tried, and *-super* , which indicates that "super" classes (including parents of types) should be tried. These flags are both present initially on the first classifications, and then as the access moves down the sub classifications, only *-sub* remains, and up the super classifications, only *-super* remains. This prevents infinite looping and getting the same classifications over and over.

mp - indicates whether the classifications in this access action are for meaning postulate axiom schemas. There is no difference in the actual access action for these, but the agenda positioning routines do look at it.

2.5.5 Subgoal Actions

If both the original subgoal and retrieved formula have variables, the variables in the retrieved formula are replaced (temporarily) by variables with the same properties, but different names. The substitution lists are updated as well. This prevents messes when substituting later. The item to replace the matching key in the retrieved formula is first calculated. For a regular backchaining inference, the key will be replaced with YES; for contrapositive, with NO. If there was a residue, the residue is used instead of YES or NO. The matching key in the retrieved formula is replaced, and applicable substitutions done on it. The resulting formula is then negated, and this result is then used to replace the matching literal in the original subgoal, and the applicable substitutions are done on it as well.

The result of this is then verified. If it evaluates to NO, the new subgoal is discarded - just because this particular set of facts led to a NO subgoal doesn't mean there aren't others that will lead to a YES. If the answer truly is the opposite of this attempt, the other proof or disproof attempt will find it. If the subgoal evaluates to YES, this path in the subgoal tree has been solved, and the answer is propagated up to the parent subgoal, which will combine it with answers from its other children to see if it has ans answer itself. Otherwise the subgoal may be simplified. If so, the simplified version is kept. The new subgoal then goes through the splitting process, eventually resulting in access actions being added to the agenda.

Subgoal action information is kept in a **subgoal-info** record, accessible to the agenda item it is there for. Fields on this record are as follows:

key - the key literal compared from the subgoal formula. This corresponds to the *literal* field of the access action which generated this subgoal action.

context - the context of *key* - negative (-) or positive (+).

kn-wff - the retrieved formula which successfully compared against the subgoal.

kn-wff-key - the key literal from *kn-wff* which was compared against *key* successfully.

kn-wff-context - the context of *kn-wff-key* in *kn-wff* - positively occurring (+), or negatively occurring (-).

comparison - a pointer to the **comparison-info** record for the comparison. This contains the substitutions required for each formula, and residues if there were any.

2.5.5.1 Subgoals Involving Embedded Existentials

For some kinds of subgoals, the above process "loses" information, resulting in incorrect inferences. These are actions where the retrieved formula contains embedded existential variables, in either a conditional wff or a disjunction. In addition, the subgoal being compared against contains variables. Using the above method, the restriction of the existentially quantified variable gets ignored. For example, if the question to be answered is *?(E x (x thing) (wolf1 live-in x))* , and the rule to be applied is *(A x (x wolf) (E y (y natural-shelter) (x live-in y)))* , the normal approach gives subgoal *(E x (x thing))* , and then answers YES if it can find any *thing* in the knowledge base - so it will answer YES even though it never checked to see if a *natural-shelter* is a *thing* .

To handle this properly, it appears that the existentially quantified variable needs to be skolemized. The system takes such cases, and does whatever substitutions in the rule that it can. After doing so, the rule is "fired" and allowed to go through the input process, which will skolemize the variable, and add the restriction and main clause as assumptions. Then the subgoal we are trying to solve can be "restarted",

from the initial classifications, and this time the required skolemized information will be found to apply in the normal manner. In the above example, *wolf1* will be substituted for x in the rule, and the result of firing the rule is the assumptions *(c1 natural-shelter)* and *(wolf1 live-in c1)* . These easily solve the original question.

If there are no constant substitutions, for example if the question were *(E x (x wolf) (E y (y thing) (x live-in y)))* , the rule is put through the forward inference process, for one step. It will look up and find facts which are applicable, apply them, and generate the skolemized constant and facts about it. In this case it may do more work than is necessary - generally only one item will be needed to solve the problem and this could generate many (for every wolf it finds) - but at least it solves it!

If the retrieved formula is a disjunction with embedded existentials (e.g. *((E x (x cave) (wolf1 live-in x)) or (E y (y den) (wolf1 live-in y)))*), nothing is done at present - the subgoal is discarded. Even if it were possible to do something with it, other information is still required to finish solving the problem, and it is easier to apply this other information first, leading to ordinary subgoals, than to handle the embedded existential here. Although this was true in our test set, it may not be in all cases.

2.5.6 Example Question

This section takes some sample input and a question, and shows the various structures involved in the question answering process. The input formulas are:

((E x (x wolf)) 0.8 (x clever))

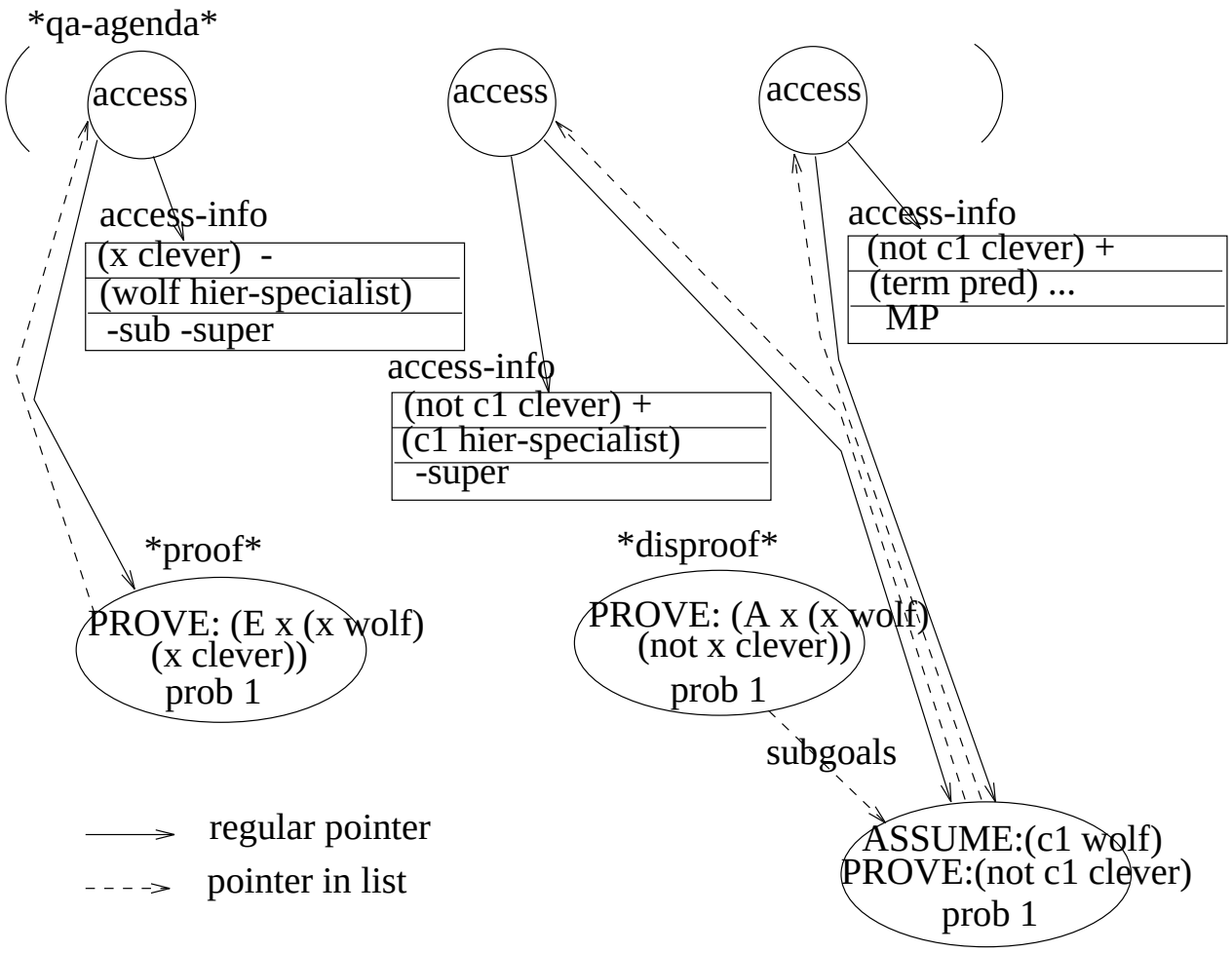
(wolf1 wolf)

and input-driven inference is not on (so the obvious conclusion is not drawn). The question is

(q '(E x (x wolf) (x clever)))

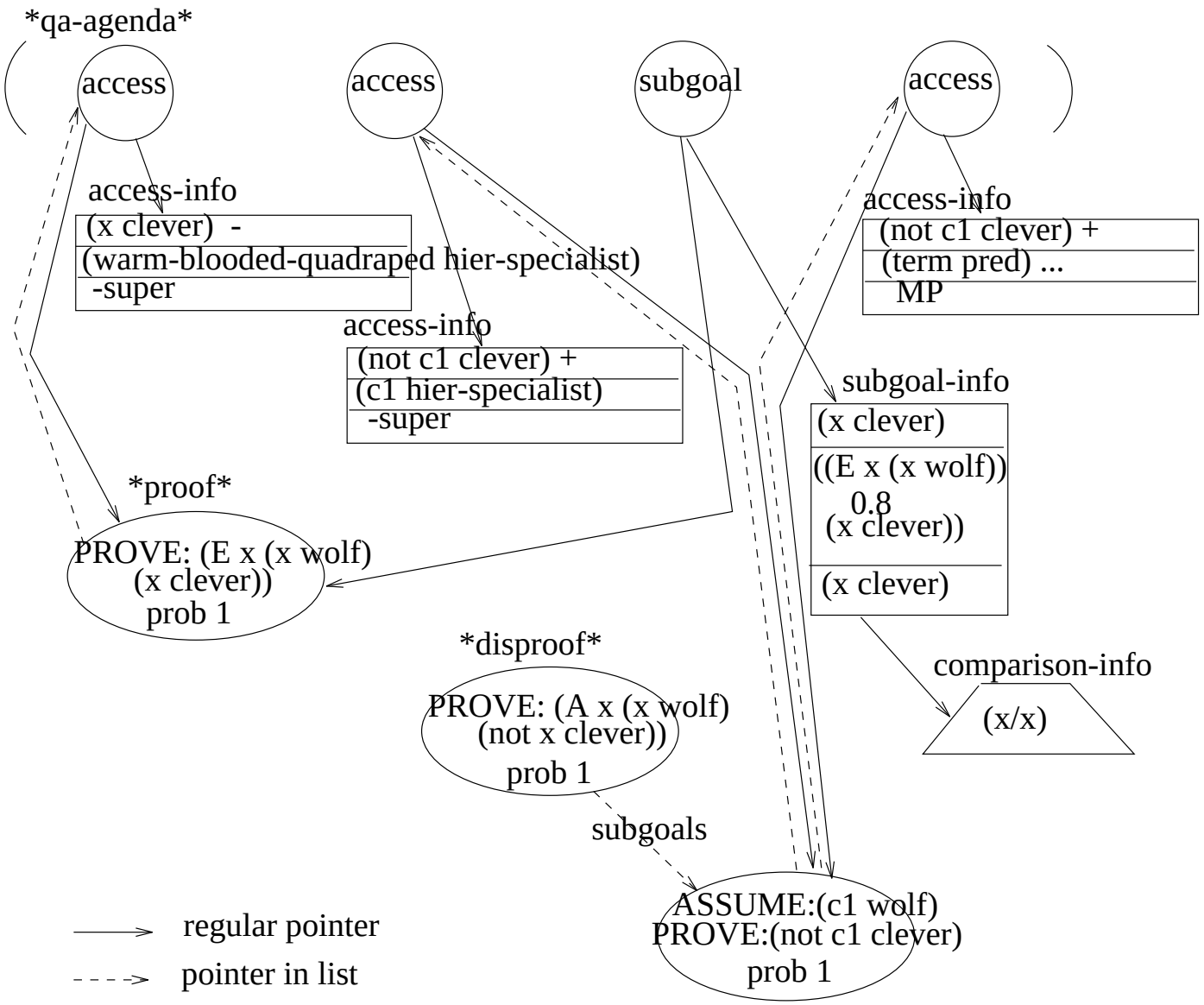
To solve this, the system looks for wolves which it already knows to be clever, as well as rules about wolves, animals, etc which involve them being clever. In this case, it will first find the rule about wolves being clever, and apply it, and then find the specific wolf to answer the question.

A series of diagrams follow, showing how the "question space" structures change as the question is answered. The first such diagram shows the initial space after the question has been asked, but before any of the agenda actions are attempted.



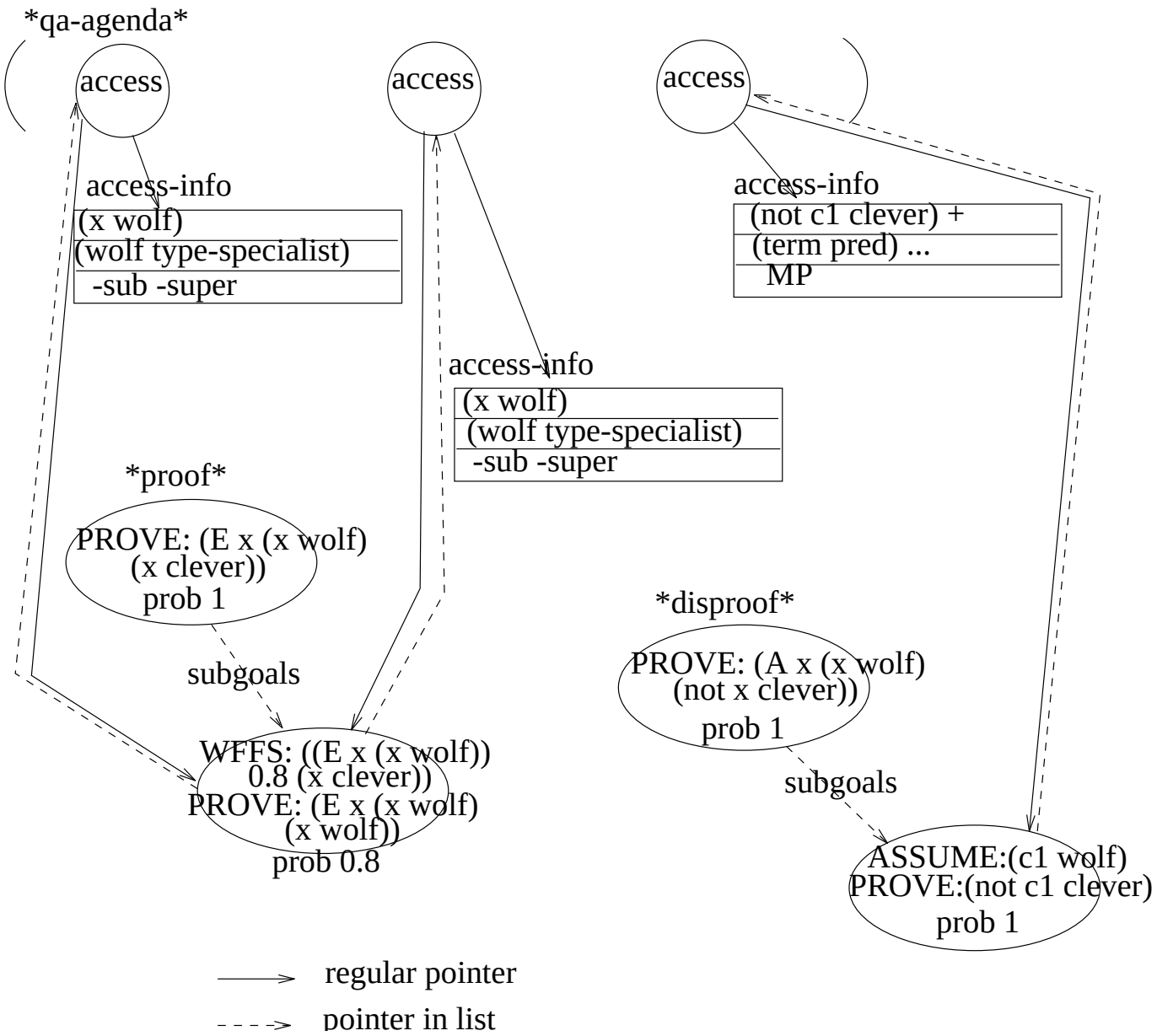
Initial Question Space

Now the system does the first access action, and finds the *((E x (x wolf)) 0.8 (x clever))* rule. This compares favorably with the subgoal we are trying to solve, so a subgoal action is added to the agenda for it. Because of agenda positioning criteria, this subgoal does not get placed at the top of the agenda, but rather a few actions from the top.



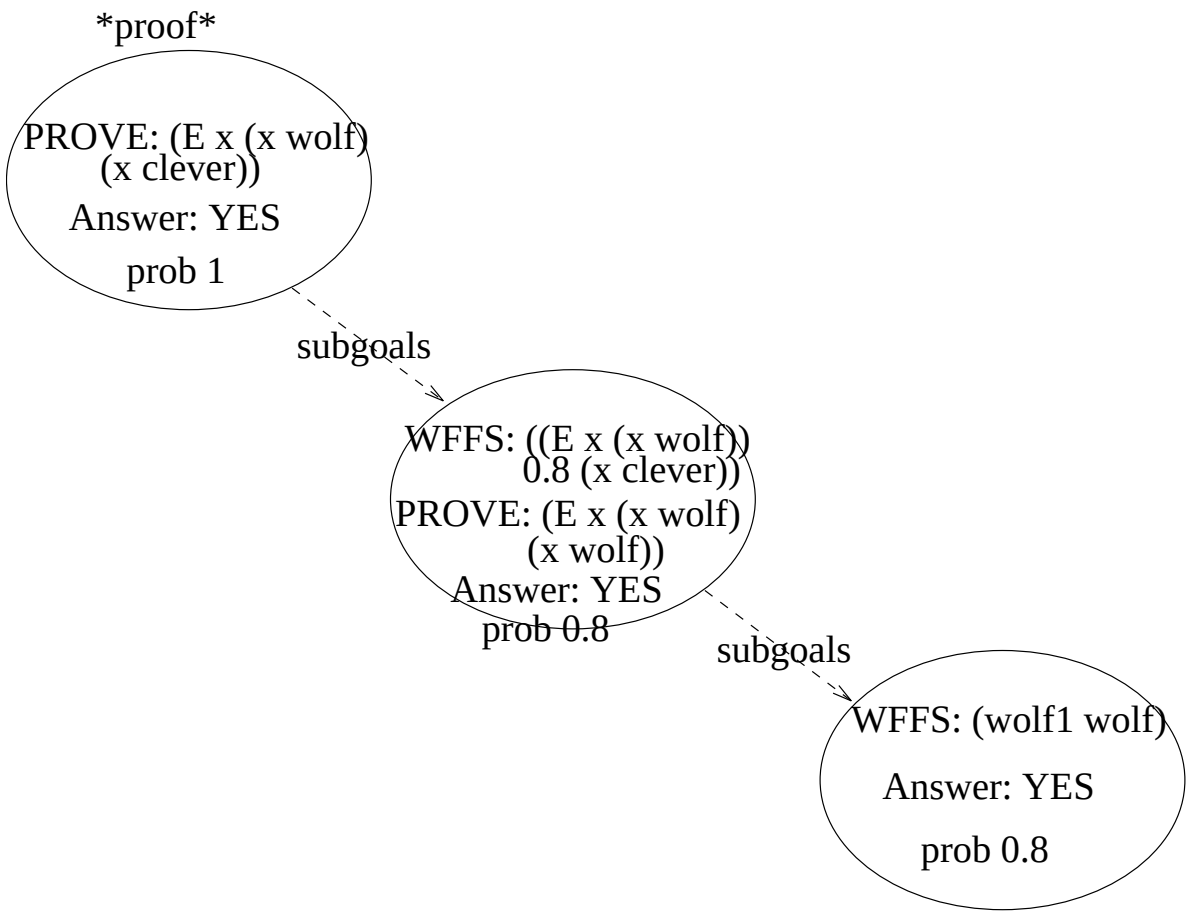
Question Space after finding *((E x (x wolf)) 0.8 (x clever))*

After exhausting the access actions in front of the subgoal action, the subgoal action is finally done, with resulting new subgoal *(E x (x wolf) (x wolf))* (factoring would help reduce this further, but the system can solve it as is). Note the new access actions added for the new subgoal (there are two for literal *(x wolf)* because it occurs twice - this doesn't happen often so is not tested for). Note also that a subgoal has been added to the ***proof*** subgoal.



Question Space after first subgoal action

Next the individual wolf is found via *(wolf1 wolf)* , and a subgoal action is added for the successful comparison between that formula and the subgoal. On the next step, the system does that subgoal action, and the result is YES. This answer is then passed up to the parent PROOF subgoal, and since it is the only subgoal required there, the parent answer is also YES. And that solves the top level ***proof*** subgoal, so the answer to the question is YES. The next diagram shows the final ***proof*** structure (without the agenda pointers). The agenda also has some additional items, but as we are finished, we can ignore those now.



Final proof subgoal after question is answered

2.5.7 Answer Combinations

When several answers are found, they may either support or contradict each other. The same formulas for combination as the assertions use are used by the question answerer. All positive (YES) answers are gathered, and the resulting probability determined using the rule

$$(1 - (1 - p_1)(1 - p_2)...(1 - p_n))$$

where the p_i are the probabilities of the different answers. All negative (NO) answers are gathered and combined in the same way. Order is not important when combining supporting answers.

If there were both positive and negative answers, the resulting probabilities of these are now combined, using the rules

$$p_{yes} = (p_{yes} - (p_{yes} * p_{no}))/(1 - (p_{yes} * p_{no}))$$

$$p_n o = (p_{no} - (p_{yes} * p_{no}))/(1 - (p_{yes} * p_{no}))$$

where the p_{yes} is the resulting probability from combining all the positive answers, and the p_{no} is the probability from combining all the negative answers. The answer is YES is the resulting p_{yes} is higher than p_{no}, NO otherwise. Note that combinations of negative and positive answers using these rules are order dependent - that is why all NO and all YES answers are combined first.

Note: the system is not yet sophisticated enough to determine inference path subsumption, or that

one path uses more specific information, so occasionally answers will be combined where one should have been discarded as redundant (e.g. two identical paths except that one uses *many humans are happy* while another uses *most girls are happy* - the *girl* rule is more specific than the *human* rule so that should be used RATHER than the other one, not in combination with it).

2.5.8 Known Problems and Possible Improvements

There may be other cases besides the embedded existentials that this method of subgoal creation does not work for. A somewhat better method of goal chaining has been devised, but not yet implemented. It will handle the cases described here, but could still have some problems with cases where more than 2 "comparisons" are needed at once.

Testing with residues has been very difficult because it is hard to think up examples that require using a residue to be solved. Testing to find that an appropriate residue is returned by a specialist during comparison is not the problem; what is done with it after is.

Application of mp's should be controlled more than is presently being done. This hasn't been a problem so far because few mp's have been used, and are not usually required in the qa process, so the penalty against using them has been enough.

The method of answer combination has the same problems as it does for input-driven inferencing: we are still looking for a better one. Identical WH answers are not being combined using this method yet either.

The agenda positioning routines ALWAYS require more work - both to find new parameters to use in the calculation for better results, and experimentation with the weighting values assigned for optimal performance.

Chapter 3

Specialists

3.1 Specialist Interface

There are a number of ways that a specialist may assist EPILOG - literal evaluation, simplification (via function evaluation), and literal comparison for compatibility or incompatibility. In addition, specialists may be involved during entry of literals so that they may update their own internal representations. Specialists may even assist each other by communicating immediate evaluations of functions or literals. When these aren't available, ties can be placed on objects so that later relevant information input will be communicated. This section describes the details of how the specialist interface is set up, how it determines what formulas are to entered, evaluated or compared, also how it decides which specialists might be able to help, and how communication is achieved.

Key considerations for the determination of interested specialists are as follows:

1) The decision must be fast
   Most formulas will not involve specialists, so we do not want a long drawn out check to see which, if any, specialists are likely to be interested in a literal or function.

2) The decision must be fairly accurate
   Before a specialist is called, there must be a fairly high degree of certainty that it will be interested. Specialists can do small, last minute checks themselves, but it would not help efficiency considerations to have a number of specialists called which ultimately throw the literal (or function) out.

The first thing we considered for the decision process was matching the given literal against a series of patterns which indicate the particular specialists. However, pattern matching can be quite slow, and as it turns out, the patterns for most specialists are simple enough that this really isn't necessary. Predicates (or operators, or functions) are the main determiner of the applicable specialists. Occasionally a predicate may be interesting to several specialists, and then the arguments may help make the decision. We found that if we checked the first argument and the predicate, this was fast and usually quite accurate for determining the interested specialists. All that remained then was for the specialist to check the remaining arguments itself. Arguments are checked by comparing the *sort* of the argument against a list of allowed first sorts for the specialist. *Sort* s are a subset of types, and serve to divide the arguments up to make the task of deciding on a specialist much easier. The above technique is like a pattern match in which the patterns consist of a single argument and a predicate, but much faster, since the specialists interested

in a predicate may be stored immediately on the property list of that predicate - so the determination of interested specialists is close to constant time - it is linear in the list of allowed argument sorts, which usually consists of only one entry, or at most two.

Function evaluation is even simpler, as there is really no formal pattern among the functions with regard to argument types. Here the function alone indicates the specialist, and it must check the arguments itself (which it can do much faster than any general purpose method).

For a literal being entered, all interested specialists are informed so that they can all represent the new fact in their own domains. For a literal being evaluated, each interested specialist is called, one after the other, until an answer other than *unknown* is achieved. This is also done for function evaluation.

For literals being compared, the intersection of the specialists interested in each literal is used as the interested specialists. Each one is called to attempt to compare the literals, so there may be many comparison results returned.

3.1.1 Specialist Setup and Activation

For EPILOG to know about a specialist, the specialist's name and the name of a file where its definition exists (usually in *epi/specname/definition.lisp*) are the only things needed. In addition, a description is associated with the specialist so that the user can tell what it is for. The actual predicates involved aren't known unless the specialist is actually activated. At that time, the definition file is loaded, which indicates the predicates, operators, and functions that a specialist deals with, and the allowed first argument sorts, as well as the place to load the specialist code from (usually *epi/specname/specname.lisp*). This code is loaded, and the specialist is added to the property list of all involved predicates, operators, and functions.

Predicates belonging to a specialist may be used without the specialist being activated, but will simply be treated as new predicates. If formulas with these predicates are entered, and later on the specialist is activated, the previous information will NOT be available to the specialist, and there may be some classification problems finding it again. If a specialist is to be activated, it should be done so before any formulas are asserted.

3.1.2 Deciding Who to Call with What

When the interface is first invoked with a literal to enter or evaluate, or a pair of literals to compare, or a functional term, a decision process is required to firstly determine exactly what is to be enter/evaluated/compared, and which specialists should be involved.

3.1.2.1 Determining the Exact Literal to Enter/Evaluate/Compare

The literal being entered, evaluated or compared may not be in a form acceptible to the specialists. If modal embedding is involved, this must be stripped off and the lowest level formula determined before the decision can be made about which specialist(s) should be involved. The modal embedding is not lost, but sent as an additional parameter to the specialists, along with the simpler, embedded formula. For example,

(john believe (that (event1 before event2)))

must have the modal context about *john* believing stripped off to get at the low level formula

(event1 before event2)

In addition, formulas may involve functions. Normally these functions are evaluated and their value replaces them before the specialists are called, but this is not always desirable in the case of literal entry, where information can be lost this way. For example,

((cardinality-of set1) lt= 3)

is asserted. If we simply evaluate the function *(cardinality-of set1)* , the result will be *nil* (unknown) - there is no information about it yet - that is what we are trying to assert! In such cases, the formula is "flattened" into two or more equivalent formulas, which are then entered into the specialists (but not the main knowledge base).

3.1.2.1.1 Modal Embedding

The stripping of the modal embedding is a recursive process which looks for modal predicates such as *say* , *think* , etc, and then tries to determine the formula actually being said or thought (this is assumed to be the last argument to the modal predicate). This embedded formula is also checked for modal embedding, recursively, until a non-modal predicate is found. The predicate *believe* is treated like other modal verbs unless the belief specialist is active, in which case it is handled as described in the section on the belief specialist (see below).

At each stage, the modal predicate, and all arguments except the actual formula are kept to build the modal context. This is then used as a subnet identifier for the specialists which maintain their own representations of facts in the knowledge base - each subnet is completely independent of any others. Subnet identifiers are of the form */name-predicate,* or */namename-predicate* for single level embedding, and */name-predicate/name-predicate* , etc., for multiple embeddings.

The argument which indicates the formula being said/thought/etc can take the form of a propositional term (of sort *propos*), in which case a knowledge base lookup is required to find a formula asserted of that term using the predicate _ . For example:

(john say p1)

and in the knowledge base, it finds

((event1 before event2) _ p1)

so *(event1 before event2)* is the formula to be entered/evaluated/compared by the specialists, with modal context */john-say* .

Alternatively, there may be an operator such as *that* operating directly on the formula (as in the example given earlier). The operators are stripped off, and again, the new formula is checked for modal embedding. (Although not necessarily legal syntax, the embedding stripping will accept a simple literal with no operator around it in this position as well.) For example, in the earlier example

(john say (that (event1 before event2)))

the embedded formula to enter/evaluate/compare is *(event1 before event2)* , and the subnet identifier would be */john-say* . If the formula were

(sue think (that (joe tell ron (that (event1 before event2)))))

the embedded formula is still *(event1 before event2)* , but now the subnet identifier would be */sue-think/joeron-tell* .

Note: for literal comparison, there are two literals involved. Both must have the same modal embedding context before any specialists will be called to compare them.

As the subnet identifier is being calculated, equivalent identifiers are also calculated, using the items

equivalent to the arguments. The result is a list of subnet identifiers rather than a single identifier. When entering information to the specialists, it will be entered to ALL of the applicable subnets, but when being evaluated or compared, only one subnet is used. When two items are asserted equal, all subnet information for them is asserted for the new equivalent item as well, so the subnets in the specialists are guaranteed to have all information that should be associated with a particular embedding context.

3.1.2.1.2 Flattening

Note that "flattening" is only done for literal entry, not for evaluation or comparison. In those cases we want the functions to be evaluated - and if there is no information to evaluate them, there won't be enough to evaluate the literal or compare two of them either.

Flattening occurs only when the formula in question contains a function, and that function has a related predicate (it must be set up this way when the function is defined). It occurs after any modal embedding is stripped off. For each such functionl term involved, a new constant is invented, with its name consisting of the function name and argument names all squished together (that way it is unique and retrievable). The functional terms in the original formula are replaced by these constants. Then for each of the terms, a new formula is created, using the related predicate instead of the function. The arguments consist of the list of arguments originally given to the function, followed by the new constant. The example given earlier

((cardinality-of set1) lt= 3)

would be flattened into two formulas:

(CARDINALITY-OFSET1 lt= 3)

(which would be interesting to the number-specialist), and

(set1 has-cardinality CARDINALITY-OFSET1)

(which would be interesting to the set-specialist).

Another example:

((duration-of event1) greater-than (elapsed event2 event3))

would be flattened to three formulas:

(DURATION-OFEVENT1 greater-than ELAPSEDEVEVT2EVENT3)

(interesting to the number-specialist),

(event1 has-duration DURATION-OFEVENT1)

(interesting to the time-specialist), and

(event2 exactly-before event3 ELAPSEDEVENT2EVENT3)

(also interesting to the time-specialist). With the communication described later, any further information given about either the duration of *event1* or the elapsed time between *event2* and *event3* will be used to update the other bounds.

If the function has a sort associated with it, the new constant is given that sort as well. All of the constants introduced above were of sort *number* . Note that the new constants do get entered into the specialists' representations, but are never sent back to the main EPILOG knowledge base.

There are only a few such function-predicate pairs in the system at present. They are:

Function	Predicate	Specialist
duration-of	has-duration	time-specialist
elapsed	exactly-before	time-specialist
cardinality-of	has-cardinality	set-specialist

The flattening capability is not really required, as it is possible for the user or high level translater to do this, but it provides a more uniform and consistent interface by allowing the most convenient form for both assertion and evaluation.

3.1.2.2 Deciding Which Specialist(s) Are Interested

At the point where the decision is to be made, all modal embedding will have been stripped off, functions evaluated if necessary, and flattening performed if necesary, so the actual predicate and arguments of the formula can be used to make the decision. A very simple test is used - the specialists associated with the predicate (or operator if an operator is acting on the predicate), or function for a functional term are obtained.

For a functional term, this is the list of specialists to try.

For formulas being evaluated or entered, an additional test is done. For each specialist in that list, the first argument's sort is compared to that specialist's allowed sorts. If they match, the specialist is kept in the list of interested specialists, otherwise it is discarded.

For literal comparison, two literals are involved. The specialists for the predicates/operators are found, and then the intersection of the two resulting lists of specialists is the list of specialists to use in the comparison. The first argument sorts are not checked at this time because during comparison, the specialist may unify the two literals and substitute something else in one of the first argument positions. Argument testing is left up to the specialists.

3.1.3 Literal Entry

The original literal asserted is first put through the modal embedding removal and flattening described above. Any functions that can be evaluated are, and replace the functional terms. Then for each literal to be entered, the interested specialists are found. For each interested specialist, the interface checks to see if there is an "enter" routine in the specialist's package. If so, this routine is called with the literal to be entered, the modal embedding context, and the predicate and arguments involved. All applicable specialists with "enter" routines are called - this ensures that each has the most up-to-date information in its own representation, for later evaluations and comparisons.

Note: negated formulas are also entered into the specialists, even though most of them will reject the formulas and not enter them. There are a few cases however, where negative information may be desirable to keep, and this allows the specialists that opportunity.

3.1.4 Literal Evaluation

The formula to be evaluated is first put through the modal embedding removal described above (but not the flattening). Any functions that can be evaluated are, and replace the functional terms. The resulting literal is then checked to see which specialists might be interested in it. For each one, the interface checks to see if there is an "evaluate" routine in the specialist's package. If so, this routine is called with the literal to be evaluated, the modal embedding context, the predicate and arguments involved, whether or

not the literal is negated, and an effort level to use to decide how hard to work evaluating the literal (this is usually ***specialist-eval-effort***). The specialists with "evaluate" routines are called one by one until one returns an answer other than *unknown* . Evaluation then stops, and this answer is returned.

Both negated and non-negated formulas are passed for evaluation to the specialists.

3.1.5 Function Evaluation

All functional terms in a function are first evaluated (recursively). Then the function itself (as described earlier) is used to determine the applicable specialists. If a routine by the name of the function exists in the specialist's package, it is called with the functional term and the arguments. The specialist itself is responsible for checking those arguments to ensure that the functional term is really in its domain. The applicable specialists are called one by one until one returns an answer (other than *nil*). Once an answer has been found, it is returned.

3.1.6 Literal Comparison

The formulas to be compared are first put through the modal embedding removal described above (but not the flattening). The resulting modal contexts from both formulas must be identical for the comparison to proceed - if they are not, the interface sends them back. Any functions that can be evaluated are, and replace the functional terms. The resulting literals are then checked to see which specialists might be interested in it.

For each specialist, the interface checks to see if there is an "incompatible-lits" (if testing for incompatibility) or "compatible-lits" (if testing for compatibility) routine in the specialist's package. If so, this routine is called with the two literals to be compared, the modal embedding context, the predicates and arguments involved, whether or not each literal is negated, and an effort level to use to decide how hard to work comparing the literals (this is usually ***specialist-eval-effort*** .) The specialists with the appropriate routines are invoked one by one, and each may produce comparison information which then adds more items to the question answering agenda. All applicable specialists are called because each may contribute different comparisons, and there is no way to know a-priori which will lead the fastest to an answer.

Both negated and non-negated formulas are sent to the specialists for comparison.

3.1.7 Communication between Specialists

Occasionally information is needed by a specialist which it does not contain, but it is possible that the information is available in some other specialist. To help with this, and to prevent any problems dealing with the ordering of input clauses, and to ensure that any changes are reflected wherever applicable, the specialists may communicate with each other through the specialist interface. They may request that a particular function or literal be evaluated by some other specialist, or may flag certain terms so that they are informed of any further input involving those terms.

3.1.7.1 Immediate Evaluation

Immediate evaluation is exactly what it sounds like - a specialist asks for an immediate evaluation of either a functional term (using **spec-eval-fn**), or a literal (using **eval-with-spec**). In both cases, the

request is sent to the interface, and it goes through the same procedure to find applicable specialists for the literal or function as it would if the main EPILOG system had sent the request. Thus the requesting specialist does not need to know the names of the specialist(s) which could answer the question, or even if there are any that can. In the case of a literal, it is assumed that there is no modal embedding to strip off - it should have the same modal context as the literal causing the specialist to make the request. The request is sent into the middle of the same process described earlier, bypassing the check for modal embedding.

When an answer is obtained, it is returned to the requesting specialist. If the request was a literal evaluation, the result will be *YES* , *NO* , or *UNKNOWN* . If a functional term, the result will either be another term, or *nil* (no answer found).

For example, if the formulas input were:

(mm_integer lt 3)

(event1 before (date 1991 mm 1 00 00 00))

the time-specialist would want to find out some information on *mm* in order to set up better absolute time bounds - in this case - the maximum. So it would send a request to evaluate the functional term *(max-of mm)* to the interface (via routine **eval-fn**). The interface would see the function *max-of* , and, using the method described earlier, invoke the number-specialist to try to find the answer. The number-specialist would check its own representation of *mm* and find that because it is an integer less than 3, its maximum must be 2. This would be returned to the interface, which would return it to the time-specialist. The time-specialist would then cause the upper bound on absolute time for *event1* to become *(1991 2 1 00 00 00)* .

3.1.7.2 Delayed Communication

Sometimes the information needed for an immediate evaluation is not yet available, or changes again later. In order to prevent ordering problems with input, and to keep everything as up-to-date as possible, specialists can make note of terms which are of interest to them. Later, when any new information about the terms is asserted or inferred, the specialist can be informed. This is done using a special property attached to each term - the **interested party list** . Each element in this list consists of a structure containing

a) the name of the specialist who is interested in the term,

b) the literal that was entered into the specialist that caused the interest in the term, and

c) the modal context for that literal.

Items are added to the interested party lists by the specialists when they are entering literals into their own domains. The specialists explicitly request that an addition be made to an interested party list.

Items to be checked for interested party lists are:

a) the terms involved in the current input formula, or

b) terms marked as changed by a specialist. Specialists explictly indicate to the interface that they have changed a particular item during entry of the input literal, or re-entry of any literals from the interested party list.

For each term to be checked, its interested party list is retrieved. If this list exists, for each entry in the list, the specialist indicated has its "enter" routine called again with the saved literal and modal context. The entry process will continue in the specialist as it did originally, but this time any immediate evaluations it does might get different results. Additionally, specialists can mark that they have changed information about specific terms, and this will also be used to trigger the specialists on the terms' interested party lists.

The interested party lists for arguments in a particular literal are obtained and used AFTER that literal has been entered into all interested specialists. These specialists may indicate that additional items have been changed and should have their interested party lists checked. Thus when the literals are being reasserted, more items may be added to be checked. Although it seems as though one could run forever, eventually the reassertion of a literal will cause no internal change in a specialist, so nothing else will be added to be checked.

In the example in the previous section, if the two formulas were reversed in order,

(event1 before (date 1991 mm_integer 1 00 00 00))

(mm lt 3)

the time-specialist would not have been able to update the bounds properly. Even if the information were available, it might later change (the bounds might become more strict) and so it would want to keep track of any changes to *mm* . To achieve this, the time-specialist would add the following entry to the interested party list of *mm* :

Specialist	Literal	Modal Context
time-specialist	*(event1 before '('time 1991 mm 1 00 00 00))*	*/*

(note that the functional term *(date 1991 mm 1 00 00 00)* has been evaluated to a record more easily recognized by the time specialist)

When *(mm lt 3)* is asserted, only the number-specialist is interested in it, so only it will be invoked to enter it. When it has finished, the interested party lists for the terms involved are checked - only *mm* needs to be checked here (numbers and strings are not classified under, nor do they have interested party lists). Then the time-specialist will be invoked with the OLD literal

(event1 before '('time 1991 mm 1 00 00 00))

and when it requests *(max-of mm)* , this time it will get a result (and so it will set the upper bound on the absolute time of *event1* to *(1991 2 1 00 00 00)*). If later we also assert

(mm lt 2)

the number-specialist would update its information on *mm* , and then from the interested party list of *mm* , the time specialist would again be invoked with its literal about *event1* , and request *(max-of mm)* again. The number-specialist would respond with *1* this time, and so the time-specialist would reset the upper bound of *event1* to *(1991 1 1 00 00 00)* .

Note that there is no special ordering for the interested party lists - items appear in the order they are entered.

3.1.8 Known Problems and Possible Improvements

One possible shortcoming of the specialist interface is that if a number of specialists are interested in helping to evaluate a literal or function, they are not ordered in any manner whatsoever. This has not been a problem with the specialists involved so far because they rarely conflict - not very many are interested in the same predicates. However, if there were numerous such specialists, it would probably

be advisable to be able to order the list of interested specialists by the amount of effort required by each specialist - with the one requiring the least first.

Specialists are not at all uniform in the effort they require to get answers. In some cases, a literal comparison is quite expensive (e.g. the time-specialist), in others, it isn't (e.g. the part-specialist). Since both actions are controlled by one flag, if one is desired, you get the others too. An effort value is provided to try to curtail operations where possible - for example, and effort level of 0 means constant-time only operations. It might be helpful to set a standard for each effort level and use the effort levels more to control what the specialists do, rather than simply using on-off flags.

Comparison of literals by specialists can be an expensive operation, but when needed, it may be the only way to answer a question (unless you've entered all the axioms that specialist encodes, and have lots of time to wait!). Some more investigation into trying to narrow down exactly the conditions the comparisons are needed under might prove useful. Residues can also be a problem. When a residue is needed to answer a question, nothing else will do, but except for these special cases, the residues tend to slow things down (because of the extra things to try). Currently EPILOG gets around this by heavily weighting the comparison results where a residue is involved, so that if something else is available to be tried, it will be tried first, but if there is nothing else left to use, the residue will be.

The subnet name calculation does not quite contain enough information and it is possible to get some inaccuracies in certain circumstances. For example, if we enter *((john say (that (e1_ep before e2_ep))) * e4_ep)* , and later ask *((john say (that (e2_ep after e1_ep))) * e5_ep)* , both formulas will indicate the same subnet identifier, *john-say* , and so the question will be answered YES, ignoring whatever relationship exists between *e4* and *e5* . Previous implementations of the system were even worse, giving one subnet for a person's entire mental world - thoughts, beliefs, hopes, etc, and assuming consistency among them (not necesarily a good assumption). The current method is better than the previous one, but still has some problems.

3.2 Belief Specialist

The belief specialist reasons about others' beliefs by simulation. When the system learns that someone believes some formula, the belief specialist stores this fact by retrieving the environment structure that represents that agent's beliefs (or creating a new one if one doesn't already exist), temporarily binding ***environment*** to that environment, and then entering the formula as if the system had just learned it. Input-driven inference may take place as a result, and any derived formulas are stored in the simulation environment. When a query is posed to the system about whether an agent believes a formula, ***environment*** is temporarily bound to the simulation environment associated with that agent (again, a new one is created if necessary), the system attempts to prove the formula, and if it succeeds, then the specialist answers that the agent does believe the formula.

Typically, there will be a great deal of knowledge which the system itself believes, and which it also thinks that everyone else beliefs. It would be impractical to duplicate all of this knowledge in the system's own reasoning environment and in all of the simulation environments. Therefore, activating the belief specialist creates a shared environment that is always accessible, whether the system is reasoning for itself or as part of a simulation.

3.2.1 Internal Representation

The belief specialist's representation is simply a hash table that stores environment structures, indexed by EL terms that name believers.

The environment structure was introduced on page 11, but we put off explaining the **use-environments** field until now. This field stores a list of environments that are accessible from the current one. When ***environment*** is bound to one environment, and a second environment is listed in the **use-environment** field of the first, the contents of both environments are accessible for inference. When the belief specialist has not been loaded, there is only one environment, and its **use-environments** field value is nil. Currently, when the belief specialist is active, every environment has exactly one environment, namely the shared knowledge environment ***shared-kb***, on its **use-environments** list (except for the shared environment itself, which has an empty **use-environments** list). But to support possible future extensions, the environment infrastructure allows the possibility of a richer structure, in which one single environment has many other environments on its **use-environments** list, and those environments may themselves have other environments on their lists.

Note that the hash table of environments is itself stored in an environment (in the belief specialist's entry in the **specialist-info** field), which allows for arbitrary nesting of belief. The system's beliefs are stored in one environment; the system's beliefs about John's beliefs are stored in a second environment which can be found from the first, and the system's beliefs about John's beliefs about Mary's beliefs are stored in a third environment which can be found through the second.

3.2.2 Entry and Evaluation

EPILOG uses a large number of global variables (rather too many for this writer's taste). The combination of their values constitutes the state of the system at a given moment. The state of the system can be divided into three parts: configuration, knowledge state, and inference state. Configuration variables indicate values for various parameters that control the system's behavior. Entering a wff into the system augments its knowledge state, and that augmentation persists through later operations. The inference state is less persistent. Whenever an input or a query is presented, the inference mechanism is reset to its initial state. It then goes through various states while processing the input. When finished, the inference mechanism remains in its final state until the next input or query is presented, at which point it is once again reset to the initial state.

When the belief specialist is invoked, the system is in the middle of an ongoing proof attempt. In order to perform a simulative subproof (whether input-driven in the case of a new input, or goal-driven in the case of a query), the **enter** and **evaluate** functions must save the state of the current proof, reset the inference mechanism to its initial state, do the subproof, and then restore the state of the ongoing proof. They do this by introducing a new scope, using **let**, for each global variable that contains inference state, and calling either **story** (for **enter**) or **question** (for **evaluate**) within that **let**. The same **let** also binds ***environment*** to the appropriate simulation environment.

3.2.3 Interaction With Other Specialists

Since it introduces the possibility of multiple environments being accessible simultaneously, the belief specialist makes inference tasks more complicated. In the current version of the system, the general-purpose reasoning mechanism has been entirely updated to accomodate the multiple-environment possibility, and the various specialists have been updated to varying degrees. As we discuss each specialist below, we

will describe the manner in which it deals with the presence of multiple environments. Some of them accomodate it fully. Of those that don't, some store their knowledge state in locations not encapsulated in the environment structure. Beliefs due to such a specialist are held by the system itself, and are also attributed to all other agents, because the information remains accessible when ***environment*** is given a different value. Others do store their knowledge state in the environment structure, but use only the current value of ***environment***, and ignore any other environments accessible from that one. For these specialists, adding information to the environment ***shared-kb*** has no effect—shared beliefs must be added explicitly to the simulation of each agent that holds them.

3.2.4 Known Problems and Possible Improvements

The specialist associates a different simulation environment with each term that names a believer. This is not correct in cases where two distinct terms are known to be equal, *i.e.* to refer to the same believer.

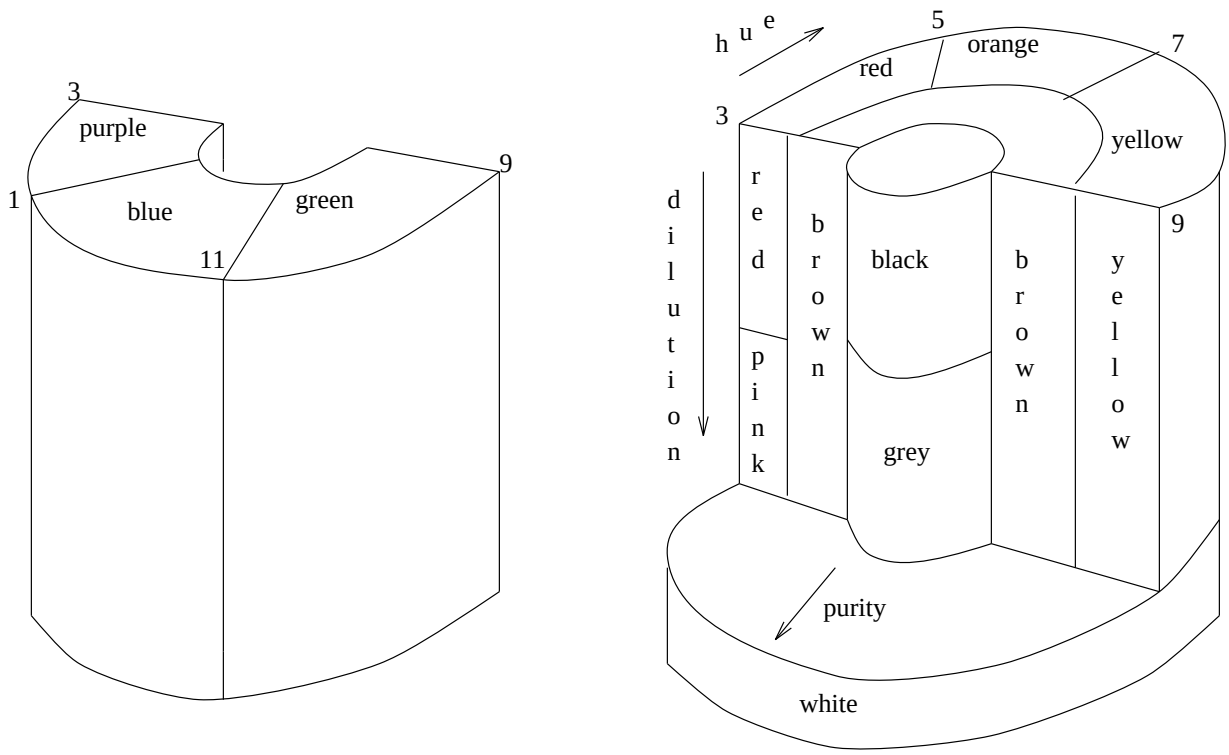
Aaron Kaplan's Ph.D. thesis (University of Rochester Computer Science Department, 2000) covers many other weaknesses and potential extensions in the implementation of simulative inference.

3.3 Color Specialist

The color specialist uses a geometric model (a cylinder) to represent the color space, and uses this to determine relationships between colors. This specialist may only assist **EPILOG** by comparing predicates. It does not maintain its own representation of color facts. However, new colors may be added.

3.3.1 Internal Representation

The color specialist maintains a representation of different color names, and where they fit on the cylindrical model.



Color cylinder with the "cool" shades lifted away.
Adapted from *Accelerating Deductive Inference ...* , p 42.

The color cylinder shown here is the same picture as the one in the User's Guide - for ease of use it has been repeated here. Note that the color cylinder is not actually stored - only the ranges of the colors on the cylinder are.

The cylindrical model has three axes:

hue - this dimension runs through the continuum of rainbow hues. It is arranged in a circle around the cylinder, and arbitrarily scaled from 0 to 12 (the numbers themselves are not important, but the relationship between the numbers shows the relationships between the hues)

purity - the amount of black in a color. This is the radial axis of the cylinder, and is scaled from 0 to 1.

dilution - the amount of white in a color. This is the axis that gives the cylinder its height, and is scaled from 0 to 1.

Each color has associated with it 6 numbers - a minimum and maximum for each axis, which define the area of the cylinder covered by that color name. For example, *white* has a purity range of 0 to 1 (the whole bottom of the cylinder), dilution of 0.8 to 1 (very high in the white content!), and a hue of 0 to 0 - i.e. all the way from 0 to 12 and back to 0 (so all the way around). By contrast, *black* has a purity range from 0 to 0.2 (a core in the center of the cylinder), dilution of 0 to 0.2 (very little white content!), and again, the entire hue continuum. *Red* has a purity of 0.2 to 1 (from a little bit dark to very dark - on the outer layer of the cylinder), dilution of 0 to 0.4 (the top half of the cylinder), and hue from 3 to 5 (arbitrarily defined as the "red" part of the rainbow).

The colors are defined as follows:

Basic Colors

Color	Purity		Dilution		Hue	
white	0.0	1.0	0.8	1.0	0.0	0.0
black	0.0	0.2	0.0	0.2	0.0	0.0
blue	0.0	1.0	0.0	0.8	11.0	1.0
red	0.2	1.0	0.0	0.4	3.0	5.0
yellow	0.7	1.0	0.0	0.8	7.0	9.0
green	0.2	1.0	0.0	0.8	9.0	11.0
purple	0.2	1.0	0.0	0.8	1.0	3.0
orange	0.7	1.0	0.0	0.8	5.0	7.0
brown	0.2	0.7	0.0	0.8	5.0	9.0
pink	0.2	1.0	0.4	0.8	3.0	5.0
grey	0.0	0.2	0.2	0.8	0.0	0.0

The basic colors are a group of colors which completely partition the color cylinder. They also have the largest color spaces. Any other colors have smaller ranges, usually near or overlapping the borders between basic colors.

Additional Colors

Color	Purity		Dilution		Hue	
chartreuse	0.9	1.0	0.0	0.2	8.8	9.2
magenta	0.8	1.0	0.0	0.1	3.0	3.2
bluegreen	0.4	1.0	0.0	0.8	10.7	11.3
lead	0.0	0.2	0.1	0.3	0.0	0.0
salmon	0.6	1.0	0.4	0.8	4.7	5.0
beige	0.2	0.7	0.4	0.8	5.0	9.0
tan	0.2	0.7	0.2	0.6	5.0	7.5
crimson	0.9	1.0	0.0	0.05	3.9	4.2
aqua	0.4	1.0	0.5	0.8	10.8	11.3
rust	0.2	0.7	0.0	0.2	5.0	6.5

3.3.2 Adding New Colors

Adding new colors is done with the command **make-color** . That is the easy part. The hard part is figuring out the ranges to give the new color. To add a new color, the ranges should be adjusted so that the appropriate relative order exists between the new color and its neighbors. For example, to add the color *rust-red* , choose numbers that make the corresponding region overlap, and extend about 1/4 of the way into the brown and red regions (hue), and not too light (dilution) and not too dark (purity). Unfortunately there is no nice way to do this automatically without using a computer monitor which can express all the possible colors.

3.3.3 Color Predicate Comparison

Comparison between color predicates consists of removing any operators, and determining the ranges for the simple colors remaining. These ranges are then compared to determine the color relationship, and if there are any hedging operators these are considered last.

3.3.3.1 Comparing Simple Colors

For each axis, the ranges of the two colors are compared, and a range relation is returned which is one of:

equal - the ranges are the same

left - the minimum of the first is the same as the maximum of the second or, the first range contains the second (after shifting)

right - the minimum of the second is the same as the maximum of the first or, the second range contains the first (after shifting)

next - the minimum of one is the same as the maximum of the other (after shifting)

sep - the minimum of one is greater than the maximum of the other (i.e. they are separated by some distance)

partial - any other relation

The range comparison must take into consideration that the hue axis is circular, so to ensure that comparisons are done properly, it may shift the ranges temporarily for the comparison to get them away from the origin. Also, a certain margin is allowed in the comparison so that the numbers don't have to be identical - they must only be within ***color-margin*** of each other to match these tests.

The axes' relations are then used to determine the color relation as follows:

apart - the two colors are some distance apart on the color cylinder - either one axis relation is *sep* , or all the axes' relations are *next* . (e.g. *red* and *blue*).

adjacent - the two colors are next to each other on the cylinder, with no intervening space (on at least one axis) - one or two of the axes' relations are *next* . (e.g. *red* and *purple*).

overlap - the color spaces overlap on the cylinder - one of the axes' relations is *partial* . (e.g. *chartreuse* and *yellow*).

leftcentre - one color space fits completely within the other color space, with no borders touching - all the axes' relations are *left* or *equal* , and none of the ranges has a common endpoint. (e.g. *brown* and *mid-brown* (defined in the middle of all axes of the brown region))

leftborder - one color space fits within the other color space, against a border - all the axes' relations are *left* or *equal* , and at least one of the ranges has a common endpoint. (e.g. *red* and *crimson*)

rightcentre - one color space fits completely within the other color space, with no borders touching - all the axes' relations are *right* or *equal* , and none of the ranges has a common endpoint. (e.g. *mid-brown* and *brown*)

rightborder - one color space fits within the other color space, against a border - all the axes' relations are *right* or *equal* , and at at least one of the ranges has a common endpoint. (e.g. *crimson* and *red*)

This relation is used in conjunction with any operators on the color predicates to determine the final relationship between the given predicates.

3.3.3.2 Operators Which Affect Color Comparisons

Operators may be applied to colors which may slightly change how they are compared. For example, *sort-of* or *almost* expand the colors they modify to include more borderline regions. Intensifying operators, like *very* are currently ignored, but might someday be used to constrain a color's space.

3.3.3.2.1 Hedging

Hedging a color supposedly doubles all its range boundaries. However, in the actual code, no explicit doubling is done. Instead, after the simple colors have been compared and their relationship found, this relationship together with whether one or both colors are hedged are then used to determine the predicate relation. If one or both are hedged, this may change whether the color relation found indicates that the two predicates are compatible or not. *Hedged1* indicates whether or not the first color was modified by a hedging operator; *hedged2* indicates the same for the second color in the comparison. *Color-relation* is the internal color relation determined between the two unmodified colors (as described above).

```
(case color-relation
    ( overlap
        if (not hedged1 ) then unknown else subsumes )
    ( adjacent
        if (and hedged1 hedged2 ) then unknown else disjoint )
    ( apart disjoint )
    ( leftborder
        if (and (not hedged1 ) hedged2 ) then unknown else subsumes )
    ( rightborder
        if (and hedged1 (not hedged2 )) then unknown else subsumed )
    ( leftcentre subsumes )
    ( rightcentre subsumed )
    (otherwise disjoint )
)
```

Another way to handle the hedging would be to have the operator change the axis numbers for the color it modifies - so it would expand those numbers in some predetermined way (perhaps "doubling", as hedging is supposed to do). This would allow the possibility of operators which have different hedging effects, although whether or not this would be useful remains to be seen.

3.3.3.2.2 Intensifiers

Intensifying operators like *very* , *extremely* , etc currently are accepted by the color specialist, but ignored. It is possible to modify the specialist to use these operators to constrain the color space of the color being modified, so that *(very red)* and *red* get treated as different colors, with *(very red)* being subsumed by *red* . As suggested for the hedging operators, the operator could narrow the numbers for

the color it modifies, on one, two, or all three axes. Each operator could have a different effect, if desired. This will remain as a possible future enhancement.

3.3.4 State of Multiple-Environment Implementation

The color specialist does not store its representation in the environment, so all beliefs about colors held by the system itself are also attributed to others when the belief specialist is active.

3.3.5 Known Problems and Possible Improvements

New colors which are added are not retractable. The properties on the colors could easily be made so (by changing the **setp** commands in **make-color** to **change-property**), but there is no way to reset the ***known-colors*** variable. This should not be a problem, as colors are rarely added, and even more rarely retracted.

Intensifying operators are currently being ignored. As discussed above, it is possible to use them to constrain the color space of the individual colors being compared, but one first has to determine exactly what they mean - for example, does *(very red)* mean unexpectedly red (no effect on the color), bright red (an adjustment on one axis only), a particular shade of red - the one most commonly recognized as red, for example (adjustment on all three axes), or something else? Other possible operators which might affect the color space are *dark* , *deep* , etc.

3.4 Time Specialist

The time-specialist maintains its own representation of facts regarding event or time point orderings, durations, and absolute times (dates). Literals in its domain include the following examples: *(e1 before e2), ((start-of e1) after-1 (date 1991 04 05 00 00 00)), (e2 has-duration 20000), (e1 at-most-after t1 3000)* . After flattening (a specialist interface task), *((elapsed e1 e2) gt= 2000)* would also be a time-specialist literal.

The time-specialist can determine relationships between points quite quickly, using the qualitative relationships given to it, or quantitative ones (absolute times and durations), or a combination of both. It can also be used to determine duration bounds, and compare literals in its domain.

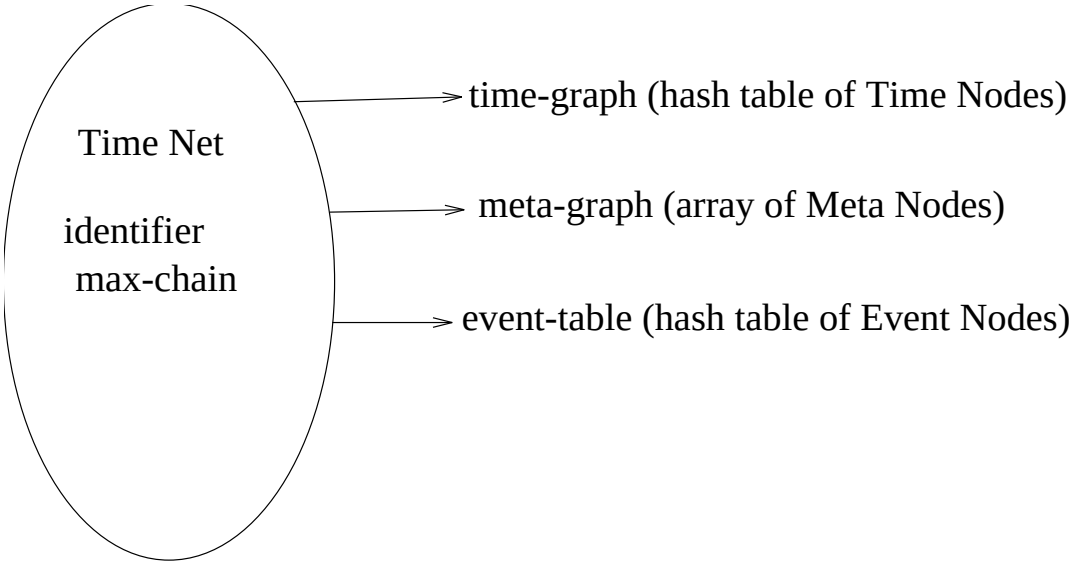
It does this by maintaining its own representation of the temporal facts given in the story. A specialized graph is created and searched to handle most operations. This section contains the detailed, low level description of the time specialist structures and how they are used to store and determine relationships, durations, and absolute times. It would probably be helpful to read the "Details" section on the time-specialist in the User's manual before wading into this.

3.4.1 Internal Representation

To achieve speed in determining relationships, a special form of graph is used - a partitioned graph. The partitioning is based on the idea that many events can be considered to be in "chains" (where one event follows another, which follows another, building a sequence of events or a chain). In most stories, the main story line is a long chain, and there are a number of much smaller chains which branch off the main chain. The time specialist partitions its graph into these "chains", and then can use the fact that

the ordering between any two points of a chain can be easily found in constant time (by introducing a "psuedo" numbering across the chain). For determining the relationship between points on different chains, a search must be performed. However, the chains can again be useful here in greatly reducing the search required - in fact, the search need only take account of the connections between the chains (the "metagraph").

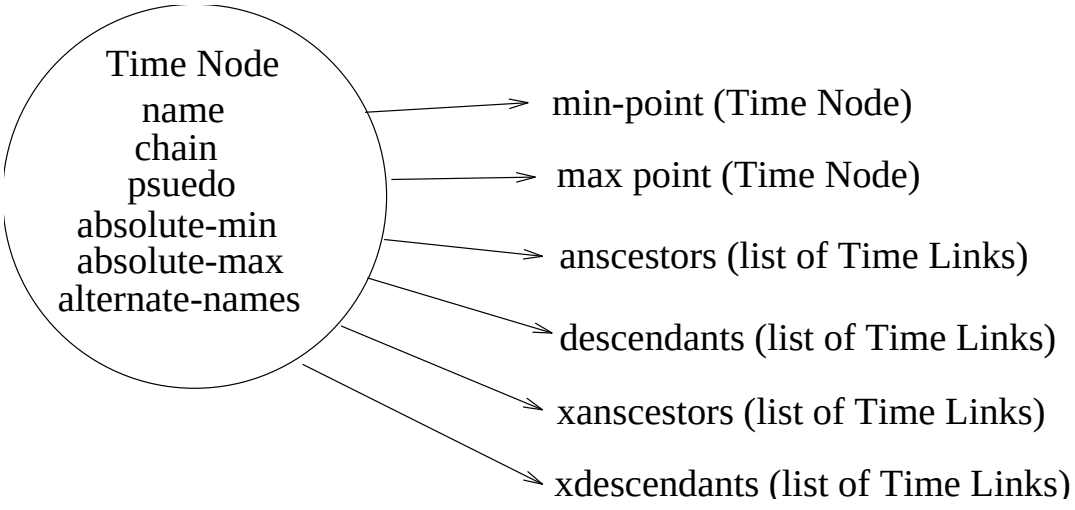
Absolute times and durations add quantitative values to the timegraph. Absolutes times are 6-tuples, whose entries are assumed to be *(year month day hour minute second)* . These entries may be numeric or symbolic. If symbolic, the time specialist will ask the main system for bounds on them, and use these if available. The absolute times may be entered as arguments using the **date** function (e.g. *((date 1991 04 03 hh 00 00) after e1)*), or as a *record* (e.g *(($ 'time 1991 04 dd 00 00 00) before e2)*). Absolute times may be used to determine the relationship between two points (on different chains) in constant time (by simply comparing the absolute times). When entered, absolute time bounds are propagated wherever appropriate, taking ny duration bounds into consideration as well.



The Time Net

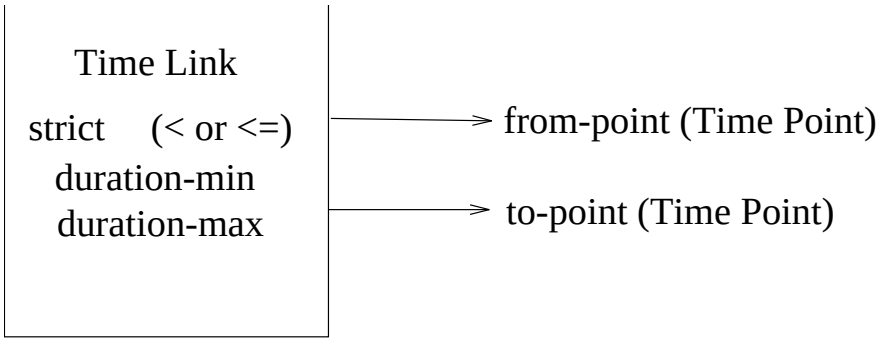
The top most entity in a time graph is the "time net". Because it is possible to have different subnets in the main net, this must be duplicated in the time specialist as well, as different people may have different beliefs about the ordering of the same events. There is one "time net" for each subnet stored. It contains the subnet identifier (*/* for the top level, */name-pred* , or */name-pred/name-pred* , etc for subnets beneath - for example: */mary-think, /john-believe/mary-think, /johnmary-tell*), the maximum chain number used so far in this timegraph, as well as pointers to the actual timegraph table of "time nodes", the metagraph, and an event table.

The table of "time nodes" is a hash table which contains for each time point a pointer to the actual time node for that point. If several points are equal, each entry for the different names will point to the same actual "time node". The metagraph is a simple array with an entry for each chain number used so far. Each entry points to a "meta node". The event table is a hash table with an entry for each event entered into the time specialist. Each entry points to the "event node" for that event name. The hash tables used are expandable, just like all the other hash tables used by EPILOG.



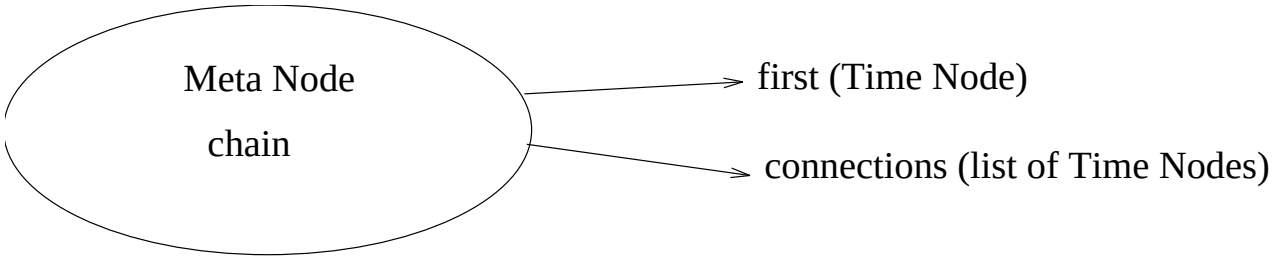
The Time Node

The "time node" is the basic timegraph entity. It contains all the known information about a specific time point, including its name, the chain number of the chain it has been assigned, the psuedo number assigned to it on that chain, any alternate names that might be used to indicate the same point, and the bounds on its absolute time, if known. In order to help determine within a chain whether a relation is strict or non-strict, pointers to the minimum and maximum points on this chain that the point could be equal to are also stored (the pointers are used so that no extra effort is needed during renumbering, when psuedo numbers are changed). Lists of ancestor and descendent links in the same chain, and ancestor and descendent links between this chain and other chains are also kept. These are not used in determining the relationships between points - but they are used to propagate absolute times, and to determine durations.



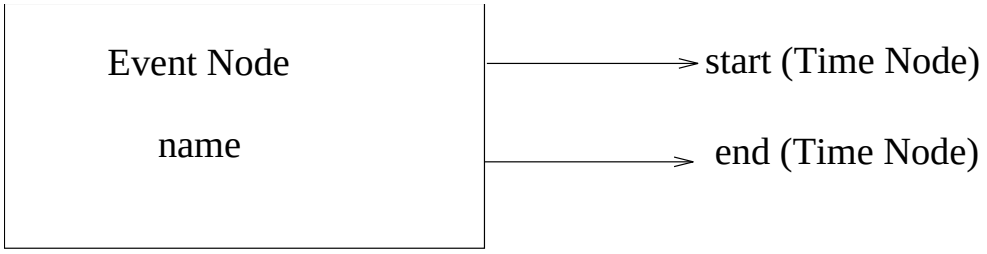
The Time Link

The "time link" is used to keep track of a link between two points. Simply knowing that there is a path between the two points is not enough - we need to know whether the path is strict or non-strict (< or <=), as well as its duration bounds. Pointers to the actual points the link is between are maintained so that no problems occur as a result of renumbering, and to save time during propagation of absolute times and minima and maxima, and searching.



The Meta Node

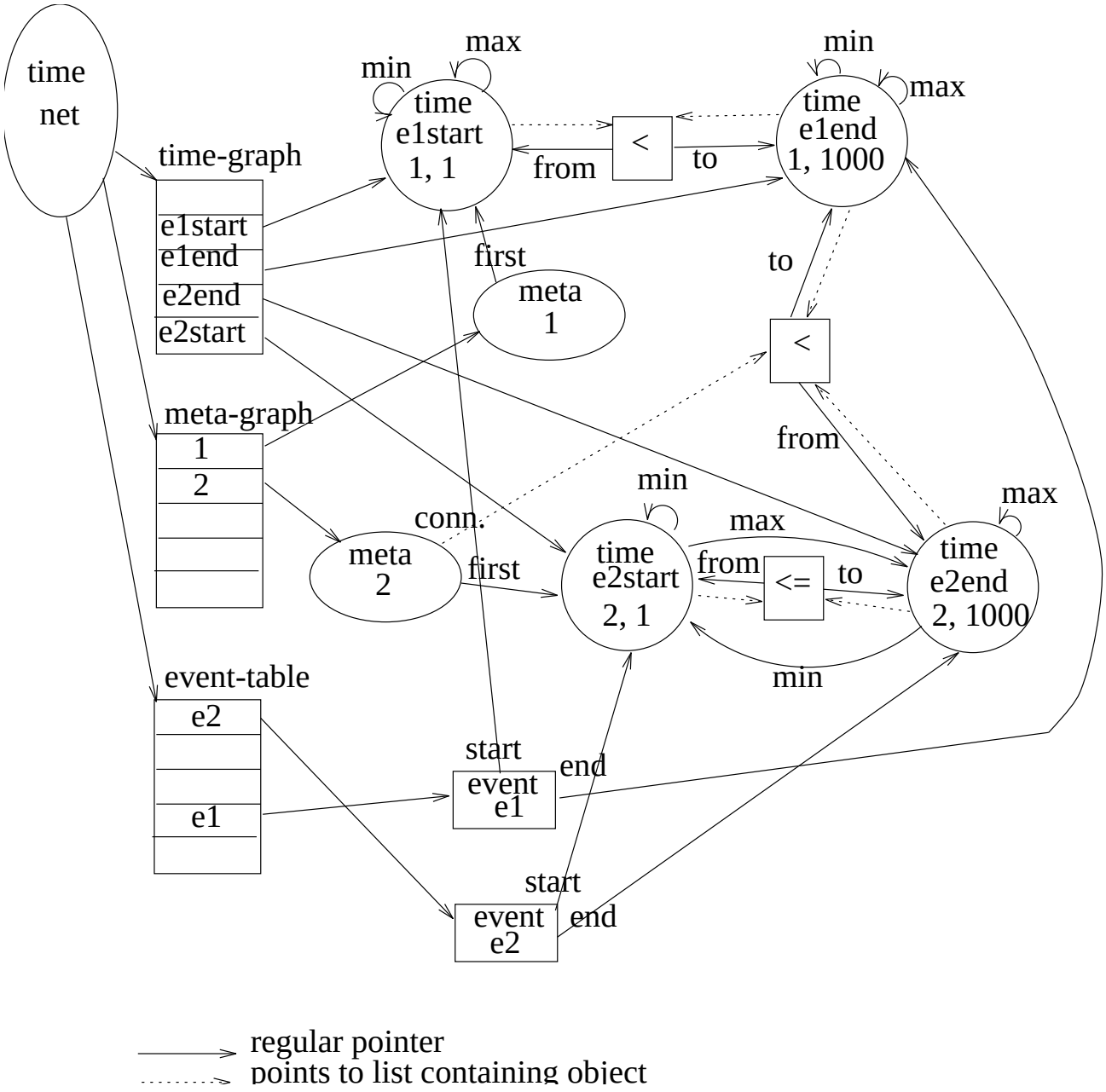
The links between particular points and points on other chains are reorganized into the metagraph, which is actually used for determining relationships between points. There is a "meta node" for each chain, which contains the chain number, a pointer to the first time point in that chain (for renumbering, or printing), and a list of connections (links) between points on this chain, and points on other chains (ordered by chain and psuedo number). It is this list that is searched during an attempt to determine the relationship between points. The in-chain relationship between the given first point and the "from" point in the first link is examined. If it is <=, this link may be useful, so it is "followed". If the "to" point is in the same chain as the given end point, another in-chain comparison will tell us whether that link was useful (the link must end before the given point). If it is after, we try another link. If the "to" point is on another chain, we continue, searching through its connections to try to find the desired end point. If a path is found, we are almost done. If the path is strict, we cannot get any better, so we are finished. If not, we must continue a little while longer to see if there is a strict path (there may be any number of paths through the graph between two points, and there is no guarantee that we'll find the most strict one first). (This algorithm will be described in more detail in a later section).



The Event Node

The last structure to mention is the "event node". This is a very simple node which simply contains an event name, and pointers to its start and end "time nodes". These are named automatically by appending "start" and "end", respectively, to the name of the event.

To see how these all fit together, a sample timegraph, metagraph, and event table are shown. This reflects the relations *((e1 before-1 (end-of e2)) and ((start-of e1) before-1 (end-of e1)))* , where *e1* and *e2* are events. Note that *before-1* is the strict version of *before* , and means the first item occurred some time before the second.



Example: *(e1 before-1 (end-of e2)), ((start-of e1) before-1 (end-of e1))*

Note in the example time graph the following: Time node *e1start* is on chain 1, with psuedotime 1; *e1end* is on chain 1 with psuedotime 1000; *e2start* is on chain 2 with psuedotime 1; *e2end* is on chain 2 with psuedotime 1000. E1start and *e1end* could not be on the same chain as *e2start* and *e2end* because the relationships between all of them are not known. The relationship between *e1start* and *e2end* is not known, for example. Another possible way to partition these would be to have *e1end* and *e2end* on the same chain, but then *e1start* and *e2start* would both have to be on completely different chains, as their relation is not known, so this would require three chains being built, which is less efficient than two.

Time node *e2start* has as its minimum itself (being the first in the chain), and maximum *e2end* (because the relation between them is non-strict). Similarly *e2end* has mximum itself (it is the end of the

chain), and minimum *e2start* (again because of the non-strict relation). Time node *e1start* likewise has minimum itself, and *e1end* has maximum itself. However, *e1start* also has maximum itself, and *e1end* has minimum itself, because the relation between them is strict - *e1start* cannot be equal to *e1end* . Also note that the link between *e1end* and *e2end* is in their anscestor and descendent lists, and is also in the metagraph connection list. The in-chain links are not in any connection lists.

This section describes the algorithms used to do the various time specialist functions. Note that functional arguments are all evaluated, if possible, before the literals or functions are sent to the time-specialist for entry or evaluation. During the entry phase, some literals involving functional terms are "flattened" by the specialist interface into several literals. Those which affect the time specialist in this way are *duration-of* and *elapsed* . For example, *((duration-of e1) gt 3)* would be flattened to *(DURATION-OFE1 gt 3)* (a number-specialist literals), and *(e1 has-duration DURATION-OFE1)* (a time-specialist literal). *((elapsed e1 e2) lt= 5000)* would be flattened to *(ELAPSEDE1E2 lt= 5000)* (a number-specialist literal), and *(e1 exactly-before e2 ELAPSEDE1E2)* (a time-specialist literal). Note that these new literals are created only within the specialist interface and only temporarily. Once the entry has finished they disappear.

3.4.2 Entry

In order to handle strictness and non-strictness, the predicates had to also relate this information. We used "strictness" endings to show this. A "1" indicates strict (<), a "0" indicates equal (meets), and nothing indicates <=. So *(e1 before-1 e2)* means *e1* some time before *e2* , *(e1 before-0 e2)* means the start of *e2* is the same as the end of *e1* , and *(e1 before e2)* means the disjunction of the previous two. If a predicate involves more than one endpoint relation, there may be another strictness ending. For example, *(e1 during-1-0 e2)* means the start of *e1* is some time after the start of *e2* , and the end of *e1* is the same as the end of *e2* . A full table of the meanings of all the predicates and their strictness values can be found in the User's Guide, in the Time Specialist section.

Literals with the following patterns may be entered into the time specialist:

{event, time, absolute time}	before, before-1, before-0, after, after-1, after-0, same-time, equal	{event, time, absolute time}	event
{event, time, absolute-time}	during, during-0, during-1, during–0, during–1, during-1-0, during-0-1, during-1-1, during-0-0	event	event
event	contains, contains-0, contains-1, contains–0, contains–1, contains-1-0, contains-0-1, contains-1-1, contains-0-0	{event, time, absolute-time}	event
event	overlaps, overlaps-0, overlaps-1, overlaps–0, overlaps–1, overlaps-1-0, overlaps-0-1, overlaps-1-1, overlaps-0-0, overlapped-by, overlapped-by-0, overlapped-by-1, overlapped-by–1, overlapped-by–0, overlapped-by-1-0, overlapped-by-0-1, overlapped-by-1-1, overlapped-by-0-0	event	event
{event, time, absolute time}	between, between-0, between-1, between–0, between–1, between-1-0, between-1-1, between-0-1, between-0-0	{event, time, absolute time}	{event, time, absolute time}
{event, time}	exactly-before, exactly-after, at-most-before, at-most-after at-least-before, at-least-after	{event, time} number	
event	has-duration	number	

Note that the third argument in the first four patterns is a "timeframe" event, and is optional. It is intended to help reduce the number of chains created by the specialist in representing the given formulas.

The specialist interface does a rough guess based on the predicate and the sort of the first argument to determine that the time-specialist might be interested, but the time-specialist itself does the final check to ensure that the correct number and sorts of arguments are involved.

3.4.2.1 Relations

Relations involving events are split into several time point relations. The events themselves are represented by a start point and an end point, and relations between events are entered and determined based on relations with these end points. In addition, the start of an event is assumed to be before its end (non-strictly, unless other information is provided).

For entering event relations:

e1 same-time, equal e2 e3:
(start-of e1) same-time (start-of e2)
(end-of e1) same-time (end-of e2)
if e3
(start-of e1) after (start-of e3)
(end-of e1) before (end-of e3)
(start-of e2) after (start-of e3)

(end-of e2) before (end-of e3)

e1 before, before-1, before-0 e2 e3 (e1 before- *strict* **e2 e3):**

if e3

(start-of e2) after (start-of e3)

(end-of e2) before (end-of e3)

if e3

(end-of e1) between- *strict* (start-of e3) (start-of e2)

(start-of e1) between- *strict* (start-of e3) (end-of e1)

else

(end-of e1) before- *strict* (start-of e2)

(start-of e1) before (end-of e1)

e1 after- *strict* **e2 e3:**

if e3

(start-of e2) after (start-of e3)

(end-of e2) before (end-of e3)

if e3

(start-of e1) between- *strict* (end-of e2) (end-of e3)

(end-of e1) between- *strict* (end-of e2) (end-of e3)

else

(start-of e1) after- *strict* (end-of e2)

(start-of e1) before (end-of e1)

e1 during- *strict1***-** *strict2* **e2 e3:**

if e3

(start-of e2) after (start-of e3)

(end-of e2) before (end-of e3)

(start-of e1) between- *strict1* (start-of e2) (end-of e2)

(end-of e1) between (start-of e1)– *strict2* (end-of e2)

e1 contains- *strict1***-** *strict2* **e2 e3:**

if e3

(start-of e1) between– *strict2* (start-of e3) (start-of a2)

(end-of e1) between- *strict1* (end-of e2) (end-of e3)

else

(start-of e1) before- *strict1* (start-of e2)

(end-of e1) after- *strict2* (end-of e2)

(start-of e1) before (end-of e1)

e1 overlaps- *strict1***-** *strict2* **e2 e3:**

if e3

(start-of e1) between– *strict1* (start-of e3) (start-of e2)

(end-of e1) between– *strict2* (start-of e2) (end-of e2)

(end-of e2) before (end-of e3)

else

(end-of e1) between– *strict2* (start-of e2) (end-of e2)

(start-of e1) before- *strict1* (start-of e2)

e1 overlapped-by- *strict1***-** *strict2* **e2 e3:**
if e3
(start-of e1) between- *strict1* (start-of e2) (end-of e2)
(end-of e1) between- *strict2* (end-of e2) (end-of e3)
(start-of e3) before (start-of e2)
else
(start-of e1) between- *strict1* (start-of e2) (end-of e2)
(end-of e1) after- *strict2* (end-of e2)

e1 between- *strict1***-** *strict2* **e2 e3:**
(end-of e1) between– *strict2* (end-of e2) (start-of e3)
(start-of e1) between- *strict1* (end-of e2) (end-of e1)
(start-of e2) before (end-of e2)
(start-of e2) before (end-of e2)

When entering point relations into the time graph, we want to ensure that the minimum number of chains are added. Whenever possible, points are added to existing chains. If there is no room left on a chain (i.e. another number cannot be added between the psuedonumbers of two adjoining points), the entire chain will be renumbered from the beginning. It is assumed that no contradictory information will be entered - the main EPILOG system will detect these and hopefully the user will not insist that they are entered. But sometimes two formulas do not contradict each other completely, but entering them will still make the system inconsistent (for example *(e1 before e2)* and *(e2 before e1)*). In this case, the time-specialist wil determine the only consistent way possible to handle this (in the above example, *e1* must be the same time as *e2* to be consistent), and it will collapse nodes into one if necessary.

t1 same-time, equal t2:
if t1 does not exist
if t2 does not exist
add t1 on new chain
make t2 equal to t1 (i.e. hash entries point to same node)
else t2 does exist
make t1 equal to t2 (i.e. hash entry for t1 points to t2 node)
else t1 does exist
if t2 does not exist
make t2 equal to t1 (i.e. hash entry for t2 points to t1 node)
else t2 and t1 exist
collapse t1 and t2 into one point - copy any links and update timegraph and metagraph

t1 before- *strict* **t2 or t2 after-** *strict* **t1:**
if t2 does not exist
if t1 does not exist
add t1 on new chain
add t2 after it on same chain
add link between t1 and t2 with *strict*
else t1 does exist
if t1 is at the end of its chain

add t2 after it on same chain
add link between t1 and t2 with *strict*
else
add t2 on a new chain
make a cross-chain link between t1 and t2 with *strict*
else t2 does exist
if t1 does not exist
if t2 is at the beginning of its chain
add t1 before t2 on the same chain
add link between t1 and t2 with *strict*
else
add t1 on a new chain
make a cross-chain link between t1 and t2 with *strict*
else t1 does exist
if t1 and t2 are on the same chain
make a transitive link between t1 and t2 with *strict*
else
make a cross-chain link between t1 and t2 with *strict*

t1 between- *strict1***-** *strict2* **t2 t3:**

if t1 does not exist
add t2 before t3 (using "before" algorithm)
if t2 and t3 on same chain
add t1 on same chain (renumber if necessary)
add *strict1* link to t2
add *strict2* link to t3
else
if t2 at the end of its chain
add t1 after t2 on same chain
add link with from t2 to t1 *strict1*
add cross-chain link from t1 to t3, with *strict2*
else
if t3 at beginning of its chain
add t1 before t3 on same chain
add link from t1 to t3 with *strict2*
add cross-chain link from t2 to t1, with *strict1*
else
add t1 on new chain
add cross-chain link to t2 with *strict1*
add cross-chain link to t3 with *strict2*
else t1 does exist
if t1 first or last in its chain
add t1 after- *strict1* t2 (using "after" algorithm)
add t1 before- *strict2* t3 (using "before" algorithm)

(this will ensure that if t2 or t3 needs to be created, the minimum number of chains will be added)
else
add t3 after t2 (using "after" algorithm)
if t1 and t2 on same chain
make a transitive link between t2 and t1 (with *strict1*)
else
make a cross-chain link between t2 and t1 (with *strict1*)
if t1 and t3 on same chain
make a transitive link between t1 and t3 (with *strict1*)
else
make a cross-chain link between t1 and t3 (with *strict1*)

When adding a new point onto an existing chain, its psuedotime is set to reflect its location - if at the end, it is 1000 greater than the psuedotime of the current end, if at the beginning, 1000 less than the current beginning, and in between two points the new psuedotime will be a number between the psuedotimes of the two points. If a large number of points are added between, it may be necessary to renumber the chain to make room for additional psuedotimes, and therefore points, to be added. In addition to reflecting the order with the psuedotimes, actual links are added between the points. Also pointers are set up to the minimum point and maximum point in the chain that the new point can be equal to - this is how the strictness is handled. A minimum is propagated forward, and a maximum backward to ensure that the necessary information is available on any point in the chain. This is similar to the absolute time propagation, but much simpler and only considers points on the chain.

Note for all of the above additions - whenever a new link is added, absolute time propagation is attempted. This propagation is described in the section on absolute times.

3.4.2.2 Absolute Times and Durations

For any of the legal patterns entered, if one of the arguments is an absolute time, the time specialist will use that to update the absolute time bounds on a point (or points), rather than adding a qualitative relation. For example, if *(t1 before (date 1991 04 01 00 00 00))* is entered, this will cause an upper bound of *(1991 04 01 00 00 00)* to be added to *t1* . Similary, *(e1 after (date 1991 01 02 12 00 00))* would cause a lower bound of *(1991 01 02 12 00 00)* to be added to the end point of *e1* . If a bound already exists on the time point, the most restrictive one is chosen - i.e. the lowest maximum, and the highest minimum. Note: if a strict predicate is used to enter the absolute time bound, that strictness information may be lost, as the bounds do not contain strictness information, and it is not always possible to subtract something from the time to show the strictness.

If any part of the absolute time is a symbolic entity rather than a number, the time-specialist will ask the main system to evaluate the function **max-of** or **min-of** on that entity to try to find a numeric bound (if the absolute time is an upper bound, **max-of** is requested, otherwise *min-f*).

Durations are entered when any of the following predicates is used: *has-duration, exactly-before, exactly-after, at-most-before, at-most-after, at-least-before* , or *at-least-after* . *Exactly-before* , *exactly-after* , and *has-duration* cause both the upper and lower bounds of the duration of the intended link to be set; *at-most-after* and *at-most-before* affect only the upper bound; and *at-least-after* and *at-least-before* affect only the lower bound. Except for *has-duration* , the other predicates in this set cause a link

to be added between the particular points or events using the routines described in the previous section (where *...-after* becomes *after-1* if the duration is non-zero, *after* otherwise, and similarly for the *...-before* predicates). After the link has been added in the usual way, it is located, and the duration added.

Whenever a link, absolute time, or duration is added, absolute time propagation is attempted. This attempts to update all relevant time points with the new bounds. For example, if a new upper bound is added to a point, all points before that must be before that point, and therefore must have that upper bound as a maximum (unless they already have something more restrictive). Similarly, minimums can be propagated forward. The propagation follows all links - in-chain and cross-chain, until it finds a point whose time bound is already better than the one being propagated, or runs out of points to go to. When propagating the bounds, the duration on links is taken into consideration in calculating the new bounds for the point on the other end of the link. So if we were propagating a maximum back, and a link had a duration maximum of 1 day, we would subtract a day from the absolute time bound, and propagate that to the point, instead of the original time bound. For time points *t1* and *t2* with lower and upper bounds *l1* , *u1* and *l2* , *u2* , respectively, and duration bounds between *t1* and *t2* (i.e. *(t2 - t1)*) *l* (lower) and *u* (upper), the bounds may be updated using the following equations:

$l2 - u <= t1 <= u2 - l$ *(use best of* $l2 - u$ *and* $l1$*,* $u2 - l$ *and* $u1$*)*

$l + l1 <= t2 <= u + u1$ *(use best of* $l + l1$ *and* $l2$*,* $u + u1$ *and* $u2$*)*

$l2 - u1 <= t2 - t1 <= u2 - l1$ *(use best of* $l2 - u1$ *and* l*,* $u2 - l1$ *and* u*)*

Note that durations may update absolute times, but absolute times do not update durations (although they may be used to calculate a duration for evaluation).

3.4.3 Evaluation

The evaluation phase of the specialist is used for consistency testing, evaluation of literals for the main EPILOG system, and evaluations during literal comparison. In addition, some functions may be evaluated by the time-specialist.

3.4.3.1 Relations

Determining whether or not a relation holds between two events is done by splitting the event evaluation into several point evaluation questions, relating the start and end points of the events.

Determining whether a relation holds between two points is done as follows:

Evaluate t1 relation t2:
if t1 and t2 on same chain
compare psuedotime to find order of times
compare minimum and maximum psuedotimes to determine strictness
else
if t1 and t2 both have absolute time bounds
compare the time bounds to see if the desired relation holds
else (also if the time bound comparison gave no information)
do a metagraph search from t1 to t2, looking for a path
(search-metagraph current-metagraph t1 t2 (chain of t1) nil)
if a path is found

```
            compare it to the desired relation
        else
            do a metagraph search from t2 to t1, looking for a path
            (search-metagraph current-metagraph t2 t1 (chain of t2) nil)
            if a path is found
                compare its inverse to the desired relation
```

```
Search-Metagraph (metagraph start end chains-used strict-so-far)
    if pt1 has 'rel property on it - use it and stop
    do for each connection in connection list for chain of start
        if link from point after start
            if this link points to chain = chain of end
                if link to point before end
                    found a path
                else
                    (Search-Metagraph metagraph (link to point) end
                        (chains-used + chain of to point)
                        (combine strict-so-far and strictness from link) )
        if we have a strict answer (i.e. found a strict path)
            stop looking - can't get any stricter
        else
            save current answer in case we don't find anything stricter
    end of do
    if stopped due to strict answer
        set answer to be the strict answer
    else if saved answer
        set answer to be the saved answer
    if answer
        save on start point (property 'rel) in case we reach this place again during search
    return answer
```

If the relation also involves durations (e.g. *at-most-before* , etc), the qualitative relation will be compared first (*before*), and then the duration will be calculated and compared with the given one.

3.4.3.2 Absolute Times

Evaluations which involve absolute times are considered to be requests to compare the given points with the given absolute times. The absolute time bounds from the given point (or particular end point if an event is specified) are retrieved and compared agains the given time to see if the specified relation holds.

3.4.3.3 Durations

To calculate the duration between two time points, a depth-first search which covers ALL possible paths between those points is undertaken. The duration bounds of each link (in-chain or cross-chain) in the path are added together. Bot duration information on arcs and duration information implicit in absolute times are used. The best possible bounds after considering each path are returned as the duration (i.e. the highest minimum (lower bound), and the lowest maximum (upper bound)).

```
determine-duration(pt1 pt2 curmin curmax already)
    bestmin = −∞
    bestmax = +∞
    For each link (in-chain and cross-chain) leaving pt1:
        If link "to point" is in already, ignore
        Calculate link duration by taking best of duration on links and implied
            duration using absolute times
        newmin = curmin + link duration minimum
        newmax = curmax + link duration maximum
        If link "to point" is pt2
            bestmin = (max bestmin newmin)
            bestmax = (min bestmax newmax)
        Else
            (determine-duration to-point pt newmin newmax (append already "to point"))
```

3.4.4 Function Evaluation

The **start-of** and **end-of** functions each take one argument, an event, and return the name of the desired end point - the event's name, with *start* or *end* , respectively appended.

The **date** function takes 6 arguments, and returns a *record* containing the sort *time* , and the 6 arguments.

The **duration-of** function takes a single event argument, and returns a number corresponding to the number of seconds between the start of the event and the end of the event. If the duration has different upper and lower bounds, nil is returned.

The **elapsed** function takes a two event or time point arguments, an returns a number corresponding to the number of seconds between the first item (or its end point if it is an event), and the second (or its start point if it is an event). If the duration calculated has different upper and lower bounds, nil is returned.

3.4.5 Literal Comparison

This is easily the most complicated and most difficult to test of all the time-specialist functions. First, unification must be done on the literals if any variables are involved. Since the predicates being compared are probably not identical, forcing the system to only do the standard unifications doesn't really make sense. Based on the predicates involved, the time-specialist determines what possible unifications could

be done (first of literal 1 with from literal 2, ...; first of literal1 with second of literal2, ...; etc). For EACH of these possible unifications, unification is attempted, and if it succeeds, substitutions are done, and the rest of the literal comparison is done. So it is possible to get several results from the comparison of the two literals.

During literal comparison for incompatibility, we are looking for a unification that makes the two literals incompatible, or incompatible with the negation of a residue. If the two relations are disjoint, we have the simplest case of incompatibility with a null residue. If another relation holds however (subsumes, intersects), we don't have as strong a result, but we can still get something - with a residue. Similarly, during literal comparison for compatibility, we look to see if there is a unification that makes the two literals compatible, or compatible with the negation of a residue. If the two relations are equivalent, we have the simplest case of compatibility with a null residue. If the first relation subsumes the second we also have a null residue.

For each comparison attempt (each successful unification, or the original literals if no variables are involved), both literals are evaluated. If both evaluate to something other than unknown, we can stop immediately with an answer. Two YES evaluations would make literals compatible, a NO and a YES incompatible, and any other NO would indicate it is useless to proceed.

If the evaluations themselves haven't been enough, the time-specialist will try to enter one literal, and then re-evaluate the other one based on this new information. It does this by determining the current relation existing between the arguments of the other literal, and comparing that with the given relation. This comparison will indicate whether or not the literals are incompatible or compatible, and if so, what residue, if any, is involved. If there is an overlap between the two relations, that overlap with the arguments for the testing literal becomes the residue. If the predicate involved is *between* , the test is split into two tests, and the results are combined.

For each successful comparison attempt, a *comparison-item* containing the substitutions and residues (if any) is returned, so that the final result of the literal comparison is a list of these *comparison-item* s.

Some examples of literal comparison:

Comparing for incompatibility *(x before y)* against *(e1 after e2)*

Unification: *x/e1, y/e2*

<=> *(e1 before e2)* against *(e1 after e2)*

<=> Entering *(e1 after e2)* into the timegraph

<=> Compare relations between *e1* and *e2* :

test *before* with graph relation *after*

<=> Incompatible with negation of residue *(e1 equal e2)*

Unification: *x/e2, y/e1*

<=> *(e2 before e1)* against *(e1 after e2)*

<=> Entering *(e1 after e2)* into the timegraph

<=> Compare relations between *e2* and *e1* :

test *before* with graph relation *before*

<=> Not incompatible

Comparing for compatibility *(x before y)* and *(e1 after-1 e2)*

Unification: *x/e1, y/e2*

<=> *(e1 before e2)* against *(e1 after-1 e2)*

<=> Entering *(e1 after e2)* into the timegraph

<=> Comparing relations between *e1* and *e2* :

test *before* with graph relation *after-1*

<=> Not compatible

Unification: *x/e2, y/e1*

<=> *(e2 before e1)* against *(e1 after-1 e2)*

<=> Entering *(e1 after-1 e2)* into the timegraph

<=> Comparing relations between *e2* and *e1* :

test *before* with graph relation *before-1*

<=> Compatible with residue null

The next sections go into a little more detail about how this is accomplished.

3.4.5.1 Unification

All possible unifications are tried here, with evaluations used after each to determine whether they are worthwhile. In determining unifications, event/time point arguments may only be unified with the same (i.e. not against durations), and the two literals do NOT need to have the same number of arguments. [1] When unifying two constants, they are considered unifiable if both exist in the time-specialist's domain already (i.e. there is information about them). No substitutions are prepared for them however - this is used merely to let the unification succeed.

Examples:

Literals are considered unifiable if both exist in the time-specialist's domain already (i.e. there is information about them). No substitutions are prepared for them however - this is used merely to let the unification succeed. Examples:

Literals	Unifications
(e1 before e2 e3) , (x after-1 y)	(e1/x, e2/y), (e2/x, e1/y)
(x after y) , (e1 between e2 e3)	(e1/x, e2/y), (e1/x, e3/y), (e2/x, e1/y), (e2/x, e3/y), (e3/x, e1/y), (e3/x, e2/y)

[1] The only restriction on the unifications is that the timeframe argument, if present, not be unified with anything. That argument is used on entry only to generate a more nearly optimal timegraph, and is not essential for later reasoning - the relation given by the first two arguments with the predicate is the important one. This means fewer unifications to be tried for factoring or resolution and therefore these attempts will be faster.

Each unification is like a shortcut for applying a number of temporal axioms to the literals to make them resolvable or factorable in the traditional sense. For instance, in the first example above, the unification (e2/x, e1/y) could be obtained by applying the temporal axioms given earlier to define $before$ and $after-1$, as well as the ordering axioms about $<$ and ,to get literals $[(end-of e_1) <= (start-of e_2)]$ and $[(end-of y) <= (start-of x)]$, which can be unified in the traditional manner, with unifier (e2/x, e1/y). (This is the second of the two unifiers listed above. The first is obtained similarly, from inferred literals $[(end-of e_1) <= (start-of e_2)]$ and $[(end-of x) <= (start-of y)]$.)

3.4.5.2 Evaluating Using the Unifications

The substitutions $(s_1, ..., s_n)$ are made in the two literals This must be done first so that we are working with the special case of the literals, using the unification. These are the literals that the temporal specialist will try to compare, not the original literals. If any functions are involved, they are evaluated if possible before continuing. The two literals are then evaluated by the time-specialist. The evaluations can answer the question of (in)compatibility immediately, and even when they can't, if one literal evaluates to NO, we cannot enter that one into the timegraph!

3.4.5.3 Testing for Residue

Before actually entering anything into the timgraph, it is checkpointed so that all changes can be retracted. The changes are retracted after each attempt with a different unification.

We now try to enter one literal and compare it against the other within the timegraph. Entering the literal into the timegraph ensures that we can detect inconsistencies between the two literals alone, even if we have an empty timegraph. Other than negated information, which cannot be entered into the timegraph, it does not really matter which literal is entered and which tested - they should show the same effects either way. However, when the residue is calculated, we would prefer it to have terms from the set of support clauses, and since the residue is determined based on the testing literal, we prefer that literal1 (which is always from the set of support) be tested, and literal2 entered. Given a choice of literal to enter (i.e. none has been ruled out because of considerations described previously), if one has a *between* predicate, it is entered. This is because a *between* predicate is compound and therefore testing it is more complicated than any of the others.

The other literal is then tested - the time specialist finds the currently existing relation between the arguments in that literal (*testreln*). This relation is then compared to the one given in this literal (*reln*).

When testing for compatible literals, compare *reln* with *testreln* :

For a non-negated literal:

equivalent	compatible with residue of nil
subsumes	compatible with residue of nil
subsumed	compatible with residue of the difference between the relations $(reln - testreln)$
intersects	compatible with residue of the more strict version of *reln*
disjoint	not compatible

For a negated literal:

equivalent	not compatible
subsumes	not compatible
subsumed	compatible with residue of *reln*
intersects	compatible with residue of *same-time*
disjoint	compatible with residue of nil

Testing for incompatible literals, compare *reln* with *testreln* :

With non-negated literal:

equivalent	not incompatible
subsumes	not incompatible
subsumed	incompatible with residue of the difference between the relations ($reln - testreln$)
intersects	compatible with residue of the more strict version of *reln*
disjoint	incompatible with residue nil

For a negated literal:

equivalent	incompatible with residue of nil
subsumes	incompatible with residue of nil
subsumed	incompatible with residue of *reln*
intersects	incompatible with residue of *same-time*
disjoint	not incompatible

If the testing predicate is *between* , split into two tests and combine the results:

if the testing literal is negated:

if both results indicate compatible/incompatible with no residue, use that

if either indicates not compatible/incompatible, the whole thing isn't

otherwise append the two residues and return both

if not negated:

if both results indicate not compatible/incompatible, the whole thing isn't

if either indicates compatible/incompatible with no residue, use that

otherwise append the two residues and return both

3.4.6 State of Multiple-Environment Implementation

The time specialist is able to reason with the contents of multiple environments simultaneously. The data structures and algorithms described above for the single-environment case are adapted to the multiple-environment case as follows. Each environment has a hash table that maps node names to nodes. When searching for a node with a particular name, the specialist looks first in the agent-specific environment, and then in the shared environment if not found in the agent-specific one. Therefore, information in the agent-specific environment takes precendence. The specialist uses structures in the shared environment as much as possible, but when it needs to modify some structure node, it makes a copy of the structure in the agent-specific environment, and modifies that copy. Chains are copied atomically—if one node needs to be modified, a copy of that node's entire chain is made in the current agent-specific environment. Absolute time information is propagated as far as possible within an environment, but not from one

environment to another. This is to prevent agent-specific beliefs from leaking into the shared knowledge base.

3.4.7 Known Problems and Possible Improvements

For some reason not all of the space used by the search mechanism is released when the search is finished. This can cause problems if a number of literal comparisons are done, which all involve evaluations and search. Attempts to locate the problem have not been successful, and it is not known whether this is a problem with the current version of Lisp being used, or a rather subtle bug in the specialist itself. It rarely shows itself however, so should not be a problem for most applications.

This code was some of the first Lisp code written by the main implementer, and could probably be more efficient.

It is probably possible to improve the efficiency of the search mechanism when it attempts to find "more strict" relations, although some attempt has been made to prevent wasted searching. However, other system considerations seem to always win out over the slight improvement this might give.

The absolute time and duration bounds are not extracted in a deductively complete way. The upper bound is the same whether the metric should be < or <= that bound, and similarly for lower bounds. While this hasn't caused any errors in the testing done so far, there are some scenarios where this might be a problem (such as tightly clustered time points).

As with other specialists, more investigation into exactly what literal comparisons are useful and when they should be tried is also needed.

3.5 Number Specialist

The number specialist helps **EPILOG** deal with arithmetic functions and numeric relationships. It maintains its own representation of story facts involving numeric relationships, and may be used to evaluate or compare similar formulas, as well as evaluating arithmetic functions.

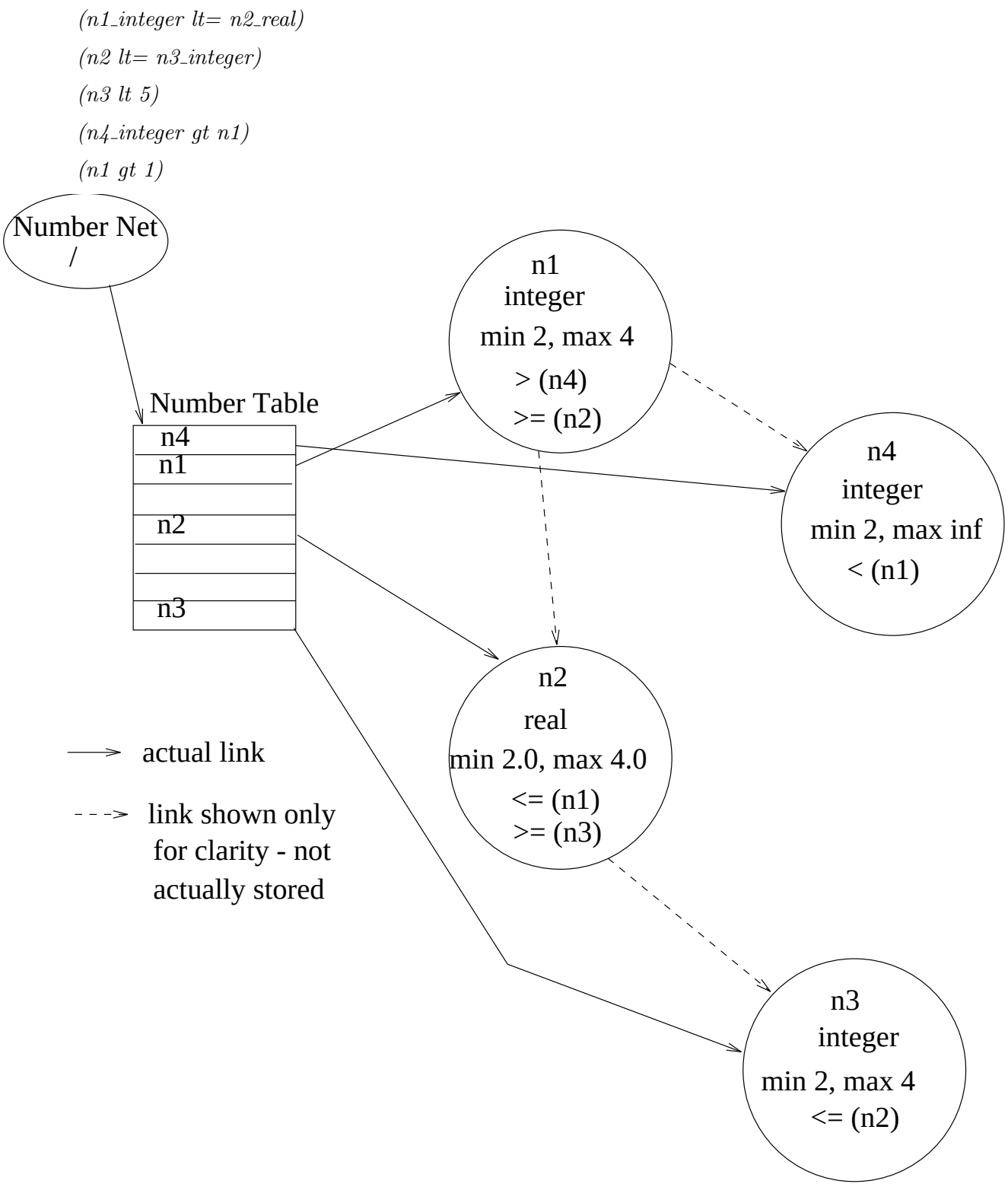
A simple graph structure is used to keep track of the relationships between numbers. Note that many of the algorithms are similar to those used by the time-specialist, but much less complicated (easier domain).

3.5.1 Internal Representation

A separate structure called a *number-net* exists for each subnet which has number information asserted in it. The *number-net* contains the subnet identifier, and a pointer to a hash table which contains all the actual number nodes for that particular subnet.

Each named number node is a graph node, and contains the number's name, minimum and maximum if known, the number's type (integer or real), lists of anscestors of this node (strictly < and <=), and lists of descendants (strictly > and >=). The links between the nodes are not actually stored - only the names of the connecting nodes are. Following links for evaluation or propagation requires separate lookups for each point in the chain.

If the following formulas are entered into the number specialist, the number specialist graph would look like the figure shown.



Example Number Graph

Note that propagation was used to set the minimums of *n2* , *n3* , and *n4* , and the maximums of *n1* and *n2* . Propagation will be discussed in detail shortly.

This section describes the various algorithms and methods used by the number specialist to store and evaluate number relations and arithmetic functions.

Literals with the following patterns are accepted for entry/evaluation/comparison by the number-specialist:

{number, integer, real}	equal less-than, lt, lt= greater-than, gt, gt=	{number, integer, real}

3.5.2 Entry

Named numbers have a number node created for them and added to the number hash table. If the relationship being entered is between two named numbers, they will be created, and the appropriate anscestor/descendant information added. If the relationship is strict (*lt, less-than, gt, greater-than*), the strict anscestor/descendent link lists are updated; otherwise the non-strict lists are. Minima and maxima are propagated, if they exist.

If two named numbers are equated, all information is copied from one to the other, and then both names are set to point to the same updated number node.

Entering relations between named numbers:

(n1 equal n2)
copy all link info from n1 to n2
remove all link info to n2, replace with n1
collapse n1 and n2 into a single point n1

(n1 less-than n2) (n1 lt n2) (n2 greater-than n1) (n2 gt n1)
add n2 to list of strict descendents of n1
add n1 to list of strict anscestors of n2
propagate minimum of n1
propagate maximum of n2

(n1 lt= n2) (n2 gt= n1)
add n2 to list of non-strict descendents of n1
add n1 to list of non-strict anscestors of n2
propagate minimum of n1
propagate maximum of n2

If the relation is between a named number and a simple number, the simple number will be used to update the minimum or maximum (as appropriate) of the named number. Propagation then follows, if applicable.

(n1 equal #) (# equal n1)
if (no current minimum of n1, or # >= current minimum)
set minimum of n1 to #

if (no current maximum of n1, or # <= current maximum)
set maximum of n1 to #
propagate minimum of n1
propagate maximum of n1

(n1 less-than #) (n1 lt #) (# greater-than n1) (# gt n1)
if n1 is an integer, set # to (# - 1) (to handle strictness)
if (no current maximum of n1, or # <= current maximum)
set maximum of n1 to #
propagate maximum of n1

(# less-than n1 (# lt n1) (n1 greater-than #) (n1 gt #)
if n1 is an integer, set # to (# + 1) (to handle strictness)
if (no current minimum of n1, or # <= current minimum)
set minimum of n1 to #
propagate minimum of n1

(n1 lt= #) (# gt= n1)
if (no current maximum of n1, or # <= current maximum)
set maximum of n1 to #
propagate maximum of n1

(# lt= n1) (n1 gt= #)
if (no current minimum of n1, or # <= current minimum)
set minimum of n1 to #
propagate minimum of n1

3.5.2.1 Propagation of Minima and Maxima

Minima propagate forward along descendant links (both strict and non-strict), and maxima propagate backward along anscestor links (both strict and non-strict). When propagating minima, if a link is strict, and the next number node is an integer, the minimum will be increased by one. For a real number, the minimum will be floated if it is an integer, but will be otherwise unchanged. Similarly, a strict link during propagation of maxima to an integer will cause the maximum to be descreased by one. A real number will cause the maximum to be floated. Propagation continues until either all the applicable nodes have been reached, or the maximum (or minimum) being propagated is no better than the one already existing on a node.

propagate-min *(num)*
min = minimum of *num*
for all links *l* in (append strict and non-strict descendents of *num*)
n = the number node for *l*
if *n* is an integer node
newmin = ceiling of *min* (i.e. round it up)
else if *n* is a real node
newmin = float *min* (i.e. make sure it is a real number)
if (*n* has no current minimum, or *newmin* > current minimum)

```
            and
            (  n  has no current maximum, or  newmin  <= current maximum)
              change minimum of  n  to  newmin
              (propagate-min  n  )
end of propagate-min

 propagate-max  (num)
     max  = maximum of  num
    for all links  l  in (append strict and non-strict anscestors of  num  )
         n  = the number node for  l
        if  n  is an integer node
             newmax  = floor of  max    (i.e. round it down)
        else if  n  is a real node
             newmax  = float  max   (i.e. make sure it is a real number)
        if (  n  has no current maximum, or  newmax  < current maximum)
          and
          (  n  has no current minimum, or  newmax  >= current minimum)
            change maximum of  n  to  newmax
            (propagate-max  n  )
end of propagate-max
```

3.5.3 Literal Evaluation

If the literal to be evaluated is a relation between two named numbers, first their minima and maxima are compared to see if the relation can be determined that way. If not, a graph search from the first to the second named number is attempted. If this too fails, then a search from the second number to the first is attempted. If found, the actual existing relation between the names will be compared against the one requested in the literal evaluation.

```
 find-relation  (n1 n2)
    ; first try comparing minima and maxima
    reln = (compare-min-max  n1   n2  )
    ; then try for path from n1 to n2
    if reln still not known
         reln = (find-path  n1   n2  lt= (list  n1  ))
    ; then try for path from n2 to n1
    if reln still not known
         reln = (reverse (find-path  n2   n1  lt= (list  n2  )))
end of find-relation
```

Reverse just changes the direction of the relation - e.g. *less-than* becomes *greater-than* , etc.

compare-min-max *(n1 n2)*
if (minimum of *n1* = minimum of *n2* (and they are not null))
and
(maximum of *n1* = maximum of *n2* (and they are not null))
relation is *equal*
if (minimum of *n1* > maximum of *n2*)
relation is *greater-than*
else if (minimum of *n1* >= maximum of *n2*)
relation is *gt=*
if (minimum of *n2* > maximum of *n1*)
relation is *less-than*
else if (minimum of *n2* >= maximum of *n1*)
relation is *lt=*
end of compare-min-max

find-path *(n1 n2 sofar already)*
; first try strict links
for all *l* in strict descendent list of *n1*
lnum = number node for *l*
unless (lnum in already list)
if (*lnum* = *n2*)
; found a strict path - stop
res = *less-than*
else ; continue searching
res = (find-path *lnum* *n2* less-than (append already *lnum*))
if res = *less-than*
stop and return res
;otherwise try the non-strict links - try them all for most strict path
for all *l* in non-strict descendent list of *n1*
lnum = number node for *l*
unless (*lnum* in already list)
if (*lnum* = *n2*)
res = *sofar*
else ; continue searching
res2 = (find-path *lnum* *n2* *sofar* (append already *lnum*))
if res2
res = res2
if res = *less-than*
stop and return res
return res
end of find-path

If a named number and a simple number are involved, the minimum and/or maximum of the named number will be compared against the simple number to determine whether or not the relation holds.

Finding relation between named number n1 and simple number val
if (minimum of n1 = val) and (maximum of n1 = val)
  relation is *equal*
if (val > maximum of n1)
  relation is *less-than*
if (val < minimum of n1)
  relation is *greater-than*
if (val >= maximum of n1)
  relation is *lt=*
if (val <= minimum of n1)
  relation is *gt=*

The relation found is then compared against the desired relation. If they are equal, the evaluation succeeds. Also, if the existing relation is more strict than the requested one, the evaluation succeeds (e.g. the existing relation is *n1 less-than n2* , and the question to evaluate was *n1 lt n2*).

3.5.4 Function Evaluation

The **add** function takes any number of named or simple numbers, and returns the sum of them. Note that if the actual value of any number is not known, the function cannot be evaluated.

The **max-of** function takes any number of named or simple numbers, and returns the maximum. If only one named number is specified, its maximum is returned.

The **min-of** function takes any number of named or simple numbers, and returns the minimum. If only one named number is specified, its minimum is returned.

The **value-of** function takes one named number and returns its value - if and only if the minimum and maximum are the same.

3.5.5 Literal Comparison

The literal comparison done by the number-specialist is similar to that done in the time-specialist, but simplified greatly because there are fewer, simpler relations to deal with, and there are never more than two arguments in a literal. The process is the same: unify the two literals, substitute in the literals and evaluate them, and finally, if necessary, temporarily enter one literal into the number graph, and compare against the other.

Some examples of literal comparison:

Comparing for incompatibility *(x lt= y) against (n1 gt= n2*
Unification: *x/n1, y/n2*
<=> *(n1 lt= n2)* against *(n1 gt= n2)*

<=> Entering *(n1 gt= n2)* into the number graph

<=> Compare relations between *n1* and *n2* :

test *lt=* with graph relation *gt=*

<=> Incompatible with negation of residue *(n1 equal n2)*

Unification: *x/n2, y/n1*

<=> *(n2 lt= n1)* against *(n1 gt= n2)*

<=> Entering *(n1 gt= n2)* into the number graph

<=> Compare relations between *n2* and *n1* :

test *lt=* with graph relation *lt=*

<=> Not incompatible

Comparing for compatibility *(x lt= y)* and *(n1 gt n2)*

Unification: *x/n1, y/n2*

<=> *(n1 lt= n2)* against *(n1 gt n2)*

<=> Entering *(n1 gt n2)* into the number graph

<=> Comparing relations between *n1* and *n2* :

test *lt=* with graph relation *gt*

<=> Not compatible

Unification: *x/n2, y/n1*

<=> *(n2 lt= n1)* against *(n1 gt e2)*

<=> Entering *(n1 gt n2)* into the number graph

<=> Comparing relations between *n2* and *n1* :

test *lt=* with graph relation *lt*

<=> Compatible with residue null

3.5.5.1 Unification and Evaluation"

For each pair of literals to compare, there are exactly two unifications to try: the traditional one - unify first from literal 1 with first from literal 2, second from literal 1 with second from literal 2; and the alternate one - unify first from literal 1 with second from literal 2, and second from literal 1 with first from literal 2. Two constants are considered unifiable if they exist in the number-specialist's domain already - although no substitutions are done for them.

For each successful unification, substitutions are done into the literals, and the resulting literals are evaluated using the number graph. The evaluations may answer the question of (in)compatibility immediately.

3.5.5.2 Testing for Residue

Before actually entering anything into the number graph, it is checkpointed so that all the changes can be retracted after each attempt with a different unification.

We now enter one literal into the number graph, and try to compare it against the other. This ensures that we can detect inconsistencies between the literals, even if no other information exists in the number graph. The choice of which literal to enter is quite simple - it doesn't really matter which is chosen, as long as it did not evaluate to NO.

To test the other literal, the number-specialist finds the currently existing relation between the arguments in that literal (*testreln*). This relation is then compared to the one given in the literal to be tested (*reln)* . The residue calculation is the same as that for the time-specialist.

3.5.6 State of Multiple-Environment Implementation

The number specialist currently uses only the current value of ***environment***, ignoring any other environments accessible from that one. A shared belief must be entered explicitly into the environment of each agent who holds it.

3.5.7 Known Problems and Possible Improvements

The maximum associated with a number is the largest number it can be equal to. In the case of real numbers, this maximum is set the same whether the number is < or <= to the maximum (for integers the exact maximum can be set; for reals it cannot). It is possible to fix this by adding another strictness value for the maximum of real numbers (and another for the minimum - the same problem exists there), but this hasn't been done yet. The minor inconsistency has not produced any problems yet.

This specialist was designed after the time-specialist, and although it is much simpler, it may be possible to simplify it even more to gain more efficiency, as it is such a simple domain.

There are lots of other arithmetic functions which could be handled by this specialist, including a multitude of Lisp functions. However, each function must have its arguments checked to ensure that if named numbers are involved, their value is used instead of the name - requiring an interface for each one. Time and need are the only considerations - the functions themselves are quite simple to implement.

3.6 Episode Specialist

This very simple specialist simply determines the relationships between the predicates **, *, and @. The information that ** subsumes * and @, and * subsumes @ is hardcoded in.

3.7 Specialists Based on Hierarchies

EPILOG supports several kinds of hierarchies, which all use the same basic hierarchy structure and algorithms. The hiearchies are for types, parts, and non-type predicates which are organized hierarchically. Type hierarchies are used by EPILOG to find new classifications to look under for wffs to apply to the current task (the hierarchies are "climbed" to look for rules to apply, and descended to find more specific

information). The type hierarchies are also used by the type-specialist. The non-type predicate hierarchies are used exclusively by the predicate hierarchy specialist (hier-specialist). The part hierarchies are also used to help find new classifications, but in a very limited way. They are mainly used by the part-specialist.

This section describes how the hierarchies are structured, built and used, as well as the specialists that use them.

3.7.1 The Hierarchies

This section describes the hierarchies themselves - what information they contain, how they are built, how they are used to compare predicates, etc.

3.7.1.1 Internal Structure

Each hierarchy node consists of a symbol with a number of properties (structures could have been used but since new properties are being added almost continually, properties are easier to deal with for now). The hierarchy properties are:

Root - the root node of the hierarchy. This is often the same as the hierarchy name (except for parts), but doesn't have to be. A number of different hierarchies can have the same root (the root must be broken down differently in each however).

Connections - connections to other hierarchies. This is a list consisting of pairs of hierarchy name and connecting node. The connections are based on the fact that the same predicate exists in more than one hierarchy - this predicate is the connection node.

Hier-type - *exclusion* or *overlap* . Sibling nodes in exclusion hierarchies are assumed to be disjoint; for overlap hierarchies this assumption is not made. This means that disjointedness and subsumption can be determined in exclusion hierarchies, but only subsumption in overlap ones.

Hier-change - indicates if there have been any changes since the last renumbering - i.e. new subnodes have been added.

Properties - a list - currently it should contain either *exhaustive* or *non-exhaustive* . In an exhaustive hierarchy, it is assumed that a breakdown of a node into subnodes is exhaustive - i.e. there are no more subnodes that could be there. In non-exhaustive hierarchies, this assumption is not made. To handle nodes where the exhaustive enumeration of subnodes is not possible, a subnode which is a remainder node may be added (these end in *-remainder, -other* , or *-rest*). Part hierarchies are assumed to be exhaustive, unless otherwise specified; while type hierarchies are assumed to be non-exhaustive.

Part-hier - t only if this is a part hierarchy - this is added automatically when **add-part-hier** is used.

Part-type-hier - t only if this is a type hierarchy which connects to a part hierarchy. This is also added automatically.

Part-type - indicates what type of part hierarchy this is. The default is FORM. FUNCTION and STATE are two other possibilities. Legal types are on ***part-hierarchy-types*** , which the user can add to. Care must be taken to give the types consistently to the part hierarchies however.

Each subnode on the hierarchy has the following properties:

Hier - a list of hierarchies the node belongs to

Super - a list of the parent nodes for this node, for each hierarchy. The list is of the form

((hier . super) (hier . super)...)

Sub - a list of the subnodes beneath this node, for each hierarchy. The list is of the form

((hier sub1 sub2 ...) (hier sub3 sub4 ...) ...)

Id - the number given to this node during hierarchy renumbering. This is actually stored as a list as there is a separate id for each hierarchy the node exists on. The list looks like

((hier . number) (hier . number))

Maximum - the maximum given to this node during hierarchy renumbering. This is actually stored as a list as there is a separate maximum for each hierarchy the node exists on. The list looks like

((hier . number) (hier . number))

Bounds - for part nodes only - a list of bounds given to this node for each part hierarchy it is on - of the form

((hier min max) (hier min max) ...)
This is used to differentiate between single parts and sets of parts.

Total - for part nodes only - a list of totals, taking into consideration higher level collections, for each part hierarchy it is on *((hier min max) (hier min max) ...)*

Parts - a list of the part hierarchies that have this node as root

3.7.1.2 Hierarchy Additions

The hierarchies are built using the **add-hier** and **add-part-hier** commands (the latter for part hierarchies, the former for all other hierarchies). These commands set up the links between parent and child nodes, and keep track of maintaining connections between hierarchies. Additional properties are added to the hierarchies using the **set-hier-type** command.

Hierarchies are all connected to the "entity" hierarchy automatically. This ensures that no matter what type is given for an individual, it will also be an "entity", and the hierarchy climbing and descending will proceed as expected. It is especially necessary when answering questions where no type is given for the variables.

Note that all hierarchy additions and changes are retractable.

3.7.1.2.1 Consistency Checking

Before adding subnodes to a hierarchy node, some testing is (optionally) done to ensure that the information being added is consistent - i.e. there are no loops being formed, either within a hierarchy or when connections are taken into consideration, and that if a relationship is already known to be disjoint, we don't try to add a subsumption relation between the same predicates (or vice versa - if the subsumption relationship holds we don't want to add disjointness). This testing includes checking to see if the parent node and subnodes already exist on a common hierarchy (in which case it is not necessary to add their relationship again). Also, each subnode and the parent node are tested to see if they are disjoint, or if the subnode has a subtype which is incompatible with the parent node (this makes the subnode also incompatible). Also, for exclusion hierarchies, checking is also done to ensure that none of the sibling

nodes have common subtypes, and that no subnode is a subtype of another subnode. This prevents loops. While additional consistency checking could be done, it is expensive, and the above catches most possible errors.

Because numbering has not yet been done, and we don't want to do it until the hierarchy is completely built (it takes too much time to do between subnode additions), the above testing is done the long way - following parent-child pointers.

Note: consistency checking for parts is much more limited - subnodes may be (and often are) disjoint from the parent node (a hand is NOT a finger, so they are disjoint). Currently there is no real consistency checking on part hierarchies, until it can be determined exactly what is legal.

3.7.1.2.2 Hierarchy Numbering

A preorder numbering scheme is used on the hierarchies which allows predicate comparisons within the same hierarchy to be done in constant time.

In addition to adding the ids and maximums to the various nodes, the "normal-level" may be set for familiar parts.

After numbering the hierarchy, if the an inferable relationship to "entity" is not found, a connection to the entity hierarchy is physically added.

number-hier-node *(hier node id level)*
  set *id* property of *node* for *hier* to *id*
  for every subnode *s* of *node* in *hier*
    if *s* is a familiar part
      use *level* and the *part-type* property of *hier* to update normal levels for *s*
    set *max* = (number-hier-node *hier s* (*id* + 1) (*level* + 1))
    set *maximum* property of *s* to *max*
  return *max*
end of number-hier-node

A small portion of the "thing" hierarchy is shown below, including the id and maximum numbers assigned to the nodes ([id,max]).

```
THING [1,112]
|      PHYSICAL-OBJECT [2,108]
|      |      LIVING-THING [3,40]
|      |      |      PLANT [4,15]
|      |      |        ...
|      |      |      CREATURE [16,40]
|      |      |      |      HUMAN [17,26]
|      |      |      |      ADULT [18,20]
|      |      |      |      |      |      MAN [19,19]
|      |      |      |      |      |      WOMAN [20,20]
|      |      |      |      |      MINOR [21,26]
|      |      |      |      |      |      ADOLESCENT [22,22]
|      |      |      |      |      |      CHILD [23,25]
|      |      |      |      |      |      |      GIRL [24,24]
|      |      |      |      |      |      |      BOY [25,25]
|      |      |      |      |      INFANT [26,26]
```

. . .

3.7.1.2.3 Adding Properties

The routine **set-hier-type** is used to add most of the hierarchy properties, although they are not all types. This includes the following properties: *hier-type, part-type, properties, pred-hier* . Internally, this routine is also called to set *part-type-hier, part-hier* .

3.7.1.3 Predicate Comparisons

Once the hierarchies have been built, the major task is comparing predicates within the hierarchies to see if they are disjoint, equivalent, or one subsumes the other. The predicates do not always exist on the same hierarchy, and connections between the hierarchies can be used to determine the relationship.

hier-rel *(a b)*
do for each *h* in sorted list of the intersection of *a* 's hierarchies and *b* 's hierarchies,
with part hierarchies removed (except for allowed ones) - sorted with exclusion hierarchies first
set *result* = (hier-rel-in-same-hier *h a b*)
if *result* is not *unknown* - stop - we have an answer
if *result* is *unknown* and effort level > 1
set *result* = (search-for-hier-res *a b*)
if *result* is still *unknown*
set result = (try-compound-hier-res *a b*)
return *result*
end-of hier-rel

3.7.1.3.1 Within A Single Hierarchy

This is the easiest kind of hierarchy predicate comparison. If both predicates exist on the same hierarchy, their id's and maximum's on that hierarchy are compared to give the relationship between the predicates. This is a constant time operation.

hier-rel-in-same-hier *(pred1 pred2 hier)*
; if necessary, renumber hierarchy before testing
if (hier-change property of *hier* is true OR
either *pred1* or *pred2* does not have an id number assigned)
renumber *hier*
; now compare ids and maximums

if id of *pred1* > max of *pred2* OR max of *pred1* < id of *pred2*
if *hier* is an exclusion hierarchy (not an overlap hierarchy)
then set *result* = *disjoint* else set *result* = *unknown*
if id of pred1 = id of *pred2*
set *result* = *equivalent*
if id of *pred1* > id of *pred2*
set *result* = *subsumed*
if id of pred1 <= id of *pred2*
set *result* = *subsumes*
otherwise set *result* = *unknown*
return *result*
end of hier-rel-in-same-hier

3.7.1.3.2 Searching Across Several Hierarchies

If the predicates to be compared do not exist on the same hierarchy, or the test within the hiearar-chy was inconclusive, a "search" is done to try to determine the relationship between them. Here the connections between hierarchies are used as kind of a "metagraph" (similar to the one used by the time specialist). The connections get us from hierarchy to hierarchy, and the simple in-hierarchy checking is used to compare the connecting nodes with the desired nodes. If a path is found between the two predicates using these connections, it may be possible to determine the relationship between them. Here we only need to look for a path from the first to the second - if no such path exists, one will not exist in the reverse direction either. Connections are sorted so that those to exclusion hierarchies are tried first - they are more likely to give an answer (because disjointedness can also be determined with them, where it can't with overlap hierarchies). Connections to part hierarchies are not used, except in special cases (see next section). Note that the "searching" is only done if the effort level *specialist-eval-effort* is greater than 0. If the effort level is = 1, only one link will be attempted during searching.

For example, if *horse* were found in the main hierarchy, under *creature* , subdivided into the various horse types - *Arabian, Morgan, Thoroughbred,* etc, and we had another hierarchy with *horse* in it where it was subdivided into *mare, stallion* and *gelding* , the "search" across the hierarchies would use *horse* as the connection between the hierarchies to determine that a *mare* is a *creature* .

search-for-hier-res *(a b)*
do for each *h* in sorted list of a's hierarchies (sorted with exclusion hierarchies first)
unless *h* is a part-hierarchy, and not in list of allowed ones
set *result* = find-hier-path *a h b* (> effort 1) *equivalent* *h*
if *result* is not *unknown* - stop - we have an answer
return *result*
end of search-for-hier-res
find-hier-path *(a hier b follow sofar hiers-seen)*
do for each *c* in sorted connection list of *hier* (sorted with exclusion hierarchies first)

```
        set newhier = hier part of c
        unless newhier in hiers-seen, or is a part-hierarchy and not in list of allowed ones
        do for each n in connecting nodes of c
            ; determine relationship between a and n
            set res1 = (hier-rel-in-same-hier hier a n ))
            ; combine that result with sofar - if they combined, continue else stop
            set newsofar = (combine-results sofar res1 )
            unless newsofar = unknown
                ; now determine relationship between n and b
                if b exists on newhier
              then set result = (combine-results newsofar (hier-rel-in-same-hier newhier n b))
                else
                    if follow
                    then set result = find-hier-path n newhier b follow newsofar
                                      (append hiers-seen newhier )
                    else set result = unknown
              if result is not unknown - stop - we have an answer
        end of do for each n
      if result is not unknown - stop - we have an answer
    end of do for each c
 return result
end of find-hier-path
```

```
combine-results (res1 res2)
    if either res1 or res2 is unknown
        set result = unknown
    if both res1 and res2 are disjoint
        ; this particular path cannot give us any information
        set result = unknown
    if res1 = res2
        ; the two are the same - subsumes and subsumes for example
        set result = res1
    if res1 = equivalent
        set result = res2
    if res2 = equivalent
        set result = res1
  if ( res1 = disjoint and res2 = subsumes ) OR ( res1 = subsumed and res2 = disjoint )
        ; subsumption in these cases helps rather than hinders the path
        set result = disjoint
    otherwise set result = unknown
```

return *result*
end of combine-results

3.7.1.3.3 Mixing Part and Type Hierarchies

The part specialist will ask for a predicate comparison, and give a list of allowed part hierarchies. When following the connections, if the connection is to an allowed part hierarchy, the search continues as if it were a regular type hierarchy, otherwise the connection is rejected as before.

3.7.1.3.4 Compound Predicates

Hyphenated predicates are internally split (if possible) into two parts, and it is assumed that the predicate is similar to the last part (so *left-hand* is similar to *hand*). If comparison using the regular methods has not produced a result, these predicate parts may be used. For example, *wolf-tail* is subsumed by *animal-tail* .

try-compound-hier-res *(a b)*
set *aparts* = predicate parts of *a*
set *bparts* = predicate parts of *b*
if both predicates are compound (i.e. *aparts* and *bparts* exist)
set *result* = (combine-results (hier-rel (first in *aparts*) (first in *bparts*))
(hier-rel (second in *aparts*) (second in *bparts*)))
if only *a* is compound
set *result* = (combine-results *subsumed* ;(assumed that "entity" is first part of *b*)
(hier-rel (second in *aparts*) *b*))
if only *b* is compound
set *result* = (combine-results *subsumes* ;(assumed that "entity" is first part of *a*)
(hier-rel *a* (second in *bparts*)))
return *result*
end of try-compound-hier-res

3.7.1.4 Hierarchy Ascending and Descending

When searching for formulas to apply to a given formula, whether in input-driven or goal-driven inference, the classification of the given formula is used as a first place to check. From there, related classifications are checked, including "super" classifications (given a particular wolf, you would want to look up rules involving wolves, animals, etc), and "sub" classifications (given a rule over animals, you might want to try to apply it to sub-types of animals, as well as specific instances of animals and all the sub-types).

3.7.1.4.1 Climbing Up the Hierarchies

To climb up the hierarchy, the parent node in each hierarchy the predicate exists on (with parts as a special case) are used. If the predicate is a part, and has a parent node in a part hierarchy, its super node is that parent node together with *-part* (e.g. a parent of *finger* is *hand-part*). If there is no parent for a part, its super node is just *part* . If there is no super node for a given predicate, *entity* will be used as a last resort.

Compound predicates, except for remainder predicates, are split into their parts, and the super nodes include the super nodes of the first part combined with the second (so *finger-part* would have super node *hand-part*), and the first part combined with super-parts of the second (so *timber-wolf* might have super node *timber-four-legged-quadraped*).

3.7.1.4.2 Descending the Hierarchies

To climb up the hierarchy, all sub-nodes in each hierarchy the predicate exists on (with parts as a special case) are used, as well as all instances of entities declared to be of that predicate.

Compound predicates have their sub-types determined much like the super-types are. *Hand-part* would have as sub-types both *finger* and *finger-part* . *Large-four-legged-quadraped* would have *large-wolf* as one of its subtypes.

3.7.2 State of Multiple-Environment Implementation

Hierarchy information is currently not encapsulated in the environment structure. Therefore, all of the system's beliefs that result from reasoning about hierarchies are attributed to other agents as well. However, see the section on the part specialist below.

3.7.2.1 Known Problems and Possible Improvements

Some consistency checking for part hierarchies should be added to ensure that valid input is given, and that no loops form (which could lead to infinite looping during searching).

Better handling and definition of compound predicates could help prevent confusion. Mainly they are used now to prevent having to create all the part types, and *part-part* and *type-part* predicates. They are created as needed only.

3.7.3 Type Specialist

Currently this specialist uses the type hierarchies as a logically true representation of the relationships among the predicates in a hierarchy. It uses the hierarchy algorithms described above to determine the relationship between predicates. A single top level operator (one of very, sort-of, almost, extremely) will be accepted and ignored by the type specialist - all other work is done by the hierarchy routines.

3.7.3.1 Known Problems and Possible Improvements

The type specialist does not allow for the possibility that different people have different conceptions of type hierarchies. Neither does the rest of EPILOG, but so far this has not been a problem.

Currently exhaustiveness of type hierarchies is available but is not being used. To be used, we would need to store all the types asserted about an individual in the type-specialist, since generally to use exhaustiveness, more than a simple comparison of two predicates is involved - there may be a large number of predicates involved, especially negative information. This would change the specialist from a simple type 1 specialist (which does not keep story facts in its own domain) into a more complicated type 2 specialist (which does keep story facts in its own representation). The type 2 specialists are more complicated to write, but have more flexibility in what they can infer, and how they can assist EPILOG. This remains as a future enhancement.

3.7.4 Predicate Hierarchy Specialist

This is the very simplest of the specialists using the hierarchies. It does not accept any predicate operators, and all its inference is done by letting the hierarchy routines described above determine the relationship between the given predicates. Only connections between other pred-hier hierarchies are allowed. Most non-type predicate hierarchies are very small, and do not have many connections to other hierarchies.

3.7.5 Part Specialist

The part-specialist handles part-of relationships, and uses part hierarchies for many, but not all, of its calculations.

3.7.5.1 Internal Representation

The representation used by the part hierarchies has already been described. Additional structures are used to keep track of specific parts, and their relations to other parts and entities. Currently only "part-of" relations are handled, although it is possible that other relations, like "joins" may someday also be handled.

A part subnet record exists for each subnet, just as for other specialists which store story facts in their own domain. This consists of a subnet identifier, and a pointer to the hash table of part records for that subnet.

Each part record contains the following fields:

name - the name of the part or entity this record is for

collection - a flag indicating whether this is a collection of parts or a single entity

types - contains a list of types applicable to this part

parts - a list of the names of sub-parts of this particular object. These are added when *part-of* assertions are made - the first argument is added to the *parts* list of the second.

super-parts - a list of the names of super-parts for this particular object. These are added when *part-of* assertions are made - the second argument is added to the *super-parts* list of the first.

owning-entity - if this record is for a specific part, the owning entity is the individual which the part belongs to. This may be given explicitly (e.g. asserting *(leg1 part-of lrrh)*), or may be inferred by the specialist using the transitivity of *part-of* .

The accompanying diagram shows what the part records would look like if the following information were entered (the part name and type fields are the top two fields, respectively):

(mary girl)

(a1 arm)

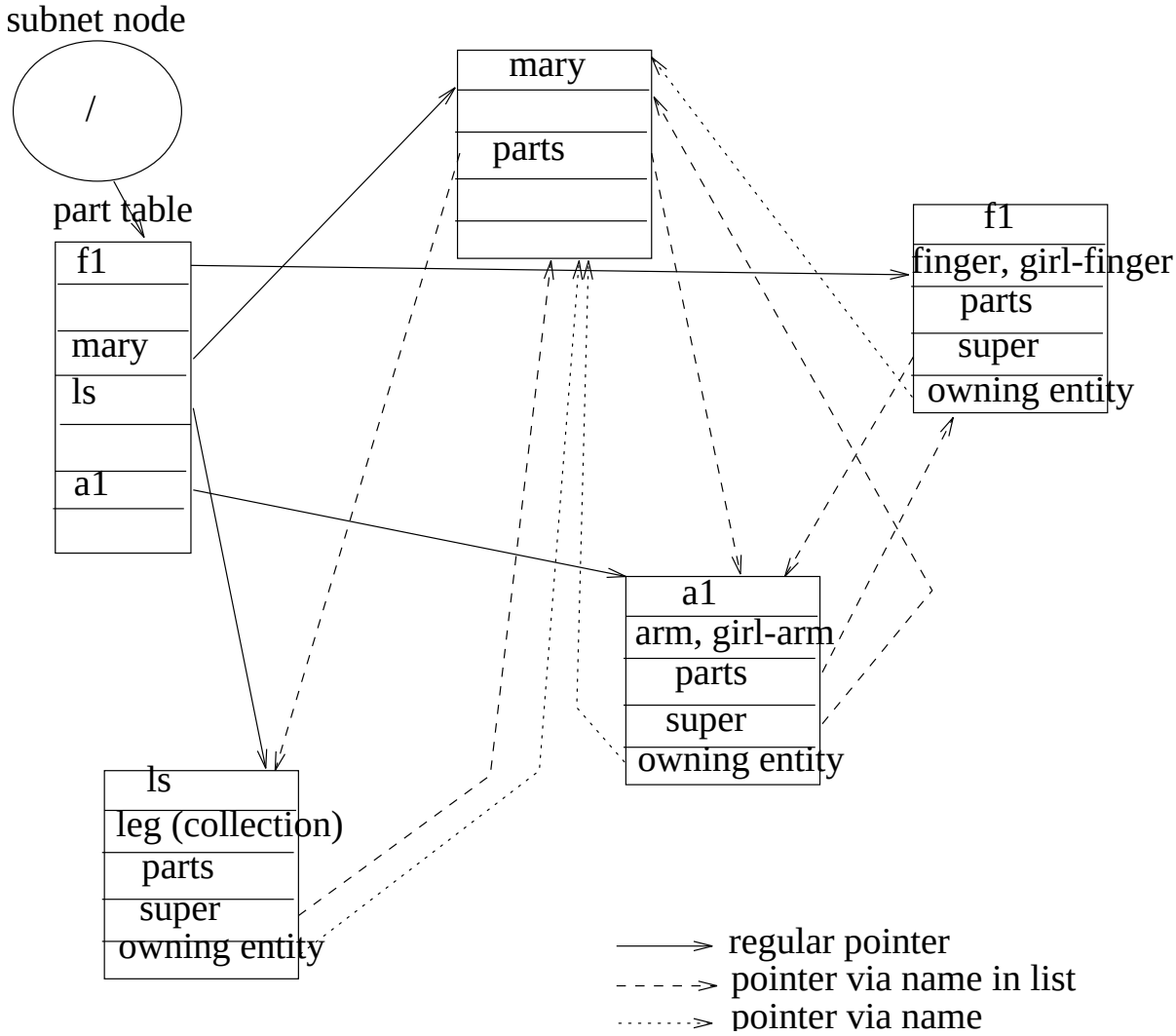
(a1 part-of mary)

(f1 finger)

(f1 part-of a1)

(ls_set ((coll-of 2) leg))

(ls part-of mary)



Example Part Records

The part specialist would also make the following assertions back to EPILOG:

(a1 girl-arm)

(f1 girl-finger)

(A x (x member-of ls) (x girl-leg))

(A x (((x leg) and (x part-of mary)) <=> (x member-of ls)))

3.7.5.2 Entry

The part specialist is interested whenever anything is asserted about a part-type, or a part-of relation. Both are stored in its internal representation. Equalities among parts are also stored - relevant information is copied from one record to the other and the two parts and then made to point to the same part record.

The specialist also makes assertions back to EPILOG about more specific part-types if the type of the owning entity is known - for example, if *leg1* is asserted to be a leg, and part-of Little Red Riding Hood, the part specialist will infer that it is also a *girl-leg* . It is possible that this could be inferred on its own during evaluation, but so far it cannot be, so this is the only way that such predicates can be used as expected.

In addition, if a collection of parts is asserted to be part of an entity, and it contains the maximum number of that type of part allowed for the entity, an assertion is made back to EPILOG that any part of that type belonging to that entity must be a member of the collection, and vice versa.

3.7.5.3 Literal Evaluation

To evaluate part-of relations of the form *(a part-of b)* , the following is used:

To answer affirmatively (YES), one of the following must hold:

- *a* is one of the items in the list of *parts* on *b* 's part record
- a path from *a* to *b* can be found using the *super-parts* field on the part records (handles the transitivity of *part-of*)

To answer negatively (NO), one of the following must hold:

- *b* is an individual, *a* is of a type which is a familiar part (i.e. on the list *familiar-parts*) and this type does not occur on any part hierarchy for *b* , within the search limits set for that part type.
- *a* and *b* have different owning entities, or *b* is an individual, but *a* belongs to an owning entity which is not equal to *b* , or has a type incomaptible with the type of *b* .
- *a* is a collection of parts whose cardinality is greater than the maximum number of that type of part allowed by *b* 's part hierarchies.

Equality of parts *(a equal b)* is determined as follows:

To answer affirmatively (YES), one of the following must hold:

- both *a* and *b* point to the same part record

To answer negatively (NO), one of the following must hold:

- a is a collection of parts, and b is not (so it is a single part), or vice versa
- a and b have incompatible types
- a and b have different owning entities, or their owning entities have incompatible types

Other literals evaluated by this specialist are of the form *(a part-type)* , where the *part-type* predicate exists on a part hierarchy. These will also be attempted by the type specialist. They are evaluated as follows:

To answer affirmatively (YES), one of the following must hold:

- using the part hierarchies available to the owning entity of a , and the type hierarchies, *part-type* is found to be equivalent to one of the known types for a .
- if *part-type* is *part* , or a compound predicate (like *girl-part*), using the part hierarchies available to the owning entity of a , and the type hierarchies, *part-type* is found to subsume one of the known types for a .

To answer negatively (NO), one of the following must hold:

- using the part hierarchies available to the owning entity of a , and the type hierarchies, *part-type* is found to be incompatible with one of the known types for a .
- if *part-type* is *part* , or a compound predicate (like *girl-part*), using the part hierarchies available to the owning entity of a , and the type hierarchies, *part-type* is found to be incompatible with one of the known types for a .
- a is known to be a collection of parts (and therefore cannot be of a simple part type)

3.7.5.4 Literal Comparison

Literal comparison in the part specialist is restricted to unifications and evaluations - no entry and testing is done as for other specialists which maintain their own representation of story facts. This should probably be changed, but will wait until the part specialist's role is better defined.

3.7.6 State of Multiple-Environment Implementation

Like the other hierarchy specialists, the part specialist uses the same information about hierarchical relationships among predicates in simulations as it does at the top level. However, the information about part relationships between individuals is encapsulated in the environment structure, and is therefore not shared by simulations. In the current version, the specialist only consults the environment indicated by ***environment***; the contents of any shared environments accessible from that one are ignored.

3.8 Specialists for Handling Equivalence and General Sets

Equivalence information is maintained by the use of equivalence sets, and so the lower level routines to handle sets are used both for maintaining equality information and general set relationships. This section

describes the low level set structures, the information they contain, how they are created and used, as well as the specialists which use them. Since one of the specialists is the equality-specialist, general equality handling is described in this section as well.

3.8.1 The Set Structures

The set structures are designed to handle general sets. The equivalence sets make use of only a few of the properties handled here - maintaing the known members of the equivalence sets. However, since the equality specialist does use set structures, it makes sense to have both the set and equality specialists use the same low level structures. This section describes those structures, and how they are built. If you are only interested in the equality specialist, you can probably ignore the set-connection discussions.

3.8.1.1 Internal Structure

Each set has an *episet* record associated with it, which contains information on the set's member-type, actual members, cardinality, connections to other sets, whether or not all the members are known, etc. The episet records are accessed via a hash table which hashes on the set's name. When two sets are made equal, they hash to the same episet record.

Episet records are connected via pointers for subset-of, set-difference, intersection and union relations.

Each *episet* structure contains the following fields:

identifier - name. This is the name supplied for the set by the user, or may be an internally created name (such as *lrrh-equal*). The equivalence sets are named *object* **-equal** , and the sets of items not equal to a particular object are named *object* **-non-equal** . Some of the more complicated relations are handled more easily be creating a new set name (and accomapnying set record) for the relation, including intersection and union, as well as the operator *set-of-all* . This will be discussed in more detail later.

properties - currently the only properties handled here are *equivalence* (this set is an equivalence set), and *non-equal* (this set contains items not equal to some item). These properties are internally assigned when the set is created.

finished - indicates whether all members are known by specialist. This is an internally maintained flag. If all members are known, certain tests may be done on the members during evaluation; if they are not all known, there may be more members, so some of those tests may not be applicable.

contents - list of currently known members. A + in this list indicates that there may be more members than are currently known.

member-type - type of members (a predicate). This is set by operators *coll* , *coll-of* , or *set-of-all* , or by an operation involving sets with member-types in relations such as intersection or union.

all - flag indicating whether this is a set of all of that type (i.e. this set was created through the operator *set-of-all*).

cardinality-min - minimum cardinality of set

cardinality-max - maximum cardinality of set

connections - list of connection records to other sets (for subset, intersection, union, and set difference relations)

connections-from - list of connection records from other sets (for subset, intersection, union, and set-difference relations)

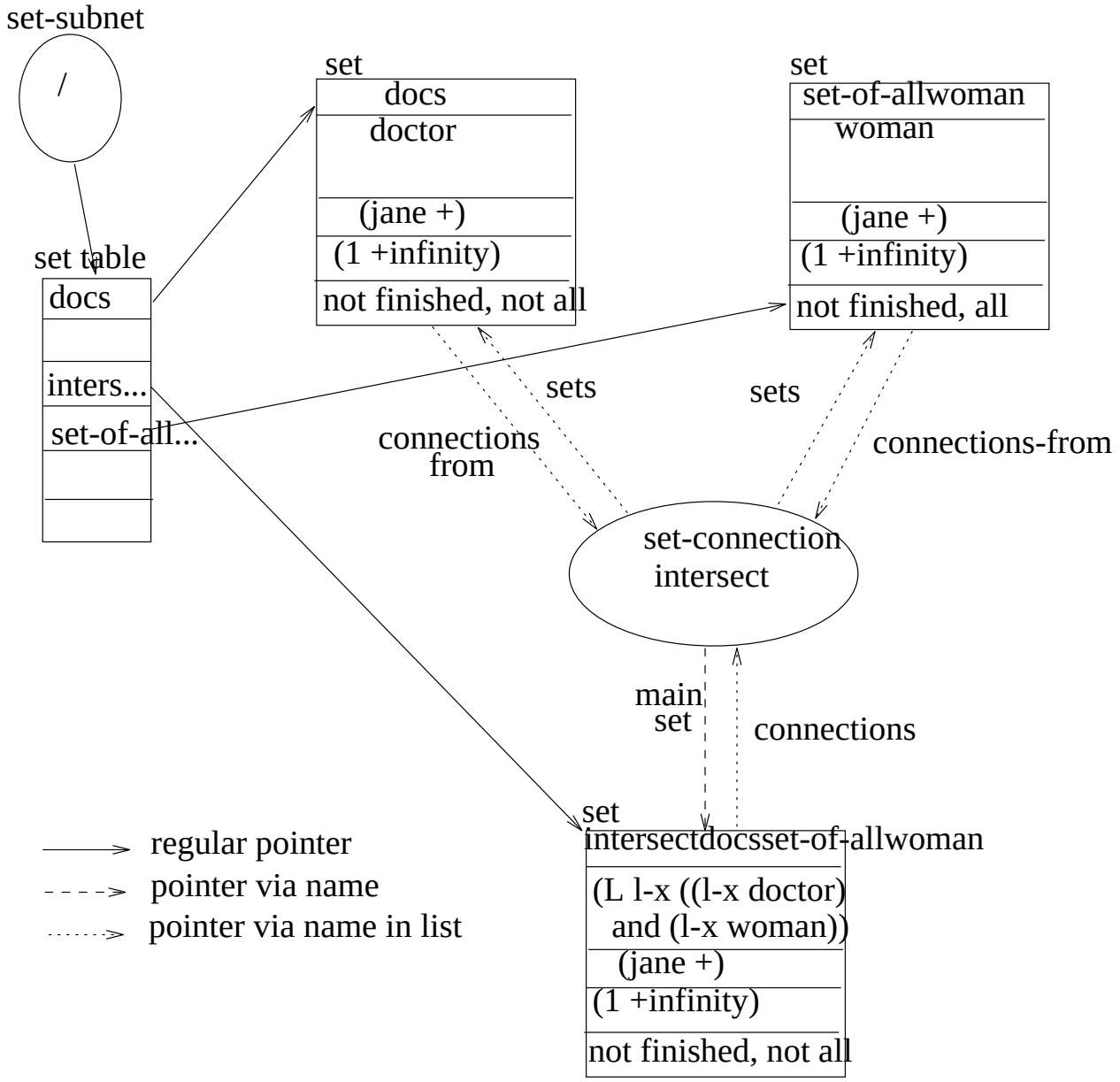
An *episet-connection* record contains the following fields:

how - how the sets involved are connected (*union-of, intersect, subset-of, set-diff*). This is then used to determine whether to propagate members from sets to connecting sets.

sets - a list of the names of the sets involved in the operation

main-set - the name of the main set affected - i.e. the set resulting from the relation

In addition, a separate subnet is kept for each person and each modal predicate. This is handled by the *episet-subnet* record, which contains the subnet identifier (*/* , */john-believe* , etc), and a pointer to the hash table of sets for that subnet. An hash table of these subnet records is kept at the highest level, indexed by subnet identifier.



Example Set Structures

The accompanying picture shows what the set records and connections would look like if the following assertions were made:

(docs_set (coll doctor))

(jane member-of (intersect docs (set-of-all woman)))

In the picture, the episet records show the fields in the following order: identifier, member-type, contents (including a + to show there may be more members), cardinality minimum and maximum, "finished", and "all" properties.

In addition to the creation of this episet structure, the following assertions would be made back to EPILOG by the set specialist:

(A x (x member-of docs) (x doctor))

(A x ((x member-of set-of-allwoman) <=> (x woman)))

(A x (x member-of intersectdocsset-of-allwoman) ((x doctor) and (x woman)))

Using the regular input driven inference mechanisms, this would ensure that *(jane doctor)* and *(jane woman)* were inferred.

3.8.1.2 Entry of Set Information

The addition of set information may come from either the set or the equality specialist. The equality specialist adds members to equivalence and non-equal sets only, and updates their cardinality. It does not touch set connections, which handle the intersection, union, or subset relations between sets.

3.8.1.2.1 Adding Members

Adding a member to a set is simply a matter of updating the contents of the episet record to show the new member. The minimum cardinality may also be adjusted so that it keeps pace with the number of known members. In addition, if the set is connected via episet-connections to other sets, it is possible that adding a member to this set also affects the other sets, so they may also have the member added (e.g. if a member is added to a set which is an intersection between two other sets, the new member must also be added to the other sets).

If new members are added to an equivalence set, the equivalence information in the main EPILOG equivalence table is also updated.

3.8.1.2.2 Updating Cardinality

Cardinality may be updated by an explicit assertion or inference from EPILOG (using *has-cardinality* or the function *cardinality-of* , which is flattened to *has-cardinality*). The minimum may also be increased when members are added to a set (the cardinality must be at least as high as the number of known members). In addition, when sets are combined in an intersection, subset or other relation, the cardinality of the new set will be set according to the cardinalities of the sets in the connection.

3.8.1.2.3 Handling Set Connections

Set relations such as intersection, union and set-difference are handled via the functions *intersect* , *union-of* , and *set-diff* . This seemed the most natural way for the user to use them. In order to make this work properly, the functions cause a special set to be created inside the specialist, with a unique name unlikely to be chosen by a user for anything. The set is created and connected, regardless of whether this is an entry or evaluation procedure - creating it for evaluation purposes does not affect other sets, and merely takes up some space. These special sets are named with a combination of the relation and the names of the sets involved. The user may equate the set to one of his own naming to refer to the set more often, but there is no guarantee that he will do so, and so the specially named set must be created.

Subset relations are entered using the predicate *subset-of* . The above operations create a unique set (there is only one set that is the intersection of two sets, etc), but subsets are not unique, so must be

handled differently.

In order for proper updating of members for sets involved in intersection, union, subset, or set difference relations, the sets themselves are permanently tied together so that updates on one can be properly reflected on the others. This is handled by creating a set-connection record to reflect the required relation information.

When new information about a set is input (such as additional members), it will be propagated to the other affected sets via the connection records using the following:

(s1 = (union-of s2 s3)) The *(union-of s2 s3)* part corresponds to a set created internally called *union-ofs2s3* . This set is equated to *s1* . Any new member of *s2* or *s3* is automatically added to *s1* . However, new members of *s1* are not added to *s2* or *s3* because there is an inherent ambiguity there - it is not known which set to add to. Since the specialist does not maintain members which do NOT belong to sets, it is unlikely that a future assertion will resolve this. This may possibly create a hole.

If *s2* and *s3* both have member types, *s1* will be given a member type which is a lambda expression which captures the disjunction of the other member types.

(s1 = (intersect s2 s3)) The *(intersect s2 s3)* part corresponds to a set created internally called *intersects2s3* . This set is equated to *s1* . Any new member added to *s1* is automatically added to both *s2* and *s3* , and any new member added to both *s2* and *s3* is also added to *s1* .

If *s2* and *s3* both have member types, *s1* will be given a member type which is a lambda expression which captures the conjunction of the other member types.

(s1 = (set-diff s2 s3)) The *(set-diff s2 s3)* part corresponds to a set created internally called *set-diffs2s3* . This set is equated to *s1* . Any new member added to *s1* is automatically added to *s2* . Any new member added to *s2* which can be proven to NOT be a member of *s3* will be added to *s1* . Because negative information is not saved, there are possibilities here which may be overlooked, with the potential of creating a hole.

If *s2* and *s3* both have member types, *s1* will be given a member type which is a lambda expression which captures the conjunction of the first member type and the negation of the second one.

(s1 subset-of s2) No special naming is done for subnets, as they are not unique. Any new member of *s* 1 is automatically added to *s2* .

If *s2* has a member type, and *s1* doesn't, *s1* will be given *s2* 's member type as well.

3.8.1.2.4 Inferences during Entry

During entry of set information, it is possible that something of interest to the rest of the system might be noted, and so this information is passed onto EPILOG in the form of an episodic logic assertion. The assertion is added to the ***assertions*** list.

Two types of assertions may be made for sets: the first has to do with member types. Whenever a set's (*s*) member type (*p*) has been determined, whether directly through an operator like *coll* , or indirectly through an intersection or other operation, the following inference is made:

(A x (x member-of s) (x p))

If the information came from the *set-of-all* operator, the inference is a somewhat stronger equivalence:

(A x ((x member-of s) <=> (x p)))

The other kind of inference made for sets is equality of members. If a set whose cardinality is 1 is equated to another set, each member of the second set is asserted equal to the single member of the first set. This can be phrased as the following axiom:

(A x_set ((cardinality-of x) = 1)
(A y_set (x equal y)
(A m1 (m1 member-of x) ; all of 1 member
(A m2 (m2 member-of y) (m2 equal m1)))))

The *(m2 equal m1)* inferences are asserted back to EPILOG.

3.8.1.3 Known Problems and Possible Improvements

Order of information input may be important - like the rest of the system, if types are available they should be input first (using *coll-of* , *coll* , and *set-of-all*).

It is possible that there are more cases where equality can be determined, other than just when two sets are equated and one has cardinality 1.

It is possible for some holes to be created with the union and set-diff operations on sets - some members which could be inferred to belong to a set may not be (because negative information is not stored). However, membership relations that can be inferred with all positive information, involving no disjunctions, and simple constants only, are all made.

3.8.2 General Handling of Equality

A hash table exists in EPILOG, which is keyed into using the name of an item, and a subnet identifier (*/* , */john-believe* , */johnmary-tell* , *john-believe/mary-believe* , etc). This table contains equivalence sets - sets of items which are equal to the item used to index there, within the modal embedding described by the subnet identifier. The table is updated by the equality specialist, which detects the equalities and builds the equivalence sets.

The table is currently used during lookup to determine equivalent classifications to look up under, during comparison of two formulas to see if they are equal (the equivalence classes of the arguments are checked), and during unification of two constants.

After an equality is entered into any applicable specialist, a retrieval is done to get all formulas within a subnet for the individuals concerned. Each of these is (re)entered into the applicable specialists for ALL equivalent subnets, effectively merging the subnets. This ensures that the specialists always have all the information applicable to a particular subnet. In the main EPILOG core, this duplication is not necessary, as the retrieval, classification, unification, and evaluation routines already take equivalences into consideration.

3.8.2.1 Known Problems and Possible Improvements

There is only one minor case where the current handling of equality and subnets may miss information by not duplicating storage in the main net - and that is if a question is asked where a top level of a subnet is not known so that the lower level equivalences cannot be determined (such as if you ask "did someone think that mary was foolish", and we have stored that "john thinks that mary equals mrs.pearson" and

"john thinks that mrs.pearson is foolish" - the question will not be answered). However, changing the system to duplicate all subnet information is much more difficult than the methods used to duplicate it for the specialists, and would increase storage requirements and program complexity. If it turns out that this is an important type of question to be answered, the appropriate changes will of course be made, but until then, this will remain as a possible future enhancement.

3.8.3 Equality Specialist

The equality specialist uses the low level set structure and routines described earlier to maintain equivalence sets for positive equality information, and to maintain non-equal sets for negative equality information. Equivalence sets are transmitted back to EPILOG into the equivalence hash table.

3.8.3.1 Entry

When two items *a* and *b* are asserted to be equal, *a* is added as a member to the set *b-equal* , and the set *a-equal* is made equal to the set *b-equal* (with any accompanying copying of members). The equivalence table in the main EPILOG space is updated. If any new equalities are detected by this, they are asserted back to EPILOG so that other specialists can store them as well.

When two items *a* and *b* are asserted to be not equal, *a* is added as a member to the set *b-non-equal* , and *b* is added as a member to the set *a-non-equal* .

3.8.3.2 Literal Evaluation

Determining whether two objects *a* and *b* are equal is done using the following (note that this is quite similar to set membership evaluation):

To answer affirmatively (YES), one of the following must hold:

- *a* is the same as *b*
- *a* is a member of the set of equivalences for *b* (i.e. a member of the set *b-equal*)

To answer negatively (NO), one of the following must hold:

- *a* is a member of the set of things which are not equal to *b* (i.e. a member of the set *b-non-equal*)
- *a* 's sort is not equal to *b* 's sort
- *a* and *b* are not skolem constants, are not equal, and the unique names assumption holds (i.e. *unique-names-assumption* is t). If this being evaluated for an assertion, the unique names assumption is not used.

3.8.3.3 Literal Comparison

This is probably the simplest literal comparison of all of the specialists that do it, because there is only one possible predicate, and no residues. The arguments may be unified in two possible orders (first of literal 1 with first of literal 2, and second with second; first of literal 1 with second of literal 2, and second with first). For each successful unification, the two literals, after substitution, are evaluated, which may answer the comparison question immediately.

No entry and testing are currently being done, although it could be. It is unlikely that comparisons of this nature will be helpful, and more likely that if they are, some other specialist will be involved which is more specific to the arguments (like the time or number specialist).

3.8.4 Set Specialist

The set specialist uses the low level set structure and routines described earlier to enter the membership, subset, cardinality, and set connection information given to it. This section describes the evaluation of set formulas.

3.8.4.1 Literal Evaluation

Formulas of the form *(m member-of s)* or *(m member-of-0 s)* are evaluated using the following:

- m is among the known set members of s Either m was explicitly asserted to be a member of s , or it was inferred to be through a set combination operation (like intersection or union).
- m is equal to a known set member of s (calling on other specialists to help determine equality)
- if the predicate is *member-0* , the above two checks are also used recursively to see if m is a member of a set which is a member of s .

To answer negatively (NO), one of the following must hold:

- any of the types which have been asserted about m conflict with the member type assigned to s through *coll* , *coll-of* , or *set-of-all* , or a combination operation.
- s 's members are all known, and m is not equal to any of them (calling on other specialists to help determine equality)
- s 's cardinality is less than 1 - i.e. it has no members

Subset relations *(s1 subset-of s2)* , including transitivity and member relations are determined using the following:

To answer affirmatively (YES), one of the following must hold:

- following subset links, a path subset links must exist between *s1* and *s2*
- *s1* 's members are all known, and all are also members of *s2* (using the above *member-of* tests)

To answer negatively (NO), one of the following must hold:

- the member-type of *s1* is not subsumed by that of *s2* (i.e. it conflicts, or subsumes). Member-type is assigned by the *coll* , *coll-of* and *set-of-all* operators, and some set combination operations.
- the cardinality of *s1* is greater than that of *s2* (i.e. *s1* has more members than *s2*)

Equality between sets *(s1 equal s2)* is determined with the following:

To answer affirmatively (YES), one of the following must hold:

- both set names point to the same set record

- both sets have all their members known, and any member of one is also a member of the other

To answer negatively (NO), one of the following must hold:

- the cardinality of one set is greater than the cardinality of the other
- the member type of one conflicts with the member type of the other
- a member of one set is not a member of the other (using the member-of tests mentioned earlier)

3.8.4.2 Function and Term-Forming Operator Evaluation

The **set-of** function returns a record consisting of the sort *set* , and all the arguments to the function as the record contents.

The **cardinality-of** function returns a number which is the cardinality of the argument given to the function. If the cardinality is not known exactly (both minimum and maximum the same), nil will be returned.

The **set-of-all** term-forming operator simply returns the internally calculated, unique name *set-of-allpred* , where *pred* is the predicate argument given to the operator. However, internally, it has the side effect of creating a set by that name, with *pred* as its member type, and property *all* t, as well as generating the assertion *(A x ((x member-of set-of-allpred) <=> (x pred)))* .

The **intersect** function simply returns the internally calculated, unique name *intersects1s2* , where *s1* and *s2* are the sets given to it as arguments. However, internally, a set is created by that name, with set connections showing that it is the intersection of sets *s1* and *s2* . Both arguments must be set names.

The **union-of** function simply returns the internally calculated, unique name *union-ofs1s2* , where *s1* and *s2* are the sets given to it as arguments. However, internally, a set is created by that name, with set connections showing that it is the union of sets *s1* and *s2* . Both arguments must be set names.

The **set-diff** function simply returns the internally calculated, unique name *set-diffs1s2* , where *s1* and *s2* are the sets given to it as arguments. However, internally, a set is created by that name, with set connections showing that it is the set difference between sets *s1* and *s2* . Both arguments must be set names.

The **member-type-of** operator returns the predicate which is the member type of the set given to it as an argument. This predicate is set by the *coll* , *coll-of* or *set-of-all* operators, or a set operation combining sets with member types.

The **set-members** and **set-members-0** functions return a record which contains the currently known members of the set given as an argument to the function. If the function is **set-members-0** , the members are recursively determined using members of members which are themselves sets. The argument may be a set name or a record.

The **number-members** and **number-members-0** functions return a number which is the number of members which are currently known for the set given as an argument to the function. If the function is **number-members-0** , members are recursively determined using members of members which are themselves sets, and then counted. The argument may be a set name or a record.

The **intersect-members** and **intersect-members-0** functions return a record corresponding to the result of intersecting the currently known members of the sets given as arguments to the function. If the function is **intersect-members-0** , the members are recursively obtained if some of them are themselves sets. Arguments may be set names or records (and there may be any number of members).

The **union-of-members** and **union-of-members-0** functions return a record corresponding to the result of calclating the union of the currently known members of the sets given as arguments to the function. If the function is **union-of-members-0** , the members are recursively obtained if some of them are themselves sets. Arguments may be set names or records (and there may be any number of members).

The **set-diff-members** and **set-diff-members-0** functions return a record corresponding to the result of calculating the difference between the currently known members of the two sets given as arguments to the function. If the function is **set-diff-members-0** , the members are recursively obtained if some of them are themselves sets. Arguments may be set names or records (but only two arguments).

3.8.4.3 Literal Comparison

Like most other specialists which compare literals, it first attempts to unify the literals, and then evaluates them after substituting. If that still hasn't given an answer, if both literals are subset-of literals, or both are membership literals, it will compare the set members involved to see if the literals are (in)comaptible, and what the residue, if any, should be. If this fails to give a result, it will temporarily enter one literal, and do a simple evaluation of the other. This last stage is not capable of calculating residues, so may miss some potential comparisons.

The following contains details of the special comparisons when both literals are subset-of, or both are membership literals:

Testing when literals have predicates member-of or member-of-0, or both literals have predicate subset-of AND the first argument of each literal matches:

When testing for **incompatibility** *(x pred A)* vs *(x pred B)* :

both are negated - calculate the intersection of set members of *A* and *B*

the first literal is negated - calculate the set difference between the set members of *B* and the set members of *A*

the second literal is negated - calculate the set difference between the set members of *A* and the set members of *B*

neither literal is negated - calculate the union of the set members from *A* and *B*

A null result indicates incompatible with a null residue

A result indicating unfinished sets (contains a +) indicates not incompatible

Otherwise the literals are incompatible with a residue which is one of the original literals (if first is negated, and second isn't, second literal, otherwise first) with its second argument replaced by a set record whose contents are the members as calculated above

When testing for **compatibility** *(x pred A)* vs *(x pred B)* :

both are negated - calculate the union of set members of *A* and *B*

the first literal is negated - calculate the set difference between the set members of *B* and the set members of *A*

the second literal is negated - calculate the set difference between the set members of *A* and the set members of *B*

neither literal is negated - calculate the intersection of the set members from *A* and *B*

A null result or a result indicating unfinished sets (contains a +) indicates not compatible
Otherwise the literals are compatible with a residue which is one of the original literals
(if both are negated, second literal, otherwise first literal) with its second argument
replaced by a set record whose contents are the members as calculated above

Testing when both literals have predicate subset-of, and the second argument of each literal matches:

When testing for **incompatibility** *(A pred x)* vs *(B pred x)* :
both are negated - calculate the union of set members of *A* and *B*
the first literal is negated - calculate the set difference between the set members of
A and the set members of *B*
the second literal is negated - calculate the set difference between the set members of
B and the set members of *A*
neither literal is negated - calculate the intersection of the set members from *A* and *B*
A null result indicates incompatible with a null residue
Otherwise the literals are incompatible with a residue which is one of the original literals
(if the second is negated, second literal, otherwise first) with its second argument
replaced by a set record whose contents are the members as calculated above

When testing for **compatibility** *(A pred x)* vs *(B pred x)* :
both are negated - calculate the intersection of set members of *A* and *B*
the first literal is negated - calculate the set difference between the set members of
A and the set members of *B*
the second literal is negated - calculate the set difference between the set members of
B and the set members of *A*
neither literal is negated - calculate the union of the set members from *A* and *B*
A null result indicates not compatible
Otherwise the literals are incompatible with a residue which is one of the original literals
(if the first is negated, but not the second, second literal, otherwise first) with its second argument
replaced by a set record whose contents are the members as calculated above

3.8.4.4 Known Problems and Possible Improvements

There are more possibilities in literal comparison than are currently being handled. This is a rather difficult area to work with as it is hard to tell what a useful comparison is.

Some disjunctions (e.g. entry with *member-of-0* , handling additional members added to a set created by *union-of*) look like they should be handled by the specialist, but are not, since definite information is not available. This can lead to confusion and possible missed inferences. In addition, some negated information (such as *m1* is NOT a member of *set1*) has potential usefulness, but is not currently being saved. Saving this negative information could help with the disjunctions. There is a tradeoff between

complexity and utility of the specialist here that should be considered.

3.8.5 State of Multiple-Environment Implementation

The set and equality specialists can reason with the contents of multiple environments simultaneously, using a shadowing scheme similar to that described for the time specialist on page 74.

3.9 String Specialist

The string specialist evaluates literals involving string comparison predicates, and handles functions which manipulate strings. It does not store story facts in its own domain, and may be used for literal and function evaluation only. Named strings are currently not handled - only simple strings, such as "HELLO".

Most of the predicates and some of the functions are actually interfaces to the Lisp functions of the same name. Note that the user must be careful what arguments are used with these functions since no checking is done to ensure that they are legal before invoking the Lisp routine.

Note that modal embedding is ignored by this specialist.

3.9.1 Literal Evaluation

The following patterns are accepted by the string-specialist:

string	string=, string-equal, string/=	string
	string<, string>, string<=, string>=	
	string-lessp, string-greaterp	string-not-lessp, string-not-greaterp
	string-not-equal	string-contain

To evaluate whether a literal with the predicate *string-contain* is true, the string-specialist looks for the second string in the first one, starting at the beginning and comparing character by character. If it finds the string, the literal will evaluate to t, otherwise nil.

The remaining predicates are handled by calling the Lisp routine of the same name. They are:

string= string1 string2	compares strings to see if characters identical
string-equal string1 string2	like string= but case ignored
string< string1 string2	compare strings lexocographically - true if <
string> string1 string2	compare strings lexocographically - true if >
string<= string1 string2	compare strings lexocographically - true if <=
string>= string1 string2	compare strings lexocographically - true if >=
string/= string1 string2	compare strings lexocographically - true if not=
string-lessp string1 string2	same as string< except case ignored
string-greaterp string1 string2	same as string> except case ignored
string-greaterp string1 string2	same as string> except case ignored
string-not-lessp string1 string2	same as string>= except case ignored
string-not-greaterp string1 string2	same as string<= except case ignored
string-not-equal string1 string2	same as string/= except case ignored

Note that the *:key* parts of these routines can be used, and the evaluation will be correct. However, if the response generator tries to say the keywords, the output may sound strange.

3.9.2 Function Evaluation

The **string-field** function takes as arguments a string and an integer. The integer refers to the portion of the string to return. The string is assumed to be divided into portions, separated by the characters in the ***string-field-divider*** parameter (initially set as -). The function looks for the next separator, or end of string, until the appropriate portion is found, and returns it.

The **sub-string** function takes as arguments a string and possibly two integer numbers, which specify the start and end positions, respectively, for the substring. If the end is not specified, the substring will be taken to the end of the string. The new string is created by copying elements from the original string to the new one, one at a time from the start position to the end position.

The **string-concat** function takes as arguments any number of strings to be concatenated together into one string. If ***string-separator*** contains a character, this character will be placed between each pair of strings in the new string (the default is to squish them all together with no separating characters). Again copying to the new string is done character by character.

The **string-number** function takes a single string argument, and returns the number corresponding to the string. So "3.5" would return 3.5. The part of the string before a decimal point is separated out, and the part after. Both are converted to numbers, and the post decimal number is divided by 10 until it is less than 1. Then the pre-decimal number is added to this to return the final number.

The **number-string** function takes a single number argument, and two optional number arguments, and returns the string corresponding to that number. So 3.5 would return "3.5". The whole number part is converted to string, and concatenated with a decimal point and the remaining part of the number, which has also been converted to a string. The optional number arguments indicate the number of decimal places and the total number of places the string should take up. If the new string is not long enough, it is padded with 0's behind to get the appropriate number of decimal places, and with 0's in front to get the appropriate number of total places.

The rest of the functions handled by the string-specialist merely invoke the Lisp function by the same name. These are:

string-trim *list-of-characters string*	trims beginning and end of string
string-left-trim *list-of-characters string*	trims beginning of string
string-right-trim *list-of-characters string*	trims end of string
string-upcase *string*	replaces lowercase letters by uppercase
string-downcase *string*	replaces uppercase letters by lowercase
string-capitalize *string*	capitalizes the string appropriately

Note that the *:key* parts of these routines can be used, and the evaluation will be correct.

3.9.3 Known Problems and Possible Improvements

Named string constants are not currently handled, nor are string relations between such objects saved. It is not clear whether this would be a useful addition to the system or not, so until a case can be made for it, such improvements will have to wait.

No checking is done on the arguments of functions or predicates handled by simply invoking a Lisp

function. If there is an error, Lisp will catch it and bomb nastily. In addition, Lisp seems to make some assumptions if actual strings are not used there - a constant name may be used as a string, rather than its contents (which are not handled by the specialist anyway).

3.10 Meta Specialist

The meta-specialist is designed to handle certain "meta" level predicates (predicates which take predicates, operators, quantifiers, etc as arguments), many of which can be best handled procedurally. The specialist is still in its infancy and will probably undergo major revisions once its role is better defined and more "meta" level information is required by the system. Currently several predicates and a very few functions are handled. Those functions are predicates beginning with % are to be used only for meta-assertions, and in meaning postulate and simplification schemas.

The meta-specialist does maintain its own representation of some story facts, but is mainly used for literal evaluation. No literal comparison is done by this specialist.

Currently the meta-specialist handles literals of the form:

meta-entity	%action-pred	
	action-type	
	%formula	
	%action-formula	
	%action-pred	
	%type-predicate	
	%numeral-term	
	%episodic-term	
	%kind-of-episode	
	%kind-level	
	%object-level	
	%stative	
	%telic	
	%nonvolitional	
	%unlocated	
	%*-or-**	
	%atemporal-formula	
	%currently-I-cannot-infer-likely	
	%contains	meta-entity

3.10.1 Entry

If an formula input to the specialist has a single argument which is a symbol, the predicate will be added as a property of the argument. This is not too helpful when evaluating that literal again later (the literal itself can be used to verify it - this property will never be seen in that case), but the property can often be used by one of the procedural evaluation routines - *action-pred* , *action-type* , or *action-formula* .

3.10.2 Literal Evaluation

For single argument formulas where the argument is a symbol, the meta-specialist will first check the property list of the argument for the predicate (in case it was added using the above entry method). It will check a little deeper to determine whether the property holds for lambda predicates, for example.

Otherwise, the procedural routine matching the name of the predicate involved is used.

%Action-pred is recursively defined and operates on both predicates and formulas. When given a formula, it needs to decide whether the predicate in the formula is an action predicate. Action predicates are specified as such by asserting *(pred %action-pred)* . Lambda expressions and expressions with the operator **to** which contain action predicates are also considered action predicates.

```
%action-pred (item)
  if item is a predicate
    case predicate-type of item
      constant: is the property %action-pred on item 's property list?
      lambda: (%action-pred main formula of item )
      modified: (%action-pred first argument to operator in item )
  if item is a formula
    case formula type of item
      episodic: (%action-pred first argument of item )
      quantified: (%action-pred main clause of item )
      modified: (%action-pred first argument to the operator in item )
      prefix: (%action-pred predicate of item )
end of %action-pred
```

Action-type only evaluates to YES if it is given a term which is of the form *(To %action-pred)* . This is quite likely not the only type of object which this should evaluate to YES for, but for current testing needs, it has sufficed.

```
action-type (item)
  ; expect item to be a (To ...) term
  if item is not a symbol
    NO
  else case term type of item
    modified
      if (operator of item is to )
          and
        (action-pred first argument of item )
end of action-type
```

%Action-formula uses *%action-pred* to do the dirty work of determining whether a given formula is an action formula. Only *episodic* and *prefix* formulas are considered to be action formulas, and only if they contain an action predicate.

%Type-predicate simply uses a low level EPILOG routine to determine whether a given predicate is a type predicate or not. A predicate is considered to be a type predicate if it exists on a type hierarchy.

%Numeral-term simply checks to see if the term given is actually a number (not a named constant with a number sort).

%Episodic-term simply checks to see if the term given has sort *episode* or sort *e-term* (the meta equivalent of *episode*).

%-or-**** simply checks to see if the predicate given as an argument is either * or **.

%atemporal-formula checks to see if the formula given as an argument is an atemporal formula. This is determined using the following:

episodic wff - YES
causal wff - YES
logical wff - atemporal if all the arguments are
    for each argument to the logical predicate
        determine if it is atemporal
        if it is not, stop, the answer is NO for the whole wff
        if it cannot be determined, stop, the answer is UNKNOWN for the whole wff
    if all arguments were atemporal, the answer is YES for the whole wff
prefix wff
    if the predicate is *equal* - YES
    if the predicate is a type predicate - YES (temporary)
    if the predicate is a temporal or causal predicate - YES
    otherwise - NO
quantified wff - atemporal if the main clause is
quasi-quoted-expression wff - atemporal if the wff in property *quote-wff* is *modified*
wff - atemporal if the first argument to the operator is
otherwise - NO

%Contains checks to see if the first expression given contains the second expression at any level of embedding. This is accomplished by taking the expanded list forms of both expressions, and recursively comparing subexpressions of the first argument with the second argument.

Most of the other predicates use a routine called *eval-for-prop* which recursively determines whether or not the property is there. If the item is a constant, the property must be on that constant (a default is in place but the defaults have not been added), but if the item is more complex - a formula, lambda expression, etc, then it is looked at more deeply. Logical expressions must have all arguments with that property before the property holds for the whole thing.

%Currently-I-cannot-infer-likely does a small inference attempt on its first argument, using effort level 2 (no assumptions) and a maximum of 10 iterations. If the question is answered YES, the meta

specialist must return NO, but otherwise it returns YES. The inference attempt not tried if this is part of a meaning postulate or simplification schema, or the system is already answering a question (i.e. already in qa mode), or if the flag **ok-to-verify** is turned off, or the thing to "question" is a variable.

3.10.3 Function Evaluation

The function **%subst** takes 3 arguments - new, old, and the object to be substituted in, and returns the normalized new object which consists of object with all occurrences of old replaced by new. This is accomplished using the expanded list versions of all three arguments, and the Lisp function *subst* .

3.10.4 Known Problems and Possible Improvements

Obviously the meta-specialist currently handles only a very small subset of the possible "meta" level information. As the role and construction of meaning postulates becomes better defined, the need for this specialist will increase and it will be expanded then.

A better algorithm for determining when to invoke the question answering mechanism to evaluate *%currently-I-cannot-infer-likely* is needed. In addition, there should be some threshold of probability involved in evaluating how the question answer should affect the evaluation of that predicate.

3.11 Other Specialist

The other-specialist provides an easy interface between EPILOG and externally defined evaluation and storage routines. The specialist does not define any predicates or functions initially. When a user defines a function or predicate and specifies a package for it, it is added to the other-specialist.

When a literal is entered which involves a predicate this specialist is interested in, it checks the property list of the predicate for the name of an entry routine. If such a routine exists, it is invoked with the argument list of the literal. NOTE: Negations and modally embedded formulas are NOT sent to the external routine. If it is desirable to save negations or modally embedded facts as well, another specialist should be set up.

When a literal is being evaluated and it involves a predicate this specialist is interested in, it checks the property list of the predicate for the name of the package it exists in. If a routine by that name exists in that package, it is invoked with the argument list. If the literal being evaluated was negated, the negation is taken into consideration AFTER the external evaluation is complete. NOTE: modally embedded formulas are NOT sent to the external routine for evaluation.

When functions belonging to this specialist are set up, their location is stored with them and they are invoked directly, without going through the other-specialist.

Chapter 4

Response Generation

The response generation subsystem is by far the most ad-hoc part of the system, and is difficult to describe coherently. This section will concentrate on some of the more interesting, and higher level concerns of the response generator. Be warned that the response generator changes continually as heuristics are added and modified, and it is difficult to ensure that the code and manual are kept in sync.

A higher level description of the response generator tasks and heuristics is given in the user manual - that section should be read before attempting this.

4.1 Overview

The response generator works by taking wffs and translation information for the predicates and operators involved, and creating partial English fragments out of them. The fragments look similar to incomplete grammar trees. These fragments are combined, and any gaps are filled in using heuristics, and occasionally looking up something in the knowledge base. The fragments created are just symbols, with various properties corresponding to parts of speech and bits of information needed to be able to fill in the gaps. Although structures would be more efficient, property lists have the advantage of being more flexible - allowing the translation information to directly add properties, and also allowing continual minor changes to the system without major recompiling.

Some of the properties available on the fragments are:

name - the name of the EPILOG entity this fragment represents

frag-type - indicates what part of speech this fragment is supposed to represent. This includes *S* (sentence), *NP* (noun phrase), *VP* (verb phrase), *AP* (adjective phrase), *ADVP* (adverbial phrase), *RelC* (relative clause).

objtype - one of *constant* , *variable* - indicates property of the name so that number and determiner can be calculated.

head - the fragment which is the parent of this fragment. Pointers back "up" the tree are kept in case negations need to be moved around.

negated - indicates whether this fragment is negated

required - indicates whether this fragment is required. If there is no information on the fragment, and it is not required, it may be ignored. Otherwise default information will have to be made up for it.

This is useful on objects of verbs, which aren't always required to be said. For example, *John eats* is ok (so the second argument is not required), but *John throws* is not.

parts - the value of this property is a list of fragments. This is how conjunctions and disjunctions are handled. If a fragment has this property, it is a conjunction, and all the parts will be "said". The following types of fragments may be conjunctive: *NP, VP, AP, ADVP, ADVS, PP, S* .

conj - *and* , *or* , *but* , etc - this property indicates how the *parts* are conjoined.

pre - this property is used for several kinds of fragments. For a conjunctive fragment, it indicates what should come before any of the *parts* and the *conj* (e.g. to say *if ... then ...* , the *pre* would be *if* , and *conj then*). It may also occur when an *NP* has an *S* beneath, or in a *PP* or *ADVP* . It may either be a single symbol, or a list of symbols.

number - indicates whether this fragment is supposed to represent something *singular* or *plural* . This is then used to make the *NP* and *VP* numbers match, as well as deciding which lexical forms to use.

adv - this property holds an adverb as value. It can occur in almost any of the types of fragments. It may be either a single symbol or a list of them.

NP - the value of this property is another fragment, which sits in the noun phrase position of this fragment. Note that the NP can actually be an S. Because episodes may be said as either sentences or as simple nouns, using NP in translations involving them allows the use of either, whichever there is more information for. This property is required for fragments of type *S* and *PP* , and may also be on *ADVP* fragments.

S - the value of this property is another fragment which is a sentence. This may occur in a noun phrase or adverbial phrase.

noun - the noun for an *NP*

proper-noun - a name (e.g. Little Red Riding Hood) Used instead of the noun if present.

pronoun - (optional) pronoun for an *NP* - used instead of the noun and other info if present

determiner - the determiner for the *NP* . This is only calculated if a noun is present.

AP - the value of this property is another fragment, which is a adjective phrase. This is "said" regardless of whether the noun information is a noun or a proper noun, but is not said if a pronoun is present.

PP - the value of this property is another fragment, which is a prepositional phrase.

RelC - the value of this property is another fragment, which is a relative clause. It has the same properties as a *VP* on it.

VP - the value of this property is another fragment, which is the verb phrase for this

verb - the verb for the verb phrase

particles - any particles for the verb (e.g. *up* , in *wake up*).

objects - this property contains a list of fragments which are objects of the verb. These may be of the type *NP* , *AP* , or *PP* .

aspect - one of *infinitive* , *passive* , or *prespart* . This is used to determine the lexical form to use for *verb* . If there is no aspect then the tense is used to determine the lexical form.

tense - one of *past* or *present* . If a formula has an episode attached, the tense is assumed to be *past* ; otherwise *present* . This is used to determine which lexical form of *verb to use.*

ADVP - this property contains another fragment which is an adverbial phrase. The adverbial phrase contains the *pre* , and *prep* properties, as well as either an *NP* or an *S* property. This occurs on *S* fragments, although it sometimes appears on *NP* fragments (where the NP is actually standing in for an S).

ADVS - this property contains another fragment which is a sentential adverb, and may be either a list of adverbs (property *adv*), or a noun phrase or sentence (*NP* or *S* properties). This occurs on *S* fragments.

prep - this property contains a preposition, and is used for *PP* and *ADVP* fragments. If there is no preposition on such a fragment, the fragment is not said. Thus sometimes the system will erase the *prep* if there is no information for the *NP* accompanying it.

adj - this property contains an adjective (or a list of them), and is found on *AP* fragments.

Sometimes these fragment properties occur on fragments other than the ones expected, and are moved to the appropriate place by the manipulation routines. This makes the translation information easier to code, and prevents missing things when a number of combinations are done.

4.2 Filtration

The filtration phase removes formulas from the list of formulas to say. These include "set-of-support" formulas - those which were temporarily created during a question answering process, formulas with a high likelihood of being already known (and therefore not of interest to the user), and those which are quite complex and may make the system waste too much time trying to say them. The resulting list of formulas will be ordered by likelihood of being known.

The set-of-support formulas are distinguished by a *proof* or *disproof* property.

A formula's likelihood of being known is calculated first, and then compared against a threshold (***filter-threshold***). Although this was quite helpful in ECoNet, tests with EPILOG have shown that often the formulas removed because of high likelihoods are required anyway, as they are types, etc that indicate how to say particular individuals. To calculate the likelihood of being known, the following is used:

Axiom schemas: likelihood is 100

General knowledge (involving all variables): likelihood is 80

Type predication: likelihood is 90, unless the type is a sort - then 101

Other fact involving only constants: likelihood is 40

Mixed variables and constants (i.e. rule about an individual): likelihood is 50

A formula's complexity is also compared against a threshold (***max-response-complexity***), and if it is higher, the formula is optionally ignored. Another flag ***prompt-if-too-complex*** indicates whether or not the user should be prompted to see if he/she really wants that formula ignored. For highly complex formulas, usually meaning postulates will be applied to break it down into more reasonable chunks anyway, so nothing important is missed if such formulas are ignored by the response generator.

4.3 Organization

Formulas are divided into rules (conditionals) and facts. The facts are all gathered together and the system attempts to say them as one sentence. Each conditional is said as its own sentence.

Episodic type literals (e.g. *((lrrh girl) ** ep1)*) are difficult for the system to handle. They are removed at this stage, along with anything which refers to the episode. Later, if the type is needed, it can be retrieved, but will be prepared without the episode part.

4.4 Translation

The translation process takes a list of formulas, and translation lists for the predicates and operators involved in the formulas, creates fragments for each formula, combines them, and manipulates them and fills in any gaps to try to make "natural" sounding output.

4.4.1 Fragment Creation

A recursive set of routines is used to create a fragment for each wff, using the list(s) of translation information available for the predicates and operators involved in the formula. If no translation information exists, the user will be prompted (if *default-trans* is nil), or the system will "guess" what might be appropriate. The user is warned in the latter case.

Formulas (wffs) are translated using the following *(wff-trans wff &optional ante)* : Note: if *ante* is non-nil, then this is a recursive call to *wff-trans* . The result of translating *wff* is combined with *ante* using the routine for combining conditionals.

prefix, causal wff - translates the predicate with the fragments resulting from translating the arguments (using translation information from the predicate).

episodic wff - makes a sentence fragment with the first argument, and sets its name to be the episode. If the episode is a constant, the tense of the sentence is set to be *past* (this will later get moved to the verb phrase part of the sentence fragment).

modified wff - translates the operator with the fragments resulting from translating the arguments.

quantified wff - For a conditional formula (any quantifier except E, WH, E!, and The), fragments are determined for the restriction, as well as a "mini-fragment" saving the quantifier information. The quantifier information fragment is combined with the restriction fragments. The *wff-trans* routine is then called to determine the fragment for themain clause, sending along with any previous *ante* fragments as well as the new restriction fragments. For a non conditional quantified formula, fragments are first determined for the restriction, if there is one. Then the fragments for the main clause is determined, and combined with the ones for the restriction.

logical wff - For a conditional formula (predicate is a number, *implies* , or <=>), fragments for the first and second arguments are determined, and combined using the routine for combining conditionals. For non-conditional formulas, i.e. conjunctions and disjunctions, the fragments for all parts are determined, and then combined appropriately according to the predicate (*and* or *or*).

quasi-quoted-expression wff - (axiom schemas only) - fragments are determined for the wff under the property *quote-wff* .

variable wff - (axiom schemas only). A sentence fragment is created that says "something happens".

otherwise - the formula is treated as a simple constant and its name is returned (rather than a fragment).

The routine for combining conditional fragments first massages the fragments (adding determiner "only" if appropriate, passivizing to make combinations easier), and then tries to combine them. If they won't combine nicely, it makes them into an *if...then...* sentence fragment.

Predicate translation is handled as follows *(pred-trans pred &optional neg args fromop)* :

constant, variable predicate - (variable will occur only with axiom schemas) - gets translation info (from predicate, or default), and uses the translation combination routines

modified predicate - uses operator translation, combining given arguments (*args*) and operator arguments.

lambda predicate - The predicate is first translated as if it were a formula (using *wff-trans*). If an operator is acting above this predicate (*fromop*), then the important piece is extraced from the resulting formula (usually the VP). The name of the fragment is set to be the last argument.

Operator translation is handled as follows *(op-trans op &optional neg args)* :

constant, variable operator - (variable will occur only with axiom schemas) - gets translation info (from operator, or default), and uses the translation combination routines

modified operator - uses operator translation, combining given arguments (*args*) and operator arguments.

Term translation is handled as follows *(term-trans term)* :

constant, variable term - the term's name (not a fragment)

modified, function term - uses operator translation with operator arguments

wff term - translated like any other wff

quasi-quoted-expression - if there is a *quote-wff* property on the term, does wff translation on *quote-wff* property, otherwise just returns the name of the term.

Default translations for predicates and operators are calculated as follows:

- If the predicate or operator is a variable, a translation list is calculated based on the sort (this should only happen during axiom schemas)

  pred - the translation is a verb phrase for "do or be something"

  operator - the translation is an adjective phrase for "some operator"

- If the predicate is a compound predicate, the translation for the second part is calculated, and the first part is added as an adverb to that. (e.g. *wolf-head*)

- If the predicate is a type predicate, the translation is a noun phrase with the predicate as noun.

- If this is a predicate with more than 1 argument, the translation is a sentence, with a verb phrase where the verb is the predicate, and all the arguments are its objects.
- If this is a predicate with only 1 argument, if the predicate indicates one of the topics on the *active-topics* list, it is treated as a verb phrase, otherwise the translation is an adjective phrase with the predicate as an adjective. For example, *(john walk)* would be said as *John walks* , while *(john tall)* would be said as *John is tall* .
- If this is an operator, the translation information says to add the operator as adverb to whatever the argument of the operator produces.

Once a default has been determined for a predicate or operator, it is stored so that future formulas using the predicate or operator don't need to recalculate the translation default.

4.4.1.1 Combining Translations and Fragments

Translation information consists of a list of properties and values. Some of the values are numbers, which refer to particular arguments. For operators, note that the arguments are combined so that the lowest level is last - which makes the numbering a bit strange on the translations for some operators. Arguments may be names or fragments. To get the desired fragment, one must take the translation information, and apply it to the argument fragments, resulting in either a new fragment, with the argument fragments as a property somewhere within it, or a modified version of the original argument fragment. Two mutually recursive combination routines are used to combine the translation information with the argument fragments.

combine-trans takes a translation list, and a list of argument fragments, and returns the resulting fragment. The translation list determines what happens here.

If the translation list is:

a number - return the argument corresponding to that. Some types of arguments are made into arguments if they weren't already (functions, records, etc).

an atom - return the trans itself (it is a property value)

a list with first element a number - take the argument fragment corresponding to that number, and add properties to it from the rest of the translation list

a list with first element a fragment type
if next element is a number take that argument fragment and coerce it into a fragment using the first element as type else just create a fragment of that type Also add properties as specified by the rest of the translation list (using *combine-properties*)

a list with first element a fragment group type (parts or objects) - recursively call this routine to create a list of the desired fragments

a list with first element a fragment routine - apply the routine to the arguments otherwise recursively call this routine with the rest of the translation info

combine-properties takes a fragment, a translation list, and a list of argument fragments and uses the translations to update properties on the given fragment.

If the property is:

a fragment type or fragment group - calculate new value for it with combine-trans. If there was already a value, make this a conjunction.

name - use combine-trans to calculate the actual name (in case of #). If a fragment is returned (instead of just a symbol), coerce it to be an NP, coerce the given fragment to be a VP, and make them into a sentence Otherwise just add the name as the value of the name property.

otherwise this is just a regular property - use combine trans to calculate the property, and then add it as the appropriate property value

4.4.2 Fragment Combination

Fragment combination is probably the hardest part of the translation process. Here several fragments, containing partial information for a sentence, are combined in such a way that some fragments fill in the gaps in the other fragments. Most fragments are combined conjunctively - i.e. they are all true at the same time, and may be combined nicely, or with *and* . *Conjunct-frags* does this. Fragments for disjunctions are combined using *disjunct-frags* , although no real combination is done here - they will simply be made into a conjunction using *or* (it is difficult enough to determine how to combine the fragments when they are "anded"!). If negations are present, the conjunction itself may change from *and* to *or* , or vice versa, or to *but* .

The fragments are combined two at a time. The decision process looks at what types of fragments are involved, and their names, and may recursively decscend a fragment looking for a place to attach the other one. Conjunctions are flattened as much as possible, so that *sharp, white, and big* would be said rather than *sharp and white and big* .

Conjunct-frags (*conjunct2* combines two at a time)
if fragments have same name
if fragments have same type
if type is *AP, PP,* or *RelC* - make a conjunction with *and*
if not directly combinable
if first is *NP* , try to add second as *RelC* beneath
otherwise copy properties from first fragment to second fragment
if one is *NP* , other *S*
if *VP* on *S* , info on *NP* , but not noun *event* or *fact*
make *S* into *NP* , *NP* into VP, combine into a new *S*
if quantifier info on *NP* but no noun, or noun is a sort
move aspect from *NP* to *S* , return *S*
if not directly combinable - can't combine
otherwise
if first fragment can be above second, try to add second there
if second fragment can be above first, try to add first there
else try to find common parent type, and add both under it
if fragments are negated and have same type - cannot combine
otherwise
look to see if first fragment can be added at a lower level of second, by recursively
calling *conjunct2*

When properties are copied from one fragment to another, if such properties already existed on that fragment, conjunctions are made for them where appropriate.

Two fragments (whose names and possibly also types are equal) are not directly combinable if:

- one is negated and the other isn't, and there is real information on both of them (*PP, AP, VP, RelC* , noun, proper noun, pronoun, etc)

- either is a *VP*
- for *NP* 's, both have nouns that are not equal
- for *S* 's, both have an *NP* and a *VP*

4.4.3 Fragment Manipulation

After a fragment has been created and combined with others, but before gaps are filled in, the fragment is checked to see if it is a sentence fragment. If not, the response generator tries to make it into one. This is very messy and probably could be simplified greatly now - but heuristics have been added gradually on an "as needed" basis.

Creating a sentence given a fragment which is:

S - if there is an adverbial phrase on this sentence, but no verb phrase, it is changed into a sentence where the sentence is a noun phrase, with verb *be* and object a prepositional phrase made from the adverbial. This handles temporal relations where no other information is available about an episode (e.g. *E1 was before E2*). Otherwise this is already a sentence, and no manipulation is needed to make it into one.

ADVP or *ADVS* - the fragment is an adverbial phrase or sentential adverb. Here the sentence is made by building a sentence of the appropriate name, with the *ADVP* or *ADVS* . The gap filling routines will ensure that the information needed to say that sentence is looked up.

VP - a sentence fragment is made by building an *NP* with the name on the *VP* , and combining the two into a sentence. The missing noun information will be looked up during gap filling.

NP with *S* beneath - just use the *S* .

NP which has already been prepared (record, quoted expression, etc) - just change the type of the fragment into *S* and use as is.

NP which has no information on it - just change the type of fragment into *S* and use as is (there is no information so nothing will print though).

NP with a relative clause - separate the relative clause out and use it as the verb phrase for the sentence.

the fragment's name has a translation (name) associated with it - use that name to be the proper noun of a new *NP* , and make the fragment itself into a verb phrase with the verb *be* .

NP with one *PP* beneath it of the form *of something* (where the *something* is not *group* or *member*) - the *PP* is separated out and made into a verb phrase *VP* with verb *have* , so that the resulting sentence is *someone has something* .

NP with *PP* beneath with preposition *of* or *for* - the *NP* without its *PP* becomes the *NP* for the sentence, and the *PP* is made into a verb phrase.

NP with multiple *PP* 's beneath - make a sentence fragment with *NP* the original *NP* with only its first *PP* , and verb phrase with verb *be* and objects the rest of the *PP* 's.

NP for a constant which is not a skolem constant - the name of the fragment is used to create a *NP* , and the given *NP* is made into a verb phrase

if the fragment is a conjunction - each part of the conjucntion is made into an *S* , and the results made into a sentence conjunction.

otherwise - an NP is made with the name of the fragment, and the fragment itself is made into a *VP* for the sentence.

To make a fragment into a *VP* , the verb *be* is used, and the fragment becomes the object of the verb.

Next the *S* fragment is recursively examined to make sure that any tense and aspect information located in the fragment gets moved to the appropriate part of the fragment (the *VP*), and negations get moved up as high as possible. Passivization is done if necessary to sentence conjunctions so that where possible they all have the same subject.

NP 's of the form *something of someone* are changed to the possessive *someone's something* where possible (i.e. where the noun is not one of *group, member, start, end, instance, duration,* or *cardinality*). The *someone* part of the *NP* is used to give the possessive determiner, and then the entire *PP* can be removed. If such a noun phrase has not combined with a verb anywhere, it will probably be said as *someone has something* , rather than using the possessive form.

4.4.4 Filling in Gaps

Special routines for filling in each type of fragment exist, as well as general routines to fill fragments within a fragment. The system keeps track of entities it has encountered so far during this response and information about them, so that this information can be copied to other instances of that entity within the sentence. The final stage will worry about what has already been said. Number and determiners for noun phrases are calculated here.

In deciding whether the number for a noun phrase should be singular or plural, constants are always singular, and the first variable decides the number of all the rest of the variables in the sentence. Sometimes the number may be included in the translation list that the fragment was initially created from, in which case we don't need to calculate it.

decide-number for an *NP*
- if number already decided (from initial translation), use that
- if "and" conjunction - *plural*
- variable
  - if "universal" number already set (i.e. this is not the first variable in this sentence), use that
    - if quantifier is E, or noun is one of *person* , *human* , or *thing* - *singular*
    - otherwise - *plural*
    - Also set "universal" number to this number
  - constant - *singular*
- if a negation is involved, reverse number unless *person* , *human* , or *thing*
- if mass noun, set number to *singular* so verb will sound ok (noun and verb must agree)
- *PP* 's with preposition *of* below this *NP* must have their number set the same

In deciding what determiner is appropriate, several things are taken into account, including whether the noun is a mass noun, if this is a constant and if so if it is indefinite or unique, the number, whether the *NP* is negated, or if there is a negation elseswhere in the sentence, if this is the first variable or another, and whether this *NP* is the same as another one with a different name. This is quite messy and probably could be simplified greatly now - but heuristics have been added gradually on an "as needed" basis. Once the determiner has been calculated, any *PP* 's with preposition *of* beneath this *NP* have their determiners calculated immediately.

decide-determiner for an *NP* which has a noun on it
- if a mass noun - determiner is *some*
- if a constant or function
  - if number is *singular*
    - if noun is one of *thing* , *human* , or *person* and there is no *AP*
      - determiner is *some*
    - if not unique, and either indefinite or the object of a "be"
      - determiner is *a*

otherwise determiner is *the*
if this *NP* is negated
if there is quantifier information on the fragment
if quantifier is A or E, - determiner is *no*
otherwise - determiner is a combination of "not" and determiner for that quantifier
otherwise - determiner is *no*
if there is a negation somewhere else in the sentence, and this *NP* is not within a *PP*
if *plural*
if this variable occurs in a positive context - determiner is *all*
otherwise - determiner is *any*
if *singular*
if this variable occurs in a positive context - determiner is *every*
if noun is one of *person* , *human* , or *thing* - determiner is *any*
otherwise determiner is *a*
if this is the first variable in the sentence
if *plural* - determiner depends on quantifier
if there is a relative clause beneath - determiner is *any*
if noun is one of *person* , *human* , or *thing* and there is no *AP*
if this variable doesn't occur in a positive context - determiner is *every*
otherwise - determiner is *some*
otherwise - determiner is *a*
if there are other *NP* 's with same noun, *AP* , etc
if number is *plural*
if the other *NP* has determiner *other* - determiner is *even more*
otherwise - determiner is *other*
if number is singular
if the other *NP* has determiner *another* - determiner is *yet another*
otherwise - determiner is *another*
if noun is one of *thing* , *person* , or *human* , there is no *AP* and it is *singular*
- determiner is *some*
if number is *plural* - determiner is blank
otherwise - determiner is *a*

4.5 Verbalization

This phase of the response generation takes the filled in fragments and actually determines the words to use to say them. To sound more natural, pronouns and shorter phrases are used for referring phrases. Lexical entries for noun and verb forms are defaulted where unknown.

4.5.1 Referring Phrases

If a particular fragment's name is on the "said" list, it has already been said and the response generator will try to determiner a shorter form of the phrase to say. This depends on whether or not the item was said in the current sentence or a previous one, if it is the subject, and how many other items that have been said already could be confused with this one. A pronoun is preferred if this won't get confusing, but if it will, a shorter version of the original phrase will be said (omitting adjectives and prepositional phrases, unless this will lead to confusion with other previous phrases).

decide-pronoun for *S* or *NP*
proper noun - use pronoun in lexicon for name if one exists
plural or group noun - if subject *they* ; object, *them*
if noun is *masculine* (lexical property) - if subject *he* ; object, *him*
if noun is *feminine* (lexical property) - if subject *she* ; object, *her*
otherwise - *it*

When a referring phrase cannot use a pronoun either because one doesn't apply, or it would be ambiguous as to which phrase it referred, the following is used.

decide-short given *argfrag* (current) and *samefrag* (previous)
if samefrag is an *S*
if not in current sentence (i.e. previous sentence)
if current fragment is *NP* that is not beneath an *ADVP* , or aspect is *prespart*
copy the *samefrag* info to *argfrag* , and say the whole thing again (with aspect *prespart*)
otherwise say the whole fragment
otherwise ignore the fragment
if *samefrag* represents a name (proper noun) - use the name
if *samefrag* is a conjunction - use *decide-short* for the first part
if *samefrag* has a *group* noun, and a *PP* beneath with preposition *of*
- use *decide-short* for the *NP* beneath the *PP*
if *samefrag* has a *member* noun, and a *PP* beneath with preposition *of*
- use *decide-short* on just the *argfrag* , ignoring the *samefrag*
if there is a noun on samefrag
- use the noun, possibly the *AP* , and a determiner calculated as follows:
if *samefrag* determiner is *another* - if subject use *the other* ; object, *that*
if *samefrag* determiner is *yet another* - if subject use *the last* ; object, *that last*
if *samefrag* determiner is *other* - use *the other*
if *samefrag* determiner is *even more* - use *those last*
otherwise - use *the*
otherwise - print the whole fragment

4.5.2 Lexical Information

One of the final steps is to look up the actual lexical form to use for a noun or a verb, depending on number, noun or verb, and aspect or tense (for verbs). This information may be stored already (given by *add-lex* commands earlier), prompted for, or may be "guessed" at by the system. In most cases the "guessing" is accurate enough that the meaning of the sentence comes through, even if the exact spelling isn't correct. Irregular verbs don't sound quite so natural, but aren't that numerous, so it is easy to keep most of them in the lexicon.

Lexical defaults are as follows:

Noun

plural - adds ending -s to the noun; for a mass noun, the singular form is used
singular - the noun itself

Verb

infinitive - "to" followed by the verb
prespart - adds ending -ing to verb
passive - adds ending -ed to verb
otherwise - uses number and tense and negation to determine
negative - chooses appropriate form of "do", use with "not" and the verb
positive - depends on tense
past - adds ending -ed to verb
present - depends on number
plural - uses the verb itself
singular - adds ending -s to verb

When adding an -s ending,

if the word ends in "y" and does not have a vowel immediately before the "y", the "y" is changed to "i" and "es" is added instead of just "s"

if the word ends in h, s, x, or z, "es" is added instead of just "s"

otherwise "s" is added to the end of the word

When adding an -ed or -ing ending

if the word ends in "e", the "e" is removed first, and then the ending added

if the word ends in "y" and does not have a vowel immediately before the "y", and the ending is -ed, the "y" is replaced by "i" before the ending is added

otherwise the ending is added to the end of the word

Note that lexical defaults are not stored.

4.5.3 Postprocessing

The final stage is responsible for combining words that should be said as one, capitalizing the first word in a sentence, and changing "a" to "an" before a noun. Words that are combined are *some, any, every,* and *no* with nouns *thing* , *human* , or *person* . These become *something/someone, anything/anyone, everything/everyone,* and *nothing/nobody* . This is all done by examining each consecutive pair of words in the sentence - not particularly efficient, but compared to the rest of the calculations, still quite fast.

4.6 Known Problems and Possible Improvements

The heuristics used are incomplete and continually being updated. They often conflict, making it difficult for the system to decide how to say something. Many parts of the system have been through so many iterations of "fixing" that they are quite messy and difficult to understand - these should be reviewed with the hope of making them more consise and easier to understand and maintain. Some of this has been done already, but more is needed.

There are probably more ways to combine the fragments than are currently being considered, so the system is more verbose than it really needs to be.

It is possible that to better handle operators, their translation could return another translation list, rather than attempt to return a whole fragment. This could make it easier to create the translation lists for operators as well, but would require some significant changes to the translation combination routines.

Axiom schemas can be "said" without the system exploding, but they do not sound very natural at all. Part of the problem is that such information is difficult to say, even for a human, but some improvements could be made to make it better.

To better handle the wide variety of things which can be represented in EPILOG, rules for combining and manipulating fragments would be preferable to hard-coding them into the program. The more information that can be coded into the rules and translation information, the easier it will be to fine-tune the system to "say" things more naturally. Of course then a new "inference" mechanism would be required for these rules.