Rochester Institute of Technology
School of Computer Science and Technology

Enhancements to The Frame Virtual Machine

by

Archna Bhandari

A thesis, submitted to
The faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by : ______________________________
Professor John A. Biles

____
Professor Kevin Donaghy

____
Professor Peter G. Anderson

October, 1989

Title of thesis **Enhancements to the Frame Virtual Machine.**

I Archna Bhandari hereby **grant permission** to the Wallance Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 11/13/89

ABSTRACT

The Enhanced Frame Package is a tool to build Expert Systems. It is a frame based system, that initially was developed in C-Prolog by LaMora S. Hiss at Rochester Institute of Technology in 1987 for her master's thesis. It was enhanced in the current thsis to provide much larger expressive power andgreater ease of use. Several operators were modified/enhanced, and several new operators were added, while providing the user a balance of computational tractability, expressive power and consistency.

Major concepts provided in the Enhanced Frame Package include - local consistency checking as opposed to global consistency checking and how the user can have the best of both options; the flexibility of loading a knowledge base file as a consistent system or as an inconsistent system; operations that work on working memory and operations that work on the original file in the working directory; the concept of a knowledge analyzer; the way one sees the human mind, knowledge and learning and its parallel in knowledge representation and the surrounding issues of consistency, expressive power and computational tractability.

TABLE OF CONTENTS

1 INTRODUCTION

'A frame virtual machine in C-Prolog' was first developed by LaMora S. Hiss at Rochester Institute of Technology in 1987 as her Master's Thesis work. A perusal of this work revealed that several enhancements could be made to the frame virtual machine. Since then it has been enhanced by the author as her Master's Thesis work. Throughout this document, the enhanced frame virtual machine in C-Prolog will be referred to as the frame package or the enhanced frame package or the package.

This document assumes that the user is familiar with the concept of frames and the Prolog language. The package implementation is in C-Prolog. The following will briefly describe the topics covered in the thesis.

Chapter 2, BACKGROUND outlines the basic information about the frame concept in general and specifically discusses the salient features of the frame virtual machine developed by LaMora S. Hiss.

Chapter 3, SUGGESTED ENHANCEMENTS TO THE FRAME VIRTUAL MACHINE discusses the major results of the critical examination of the frame virtual machine and presents conceptual development, issues and proposed enhancements.

Chapter 4, KNOWLEDGE REPRESENTATION ISSUES talks about some fundamental issues such as consistency, computational tractability and expressive power related to knowledge representation systems. These issues surfaced during critical examination of the frame virtual machine and while developing conceptual thoughts for enhancements. It also discusses the techniques used in this package to deal with these issues.

Chapter 5 IMPLEMENTATION CONCEPTS describes the concepts that were developed for the enhanced frame package keeping in mind the knowledge representation issues, the proposed enhancements and the basic framework of the earlier version.

Chapter 6 FRAME PACKAGE OPERATORS defines each individual operator in terms of name, syntax, purpose and an illustrative example. Functionality of the operators explaining all the major logical steps that are followed when an operation is performed has been described in Chapter 7 FUNCTIONAL DETAILS.

The conclusions have been included in Chapter 8 CONCLUSIONS.

Four appendices have been included that cover - excerpts from [HISS87a] about other frame packages, keywords and definitions, an example knowledge base file and an interactive example session.

For the purpose of just using the package, a self-sufficient USER MANUAL and its companion a TECHNICAL MANUAL also have been prepared. This way, a user does not have to page through the details of this thesis.

The source code for the Enhanced Frame Package has been included separately in the document "ENHANCED FRAME PACKAGE - SOURCE CODE."

2 BACKGROUND

2.1 THE FRAME CONCEPT

In 1974, Marvin Minsky proposed a theory of "frames" as a mechanism for representing knowledge in a computer. Minsky mentioned in his paper that the theories both in artificial intelligence and in psychology have been on the whole too minute, local and unstructured to account, either practically or phenomenologically, for the effectiveness of common-sense thought. The "chunks" of reasoning, language, memory, and perception ought to be larger and more structured, and their factual and procedural contents must be more intimately connected in order to explain the apparent power and speed of mental activities [MINS85,246].

According to Minsky, a frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed.

Minsky perceived frames as a good model of human memory. He even went so far as to propose a frame-based explanation of Freud's finding of early cognitive structures in free-association thinking [MINS85,248]. He argued that researchers must model human mind design and suggested that logical reasoning is not flexible enough to serve as a basis for thinking[MINS85,262].

Minsky's first paper on frames was published in 1975 and has evoked a great deal of discussion and interest in exploring further levels of detail. It presents plausible and provocative examples of the applications of frames to different problems in artificial intelligence [KUIP75,152]. Since then, a lot of researchers have begun to distinguish the various theoretical and technical issues in discussions of frames, and many variations of the frame concept have been formulated.

Fikes and Kehler discuss the advantages of frame languages in their paper:

"Frames capture the way experts typically think about much of their knowledge, provide a concise structural representation of useful relations, and support a concise definition-by-specialization technique that is easy for most domain experts to use. In addition, special purpose deduction algorithms have been developed that exploit the structural characteristics of frames to rapidly perform a set of inferences commonly needed in knowledge system applications.

In addition to encoding and storing beliefs about a problem domain, a representation facility typically performs a set of inferences that extends the explicitly held set of beliefs to a larger, virtual set of beliefs. Thus, the representation facility participates in the system's reasoning activities by providing these "automatic" inferences as part of each assertion and retrieval operation. Frame languages are particularly powerful in this regard because the taxonomic relationships among frames enable descriptive information to be shared among multiple frames (via inheritance) and because the internal structure of frames enables semantic integrity constraints to be automatically maintained." [FIKE85,904-905]

During the process of acquiring knowledge for this thesis and while investigating current advancements in the area of frame based knowledge representation systems; the following frame packages were studied:

- FRL [ROSE79]
- KRL [BOBR85]
- KRYPTON [BRAC85][PIGM84]
- UNIT PACKAGE [STEF79]
- KANDOR [PATE84]
- KEE [FIKE85]
- FRAME VIRTUAL MACHINE [HISS87a]

Since the thesis relates to the enhancements of the frame virtual machine, the domain of the frame package discussions here will be limited to the frame virtual machine. A comprehensive description of other frame packages has been included in the Appendix A.

2.2 THE FRAME VIRTUAL MACHINE (Earlier Version)

The Frame Virtual Machine was developed at the Rochester Institute of Technology in 1987 by LaMora S. Hiss using C-Prolog [HISS87a]. The basic conceptual unit of the Frame Virtual Machine is a frame, which is implemented as an abstract data type.

A frame is composed of a unique name and a list of any number of slots representing knowledge about a frame. A slot is composed of a name and a list of one or more facets. Facets represent descriptive knowledge, procedural knowledge and constraints for a slot's value. There is a predefined set of possible facets : value, type, max, min, default, if_added, if_needed and if_removed.

The value, type, max, min and default facets store declarative knowledge about a slot's value. A value facet holds the value for a given slot. The value facet always contains a list, and the cardinality determines how many values may be in that list. The cardinality of a slot is constrained by the min and max facets. The type facet serves as a constraint on the possible types of values that can appear in the value facet. The default facet provides a list of values that is to be used for the slot's value if no other values are present.

The remaining three facets; if_needed, if_added and if_removed store procedural knowledge in the form of calls to prolog predicates or demons. Conceptually, a demon is a predicate that will be invoked under the appropriate circumstances. These circumstances differ with each of the three facets. A demon in the if_needed facet will be invoked when a value is requested for a given slot but none is present in the slot. A demon in the if_added or if_removed facet will be invoked when a value is added to a slot or removed from a slot, respectively.

Frames in the Frame Virtual Machine exist in hierarchies established by AKO (a_kind_of) links. These links are established by the system defined slots is_a and instance_of. These two links distinguish between generic frames and frames that are instantiations of those generic frames. An is_a link represents a subsumptive relationship between two classes, whereas instance_of represents a subsumptive relationship between a class and an individual member of that class.

The hierarchical structuring of frames allows for inheritance of both descriptive and procedural knowledge. Inheritance of information from a frame's ancestor, however, is always overridden by specific information in that frame. Inheritance in the Frame Virtual Machine does not occur at the time of frame definition but rather when a value is needed. At that time, if no value is found in a given slot, the frame hierarchy will be searched recursively in a breadth-first manner until the slot is found in an ancestor frame or until there is nowhere else to look. This ensures that the most current values will be inherited and that a local value will always override an inherited value.

Frames can be created in a knowledge base using the def_frame operator. There is one global slot defining operator also, def_slot, which allows for specifying the slot name and facets within the slot. The slot defined by def_slot can be incorporated in any frame simply by creating a slot with the defined name, without having to specify the facets again.

Three slot operators are defined to access the value facet of a given slot for a given frame, once a frame is created in the knowledge base:

add_value_to_slot(FrameName, SlotName, Value)
remove_value_from_slot(FrameName, SlotName, Value)
return_value_from_slot(FrameName, SlotName, Value)

where:
FrameName is the name of the target frame
SlotName is the name of the target slot and
Value is the value to be added/removed/returned.

The add_value_to_slot operator allows for the addition of a value to a specified slot in a specified frame. Similarly, remove_value_from_slot operator allows the removal of a value from a specified slot in a specified frame and return_value_from_slot retrieves the value of a specified slot from a specified frame.

The Frame Virtual Machine provides two frame match operators:
frame_match_exact(Prototype_List, List_of_Frames)
frame_match_subset(Prototype_List, List_of_Frames)

The Prototype_List is defined as a list of slots, including values and other facets, in a frame. The Prototype_List determines what is to be searched for, and List_of_Frames is the list of frames that matches the Prototype_List. The only difference between these two frame match operators is the value specification interpretation. For the frame_match_exact operator, the list of values in the value facet of the frame's slot must be identical to those in the Prototype_List, while frame_match_subset will match a frame whose list of values in the value facet of a slot is a superset of the list of values in the value facet of the prototype's list.

Besides the three slot operators and two frame match operators, the frame virtual machine provides a frame_subsumes operator to determine the relationship among frames in a hierarchy:

```
frame_subsumes(Subsuming_Frame,Subsumed_Frame)
```

Where Subsuming_Frame is the potential ancestor frame and Subsumed_Frame is the potential progeny frame. The frame_subsumes operator succeeds if the first argument subsumes the second argument. The subsumptive relationship is based on the AKO links.

Also a utility operator print_frame(framename) is provided that allows an organized printing of the specified frame framename.

3 PROPOSED ENHANCEMENTS TO THE FRAME VIRTUAL MACHINE

3.1 A START

An assignment was due 9 days from the first day of the class in ICSS782, Knowledge Based Systems, at RIT in Fall, 1987. A copy of LaMora Hiss's Thesis [HISS87a] and the User's manual [HISS87b] was distributed to every one in the class. The objective was to understand the package, write a small knowledge base, test and critique the frame virtual machine and its documentation. That is how this thesis got started.

The author of this thesis undertook the task of doing the same thing in greater detail and in fact go beyond that and make some enhancements to the package as a final class project during the last four weeks of the quarter.

Bits and pieces of information from the critique of all the students in the class were put together and more extensive tests were performed on the complete range of features of the frame package. A final critique was then prepared which not only organized and included all the relevant points from the set of critiques but contained several additional ones that were discovered during the test. It was quite interesting to create a comprehensive critique and more interestingly a byproduct - critique of critiques, that emerged inadvertently.

Development of additional interest later resulted into a thesis proposal on enhancements to the frame virtual machine in May, 1988. During implementation, the scope of the thesis work widened as more thinking was done and more time was spent experimenting with the package. As a result, several enhancements in addition to the proposed ones, were made.

3.2 DISCUSSION

Critical examination of the frame virtual machine (the earlier version) in a nutshell showed that it rested on a good theoretical foundation but could be enhanced and revised. There were some bugs that existed, a lot of areas were identified where enhancements could be made and several new features could be added. Also, the user manual needed major improvements and a complete redesign.

A lot of time was invested in developing a conceptual framework to accomplish the aforementioned tasks. This involved:

- study of the frame virtual machine - code and concepts.
- planning enhancements without disturbing the underlying concepts of the frame virtual machine.
- keeping knowledge representation issues in mind (covered in Chapter 4).
- literature search (which only turned out to be of a little additional help).
- developing additional concepts (covered in Chapter 5) to accomplish enhancements as well inclusion of new features.

The major enhancements have been divided into four parts:

- Primary Feature Enhancements
- Utility Related Feature Enhancements
- Bugs in the Frame Virtual Machine
- Documentation Improvements

Enhancements discussed under these topics are grouped together for similar functions rather than by individual operators.

While going through the rest of this Chapter, one would note that Chapters 4 and 5 have been referenced several times. Since a lot of issues and concepts were identified that were either common to several of these enhancements or closely related to each other, it was decided that it will be best to discuss them in one place. Chapters 4 and 5 cover these issues and concepts. The functionality of each individual operator accounting for the application of these issues and concepts, have been described in Chapter 7. General description of the operators and the illustrative examples are covered in Chapter 6.

3.2.1 Primary Feature Enhancements

Primary feature enhancements increase the representational power of the frame package. In this section, several new operators are considered that operate on facets, slots and frames. It also includes improvements of the earlier frame virtual machine operators. Each one of these enhancements have been discussed below.

3.2.1.1 Access to All Facets

The earlier version allows access to only the value facet through the use of three slot operators - add_value_to_slot, remove_value_from_slot and return_value_from_slot. If the package allowed similar access to other facets, it would provide additional flexibility to the user. This will be a useful feature in order to provide increased information retrieval and an augmented knowledge capability. For example, retrieval of the value in type facet or max or min facets would provide additional information to the user. Addition or removal or change of information in these facets would augment knowledge capability of the system.

This flexibility can be provided at the cost of greater consistency checking. For example, when min or max facets are changed, the cardinality of the existing value facet will need to be rechecked for consistency.

Whether complete consistency should be maintained or not is an issue that needs considerable thought. This is being covered in Chapters 4 and 5.

3.2.1.2 change value of slot Operator and if changed Facet

To change the value of the value facet in a slot, two operations are required in the earlier version. First the use of:

remove_value_from_slot(FrameName,SlotName,OldValue)

operator followed by:

```
        add_value_to_slot(FrameName,SlotName,NewValue)
```

operator. For example, in the following frame:

```
        def_frame(jim:
                [is_a:[value:[male]],
                status:[value:[single],
                        min:1]
                ]).
```

In order to change the marital status, one has to first remove the value single and then add the value married. Thus the min facet must equal 0 to accommodate the removal, when in fact it should always be equal to 1 to reflect that the slot should always have a value.

Both of these slot operators can be combined into one, and a new slot operator can be defined as change_value_of_slot(FrameName, SlotName, OldValue, NewValue). Correspondingly an if_changed facet can be define to activate a demon.

In fact, the change_value_of_slot operator can be implemented in such a way that it can also provide the function of add_value_to_slot (if the OldValue is a null list) and remove_value_from_slot (if the NewValue is a null list). However, one may still like to keep add_value_to_slot and remove_value_from_slot operators so that efficiency is maintained, as it is obvious that add_value_to_slot and remove_value_from_slot will be more efficient than using change_value_of_slot to accomplish the task of only add_value or only remove_value. When both remove and add need to be accomplished, using change_value_of_slot will be much more convenient.

Change can also be implemented for facets other than value facet. The issues of consistency checking also surface here and are covered in Chapters 4 and 5.

3.2.1.3 Delete/Undefine Frame Operator

The inclusion of an undefine or a delete operator for a frame would be very useful when the knowledge base needs to be changed or updated frequently. In the earlier setup a frame cannot be removed at all.

However, before implementation of the removal of a frame is considered, its impact on knowledge base consistencies and computational tractability needs to be examined thoroughly. Removal of a frame can affect the inheritance for example.

A solution to consider is to have the knowledge base file reconsulted every time a slot or a frame is removed. This would confront the user with all the inconsistencies that the package finds. However, this would mean that the removal is effected first, and then all inconsistencies are brought up on reconsultation of the entire knowledge base file. Instead, the system should warn the user of inconsistencies without actually effecting the removal of the slot or frame in question. The user at that point should have the option of making changes to remove some or all of those inconsistencies and then only actually confirm that the system can proceed with the process of removal of the slots or frames in question. Consideration needs to be given to the fact that reconsulting the whole knowledge base could overburden the system (poor computational tractability), and an attempt should be made to reduce the overhead.

Another solution would be to initiate some kind of a tracing function that would provide the user a list of all those frames/slots that reference the underlying frame/slot. This tracing function would be activated when an undefine request is made. The user can then check each of those frames and slots and make appropriate changes in them if desired. Only after a final approval of the user will these changes take effect.

Another alternative would be to let the knowledge base simply become inconsistent.

A lot needed to be examined before any solution could be implemented. Again Chapters 4 and 5 deal with related issues and concepts that finally lead to the specific implementation.

3.2.1.4 Nonduplication in Value Facet

The earlier version allowed adding the same value more than once in the value facet if the max cardinality constraint allowed.

An example from the knowledge base file animalkingdom (Appendix C):

```
|?- def_frame(bird:[....  reproduction:[value:[oviparous,oviparous]....]).
yes
```

This allowed 'oviparous' to appear twice in the value facet above. If we add

```
|?- add_value_to_slot(bird,reproduction,oviparous).
yes
```

This will even add it a third time.

It is not quite clear if there was any specific purpose for allowing this. In fact it creates a problem when remove_value_from_slot(bird,reproduction,oviparous) is used as it removes only one instance of 'oviparous' from the facet list.

It seems appropriate to only allow single instance of the same value in any facet. This will require additional search in the knowledge base; however, it will eliminate redundantly stored knowledge. Additionally, a value facet is a set of possible values that a slot can have and theoretically by definition, a set can have only have unique values.

3.2.1.5 Addition/Removal of Multiple Elements in Value Facet

In the previous version, if more than one element needs to be added to or removed from the value facet using slot operators, one needs to use the operator as many number of times as the number of elements.

Consider the following example session from knowledge base file animalkingdom (Appendix C):

```
|?- add_value_to_slot(animal,food_source,egg).
```

```
yes
/?- add_value_to_slot(animal,food_source,plant).
yes
/?- add_value_to_slot(animal,food_source,  [cake,coffee]).
yes
```

It adds [cake,coffee] as one element instead of two entries in the value list. The frame at this point looks like:

```
animal:
    [
        .
        .
        food_source:[value:[[egg, plant,[cake,coffee]]
        .
        .
    ]
```

```
/?- remove_value_from_slot(animal,food_source,[[cake,coffee],plant,egg]).
no
/?- remove_value_from_slot(animal,food_source,[coffee,cake]).
no
/?- remove_value_from_slot(animal,food_source,[cake,coffee]).
yes
/?- remove_value_from_slot(animal,food_source,  egg).
yes
```

Some convenient way of handling more than one element at a time can be a useful feature. Similarly the order of atoms in the list [coffee,cake] matters. [coffee,cake] and [cake,coffee] should be considered equivalent for this purpose.

Another problem with add_value_to_slot is encountered when a list is specified in value parameter. When a list is specified in add_value_to_slot, the value facet becomes a nested list. This can lead to problems with demon execution that relied on a single value in the earlier package. Consider the following example session from the knowledge base bookprototype (Appendix D).

```
|?- add_value_to_slot(typeface_type,available_types,[baskerville,times]).
You want to add the typeface [baskerville,times]
Is this a regular font y,or,n?
|: y.
What is the italic form of [baskerville,times]?
|:
.

.
```

Instead of considering baskerville and times as two different typefaces, it considers [baskerville,times] as single typeface which is not correct. There should be a better way to handle these types of situations.

The problem lies in the fact that in the above case the system recognizes atoms only. It considers the list [baskerville,times] as an atom. A solution will be to make it recognize the above as a list.

3.2.1.6 Loading vs. Working Environment

[HISS87a] gives the impression that the frame virtual machine maintains complete knowledge base consistency from start to end. At the time of loading a knowledge base file into the frame virtual machine (or in other words working memory), in the earlier version, the frame virtual machine performs a complete consistency checking and does not load any inconsistent frame into the working memory. However, during the session, some operators for example - 'remove_value_from_slot' can create inconsistencies in the knowledge base but the user is still allowed to work with that inconsistent knowledge base during that session.

This way on one hand, the frame virtual machine does not allow to load inconsistent frames to begin with but on the other hand, it allows creation of inconsistencies through the frame virtual machine operators during the session.

consider the situation if a utility existed that allowed saving this knowledge base during the session then at the time of reloading the saved knowledge base, the frame virtual machine will not allow the the loading because it has inconsistencies. The same frame virtual machine that allowed use of a knowledge base at one time, will not allow reuse of the same knowledge base the next time!

A solution to consider would be that the package maintains complete consistency throughout the session. But in order to maintain a complete consistent system, a lot of checking is required throughout the knowledge base before performing any operation the user requests. Maintaining a completely consistent system can therefore result in a poor computational tractability of the package.

Another solution could be that the package provides a choice to the user at the start of the session if he/she wants to start with a slower but consistent system or a faster but inconsistent system.

Again before deciding the exact implementation, one needs to weigh the issue of consistency checking and computational tractability.

These issues and final choice of implementation are discussed in Chapters 4 and 5.

3.2.2 Utility Related Feature Enhancements

These features will help in development and debugging of knowledge base. These features include knowledge base edit and save facility, a trace facility, a facility to print hierarchical tree of frames, etc.

3.2.2.1 Edit and Save Changes in Knowledge Base

Sometimes a user can make typographical errors, accidentally entering wrong values while creating the knowledge base. In the earlier version, attempts to overwrite a frame while in the frame package results in an error stating that the frame already exists. Therefore, it becomes necessary to exit from the package, reenter prolog and reconsult the knowledge base file. Of course, one loses whatever the user has done during the previous session except for the changes made specifically in the knowledge base file using the vi editor. The user will have to reenter all other information.

To overcome this problem, editing facilities inside the frame package would be helpful, so that to edit the knowledge base, one does not need to exit from the package and the prolog Environment. Several editing features have been provided in the enhanced frame package which have been covered in Chapters 5 and 6.

A user may also need to interactively develop a knowledge base or make several changes for some purpose by testing various scenarios by trial and error. It is obvious that the user will want to save the knowledge base as it exists in its final form but in the earlier frame package, the user cannot do this. The only way is to actually go into the knowledge base file(s) outside of the frame virtual machine and make changes, which can be done accurately only if the user keeps a detailed record of all the things that were done during the interactive session. Obviously there is a need for a knowledge augmentation technique that will not only allow the system to learn but also retain new information.

An ability to save the most current version of the knowledge base in the frame package would be ideal. It should be possible to save it under a new name or replace the old one as per the requirement of the user.

3.2.2.2 Tracing Feature

Tracing is the only feature other than syntax checking (of frames and slots) that helps the user in debugging and understanding the knowledge base in the earlier package. Since the user can best relate to the knowledge base he/she has created/used, the trace must conform to the steps as they relate to the user's knowledge base. The trace provided in the earlier version, by the prolog interpreter is too detailed and difficult to understand because it goes into the details of the source code of the frame virtual machine. Hence the trace needs to be modified in such a way that it only relates to the user's knowledge base. It would be useful to have a facility to switch on the detailed trace (as it exists in the previous version) when required.

3.2.2.3 Printing Hierarchical Tree of Frames

The frame virtual machine represents knowledge in a tree structure of frames. Generic knowledge is stored higher up in the hierarchy and shared by frames lower down. A facility to quickly print the hierarchical tree of frame names of the knowledge base would be a very useful utility. This would help in the debugging process and also serve as a nice tool for displaying development ideas, especially when the knowledge base is very large and there is a lot of inheritance.

3.2.2.4 Short Form of Slot Operator Names

The operator names are very long, for example,

```
remove_value_from_slot(Framename,Slotname,Value),
```

and therefore, are cumbersome to use. Such long names may be helpful from the point of view of providing a quick mini-definition of the operator embedded in the name of the operator itself. However, for convenience, short operators are better; rem_val or some such short name seems more appropriate.

3.2.3 Bugs in the Frame Virtual Machine

Testing of the frame virtual machine lead to the identification of bugs that existed in the earlier version. Most of these bugs were related to the functionality of the operators, some were related to the differences in implementation from what was intended etc.

Some of these bugs were discussed in the thesis proposal and some were identified later on. All of these bugs were fixed during the enhanced process.

3.2.4 Documentation Improvements

The User's Manual [HISS87b] is very brief and does not clearly explain many of the functions that are available in the frame virtual machine. The operators, facets, predefined demons, etc. are described, but at times the descriptions are not clear on exactly how to use them. Good examples on the proper and incorrect use of the frame and slot operators are missing, and the use of demons has not been explained at all. Reference to the thesis [HISS87a] is necessary in order to use the package, or else one has to resort to trial and error.

For the enhanced package, the User Manual has completely redesigned. It includes the purpose, the syntax, example(s) of usage of each operator. Additionally, a Technical Manual has also been prepared that provides the functionality of each operator. The order in which the information on the operators has been provided in the User Manual as well as the Technical Manual is based on the type and functionality grouping of the operators. For user's convenience an alphabetical index of the operators has also been included at the end of both manuals.

To make it self sufficient, some major concepts of the package have also been discussed in the User Manual.

4 KNOWLEDGE REPRESENTATION ISSUES

The purpose of this chapter is to explain knowledge representation issues that played an important role in the enhancement of this package. This will provide a basis for a lot of answers to the questions such as - "why a particular operation performs in a certain fashion under certain conditions, for example, under consistent versus inconsistent option of the package.

4.1 ISSUES

A perusal of the Frame Virtual Machine coupled with an investigative on-line usage brought up the following issues:

- Expressive Power
- Consistency
- Computational Tractability

One of the observations made during the study of the Frame Virtual Machine was, that the expressive power of the system could be increased by introducing new facets, slots and frame operators. For example, access to all facets in a slot; removal of a frame from the knowledge base, etc. Removal of a frame may be a desirable thing to include; however, a question comes up about consistency of the knowledge base after removal of a frame. Removal of a frame can affect inheritance for example. Even allowing the deletion of a terminal frame can result in inconsistencies. Assume that a type constraint allowed the addition of a value to a slot because of frame 1's existence and then frame 1 was removed. A value that exists in a constrained slot would no longer meet the type constraint. Should the system provide such inconsistencies or not? If the system maintains complete consistency, then before performing the operation, it has to do a thorough consistency checking for each slot of each frame in the knowledge base. If the knowledge base is large, consistency checking is going to take forever and computational tractability will be poor.

The above discussion was intended to quickly provide a basic familiarity to these fundamental issues. Several researchers in artificial intelligence have also talked about these issues, and some of the selected views are presented below:

An excerpt from Minsky's paper highlights the issue of consistency:

> "Completeness is a trivial consequence of any exhaustive search procedure, and any system can be "completed" by adjoining to it any other complete system and interlacing the computational steps. Consistency is more refined; it requires one's axioms to imply no contradictions. But I do not believe that consistency is necessary or even desirable in a developing intelligent system. No one is ever completely consistent. What is important is how one handle paradoxes or conflict, how one learns from mistakes, how one turns aside suspected inconsistencies." [MINS85,261]

Minsky further concludes "logical" reasoning is not flexible enough to serve as a basis for thinking. He prefers to think of it as a collection of heuristic methods, effective only when applied to starkly simplified schematic plans. Minsky believes that the consistency that logic demands is not otherwise usually available and probably not even desirable (!) because consistent systems are likely to be too weak.

Bobrow, while discussing the features of KRL, emphasizes the following:

> "A knowledge representation language must provide a flexible set of underlying tools, rather than embody specific comments about either processing strategies or the representation of specific areas of knowledge." [BOBR77,984]

Bobrow believes that a knowledge representation system should have an extensive and varied repertoire of mechanism in order to achieve intelligent performance [BOBR85,284]. KRL (Knowledge Representation Language) was therefore designed to provide a large expressive power.

The builders of the KNOBS system, which is a highly interactive experimental knowledge based planning system for tactical air command and control, chose FRL (Frame Representation Language) as a base for the representation, access and manipulation of frames

[ENGE80,184]. The primary role of KNOBS is checking the completeness and consistency of a plan as it evolves through an interchange in which the user normally makes the significant choices. The program supports the user by explaining clearly and judiciously what inconsistencies have arisen, by understanding what has to be rechecked as the planner changes elements of the plan in response to the program's criticism, and by providing the planner with dynamically generated lists of recommendations for various plan elements [ENGE81,141].

Designers of KANDOR believe that limiting the expressive power of a knowledge representation system can guarantee that all its operations terminate in reasonable time. This makes the system usable as part of larger knowledge based systems that have to deal with the real world [PATE84,11]. For better computational tractability, the designers of KANDOR had to trade off several features, one of them being the lack of ability to change information, such as changing the slot fillers for some individual.

There is a tradeoff between the expressiveness and the tractability of a knowledge representation scheme. Neither expressiveness nor tractability by itself determines the value of a representation language[LEVE85,41]. Levesque and Brachman feel that there are many interesting issues to pursue involving the tradeoff between expressiveness and tractability. Although there has always been a temptation in knowledge representation to set the sights either too low (and provide only a data structuring facility with little or no inference) or too high (and provide a full theorem proving facility), their paper argues for the rich world of representation that lies between these two extremes.

What should one go for, then?

Better Computational Tractability?
or
Large Expressive Power?
or
Consistent System?

One would ideally like to have all the three capabilities. However, this may not be possible all the time. It seems that there is no simple answer. If the knowledge representation system has less expressive power, computational tractability will be better and operations can be performed quickly. On the other hand, if the knowledge representation system incorporates a lot of expressive power, the system would be able to do a lot; however, there may be a possibility that the system never completes a particular operation when asked for it. Similarly, to maintain complete consistency, a knowledge representation system may need to perform a very large number of checks resulting in poor computational tractability.

A knowledge representation system developed for a very small, limited domain can perhaps have very good computational tractability while maintaining complete consistency and full expressive power. In the real world however, applications can tend to be large and demand knowledge representation schemes with large expressive power. In such cases maintaining good tractability may require trading off expressive power for consistency or vice versa. For example, KANDOR trades expressive power in order to provide good tractability. On the other hand, KNOBS, using FRL as a base, maintains consistency and hence trades off computational tractability. KRL provides a large expressive power but ultimately it was reported as a failure, collapsing perhaps under the weight of its own features [BOBR85,263]. As one thinks more and more about these issues, perhaps the best is to choose an intermediate solution that will provide acceptable tractability and some kind of limited expressive power and consistency. As computational power increases with technological advances, obviously more can be achieved within the limits of such computational power. These limits however, should be properly recognized. If the program consumes a quantity of resources that is exponential in the size of the task, $O(k^n)$, then each doubling of available resources only means an additional (ln 2/ln k) words, regions or clauses can be handled [MACK81,69]. It is the proper combination of all these attributes that will determine the value of the knowledge representation system.

For this enhanced frame package, the concept that was used in deciding the tradeoffs is taken from the way humans acquire knowledge and expertise.

4.2 THE HUMAN MIND, KNOWLEDGE AND LEARNING

Consider the human learning process. The human mind constantly keeps acquiring knowledge by somehow absorbing various facts and reasonings. When a new fact is absorbed that does not fit well with previously acquired knowledge, the mind rejects that fact; it accepts the fact if it is not found inconsistent with the previously acquired knowledge. However, the level and depth of such consistency checking varies from time to time based on several practical factors, for example, how much knowledge is already available on the subject, how much time is available to analyze (i.e. how much detail can one afford to go into) and how important the fact might be. At times one only checks for some immediate concerns, and if there is nothing inconsistent found, then this new fact is accepted. Later, if time permits, one may sit down and go through all the details to thoroughly analyze them. It is quite possible that the detailed analysis later on reveals some inconsistencies that may need to be resolved. Sometimes some of these inconsistencies do not get resolved, and the individual may decide to live with them (as opposed to discarding them).

On the other hand, one may want to thoroughly satisfy all concerns every time some facts are available to be absorbed by making sure that they do not generate any inconsistency with any other knowledge acquired so far. This detailed analysis is generally very difficult and is not done very often for complex situations.

The enhanced frame package reasoning attempts to parallel the above. Human knowledge operations can be equated to the expressive power of the knowledge base developed by using the enhanced frame package. How much detail can a human mind afford to go into can be equated to the computational tractability. This analogy to the human mind makes it easier to understand the tradeoffs between consistency, expressive power and computational tractability. A completely consistent system with very large expressive power could become so slow that it may take forever to analyze a situation (poor computational tractability). To improve the computational tractability, therefore, some tradeoff is necessary. This could be limited consistency checking to maintain a certain level of expressive power and computational tractability in the system. This may be what one could describe as "normal busy mode" of the busy system, perhaps equivalent to something like humans making decisions "on the go". However, one can go a step further by allowing the option where the computer would do detailed consistency checking when asked for it (knowingly taking a long time, of course). This would be done only once in a while when needed, for efficiency reasons. This additional feature is provided in the enhanced package to analyze the full details from time to time. With this operation, a complete consistency check can be run on all the knowledge in the knowledge base when desired. This feature has been named the

Knowledge Analyzer (KA).

The knowledge analyzer only reports the inconsistencies present at the time it is invoked and leaves the option with the user to modify the knowledge base if so desired. A lot of times, just as in the human learning process, certain inconsistencies may be left for an indefinite period of time because there may not be a desirable solution, but that does not stop the human mind from working. Similarly, the package does not freeze in such situations. Eventually if a solution is found, the inconsistencies may be resolved, otherwise life still goes on.

With this philosophy in mind, most of the enhanced frame package operators perform local consistency checking all the time, just like a human mind, which on a regular basis can only afford to do a limited checking on immediate concerns. For example, before adding a new value to a specified slot in a specified frame, cardinality and type constraint checking will be done for that slot only. This is a case of **local consistency** checking or what will also be referred to as limited consistency checking. The consistency checking done by the KA (Knowledge Analyzer) will fall into the category of a complete or global consistency checking.

5 IMPLEMENTATION CONCEPTS

Keeping enhancements and the underlying conceptual framework in mind, the following concepts were developed for the enhanced frame package.

5.1 DEFINITION OPERATORS

Definition operators are provided to create objects of the frame data type. These include two operators a frame defining operator and a slot defining operator. The frame defining operator allows for specifying the frame name, defining slots within that frame and defining facets within those slots. The global slot defining operator allows for specifying the slot name and the facets within that slot.

Since in this knowledge representation tool, the knowledge is stored in the form of frames and slots, a knowledge base file is created using these definition operators. At the time of loading a knowledge base file (or in other words, loading a frame/slot), these definition operators are executed and if possible, frames and slots are loaded into working memory.* Once the knowledge base is loaded into working memory, several operations can be performed to access, modify or analyze the knowledge base (for example, various facet operations, editing knowledge base, saving knowledge base, knowledge analyzer etc; details in Chapter 6).

5.2 LOCAL CONSISTENCY CHECKING VS. GLOBAL CONSISTENCY CHECKING

As explained in the previous chapter, there is always a trade off between the three major knowledge representation issues: consistency, computational tractability, and expressive power, for any knowledge representation system. The philosophy that has been adopted for

* These definition operators check for the uniqueness of the chosen frame or slot name, check the syntax of the frame or slot, check the values in the slot against any constraints (described in Section 6.2) and load/add all acceptable frames and slots to the knowledge base/working memory. Frame(s) or slot(s) with error(s) do not get loaded into working memory (constrained by the way prolog works).

the enhanced frame package is parallel to human reasoning. Most of the enhanced frame package operators perform local consistency checking all the time, just like a human mind, which on a regular basis generally can afford to do only a limited checking on immediate concerns. Before adding a new value to a specified slot in a specified frame, cardinality and type constraint checking is done for that slot. The impact of this operation on the rest of the knowledge base is not analyzed at that time. This is a case of local consistency checking or limited consistency checking. When global or complete consistency checking is required, then it can be done by the knowledge analyzer (analyzing the whole knowledge base). In order to analyze a single frame with respect to the rest of the knowledge base, one could use the check_frame operator (Section 6.18.2).

5.3 LOADING AS A CONSISTENT/INCONSISTENT SYSTEM

The enhanced package provides a choice for the user of loading a knowledge base file as a consistent or an inconsistent system.

The set of operators available in both cases are the same; however, the system behaves a little differently when run as a consistent knowledge base versus an inconsistent one. In the case of a consistent system, complete consistency checking of the knowledge base is performed at the time of loading, in addition to the syntax checking of the knowledge base. In the case of an inconsistent system, only syntax checking is performed, and no consistency checking is done at the time of loading.

5.4 CREATING AND LOADING THE KNOWLEDGE BASE

As mentioned earlier, the enhanced frame package is an expert system development tool. In order to work with an expert system, one has to have a knowledge base. There are two ways to create a knowledge base in this package: one - outside the frame package using vi editor, and two - inside the frame package using the file_create operator (similar to the vi operator - explained in Section 6.4). Allowing initial creation of the knowledge base in two ways is primarily for convenience. Creating the knowledge base outside the package saves the user from entering the package, in the case when the user just wants to create a knowledge base but does not want to work with it at that time. On the other hand, once in the package, the facility to create a knowledge base from inside the package avoids repeatedly going out and coming into the package in order to create a knowledge base.

An existing knowledge base file can be loaded into the frame package using the 'Load' operator (details in Section 6.3). If the knowledge base file was created from inside the package, loading of the file is done automatically right after the creation.

The loading operation brings the contents of the knowledge base file into working memory. At the time of loading a knowledge base file, the frame package performs the syntax checking. Constraint checking is done if the system was loaded as a consistent system. In case of any error(s), the following happens:

- An error message is displayed.
- The frame or slot that has the error is not loaded into working memory. (This is constrained by the way prolog works.)

Hence, in order to correct the errors, one needs to access the knowledge base file as it exists in the working directory. The access to this file is allowed from inside the frame package using the 'file_edit' operator (details in Section 6.5). The primary purpose of this operator is to allow the user to correct syntax errors or any other errors encountered right after loading or creating the knowledge base file without exiting from the package.

5.5 OPERATIONS ON WORKING MEMORY

'file_edit' is the only operator that updates the knowledge base file present in the working directory. All other operators act only on the material that is present in **working memory** (they do not disturb the original knowledge base file in the directory). These operators allow the user to modify the existing knowledge base in working memory. All of this is transparent to the user. The concept used will be clear from the following example:

> Consider a scenario where a user who, after loading a knowledge base in the frame package, performs a few operations on frames/slots and modifies the

knowledge base. At that point, suppose the user realizes that he/she should not have made these changes and wants to restart from the original knowledge base file. If all the operations were performed on the original knowledge base file, then the original file gets modified, in which case the user must remember what changes were made and undo those changes manually. On the other hand, if these operations were performed on the copy in working memory, then the user just needs to replace the current contents of working memory for that particular knowledge base file by reloading the contents of the original file from the working directory. To do so, the enhanced frame package provides a 'reload(filename)' operator (details in Chapter 6). This way, one does not even have to go out and come back in to meet the above need.

Another useful operator available in the package is the 'save' operator. The 'save' operator allows a user to save the updated version of the knowledge base file that is present in working memory, into a user specified file at any time during the session. The user specified file name can be the same as the original file name or some other new file name. The saved version can be used later for further modifications. This comes in very handy for the original development of a knowledge base. The combination of the ability to manipulate the working memory and to save the updated version provides the best of everything to the user. The only negative aspect is the fact that any comments in the file are not saved.

5.6 OPERATORS THAT MODIFY THE KNOWLEDGE BASE

The enhanced frame package provides several operators for different needs. Some operators are for the purpose of obtaining information from the knowledge base, some for the analysis of the knowledge base, some for modifying the knowledge base, and some for utility operations. All operators in these categories except for some of the modifying operators behave similarly under the consistent and inconsistent system options.

The operators that **modify** the knowledge base can be divided in two parts:

1. <u>Type One Operators</u>: Those operators that involve definition operator execution. These are load, temp_edit, file_create, frame_edit and reload operators.

2. Type Two Operators: All modifying operators other than the ones included above. These include various facet, slot, and frame operators.

The details on these individual operators are included in Chapter 6.

As explained in Section 5.3, the execution of the definition operators is different for a consistent system choice compared to an inconsistent system choice. Therefore, all Type One Operators behave differently based on the user opted choice of a consistent versus inconsistent system.

Type Two Operators and all other operators are independent of the definition operators, and they behave the same way in both the options.

5.7 ADDITION/REMOVAL OF MULTIPLE ELEMENTS IN THE VALUE FACET

The enhanced frame package allows addition/removal of a single value as well as multiple values in the value facet through the following operators:

```
add_value_to_slot(framename,slotname, value).
remove_value_from_slot(framename,slotname, value).
```

Where value argument can be an atom or list of atoms. If the value argument is an atom, it allows the user to add/remove a single value; whereas a list of atoms as value argument allows the user to add/remove multiple values at a time. A list of atoms as value argument can have one entry also, in that case it will be the same as adding/removing just one value to/from the value facet.

5.8 A NOTE ABOUT FACET OPERATORS

A value facet contains a list of entries as its value and the cardinality of the number of entries in the list is determined by the max and min constraints. Max, min and type facets on the other hand can have only one entry as their value. Because of this difference, add, remove and change operators work on these facets differently.

In value facet, one can add a value or a multiple number of values to the existing list of values, or remove a value or a multiple number of values from the existing list, or change part of the value list. However, for max, min and type facets, since they can have only one entry as their value, add will take place only if that facet does not already exist locally. Remove operators for such facets upon success, will remove that facet completely from the slot and change operators will change the value of that local facet completely (one single value replaced by another single value).

To remove the value facet completely, the remove_all operator has been provided. To change the value facet completely, the change_all operator has been provided.

For change and remove operation on a facet, the facet must be present locally in the slot.

5.9 PACKAGE BEHAVIOR: a difference from Prolog

The package is implemented in C-Prolog. However, one should not expect the exact same behavior as C-Prolog from the package because the behavior of the package is geared towards the functionality of the operations. Consider the following operator for example:

```
add_value_to_slot(framename,slotname,value)
```

from Prolog's perception, framename and slotname can be uninstantiated variables, however, from the perception of the frame package, both of these should be instantiated variables. The purpose of this operator is to put the knowledge about a slot's value into the knowledge base and therefore the user must know the frame and slot names in which the value should be added. In general, for all the facet operators, it is assumed that the framename and slotname will be instantiated variables.

Consider another case. In the add_value_to_slot operator, value argument is assumed to be an instantiated variable and it must either be an atom or a list of atoms. Value argument as a list of atoms implies the addition of more than one value at a time to the value facet. As an example, successful execution of add_value_to_slot(jim,kitty,[annie,minnie]); values annie and minnie will be added as two atoms rather than a list (of two atoms) in the value facet of slot kitty in frame jim. Consider the following special case for better understanding:

```
add_value_to_slot(jim,kitty,annie)
add_value_to_slot(jim,kitty,[annie]).
```

Upon successful execution, from Prolog's perception the outcome should be different in both the cases. In first case, it should add annie as an atom in the value facet and in the second case, list [annie] to the value facet. However, from the package's perception, the outcome in both of these situations will be the same. In both cases, it will add annie as an atom to the value facet list.

5.10 INTERNAL FEATURES OF THE FRAME PACKAGE

When the frame package is being used, a lot of things happen that are completely transparent to the user; however, they are very critical to the functionality and performance of the frame package. For example, whenever a knowledge base file is loaded, it automatically remembers what frames and slots belong to that file and in what order they appear on the file. In the case of several knowledge base files, it remembers the order in which these files were loaded. All this information will be referred to as 'internal information to the package' in the documentation. This internal information is very important throughout the frame package session. For example, during a temp_edit operation, the frame package identifies the frames and slots and their order, then displays them in that order for the user to edit. If the frames are moved in sequential position or if the frames are added, deleted or renamed during editing, the package keeps track of that and, at the end of the temp_edit, utilizes the revised status for all future operations.

Such automatic update of this internal information always takes place whenever a frame is created(added), loaded, deleted or renamed through any of the frame package operations.

5.11 TEMPORARY FILES

To perform tasks such as analyzing the knowledge base or editing the knowledge base (ka, check_frame, temp_edit, frame_edit operators), the frame package uses temporary files. These temporary files not only allow the user to perform the above tasks but also allow the knowledge base to return to the same status as it was before such operations.

Two temporary files are created in a remote directory for this purpose. For the purpose of explanation, lets call these files - temp and temp2. One of the files say temp2, is used to store the current knowledge base (or part of the knowledge base) facts as they exist so that when needed, they can be consulted to restore the knowledge base as it was. The other file temp is also used to store the current knowledge base (or part of the knowledge base), but it is subjected to the operations. These operations are consistency checking in case of analyzing the knowledge or editing in case of temp_edit and frame_edit operations. At the end of such operations, file temp2 is reconsulted to get back to the exact same status of the knowledge base as it was just before these operations, when needed. The details about the functionality of these tasks are available in the Technical Manual (Section 2.8 and Section 2.18).

5.12 KNOWLEDGE BASE FILE VS. DEMON PREDICATE FILE

Another important concept of the enhanced package is related to the difference between definition operators and demon predicates. A knowledge base file is composed of active definition operators, which create unique instances of frames and slots. Demon predicates, which are like any other Prolog predicates, are used to store procedural knowledge about the knowledge base. Editing, loading and saving of demon predicates are completely different from the definition operators. Hence it is highly recommended that the user create separate files for the demon predicates and the knowledge base (definition operators). Available demon operators are explained in the next chapter.

6 FRAME PACKAGE OPERATORS

This chapter describes all the operators available in the frame package. Some or all of the following aspects are included in the discussion for each operator as deemed necessary:

- the name and syntax of the operator
- the purpose of the operator
- an example to illustrate the operation(s)
- the functionality of the operator, explaining all the major logical steps that are followed when the operation is performed. The functionality has been included separately in Chapter 7 for all the operators.

Material covered in the previous chapters should help in better understanding of the frame package operators discussed here.

Please note that the illustrative examples use a different font for clarity. All system prompts and responses are in regular Helvetica italic and the user responses are in bold Helvetica.

The order of the operators in this section is not alphabetical so that they can be grouped by their functionality. This should facilitate easier understanding. However, an alphabetical index for the operators is provided at the back of this manual for convenience.

6.1 RUNNING THE FRAME PACKAGE

To load the package from the system prompt, type 'prolog /usr/local/lib/prolog/frameboot' and press the enter key. This will load the frame package. The command '/usr/local/lib/prolog/frameboot' is installation dependent and may differ from system to system. After loading the package, the user should respond with

```
start
```

The system will provide an option to start with a consistent or an inconsistent system. The user may respond with a 'yes' for a consistent system or a 'no' for an inconsistent system ('y'/'n' respectively). The default is 'no'.

The operation **'exit'** will allow the user to exit from the Prolog and the package environment, at any point. [Although 'halt' (as available in C-Prolog) can be used to terminate the session; 'exit' provides an opportunity to the user for saving updated knowledge base files.]

Example

```
system prompt[3] prolog /usr/local/lib/prolog/frameboot
C-Prolog version 1.4
[ Restoring file /usr/local/lib/prolog/frameboot ]

yes
|?- start.
Do you want to enforce a consistent system?
|:                /* The user can answer a 'yes'/'y' or 'no'/'n' */
```

6.2 STORING KNOWLEDGE IN A KNOWLEDGE BASE FILE

The frame package stores knowledge in the form of frames and slots. Frames are stored in the prolog knowledge base as structures that have the functors as frames. Similarly, global slots are stored in the prolog knowledge base as structures that have the functor slot. The operators that allow for defining frames and slots are called definition operators.

6.2.1 Definition Operators

Knowledge is stored in a knowledge base file using the definition operators. Two definition operators have been provided: a frame defining operator and a global slot defining operator.

6.2.1.1 The Frame Defining Operator

The structure of the frame defining operator is:

def_frame(Frame)
where Frame is
framename : [list of slots]

where framename is any unique, legal Prolog atom, and the structure of the list of slots is:

```
[slot1name : [list of facets],
 slot2name : [list of facets],
 .
 .
 .
 slotNname : [list of facets]]
```

where the slot names are atoms that are unique to the frame and the structure of the list of facets is:

```
[facet1name : facetvalue,
 facet2name : facetvalue,
 .
 .
 .
 facetNname : facetvalue]
```

where a facetvalue is dependent upon the facetname.

The only valid facetnames are value, type, max, min, default, if_added, if_removed, if_changed and if_needed. Please refer to the figure on page 40. All the facets have been described in Section 2.1. Only one value can be stored in a min or max facet, and that value must be an integer. Min by default is zero and max is one. The value in the value and default facets must be a list. The value in a type facet must be a legal type predicate, where a legal type predicate is:

a frame name (legal type)

a legal type @@ a legal type (or)
a legal type ## a legal type (and)
~ a legal type (not).

The value in the other facets may be either a single value or a list.

Example

```
def_frame(jim:
          [in_of :
                  [value:[male]],
           education:
          [value:[ms,phd],
                  max :5],
           complexion:
                  [value:[fair]],
          status:
                  [value:[unmarried],
                   if_changed:change_status]]).
```

6.2.1.2 The Slot Defining Operator

The slot defining operator is very similar to the frame defining operator. It is used to introduce predefined global slots into the knowledge base and has the following syntax:

def_slot(slotname:[list of facets]).

Where list of facets is as defined in Section 6.2.1.1. Also, refer to the figure on page 40.

Example

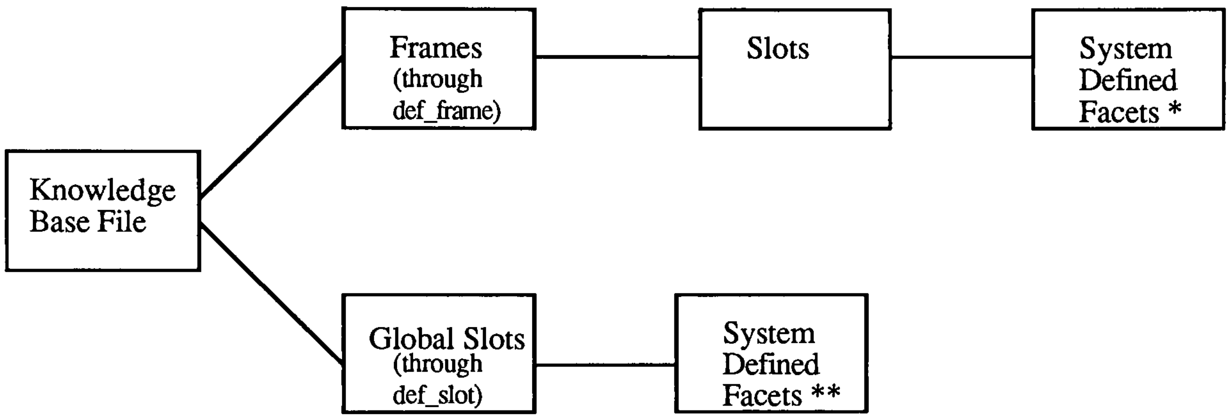
```
def_frame(male:
          [is_a:[value:[living_thing]]).
```

```
def_slot(father:
        [min:0,
        max:1,
        type:male]).
```

Since the frame defining operator allows for slot definitions, the slot-defining operator might seem redundant. It seems desirable, however, to be able to define a global slot independently that might occur in several unrelated frames. This slot definition would reduce redundant entry of information. An example of this is the "system defined" slot is_a and in_of. It would be redundant to include all facets of the is_a or in_of slot in every frame that contains this slot. The slot definition allows for facets of a slot to be listed once but at the same time makes it available to all frames using that slot.

is_a and in_of are system defined slots. is_a represents a subsumptive relationship between two classes. An in_of(instance_of), on the other hand, represents a subsumptive relationship between a class and an individual member of that class. The is_a and in_of slots can have maximum of 999 values in the value facet by default.

STRUCTURE OF A KNOWLEDGE BASE FILE



* max, min, type, value, default, if_added, if_needed, if_removed, if_changed

** max, min, type, value, default

6.2.2 Demons (Prolog Predicates)

Demons store procedural knowledge about the world in the form of Prolog Predicate.

The syntax for a demon in case of:

a) facets: if_added, if_removed, if_needed -

```
        demon_name(FName,SName,Value) :-
                ......
                ......
```

b) facet: if_changed -

```
        demon_name(FName,SName,OldValue,NewValue) :-
                ......
                ......
```

where:

- demon_name is a prolog functor
  FName is the name of the frame from where the demon gets invoked
- SName is the slot name in the frame FName
- Value is the value of the slot SName
  Value can be an atom or list of atoms

- OldValue is the current value in the slot that gets replaced by NewValue
- NewValue is the new value that replaces the OldValue present in the slot

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
          [in_of :
                [value:[male]],
           education:
                [value:[ms,phd],
                max  :5],
                complexion:
                        [value:[fair]],
                status:
                        [value:[unmarried],
                         if_changed:change_status]
                ]).

        ?-  def_frame(john:
                [in_of :
                        [value:[male]],
                 education:
                        [value:[bs],
                        max  :5],
                 complexion:
                        [value:[dark]],
                status:
                        [value:[widowed],
                        if_changed:change_status],
                wife:
                        [value:[jane],
                        min : 0,
                        max : 1]]).
```

Frames jim and john both have if_changed facets with the demon change_status in slot status. An example of the change_status demon can be written as follows:

```
        change_status(FName,SName,OldValue,NewValue)  :-
        (
           OldValue == unmarried,
           NewValue == married,
           write_sent('What is the name of the wife of', FName),
           read(WifeName),
           add_value_to_slot(FName,wife,WifeName)
        ;
        OldValue == widowed,
        NewValue == married,
        write_sent('What is the name of the new wife of', FName),
        read(NewWifeName),
        change_value_of_slot(FName,wife,OldName,NewWifeName)
    ).
```

Upon the execution of change_value_of_slot(jim,status,OldValue,married) (whose operator is described in Section 6.12.4), the value of slot status will be changed from unmarried to married and the change_status demon will be invoked. The change_status demon will ask the user to enter Jim's wife's name and then it will add the slot wife to frame jim with the wife's name as a value.

Upon the execution of change_value_of_slot(john,status,OldValue,married), the value of slot status will be changed from widowed to married and the change_status demon will be invoked. The change_status demon will ask the user to enter Jim's new wife's name, and then it will change the value of slot wife from jane to the new wife's name.

Additional user supplied arguments also can be passed to demons. Such an example is provided in Appendix D.

6.3 load(filename)

This operator loads an existing knowledge base file into working memory. For every frame or slot loaded successfully, a message will appear indicating the successful load, otherwise an appropriate error message will be displayed.

Example

```
| ?- load(animalkingdom).
```

Upon execution, the frames and slots defined in the knowledge base file animalkingdom will be loaded into working memory.

Please note that the **load operator** must be used as explained above to load an existing knowledge base file in to working memory. Do not try loading a knowledge base file as follows:

```
|?- [filename].
```

which is what one would normally do, when loading or consulting a file filename in the prolog environment. If a **knowledge base file** is loaded this way, it **will not perform as intended** while using certain operators of the frame package e.g. temp_edit, frame_edit, ka...etc. In order to access the frame package environment properly, one must use the load operator for loading a knowledge base file.

To load a demon file (Section 6.20), one must use the **load_demon** operator or simply consult the demon file.

6.4 file_create(filename)

This operator creates a knowledge base file from inside the frame package. The file_create operator is similar to the vi operator. At the end of the creation, the user can end the operation exactly like leaving the vi editor by typing ":wq" on the command line. At that time the knowledge base file is saved in the working directory and then the contents are loaded in working memory automatically as if the operation load(filename) were performed.

Example

```
| ?- file_create(animalland).
```

Upon execution, the user gets into the edit mode for the knowledge base file animalland.

6.5 file_edit(filename)

This operator allows the user to edit the contents of the knowledge base file filename present in the working directory (please note, working directory file as opposed to the working memory file). The best place to use this operator is just after the load or create operation, as explained in Section 5.4. Any erroneous segments of the knowledge base file will not be loaded in the prolog environment. The main purpose of this operator is to allow the user to fix syntax or other errors that exist in the erroneous segments of the knowledge base from inside the frame package. The edit environment here is also exactly like the vi edit environment.

Example

```
| ?- file_edit(animalkingdom).
```

Upon execution, the contents of the knowledge base file animalkingdom, as present in working memory will get deleted. The contents present in the working directory, will be displayed on the screen for editing. Loading of the file is performed automatically at the end of editing. It also updates the knowledge base file in the working directory.

6.6 purge(filename)

This operator purges the contents of the knowledge base file filename from working memory. It does not purge that knowledge base file stored in the working directory. It is useful when the knowledge stored in a file is no longer required, e.g. while using the file, the knowledge has been modified drastically to the extent that it is not useful anymore or maybe the user just wants to end working any further on that knowledge base file and possibly start working with another knowledge base file.

Example

```
| ?- purge(animalkingdom).
```

Upon execution, frames and slots stored in the file animalkingdom will no longer be available in working memory for further frame package operations.

6.7 reload(filename)

This operator purges the current contents of the knowledge base file filename from working memory and loads the contents of the same knowledge base file from the working directory into working memory. When there are several knowledge base files already loaded in working memory, the order in which they are loaded may be critical based upon the relationship between the files. Reload maintains the original sequential position of the knowledge base file in working memory. This operator comes in handy if the knowledge stored in a file filename was modified drastically and for some reason the user wanted to go back to the last saved version quickly.

Example

```
| ?- reload(animalkingdom).
```

6.8 temp_edit(filename)

The temp_edit is useful when knowledge needs to be modified in a knowledge base file at several places. This operator moves the contents of the knowledge base file filename from working memory into a temporary file 'temp' for editing. All the editing takes place in the temp file. At the end of editing, the contents of the temp file are automatically loaded back into working memory under the same filename. The edited file is not saved in the working directory in this process. Edit commands are exactly like the vi editor.

As mentioned earlier in Section 5.4, frames and slots with error(s) do not get loaded back into working memory. Hence, in a situation where the user makes a few errors while editing frames/slots in a knowledge base file, upon loading, those frames/slots will not be loaded back into working memory. The user then would not be able to perform any operation(s) on those frames or slots during that session. To avoid that, the user is always provided with the list of error messages and given a choice to undo the editing, as well as continue the editing, before the actual loading.

The undo edit option gives the user a chance to go back to the point just before the temp_edit operation (as if no editing was done). The continue edit option allows the user to pick up the editing from the point where it was last left.

Example

Suppose that the knowledge base file **information** looks like:

```
?- def_frame(kim:
       [father:
            [value:[roy],
             max:1],
        status:
            [value:[unmarried],
             min:1,
             max:1]
       ]).

?- def_frame(reena:
       [father:
            [value:[ron],
             max:1],
        status:
             [value:[married],
              min:1,
              max:1],
        husband:
             [value:[ravi],
              min:0,
              max:1]
        ]).
```

now suppose the temp_edit operation is performed.

```
| ?- temp_edit(information).
```

Upon execution, the contents of the file **information** would be placed into a temporary file for editing. Assume after editing, the contents of the knowledge base file in file temp looks as follows:

```
?- def_frame(kim:
        [father:
             [value:[roy],
              max:nine],
         status:
             [value:[unmarried],
              min:1,
              max:1]
        ]).

?- def_frame(reena:
        [father:
             [value:[ron],
              max:1],
         status:
             [value:[divorced],
            min:1,
            max:1],
        husband:
           [value:[ravi],
            min:0,
            max:1]]).
```

At the end of editing, the user will input **:wq.** At that time loading of the above knowledge base file in working memory will take place and the following messages will appear on the screen:

```
The max value of nine is not valid.
A facet structure in father:[value:[roy],max:nine] is bad.
The syntax of kim:[father:[value:[roy],max:nine],status:[value:[unmarried],
min:1,max:1]] is bad.
no
frame reena loaded successfully.
yes
temp reconsulted 248 bytes 0.766679 sec.
Would you like to
1.      exit the edit.
2.      undo the edit.
3.      continue the edit.
Input your option.
```

In case of option 1, only frame reena (the updated version) will get loaded into working memory. Frame kim (the edited version) has an error and, therefore, will not get loaded back in working memory.

If case of option 2, the package will undo the editing and will load frame kim and reena in working memory as they were before the editing was even started.

In case of option 3, the edited file with frames kim and reena will be displayed for further editing.

6.9 frame_edit(framename)

When several changes in several frames of a knowledge base file are required, temp_edit is the most convenient operator to use; however, if several changes are required in only one or two frames, the frame_edit operator is more convenient and faster. It allows the user to edit one frame at a time. The operation of this operator is similar to the temp_edit operator.

Example

```
| ?- frame_edit(reena).
```

6.10 save_file(filename,newfilename)

This operator allows saving the updated version of the knowledge base file filename present in working memory into the user defined file newfilename. The saved version can be used later on. It is important to note that the comments are not saved using any of the save operators. Only frames and slots are loaded into working memory. Comments are not available in working memory, hence they obviously cannot be saved.

If newfilename already exists then the user is provided with an option to replace this file. A 'yes'/'y' response would replace it, and any other response will not result in any file saving operation. If the newfilename is not in the directory, the save_file operator creates it.

One can use these operators at any time during the session. Saving does not delete the contents of that knowledge base file from working memory, so it is still available to the user for further operations.

Example

```
| ?- save_file(animalkingdom,ak).
```

Upon execution, contents of the knowledge base file animalkingdom from working memory will be copied in the knowledge base file ak in working directory.

6.11 save_it(filename)

The operation of the operator save_it is almost exactly like save_file except that the newfilename is the same as the filename. The operator always replaces the contents of the knowledge base file filename present in the directory with the contents of the knowledge base file filename present in working memory.

Example

```
| ?- save_it(animalkingdom).
```

6.12 VALUE FACET OPERATORS

The value facet contains a set of values that a particular slot can have. A set of operators is provided to access and modify the value facet. These operators will add, remove, modify(change) and return the slot values for the value facet.

6.12.1 add_value_to_slot(framename,slotname,value)

short_form **add_val**(framename,slotname,value)

It allows the addition of a value to a specified slot in a specified frame. Since this operator also allows the addition of more than one value at a time, the value argument should be either an atom or a list of atoms. Duplication in the value argument is automatically identified and eliminated.

Upon successful execution of the add_value_to_slot operator, if_added demons get activated if present in the specified slot.

Example

Suppose the knowledge base file contains:

```
?- def_frame(photosynthesis:[]).
?- def_frame(animal:
        [is_a:[value:[living_thing]],
         food_source:[type:(~photosynthesis),max:999],
        ]).
```

Upon performing:

```
| ?- add_val(animal,food_source,photosynthesis).
There is a type violation in slot food_source:[value:[photosynthesis], type:
(~photosynthesis),max:999] in frame animal
```

```
Value not defined as ~photosynthesis  YET.
The operation fails.
no
```

The above operation fails because of the type constraint that exists in the frame animal. It is specified that a value in the slot food_source must be other than photosynthesis.

Upon performing:

```
| ?- add_val(animal,food_source,[egg,milk]).
yes
```

Since the addition of values egg and milk in the slot food_source do not violate any constraints, add_val succeeds.

6.12.2 remove_value_from_slot(framename,slotname,value)

short_form **rem_val**(framename,slotname,value)

This operator allows the removal of a value from a specified slot in a specified frame. Since this operator allows removal of multiple values at a time, the value argument can be either an atom or a list of atoms. In the case of an uninstantiated variable as the value argument, the first available value from the value facet gets removed. This operator identifies and ignores any duplication present in the value argument.

Upon successful execution of the remove_value_from_slot operator, if_removed demons get activated if present in the specified slot.

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
        [is_a:[value:[male]],
         status:[value:[married],min:1],
         kitty:[value:[annie,minnie,kannie],max:7]
        ]).
```

Upon performing:

```
| ?- rem_val(jim,status,married).
Removing the value would violate cardinality constraints.
The operation fails.
no
```

It is specified in the frame jim that the value facet of the slot status must contain at least one value all the time because min=1.

```
| ?- rem_val(jim,kitty,[annie,minnie]).
yes
```

Removal of values annie and minnie do not violate any constraints, so rem_val succeeds.

6.12.3 remove_all(framename,slotname)

short_form **rem_all**(framename,slotname)

This operator allows the removal of a value facet from a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
        [is_a:[value:[male]],
        status:[value:[married],min:1],
        kitty:[value:[annie,minnie,kannie],max:7]
       ]).
```

Upon performing:

```
| ?- rem_all(jim,status).
Removal of the value facet would violate the min constraint.
Operation fails.
no

| ?- rem_all(jim,kitty).
yes
```

6.12.4 change_value_of_slot(framename,slotname,oldvalue,newvalue)

short_form - **change_val**(framename,slotname,oldvalue,newvalue)

This operator can be used to change the value oldvalue, which must be present in the value facet of a specified slot in a specified frame, to newvalue. It combines the operations of rem_val and add_val. This operator is implemented in such a way that it provides the function of add_val (if the oldvalue is a null list) and rem_val (if the newvalue is a null list). The arguments oldvalue and newvalue, can be atoms or a lists of atoms.

Upon successful execution of the change_value_of_slot operator, if_changed demons get activated if present in the specified slot.

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
        [is_a:[value:[male]],
        status:[value:[married],min:1],
        kitty:[value:[annie,minnie,kannie],max:7]
       ]).
```

Upon performing:

```
| ?- change_val(jim,status,marrled,dlvorce).
yes
```

The value married in slot status of frame jim will be changed to value divorce.

6.12.5 change_all(framename,slotname,newvalue)

This operator allows replacement of the entire contents of the value facet in a specified slot of a specified frame, with newvalue.

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
       [is_a:[value:[male]],
        status:[value:[married],min:1],
        kid :[min:0,max:1],
        kitty:[value:[annie,minnie,kannie],max:7]
       ]).
```

Upon performing:

```
| ?- change_all(jim,kid,[lauren,hardy]).
Slot kid has no value facet in frame jim.
The operation fails.
no
```

```
| ?- change_all(jim,kitty,[barney,marry]).
yes
```

Value facet of the slot kitty will now have the values barney and marry instead of annie, minnie and kannie.

6.12.6 return_value_from_slot(framename,slotname,value)

short_form - **ret_val**(framename,slotname,value)

This operator allows the retrieval of a value facet from a specified slot in a specified frame. Value parameter is assumed to be an uninstantiated variable or an exact match of the entire value facet.

If no value facet or default facet is available, if_needed demons get activated if present in the specified slot.

Example

Suppose the knowledge base file contains:

```
?- def_frame(jim:
        [is_a:[value:[male]],
         status:[value:[married],min:1],
         kitty:[value:[annie,minnie,kannie],max:7]
        ]).
```

Upon performing:

```
| ?- ret_val(jim,kitty,Value).
Value = [annie,minnie,kannie]
```

Values annie, minnie and kannie are retrieved from the local slot kitty of frame jim.

6.13 MAX FACET OPERATORS

The max facet contains the maximum number of values that a particular slot can have. A set of operators is provided to access and modify the max facet. These operators will add, remove, modify(change) and return the max facet. The system default for the max facet is one except for the system defined slots is_a and in_of, where the max facet is defined as 999.

6.13.1 add_max_to_slot(framename,slotname,max)

short_form **add_max**(framename,slotname,max).

This operator allows the addition of a max facet to a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(person:[]).
?- def_frame(boy:[is_a:[value:[male]]]).
?- def_frame(girl:[is_a:[value:[female]]]).
?- def_frame(class:
            [teacher:[type:person],
             student:[type: boy @@ girl,
                      max:30]
            ]).

?- def_frame(grade4:
            [is_a:[value:[class]],
             teacher:[value:[ms_smith]],
             student:[value:[kim,neena,reena],min:2]
            ]).
```

Currently, value facet [kim,neena,reena] does not violate max constraint because it inherits 30 for max from slot student in frame class.

Upon performing:

```
| ?-add_max(grade4,student,2).
Entries in value facet outside new min max range.
There is a cardinality violation in slot student:[max:2, value:[kim, neena,reena],min:2].
in frame grade4 max=2 and min = 2.
New value of max results in cardinality violation , Operation fails.
no

| ?-add_max(grade4,student,10).
yes
```

Max = 10, does not violate the cardinality constraints.

6.13.2 remove_max_from_slot(framename,slotname,max)

short_form - **rem_max**(framename,slotname,max)

This operator allows the removal of the max facet from a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(person:[]).
?- def_frame(boy:[is_a:[value:[male]]]).
?- def_frame(girl:[is_a:[value:[female]]]).

?- def_frame(class:
          [teacher:[type:person],
                student:[type: boy @@ girl,
                       max:30]
               ]).

?- def_frame(grade4:
               [is_a:[value:[class]],
                teacher:[value:[ms_smith]],
                student:[value:[kim,neena,reena],min:3,max:10]]).
```

Upon performing:

```
|?- rem_max(grade4,student,Max).
Max = 10
yes
```

Removal of the max facet does not violate any constraints.

6.13.3 change_max_of_slot(framename,slotname,oldmax,newmax)

short_form **change_max**(framename,slotname,oldmax,newmax)

This operator can be used to change the max facet with value oldmax of a specified slot in a specified frame to newmax.

Example

Suppose the knowledge base file contains:

```
?- def_frame(person:[]).
?- def_frame(boy:[is_a:[value:[male]]]).
?- def_frame(girl:[is_a:[value:[female]]]).
?- def_frame(class:
```

```
                [teacher:[type:person],
                 student:[type: boy @@ girl,
                            max:30]
                ]).
?- def_frame(grade4:
                [is_a:[value:[class]],
                 teacher:[value:[ms_smith]],
                 student:[value:[kim,neena,reena],min:2,max:5]
                ]).
```

Upon performing:

```
|?- change_max(grade4,student,Max,10).
Max = 5
yes
```

In slot student, max is changed from 5 to 10.

6.13.4 return_max_from_slot(framename,slotname,max)

short_form **ret_max**(framename,slotname,max)

This operator allows the retrieval of the max facet of a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(person:[]).
?- def_frame(boy:[is_a:[value:[male]]]).
?- def_frame(girl:[is_a:[value:[female]]]).
?- def_frame(class:
                [teacher:[type:person],
            student:[type: boy @@ girl,
                     max:30]
           ]).
```

Upon performing:

```
|?- ret_max(class,student,Max).
Max = 30
yes
```

6.14 MIN FACET OPERATORS

Min facet contains the minimum number of values that a particular slot can have. A set of operators is provided to access and modify the min facet. These operators will add, remove, modify(change) and return the min facet. The system default for the min facet is zero.

6.14.1 add_min_to_slot(framename,slotname,min)

short_form **add_min**(framename,slotname,min).

This operator allows the addition of a min facet to a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(photosynthesis:[]).
?- def_frame(animal:
   [is_a:[value:[living_thing]],
    food_source:[type:(~photosynthesis), max:999]]).
```

Upon performing:

```
|?- add_min(animal,food_source,1).
Entries in value facet outside min max range.
There is a cardinality violation in slot food_source:[min:1, type: (~photosynthesis),
max:999]. In frame animal max = 999 and min =1.
New value of min results in cardinality violation.  Operation fails.
no

|?- add_val(animal,food_source,milk).
yes
|?- add_min(animal,food_source,1).
```

yes

After adding the value milk in the value facet of the slot food_source, min =1 does not violate the cardinality constraints.

6.14.2 remove_min_from_slot(framename,slotname,min)

short_form - **rem_min**(framename,slotname,min)

This operator allows the removal of the min facet from a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(photosynthesis:[]).
?- def_frame(animal:
   [is_a:[value:[living_thing]],
    food_source:[type:(~photosynthesis), max:999,
                 value:[milk,water,bread,meat,fruits],
                 min:5]
   ]).

?- def_frame(bird:
   [is_a:[value:[animal]],
    food_source: [value:[water,vegetables], min:2]]).
```

Upon performing:

```
| ?- rem_min(bird,food_source,Min).
Removing this Min = 2 would create cardinality violation in value facet [water,vegetables]
Operation fails.
no
```

```
| ?- rem_min(animal,food_source,Min).
Min = 5
yes
```

Removal of the min facet does not violate any constraints.

6.14.3 change_min_of_slot(framename,slotname,oldmin,newmin)

short_form - **change_min**(framename,slotname,oldmin,newmin)

This operator can be used to change the min facet oldmin of a specified slot in a specified frame with newmin.

Example

Suppose the knowledge base file contains:

```
?- def_frame(bird:
   [is_a:[value:[animal]],
    food_source: [value:[water,vegetables], min:2]
   ]).
```

Upon performing:

```
| ?- change_min(bird,food_source,Min,3).
Number of entries in valuefacet less than new min  Operation fails.
no

| ?- change_min(animal,food_source,Min,1).
Min = 2
yes
```

In slot food_source, min gets changed from 2 to 1.

6.14.4 return_min_from_slot(framename,slotname,min)

short_form **ret_min**(framename,slotname,min)

This operator allows the retrieval of the min facet of a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(bird:
       [is_a:[value:[animal]],
        food_source: [value:[water,vegetables], min:2]
       ]).
```

Upon performing:

```
| ?- ret_min(bird,food_source,Min).
Min = 2
yes
```

6.15 TYPE FACET OPERATORS

A set of operators are provided to access and modify the type facet. These operators will add, remove, modify (change) and return the type facet. The value in a type facet must be a legal type predicate. A legal type predicate is defined as follows:

a frame name
a legal type ## a legal type (ANDed types)
a legal type @@ a legal type (ORed types)
~ a legal type (NOT type)

6.15.1 add_type_to_slot(framename,slotname,type)

short_form - **add_type**(framename,slotname,type).

This operator allows the addition of a type facet to a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(body_covering_type:
        [description: [value:[possible_types_of_body_covering]]]).
?- def_frame(hair:
        [is_a:[value:[body_covering_type]]]).
?- def_frame(feathers:
        [is_a:[value:[body_covering_type]]]).
?- def_frame(black_feathers:
        [is_a:[value:[feathers]]]).
?- def_frame(crow:
        [is_a:[value:[bird]],
        body_covering:[value:[black_feathers]]
        ]).
```

Upon performing:

```
| ?- add_type(crow,body_covering,hair).
Adding type hair would create type violation in value facet [black_feathers] , Operation fails.
no

| ?- add_type(crow,body_covering,feathers).
yes
```

Type feathers does not violate any constraints.

6.15.2 remove_type_from_slot(framename,slotname,type)

short_form **rem_type**(framename,slotname,type)

This operator allows the removal of the type facet from a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(feathers:[]).
?- def_frame(black_feathers:
            [is_a:[value:[feathers]]]).
?- def_frame(crow:
            [is_a:[value:[bird]],
            body_covering:[value:[black_feathers],type:feathers]]).
```

Upon performing:

```
| ?- rem_type(crow,body_covering,hair).
Hair does not exist in type facet for slot body_covering in frame bird , Operation fails.
no
```

```
/ ?-  rem_type(crow,body_covering,feathers).
yes
```

Removal of the type facet does not violate any constraints.

6.15.3 change_type_of_slot(framename,slotname,oldtype,newtype)

short_form - **change_type**(framename,slotname,oldtype,newtype)

This operator can be used to change the type facet oldtype of a specified slot in a specified frame with newtype.

Example

Suppose the knowledge base file contains:

```
?- def_frame(feathers:[]).
?- def_frame(black_feathers:
        [is_a:[value:[feathers]]]).
?- def_frame(crow:
        [is_a:[value:[bird]],
        body_covering:[value:[black_feathers],type:feathers]
        ]).
```

Upon performing:

```
/ ?- change_type(crow,body_covering,feathers,hair).
Changing type with hair would create type violation in value facet [black_feathers]
Operation fails.
no

/ ?-  change_type(crow,body_covering,Type,feathers).
Type = feathers already exists in slot body_covering in frame crow, hence ignored.
```

```
no
/ ?-  change_type(crow,body_covering,Type,black_feathers).
Type = feathers
yes
```

In slot body_covering of frame crow, type gets changed from feathers to black_feathers.

6.15.4 return_type_from_slot(framename,slotname,type)

short_form **ret_type**(framename,slotname,type)

This operator allows the retrieval of the type facet of a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
?- def_frame(feathers:[]).
?- def_frame(black_feathers:
            [is_a:[value:[feathers]]]).
?- def_frame(crow:
            [is_a:[value:[bird]],
            body_covering:[value:[black_feathers],type:feathers]
            ]).
```

Upon performing:

```
/ ?-  ret_type(crow,body_covering,Answer).
Answer = feathers
yes
```

6.16 DEFAULT FACET OPERATORS

The default facet provides a list of values that is to be used for the slot's value if no values are present in the value facet. Once the value for a value facet is determined, then one might like to create or modify the value facet itself and at that point might want to remove the default facet. Hence, the operator remove_default is provided. change_default and add_default are not provided because conceptually there will be little need to use those functions. return_default is useful when one would like to find out what the default value is.

6.16.1 remove_default_from_slot(framename,slotname)

short_form **rem_default**(framename,slotname)

This operator allows the removal of a default facet from a specified slot in a specified frame.

Example

Suppose the knowledge base file contains:

```
? - def_frame(grade4:
          [is_a:[value:[class]],
           teacher:[default:[miss_maria]],
           student:[value:[jim,ken],min:2,max:30]]).
```

Suppose, that initially it was not decided who is going to teach grade4. Therefore, by default, it was assumed that Miss Maria will teach grade4. However, at the starting of the session, it was decided that Miss Jinnie will teach the class. In that case, one would like to do the following:

```
/ ?- add_val(grade4,teacher,miss_jinnie).
yes
/?-rem_default(grade4,teacher).
yes
```

6.16.2 return_default_from_slot(framename,slotname,default)

short_form **ret_default**(framename,slotname,default)

This operator allows the retrieval of a default facet from a specified slot in a specified frame. The default parameter is assumed to be an uninstantiated variable or an exact match of the default facet.

Example

Suppose the knowledge base file contains:

```
?- def_frame(grade4:
             [is_a:[value:[class]],
              teacher:[default:[miss_maria]],
              student:[value:[jim,ken],min:2,max:30]
             ]).
```

On performing:

```
| ?- ret_default(grade4,teacher,Default).
Default = [miss_maria]
yes
```

6.17 FRAME OPERATORS

6.17.1 add_frame(filename,frame)

This operator allows the addition of a new frame while working with the knowledge base from inside the frame package. filename is the name of the knowledge base file in which the user wants to put the new frame. The new frame gets appended at the end of the knowledge base file when the knowledge base file is saved. If the filename is specified as [], then that frame will be available only during that session and it will not be saved with any knowledge base file.

Example

```
| ?- add_frame(personalinfo,
                    joy:[is_a:[value:[kid]],
                       father:[value:[ron]],
                       mother:[value:[kim]]]).
```

Here personalinfo is the filename, in which the frame joy gets added to working memory.

The operation is very similar to the def_frame operation. There is a small difference however. def_frame is used when creating a knowledge base file whereas add_frame is used when new frames are added interactively during the session or through procedural knowledge. Since the add_frame operator specifies a filename, this allows the system to know and keep track of the fact that the particular frame belongs to a specified file.

6.17.2 rename_frame(oldframename,newframename)

This operator allows the frame oldframename to be renamed as newframename in working memory.

Example

Suppose the knowledge base file contains:

```
?- def_frame(bird:
       [is_a:
           [value:[animal]],
       body_covering:
           [value:[feather]]
       ]).
```

Upon performing:

```
| ?- rename_frame(bird,pretty_bird).
yes
```

Now the knowledge base file in working memory will have pretty_bird as the frame name instead of bird.

6.17.3 undef_frame(framename)

This operator deletes the frame framename from working memory.

Example

Suppose the knowledge base file contains:

```
?- def_frame(kitty:
       [is_a:
           [value:[cat]],
       body_covering:
           [value:[hair]]
       ]).
```

When kitty dies, one would like to update the knowledge base by deleting the frame kitty.

On performing:

```
/?- undef_frame(kitty).
yes.
```

Frame kitty will no longer be available in the knowledge base in working memory. However, it will remain in the knowledge base file in the working directory in which it currently resides.

6.17.4 frame_match_exact(prototype,list_of_frames)

This operator returns a list of the names of frames that match a given pattern. This pattern is defined by the argument prototype, which is considered to be a subset of slots in a matching frame. The prototype list determines what is to be searched for, and list_of_frames is the list of frames that matches the prototype list.

Prototypes are specified like the list of slots in a frame. However, only four facet types may appear in the list of facets that describe the slots in a prototype. The allowed facets are value, type, max and min. When they appear in a prototype, these facets are interpreted differently from when they appear in a frame or slot definition. The value specification indicates a value that must be matched literally. A type specification in a prototype's slot indicates that the value in the corresponding slot in any frame must meet this type constraint. (Note that it does not mean that the matching frame's slot's type facet matches the prototype's slot's type facet). The min and max specifications similarly define a cardinality requirement for the slot's value match. For example, if the min facet's value is three and the max facet's value is five in a given prototype slot, only frames with a list containing three, four or five values in the value facet could match.

In frame_match_exact operator, the list of values in the value facet of the frame's slot must be identical to those in the prototype list.

Example

Suppose the knowledge base file contains:

```
?- def_frame(carnivorous:
       [is_a:
           [value:[food_source_type]],
        characteristic_of :
           [value:[cats,dogs],
            max:999],
        description:
           [value:[eats_animal]]
       ]).
```

Upon performing:

```
|?-frame_match_exact([characteristic_of:[value:[dogs]]],Answer).
no                           /* value match is not exact */
|?- frame_match_exact([characteristic_of:[value:[dogs,cats]]],Answer).
Answer= [carnivorous]
yes
```

6.17.5 frame_match_subset(prototype,list_of_frames)

frame_match_subset operator is the same as frame_match_exact except that it will match frames whose list of values in the value facet of a slot is a superset of the list of values in a value facet of the prototype list.

Example

Suppose the knowledge base file contains:

```
?- def_frame(carnivorous:
        [is_a:
            [value:[food_source_type]],
         characteristic_of:
            [value:[cats,dogs],
             max:999],
         description:
            [value:[eats_animal]]]).
```

Upon performing:

```
| ?- frame_match_subset([characteristic_of:[value:[dogs]]],Answer).
Answer=[carnivorous]
yes
```

6.17.6 frame_subsumes(subsuming_frame,subsumed_frame)

frame_subsumes operator determines the relationship among frames in a hierarchy, where subsuming_frame is the potential ancestor frame and subsumed frame is the potential progeny frame. This operator succeeds if the first argument subsumes the second argument.

Example

Suppose the knowledge base file contains:

```
?- def_frame(animal:
        [is_a:
            [value:[living_thing]]
        ]).

?- def_frame(mammal:
        [is_a:
            [value:[animal]],
         body_covering:
            [value:[hair]]
        ]).

?- def_frame(bird:
        [is_a:
            [value:[animal]],
         body_covering:
            [value:[feather]]
        ]).
```

Upon Performing:

```
|?- frame_subsumes(animal,bird).
yes                     /* to determine bird is an animal */

|?- frame_subsumes(animal,Answer).
Answer = mammal;
Answer = bird;
no              /* to find all the frames that are progeny of frame animal */

|?- frame_subsumes(Answer,bird).
Answer = animal;
yes           /* to find ancestor frame of frame bird */
```

6.18 KNOWLEDGE ANALYSIS OPERATORS

The enhanced frame package provides two commands to analyze the knowledge base. Both of these commands tell the user only about inconsistencies that are present in the system. They do not interfere with anything else and let the user work with the system as it stands (whether consistent or inconsistent).

6.18.1 ka

ka, i.e. knowledge analyzer, analyzes the complete knowledge present in working memory to check for consistency. It reports all the inconsistencies that it encounters during its analysis. (use this command to find out what it can do!)

6.18.2 check_frame(framename)

ka goes through the whole knowledge base, whereas check_frame operator checks only a particular frame framename for inconsistencies. This operator was created for efficiency purposes when only a selected frame needs to be analyzed instead of the complete knowledge base.

Example

```
/ ?- check_frame(animal).
```

6.19 UTILITY OPERATORS

Some other useful operators available in the enhanced frame package are:

6.19.1 trace_on/trace_off

The trace feature helps the user in debugging and understanding the knowledge base. The trace provided by the prolog interpreter turns out to be very detailed and difficult to understand because it traces through the details of the source code of the frame package. Since the user can best relate to the knowledge base he/she has created/used, a new trace feature has been developed for the enhanced frame package. This trace confines itself to high level frame package operators as they relate to the user's knowledge base.

Normally trace is kept off. The user can ask for the trace on by typing trace_on, in which case the package traces and displays the decision path for the user. All comments that are displayed as part of the tracing operation are preceded by an arrow '-->'.

If the trace is on during any loading function (e.g. load, file_create, file_edit ,etc.) the number of messages displayed becomes very large and is not necessarily meaningful. If the user leaves the trace on before any of these operations, the package recommends that the user turn the trace off.

Once the user puts trace_on, trace will be on until the user puts trace_off again and vice versa.

Example

Suppose the knowledge base file contains:

```
?- def_frame(animal:
        [is_a:
            [value:[living_thing]],
         food_source:
                [value:[milk,water,bread],
                 min:5],
            description:
                [value:[partition_of_living_thing]]
            ]).
?-  def_frame(bird:
            [is_a:
                [value:[animal]],
             body_covering:
                [value:[feathers]]
            ]).
```

On performing:

```
/ ?- trace_on.
yes
/ ?- ret_val(animal,description,Value).
--> value = [partition_of_living_thing] inherited  from slot description in frame animal
Value = [partition_of_living_thing]
yes
```

6.19.2 print_tree(framename)

Given the framename, this operator prints the immediate parent and children frame names. This can be a good help in the debugging process. The package does not go beyond the immediate parents and children for the sake of simplicity.

Example

Suppose the knowledge base file contains:

```
?-  def_frame(living_thing:[]).
?-  def_frame(animal:
```

```
            [is_a:
                  [value:[living_thing]]
            ]).
   ?- def_frame(mammal:
            [is_a:
                  [value:[animal]],
             body_covering:
                  [value:[hair]]
            ]).
   ?- def_frame(bird:
            [is_a:
                  [value:[animal]],
             body_covering:
                  [value:[feathers]]]).
```

Upon performing:

```
/?-print_tree(animal).

PARENT FRAME(S):
              living_thing
QUERY FRAME:
              animal
CHILD FRAME(S):
              mammal
              bird
```

6.19.3 print_frame(framename).

short_form **pf(framename)**

The print_frame operator prints the frame framename that is present in the working memory in a well organized format. This operator comes in handy especially to look at a frame after modifications have been made using the frame package operators.

Example

Suppose the knowledge base file contains:

```
?- def_frame(bird:[is_a:[value:[animal]], body_covering:[value:[feathers]]]).
```

Upon performing:

```
|?-pf(bird).
bird:
   [
   is_a:
       value:[animal]
   body_covering:
       value:[feathers]
   ]
yes
```

6.19.4 show_all

This operator shows all the knowledge base file names along with a list of frame names that are present in working memory for each one of the files. A slotname, if defined through a def_slot operator, also appears as a list entry in the list of frame names.

Example

Suppose the knowledge base file **parent** contains:

```
?- def_slot(children:
            [min:0, max:5]]).
?- def_frame(jim:
        [is_a:[value:[father]],
        children:[value:[annie,kannie]],
        wife:[value:[kati]]
        ]).
?- def_frame(johny:
        [is_a:[value:[father]],
        children:[value:[minnie]],
        wife:[value:[kavita]]]).
```

Suppose the knowledge base file **kid** contains:

```
?- def_frame(male:[is_a:[value:[living_thing]]]).
?- def_frame(female:[is_a:[value:[living_thing]]]).
?- def_frame(leather:[is_a:[value:[material]]]).
?- def_slot(has_leather_shoes:
            [min:0, max:5,type:leather]]).
?- def_frame(father:
        [is_a:[value:[parent],
              type:[male]]
        ]).
?- def_frame(mother:
        [is_a:[value:[parent],
              type:[female]]
        ]).
?- def_frame(annie:
        [father:[value:[jim]],
        mother:[value:[kati]]
        ]).
?- def_frame(kannie:
        [has_leather_shoe:[value:[white_italian]],
        father:[value:[jim]],
        mother:[value:[kati]]]).
?- def_frame(minnie:
        [father:[value:[johny]],
```

mother:[value:[kavita]]]).

Upon performing:

```
/ ?-show_all.
File --> parent          Frames/Slots --> [[children],jim,johny]
File-->kid               Frames/Slots-->[male,female,leather,[has_leather_shoes], father,
                                  mother,annie,kannie, minnie]
yes
```

Here children and has_leather_shoes are defined through def_slot operator.

6.19.5 show_file(filename)

This operator shows names of all the frames and slots that belong to the file filename present in working memory.

Example

Suppose the knowledge base file parent is declared as above in Section 6.19.4.

Upon performing:

```
/ ?- show_file(parent).
File --> parent              Frames/Slots --> [[children],jim,johny]
yes
```

6.20 DEMON FILE OPERATORS

A set of operators are available for the demon predicate file.

6.20.1 load_demon(filename)

This operator is used to load a demon predicate file in the frame package.

Example

```
/ ?- load_demon(ak_demon).
```

Upon execution, demon file ak_demon will be consulted into working memory.

To load a demon file, one must use the load_demon operator or simply consult the demon file and to load a knowledge base file one should always use the load operator (Section 6.3).

6.20.2 edit_demon(filename)

This operator allows the user to edit a demon predicate file from inside the enhanced frame package.

Example

```
/ ?- edit_demon(ak_demon).
```

Upon execution, demon file ak_demon will be displayed for editing.

6.20.3 create_demon(file)

This operator allows the creation of a demon predicate file from inside the frame package.

Example

```
/ ?- create_demon(ak2_demon).
```

Upon execution, demon file ak2_demon will be displayed for creation.

While Writing Demons ....

PLEASE NOTE :

1. If the user is creating new frames using procedural knowledge in demon predicates, then the user must use add_frame instead of def_frame (because through procedural knowledge, new frames are being added as opposed to defining a frame at the time of creation of the knowledge base). Reference: Appendix D.

2. The user must keep in mind while writing the demon predicates that the enhanced package allows addition, removal or modification of a single value as well as a list of values. Reference: Appendix D.

7 FUNCTIONAL DETAILS

This chapter covers the additional technical details that include the functionality, criteria and constraint checking aspects of the various frame package operators; complementing the information provided in the previous chapter.

The order of the operators in this chapter has been kept the same as the order of operators in Chapter 6. It was not kept alphabetical for the reason of being able to group them together by their functionality. However, an index of operators has been provided at the back of this thesis for a quick alphabetical reference.

7.1 Definition Operators

Definition operators - def_frame and def_slot are commands not just the simple prolog clauses in the frame package. Definition operators therefore must be preceded by a ?- in any knowledge base file.

7.1.1 def_frame(Frame)

Functionality:

(1) Check to assure that the argument is instantiated.

(2) Check the frame for correct syntax by assuring all of the following:

- the frame name is unique.
- the structure of the frame is:

    framename:[list of slots]
- the list of slots is indeed a list.
- the list of facets for each slot is indeed a list.
- the structure of each slot is:

    slotname:[list of facets]
- the structure of each facet is:

facetname:facetvalue
- the facetvalue is an atom, a list, or a type predicate
- the facetname is one of the predefined names.
- the type facet is one of the following legal types:
  a frame name (legal type)
  legal type && legal type
  legal type $$ legal type
  ~legal type
- the max and min facets, if they exist,
  contain a single integer
- the value and default facets, if they exist,
  contain lists

(3) Check the frame for correct values by assuring the following:
(applicable only in the case of a consistent system option)
- type constraints have not been violated.
  (all data in value and default facets are checked against any type constraints that exist)
- cardinality constraints have not been violated
  (all data in value and default facets are checked against max and min constraints)

(4) Assert the frame into the knowledge base or return an error message and fail.

7.1.2 def_slot(slotname:[list of facets]).

<u>Functionality:</u>

(1) Check to assure that the argument is instantiated.

(2) Check the slotname is unique among the predefined slots.

(3) Check the slot for the correct syntax by assuring the following:

- the slot has at least one facet.
- the structure of each slot is:
  slotname:[list of facets]
- the list of facets for each slot is indeed a list.
- the structure of each facet is:
  facetname:facetvalue
- the facetvalue is an atom, a list, or a type predicate
- the facetname is one of the predefined names.
- the type facet is one of the following legal type:
  a frame name
  legal type && legal type
  legal type $$ legal type
  ~legal type
- the max and min facets, if they exist,
  contain a single integer
- the value and default facets, if they exist,
  contain lists

(4) Check the slot for correct values by assuring the following:
(applicable only in the case of a consistent system option)
- type constraint have not been violated.
  (all data in value and default facets are checked
  against any type constraints that exist)
- cardinality constraints have not been violated
  (all data in value and default facets are checked
  against max and min constraints)

(5) Assert the slot into the knowledge base or return an error message and fail.

7.2 Demons

Functionality:

(1) - a demon in the if_added facet will be in invoked when a value is added to a slot
- a demon in the if_removed facet will be in invoked when a value is removed from a slot

a demon in the if_needed facet will be in invoked when a value is needed but none is present in the slot
- a demon in the if_changed facet will be in invoked when a value is changed from a slot

(2) When a demon is invoked, arguments are sent automatically in the following order for the first three facets listed above:
- user-defined arguments (if present), framename, slotname and value

the frame name and the slot name arguments are passed in addition to the value argument to identify the source which activated the demon.

For the fourth facet i.e. for the if_changed facet, the following arguments are sent automatically:
- user-defined arguments (if present) framename, slotname, oldvalue, and newvalue

where the additional argument, oldvalue is the old value in the slot slotname of the frame framename that is getting changed by the new value newvalue.

7.3 load(filename)

Functionality:

(1) Assure that filename is an instantiated atom.

(2) If the file filename does not exist in the directory, report an error and fail.

(3) Load the frames and slots from the file filename into working memory based upon the following criteria:

```
if the user started with the choice of a consistent system
    then perform syntax and consistency checking both
    else perform syntax checking only.
```

(4) Update the internal information maintained by the package regarding the knowledge base.

7.4 file_create(filename)

Functionality:

(1) Assure that filename is an instantiated atom.

(2) If the file filename already exists in the working directory, report an error and fail.

(3) Provide edit mode for the editing and the creation of file filename.

(4) At the end of creation, load the frames and slots from the file filename into working memory based upon the following criteria:

```
if the user started with the choice of a consistent system
    then perform syntax and consistency checking both
    else perform syntax checking only.
```

(5) Save the file filename in the working directory also.

(6) Update the internal information maintained by the package regarding the knowledge base.

7.5 file_edit(filename)

Functionality:

(1) Assure that filename is an instantiated atom.

(2) If the file filename does not exist in working memory, report an error and fail.

(3) Delete all the frames and slots that belong to the knowledge base file filename from working memory.

(4) Display the file filename as it exists in the working directory for editing using the vi editor.

(5) At the end of editing, save the file filename in the working directory and load the frames and slots from the file filename into working memory based upon the following criteria:

```
if the user started with the choice of a consistent system
    then perform syntax and consistency checking both
    else perform syntax checking only
```

(6) Update the internal information maintained by the package regarding the knowledge base.

The file_edit operator updates the knowledge base file in the working directory!

7.6 purge(filename)

<u>Functionality:</u>

(1) Assure that filename is an instantiated atom.

(2) If the knowledge base file filename does not exist in working memory, report an error and fail.

(3) Retract all the frames and slots of the knowledge base file filename from working memory.

(4) Update the internal information maintained by the package regarding the knowledge base.

7.7 reload(filename)

Functionality:

(1) Assure that filename is an instantiated atom.

(2) If the knowledge base file filename does not exist in working memory, report an error and fail.

(3) Retract all the frames and slots of the knowledge base file filename from working memory.

(4) Load the knowledge base file filename from the working directory in the same sequential position with respect to the other loaded knowledge base files in working memory based upon the following criteria:

```
if the user started with the choice of a consistent system
    then perform syntax and consistency checking both
    else perform syntax checking only.
```

(5) Update the internal information maintained by the package regarding the knowledge base.

The **reload** operator replaces the contents of the knowledge base file while maintaining the existing sequential position of the file in the working memory; where as the **load** operator always loads the knowledge base file at the end of the sequence of all the loaded files.

7.8 temp_edit(filename)

Functionality:

(1) Assure that filename is an instantiated atom.

(2) If the knowledge base file filename is not present in working memory, report an error and fail.

(3) Copy the frames and slots of the file filename from working memory into a storage file say temp2 in a remote directory (needed for an undo option).

(4) Copy the contents of the file filename from working memory into another storage file say temp in a remote directory for editing using def_frame and def_slot operators.

(5) Purge the contents of the file filename from working memory.

(6) Display file temp for editing.

(7) At the end of editing, load the edited file 'temp' based upon the following criteria:

```
if the user started with the choice of a consistent system
    then perform syntax and consistency checking both
    else perform syntax checking only.
(for frames and slots defined in the temporary file temp.)
```

Keep in mind that temp_edit operates on working memory. It does not change the original knowledge base file in the working directory. In step 7, loading is done in such a way that valid/correct frames and slots get loaded as part of the knowledge base file filename in working memory. Step 7 also lists all the errors that are present in the edited version.

(8) Provide a list of options to the user as follows:

1. exit edit

2. undo edit
3. continue edit

(9) Option 1 exit edit. The file edit operation ends and the edited version is to be used for the file filename in working memory. Since the edited version is already loaded in working memory (step 7), nothing further needs to be done.

(10) Option 2 undo edit. The file edit operation should be undone, i.e. the system should return to the status as it was just before the editing was started. To achieve this, the following steps are executed:

a. Purge all frames/slots from working memory that were loaded as part of step 7.
b. Consult file temp2.

(11) Option 3 continue edit. The file that was just being edited is redisplayed for further editing. To achieve this, the following steps are executed:

a. Purge all frames/slots from working memory that were loaded as part of step 7.
b. Display file temp (edited version) for further editing.
c. Go back to step 6, above.

(12) At the end of the editing session, update the internal information maintained by the package regarding the knowledge base.

7.9 frame_edit(framename)

Functionality:

functionality of the frame_edit operator is similar to the temp_edit operator except that it allows editing of a particular frame instead of a particular file.

7.10 save_file(filename,newfilename)

Functionality:

(1) Assure that the filename and newfilename both are instantiated atoms.

(2) If file filename is not present in working memory, report an error and fail.

(3) If file filename does not have any frames or slots, display a message and fail.

(4) If the newfilename already exists in the working directory, provide an option to the user for replacing it with the contents of the file filename present in working memory.

(5) If the file newfilename does not exist in the working directory, create a new file newfilename in the working directory.

(6) Copy the contents of file filename from working memory into the file newfilename in working directory.

7.11 save_it(filename)

Functionality:

functionality of the save_it operator is similar to the save_file operator except that newfilename in save_file is the same as filename in save_it.

7.12 VALUE FACET OPERATORS

7.12.1 add_value_to_slot(framename,slotname,value)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the value to be added is not valid (ie not a list or an atom), report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) Remove any duplication in the list of values to be added or if the value to be added already exists in the value facet.

(5) If possible (see the criteria below), add the value by retracting the old frame and asserting the frame with the amended value.

(6) If the addition was executed, activate any if_added demon(s), if present, in the specified slot.

Criteria:

```
if the slot is local to the frame and has a value facet
   then if constraints allow (see constraint checking below)
       then add the value to the list
       else report an error and fail
   else if the slot is local and does not have a value facet
           then if constraints allow
               then put the value in the value facet
               else report an error and fail
           else the slot is not local
               then if constraints allow
                   then add the slot to the list
```

```
                    and put the value in the value facet
                else report an error and fail
```

Constraint Checking:

check the value argument by assuring the following:

```
cardinality constraints have not been violated
    (number of values present in the value facet along with
     number of values to be added (Value argument) is checked
     against the max constraint)

type constraint has not been violated.
    (value(s) to be added is checked against
     any type constraints that exist.)
```

This operator adds to local information only or creates local information if there was none. This allows for local information to override inherited information.

7.12.2 remove_value_from_slot(framename,slotname,value)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the frame does not exist, report an error and fail.

(3) Remove any duplication in the list of values to be removed.

(4) If possible (see the criteria below), remove the value by retracting the old frame and asserting the amended frame.

(5) If the removal was executed, activate any if_removed demon(s), if present, in the

specified slot.

Criteria:

```
if the slot is local to the frame and has a value facet
    then if the value(s) to be removed is present
    (in case of an uninstantiated variable as value argument, first entry
    of the value facet becomes the value argument)
        then if constraints allow (see below)
            then remove the value
            else report an error and fail
        else fail
    else fail
```

Constraint Checking:

```
check the value argument by assuring the following
    cardinality constraints have not been violated
        (number of values present in the value facet minus
         number of values to be removed (Value argument) is checked
         against the min constraint)
```

Removal of a value(s) takes place only if the value is present locally in a slot. If the removal of a value produces an empty value facet, the facet is removed from its slot. If this removal of the value facet produces an empty facet list, then that slot is removed from the frame.

7.12.3 remove_all(framename,slotname)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the frame does not exist, report an error and fail.

(3) If possible (see the criteria below), remove the value facet by retracting the old frame and asserting the amended frame.

(4) If the removal of the value facet was executed, activate any if_removed demon(s), if present, in the specified slot.

Criteria:

```
if the slot is local to the frame and has a value facet
    then if constraints allow (see below)
        then remove the value
        else report an error and fail
    else fail
```

Constraint Checking:

```
check the following:
    cardinality constraints have not been violated
    (min must be zero for that slot)
```

If this removal of the value facet produces an empty facet list, then that slot is removed from the frame.

7.12.4 change_value_of_slot(framename,slotname,oldvalue,newvalue)

Functionality:

(1) Assure that framename, slotname and newvalue arguments are instantiated.

(2) If the value to be removed (oldvalue) is not an atom or a list of atoms, report an error and fail.

(3) If the value to be added (newvalue) is not an atom, a list of atoms, or an

uninstantiated variable, report an error and fail.

(4) If the frame does not exist, report an error and fail.

(5) Remove the duplication if present in the oldvalue and newvalue arguments.

(6) If possible (see the criteria below), change the values by replacing oldvalue with newvalue, and retracting the old frame and asserting the frame with the changed value(s).

(7) If the change was executed, activate the if_changed demon(s), if present, in the specified slot.

Criteria:

```
if the slot is local to the frame and has a value facet
    then if constraints allow (see below)
        then change the value of the value facet
        else report an error and fail
    else fail
```

Constraint Checking:

```
check the oldvalue argument by assuring the following:
    value(s) to be removed should be present in the value facet
        (if oldvalue is an instantiated variable, it must be present in the value facet.
        In case of an uninstantiated variable as oldvalue argument, the first entry
        of the value facet becomes the oldvalue)

check the new argument by assuring the following:
    type constraints have not been violated
        (newvalue is checked against any type constraint that is present)

once the above two conditions are satisfied, ensure that the cardinality
```

constraints have not been violated
(number of values present in the value facet plus
number of entries in newvalue minus number of entries in oldvalue
should be greater than or equal to min and
less than or equal to the max constraint for that slot)

The change_value_of_slot operates only when the slot has a local value facet. If oldvalue argument is [], then the change_value_of_slot will change [] to the newvalue; which is equivalent of adding newvalue to the valuefacet. Similarly, if newvalue argument is [], then the change_value_of_slot will change oldvalue to []; which is equivalent of removing oldvalue from the value facet. In all cases if_changed demon will be fired if present.

If the change of a value produces an empty value facet, the facet is removed from its slot. If this removal of the value facet produces an empty facet list, then that slot is removed from the frame.

7.12.5 change_all(framename,slotname,newvalue)

Functionality:

(1) Assure that framename, slotname and newvalue arguments are instantiated.

(2) If the value to be added (newvalue) is not an atom, a list of atoms, or an uninstantiated variable, report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) Remove any duplicate entries in the newvalue argument.

(5) If possible (see the criteria below), change the value facet by retracting the old frame and asserting the frame with the newvalue.

(6) If the change was executed, activate any if_changed demon(s), if present, in the

specified slot.

Criteria:

```
if the slot is local to the frame and has a value facet
   then if constraints allow(see below)
       then change the value of the value facet
       else report an error and fail
   else fail
```

Constraint Checking:

```
check the new argument by assuring the following:
   type constraints have not been violated
       (newvalue is checked against any type constraint that is present)

ensure that the cardinality constraints have not been violated
   (number of unique entries present in the newvalue >= min and <= max
    constraints for that slot)
```

change_all operates only when the slot has a local value facet.

7.12.6 return_value_from_slot(framename,slotname,value)

Functionality:

(1) Framename and slotname must be instantiated variables.

(2) Value argument must be an uninstantiated variable or an exact match of the value facet.

(3) If the frame does not exist, report an error and fail.

(4) If a value facet can be found based upon the following value facet inheritance algorithm, return that value facet.

(5) If a value facet cannot be found, fail.

The <u>Value Facet Inheritance Algorithm:</u>

```
if the value facet is local to the current slot
    then use that value
    else if the current slot contains a default facet
        then use that value
        else if the current slot contains an if_needed facet
            then activate the demon(s)
                and return the value returned by the last demon
            else search the ancestors recursively*
                {if a value can be found
                    then use it}   /*using the same criteria as above in the algorithm*/
                    else if the value facet is in a predefined slot
                        then use that value
                        else there is no value
```

*This search is done as a breadth-first search on all ancestors, so that the "nearest" ancestor supplies the value.

7.13 MAX FACET OPERATORS

7.13.1 add_max_to_slot(framename,slotname,max)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the max to be added is not valid (ie not an integer), report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) If possible (see the criteria below), add the max by retracting the old frame and asserting the frame with the amended max.

Criteria:

```
if the slot is local to the frame and has no max facet
  then if constraints allow (see constraint checking below)
    then add the max facet to the slot
    else report an error and fail
  else if the slot is local and has a max facet
       then report an error and fail
       else if the slot is not local
         then if constraints allow
           then create the slot to the list and add the max facet
           else report an error and fail
```

Constraint Checking:

```
check the value argument by assuring the following:
  cardinality constraints have not been violated for value/default facet(s) if they exist
    (number of entries in the value facet and/or default facet(s) must be <= max)
```

It is worth noting that the max facet can have only one entry as its value; therefore, add_max succeeds only if there is no local max facet.

7.13.2 remove_max_from_slot(framename,slotname,max)

Functionality:

(1) Assure that the framename and slotname are instantiated.

(2) If the frame does not exist, report an error and fail.

(3) If the max to be removed is not valid (i.e. neither an uninstantiated variable nor an integer), report an error and fail.

(4) If possible (see the criteria below), remove the max by retracting the old frame and asserting the amended frame.

Criteria:

```
if the slot is local to the frame and has a max facet
    then if constraints allow (see below)
        then remove the max facet from the slot
        else report an error and fail
    else fail
```

Constraint Checking:

```
check the max argument by assuring the following:
    if max argument is an instantiated variable, then it must exist in the max facet.
```

cardinality constraints have not been violated
(number of entries in the value facet (if it exists), and in default facet (if it exists) must be <= the new max. New max will be either inherited from an ancestor

frame or a predefined global slot or the system default.)

Since the max facet can have only one entry, rem_max removes the max facet completely from the specified slot. If this removal of the max facet produces an empty facet list, then that slot is removed from the frame.

7.13.3 change_max_of_slot(framename,slotname,oldmax,newmax)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the oldmax is not valid (i.e. not an integer), report an error and fail.

(3) If the newmax is not valid(i.e. not an integer), report an error and fail.

(4) If the frame does not exist, report an error and fail.

(5) If possible (see the criteria below), change the oldmax by retracting the old frame and asserting the frame with the newmax.

Criteria:

```
if the slot is local to the frame and has a max facet
    then if constraints allow(see below)
        then change the max of the max facet
        else report an error and fail
    else report an error and fail
```

Constraint Checking:

if oldmax is an instantiated variable, it should be the same as max facet entry of the slot.

ensure that the cardinality constraints have not been violated
(number of values present in the value facet (if it exists) and in default facet
(if it exists), must be <= newmax)

change_max_of_slot operates only when the slot has a local max facet.

7.13.4 return_max_from_slot(framename,slotname,max)

<u>Functionality:</u>

(1) The framename and slotname must be instantiated variables.

(2) The max must be either an uninstantiated variable or an integer.

(3) If the frame does not exist, report an error and fail.

(4) If a max can be found based upon the following max facet inheritance algorithm, return that max.

(5) If a max cannot be found, fail

The <u>Max Facet Inheritance Algorithm:</u>

```
if the max facet is local to the current slot
   then use that max
   else search the ancestors recursively*
      { if a max can be found
          then use it }
      else if the max facet is in a predefined global slot
          then use that max
          else use system default max
```

* This search is done in a breadth first search manner.

7.14 MIN FACET OPERATORS

7.14.1 add_min_to_slot(framename,slotname,min)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the min to be added is not valid (i.e. not an integer), report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) If possible (see the criteria below), add the min by retracting the old frame and asserting the frame with the amended min.

Criteria:

```
if the slot is local to the frame and has no min facet
  then if constraints allow (see constraint checking below)
    then add the min facet to the slot
    else report an error and fail
  else if the slot is local and has a min facet
       then report an error and fail
       else if the slot is not local
         then if constraints allow
           then create the slot to the list and add the min facet
           else report an error and fail
```

Constraint Checking:

check the value argument by assuring the following:

cardinality constraints have not been violated for value and default facets if they exist
  (number of entries in the value facet and/or default facet must be >= min.)

It is worth noting that the min facet can have only one entry as its value; therefore, add_min succeeds only if there is no local min facet.

7.14.2 remove_min_from_slot(framename,slotname,min)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the frame does not exist, report an error and fail.

(3) If the min to be removed is not valid (i.e. neither an uninstantiated variable, nor an integer), report an error and fail.

(4) If possible (see the criteria below), remove the min by retracting the old frame and asserting the amended frame.

Criteria:

```
if the slot is local to the frame and has a min facet
    then if constraints allow (see below)
        then remove the min facet from the slot
        else report an error and fail
    else fail
```

Constraint Checking:

```
check the min argument by assuring the following:
    if min argument is an instantiated variable, then it must exist in the min facet.
    cardinality constraints have not been violated
        (number of entries in the values facet (if it exists), and in default facet (if it
         exists),  is checked against the new min.
        new min will be either inherited from the ancestor frame or a predefined global
```

slot or the system default.)

Since min facet can have only one entry, rem_min removes the min facet completely from the specified slot. If this removal of the min facet produces an empty facet list, then that slot is removed from the frame.

7.14.3 change_min_of_slot(framename,slotname,oldmin,newmin)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the oldmin is not valid (i.e. not an integer), report an error and fail.

(3) If the newmin is not valid(i.e. not an integer), report an error and fail.

(4) If the frame does not exist, report an error and fail.

(5) If possible (see the criteria below), change the oldmin by retracting the old frame and asserting the frame with the newmin).

Criteria:

```
if the slot is local to the frame and has a min facet
    then if constraints allow(see below)
        then change the min of the min facet
        else report an error and fail
    else report an error and fail
```

Constraint Checking:

if oldmin is an instantiated variable, it should be the same as min facet entry of the slot.

```
ensure that the cardinality constraints have not been violated
    (# of values present in the value facet (if it exists) and in default facet
    (if it exists), must be >= newmin)
```

change_min_of_slot operates only when the slot has a local min facet.

7.14.4 return_min_from_slot(framename,slotname,min)

Functionality:

(1) Framename and slotname must be instantiated variables.

(2) Min must be either an uninstantiated variable or an integer.

(3) If the frame does not exist, report an error and fail.

(4) If a min can be found based upon the following min facet inheritance algorithm, return that min.

(5) If a min cannot be found, fail

The Min Facet Inheritance Algorithm:

```
if the min facet is local to the current slot
    then use that min
    else search the ancestors recursively*
        { if a min can be found
            then use it }
        else if the min facet is in a predefined global slot
            then use that min
            else use system default min
```

* This search is done in a breadth first search manner.

7.15 TYPE FACET OPERATORS

7.15.1 add_type_to_slot(framename,slotname,type)

Functionality:

(1) Assure that framename, slotname and type are instantiated.

(2) If the type to be added is not valid (ie a legal predicate), report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) If possible (see the criteria below), add the type by retracting the old frame and asserting the frame with the amended type.

Criteria:

```
if the slot is local to the frame and has no type facet
   then if constraints allow (see constraint checking below)
      then add the type facet to the slot
      else report an error and fail
   else if the slot is local and has a type facet
      then report an error and fail
      else if the slot is not local
          then create the slot to the list
              and add the type facet
          else report an error and fail
```

Constraint Checking:

```
check the value and default argument by assuring the following:
   type constraint has not been violated
      (entries in value facet (if it exists) and in type facet (if it exists),
       must satisfy the new type constraint)
```

It is worth noting that the type facet can have only one entry as its value, therefore add_type succeeds only if there is no local type facet.

7.15.2 remove_type_from_slot(framename,slotname,type)

<u>Functionality:</u>

(1) Assure that framename and slotname are instantiated.

(2) If the type to be removed is not valid (ie neither an uninstantiated variable, nor an integer), report an error and fail.

(3) If the frame does not exist, report an error and fail.

(4) If possible (see the criteria below), remove the type by retracting the old frame and asserting the amended frame.

<u>Criteria:</u>

```
if the slot is local to the frame and has a type facet
    then if constraints allow (see below)
        then remove the type facet from the slot
        else report an error and fail
    else fail
```

<u>Constraint Checking:</u>

```
ensure the type constraint has not been violated:
    entries in the value facet and in default facet  (if they exist), must
    satisfy the new type constraint if present.  New type constraint
    will be either inherited from the ancestor frame or a predefined
    slot.
```

Since type facet can have only one entry, rem_type removes the type facet completely from the specified slot. If this removal of the type facet produces an empty facet list, then that slot is removed from the frame.

7.15.3 change_type_of_slot(framename,slotname,oldtype,newtype)

Functionality:

(1) Assure that framename and slotname are instantiated.

(2) If the newtype is not valid (ie not a legal predicate or a combination of that), report an error and fail.

(3) If the oldtype is not valid (neither an uninstantiated variable nor a legal predicate), report an error and fail.

(4) If the frame does not exist, report an error and fail.

(5) If possible (see the criteria below), change the oldtype by retracting the old frame and asserting the frame with the newtype).

Criteria:

```
if the slot is local to the frame and has a type facet
    then if constraints allow(see below)
        then change the type of the type facet
        else report an error and fail
    else report an error and fail
```

Constraint Checking:

```
check the oldtype argument by assuring the following:
    if oldtype is an instantiated atom, it must be present as the
    type facet entry.  (in case of an uninstantiated variable, type facet
    entry becomes the oldtype)

check the newtype argument by assuring the following:
    type constraint has not been violated
    (existing entries in value facet and default facet are
     checked against the newtype)
```

7.15.4 return_type_from_slot(framename,slotname,type)

Functionality:

(1) Assure that framename and slotname must be instantiated variables.

(2) If the frame does not exist, report an error and fail.

(3) If a type can be found based upon the following type facet inheritance algorithm, return that type.

(4) If a type cannot be found, fail

The Type Facet Inheritance Algorithm:

```
if the type facet is local to the current slot
    then use that type
    else
        search the ancestors recursively*
            { if a type can be found
                then use it }
            else if the type facet is in a predefined slot
                then use that type
```

* This search is done in a breadth first search manner.

7.16 DEFAULT FACET OPERATORS

7.16.1 remove_default_from_slot(framename,slotname,default)

Functionality:

(1) Assure that the framename and slotname are instantiated.

(2) If the frame does not exist, report an error and fail.

(3) Remove the default facet by retracting the old frame and asserting the amended frame without the default facet.

If the removal of the default facet produces an empty facet list, then the slot automatically gets removed from the frame.

7.16.2 return_default_from_slot(framename,slotname,default)

Functionality:

(1) Assure that framename and slotname must be instantiated variables.

(2) Default argument must be an uninstantiated variable or an exact match of the default facet.

(3) If the frame does not exist, report an error and fail.

(4) If a default can be found based upon the following default facet inheritance algorithm, return that default.

(5) If a default cannot be found, fail

The <u>Default Facet Inheritance Algorithm:</u>

```
if the default facet is local to the current slot
    then use that default
    else search the ancestors recursively*
        { if a default can be found
            then use it }
            else if the default is present in the predefined slot
                then use that default
                else there is no default
```

*This search is done as a breadth-first search on all ancestors.

7.17 FRAME OPERATORS

7.17.1 add_frame(filename, frame)

Functionality:

(1) Assure that the filename and frame both arguments are instantiated.

(2) If the frame already exists, report an error and fail.

(3) Check the frame for the correct syntax as performed in def_frame operator (Section 7.1.1).

(4) If started the session with the consistent system option, perform constraints checking as performed in def_frame operator(Section 7.1.1).

(5) If the filename is [], assert the frame into the working memory temporarily in a file by the name just_for_session. just_for_session file is internal to the frame package.

(6) Else assert the frame at the end of the knowledge base file filename in the working memory.

(7) If the file filename does not exist, create a new file filename in the working memory and assert the frame in that file.

(8) If the frame gets added, update the internal information related to the knowledge base file and the frame.

7.17.2 rename_frame(oldframename, newframename)

Functionality:

(1) Assure that the oldframename and newframename are instantiated.

(2) If the frame oldframename does not exist, report an error and fail.

(3) If the frame newframename already exists, report an error and fail.

(4) Rename the oldframename by newframename and also update the internal information related to the knowledge base file in which frame oldframename was present.

7.17.3 undef_frame(framename)

Functionality:

(1) Assure that the framename is instantiated.

(2) If the frame framename does not exist, report an error and fail.

(3) Delete the frame framename from the working memory.

(4) Update the internal information related to the knowledge base file in which frame framename was present.

7.17.4 frame_match_exact(prototype,list of frames)

Functionality:

(1) If the prototype is uninstantiated, fail.

(2) If the syntax is incorrect, report an error and fail.

(3) If there are exact matches in the value facet, return the list of matches.

(4) If there are no matches, fail.

the algorithm for determining if a frame matches the prototype:

```
for each of the prototype's slots do
  if the frame does not have a value for the slot
    then fail
    else if the frame's value is not the same
      then fail
      else if the frame's value's types are not acceptable
        then fail
        else if the frame's value's cardinality is not acceptable
          then fail
          else add the frame's name to the list of matches
```

7.17.5 frame_match_subset(prototype,list of frames)

Functionality:

Functionality of frame_match_subset operator is same as frame_match_exact operator except the difference in the matching criteria:

> for the frame_match_exact operator, to find a match, the list of values in the value facet in the prototype's slot and list of values in the value facet of the frame's slot must be identical (except for order). The frame_match_subset operator, on the other hand, will match a frame whose list of values in the value facet of a slot is a superset of the list of values in the value facet of the prototype's slot.

7.17.6 frame_subsumes(subsuming_frame,subsumed_frame)

Functionality:

(1) If an instantiated argument does not exist, report an error and fail.

(2) If a subsumptive relationship does not exist, fail.

One frame is considered to subsume another if either of the following is true:
there exists a chain of is_a links from the second frame to the first frame;
or there exists an instance_of link from the second frame to the first frame.

(3) If a subsumptive relationship does exist, succeed.

The frame_subsumes operator succeeds on backtracking.

7.18 KNOWLEDGE ANALYSIS OPERATORS

7.18.1 ka

Functionality:

(1) Store all the frames and slots present in working memory in a temporary file say temp2 in a remote directory.

(2) Also, store in a remote directory all the frames and slots in another temporary file say temp using def_frame1 and def_slot1 operators. def_frame1 and def_slot1 operators are internal to the package . These operators are similar to the def_frame and def_slot operators but they always perform consistency checking upon execution.

The order of frames and slots in the file temp is the same as the order in which frames and slots were defined in the existing knowledge base in working memory.

(3) Purge all the frames and slots from working memory.

(4) Load the file temp. At the time of loading, complete consistency checking is performed through the def_frame1 and def_slot1 operators. All the inconsistencies are displayed to the user.

(5) Delete all the frames and slots from working memory that were loaded in step 4 and then consult the file temp2. This way, ka analyze the whole knowledge base (using temp) and at the same time, does not change anything in working memory (using temp2).

7.18.2 check_frame(framename)

Functionality:

(1) Assure that the framename is an instantiated variable.

(2) If the frame framename does not exist, report an error and fail.

(3) Rest of the functionality is similar to the KA (knowledge analyzer), accept that instead of analyzing the whole knowledge base, it analyzes a particular frame.

7.19 UTILITY OPERATORS

7.19.1 trace_on/trace_off

Functionality:

(1) The frame package utilizes the concept of a trace switch. The trace switch is defined as trace_sw. By default trace_sw is off. It is turned on by the trace_on operator, and off by the trace_off operator.

(2) The entire coding of the frame package is embedded with trace_sw logic that displays appropriate messages if the trace_sw is on. These messages allow the user to trace the path for any operation. The trace confines to the steps as they relate to the user's knowledge base (as opposed to the frame package details).

7.19.2 print_tree(framename)

Functionality:

(1) Check to assure that the framename is an instantiated atom.

(2) If the frame framename does not exist, report an error and fail.

(3) Find the immediate parent(s) and child(ren) of the query frame framename by tracing a_kind_of links (is_a and in_of).

7.19.3 print_frame(framename)

Functionality:

(1) Check to assure that the framename is an instantiated atom.

(2) If the frame framename does not exist, report an error and fail.

(3) Print the frame details by utilizing proper indentation and line feeds at the level of slots, facets and facet values.

7.19.4 show_all

Functionality:

(1) Display knowledge base file names along with the frame names based upon the internal information that the frame package maintains throughout the session. a global slot name if defined through a def_slot operator, appears as a list entry in the list of frame names.

7.19.5 show_file(filename)

Functionality:

(1) Assure that the filename is an instantiated atom.

(2) If the file filename was not loaded as a knowledge base file (i.e. using load operator), report an error and fail.

(3) If file filename does not have any frames or slots, display a message and fail.

(4) Display the knowledge base file filename along with the frame names that belong to the file filename; based upon the internal information that the frame package maintains throughout the session. A slot name if defined through a def_slot operator, appears as a list entry in the list of frame names.

7.20 DEMON FILE OPERATORS

7.20.1 load_demon(filename)

Functionality:

(1) Consult the file filename.

7.20.2 edit_demon(filename)

Functionality:

(1) Display the file filename for editing using vi operator.

7.20.3 create_demon(filename)

Functionality:

(1) Assure that the file filename is an instantiated variable.

(2) If file filename already exists in the working directory, report an error and fail.

(3) Display file filename for creation using vi operator.

8 CONCLUSIONS

The enhanced frame package now provides much larger expressive power for a knowledge base development. It has been made simpler to use, very flexible and a little more user friendly. Several operators have been modified/enhanced and several new operators have been added while providing the user a balance of computational tractability, expressive power and consistency.

Some of the interesting **concepts** that were a major part of the development of the enhanced frame package include local consistency checking as opposed to global consistency checking and how the user can have the best of both options; the flexibility of loading a knowledge base file as a consistent system or as an inconsistent system; operations that work on the working memory and operations that work on the original file in the working directory; the concept of a knowledge analyzer; the way one sees human mind, knowledge and learning and its parallel in knowledge representation and the surrounding issues of consistency, expressive power and computational tractability.

Some of the prominent **features** of the frame package include the ability to use a knowledge base created from outside as well as inside the frame package, a large number of operators to operate on the knowledge base, the option to start with a consistent or an inconsistent system, the ability to work with several knowledge base files during a session, an editing and saving facility of the knowledge base from inside the frame package, an ability to explain its action during an operation and printing the hierarchical tree of frame names etc. One of the most important features of the package is to analyze the stored knowledge in a knowledge base using a knowledge analyzer.

Availability of the knowledge analyzer feature provides a very flexible method to achieve a desired level of knowledge base consistency. Level of consistency requirement in reality is application dependent. For example, a knowledge based system developed for tactical air command and control for military applications cannot possibly afford to have inconsistencies. On the other hand, it may be very difficult to make a completely consistent knowledge based system for home furnishings. It may not even be desirable. One's perception of consistencies for home furnishings may not be the same as other's perception. Obviously the level of consistency requirement changes from one application to the other. Hence any package that provides a fixed level of consistency may not suit all the applications. The enhanced frame package provides a flexible approach through the knowledge analyzer feature. The user can run the knowledge analyzer as often as needed and selectively correct the inconsistencies where and when desired. The control stays in the hands of the user.

For applications where complete consistency is required, one could consider another version of this package that enforces consistency all the time by analyzing the entire knowledge base before performing every single operation. The potential risk one may run into with this solution will be of a poor computational tractability.

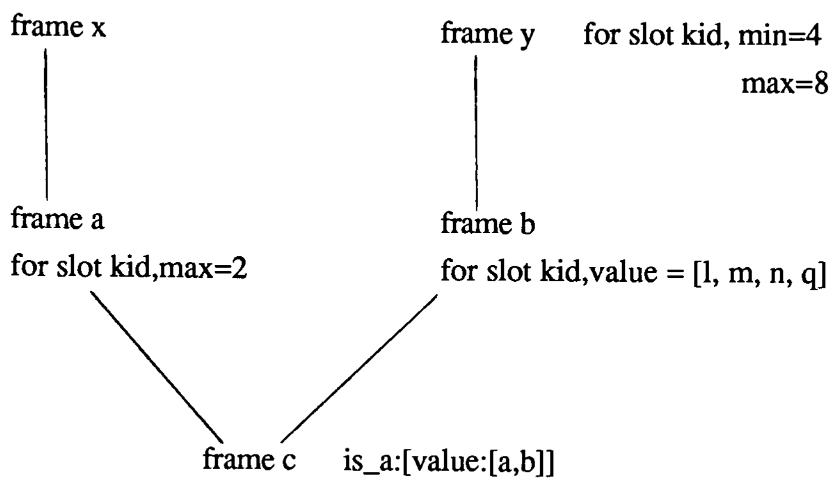
During the thesis work, some additional areas with scope for future work were identified. These areas have been discussed below.

8.1 INHERITANCE MECHANISM

Inheritance in this package does not occur at the frame definition time, but rather when a value for a facet is needed. At that time, if no value is found locally in a slot for a facet, it will be searched for recursively until found or until there is nowhere else to look for. This assures that the most current value will be inherited and that a local value will always override an inherited value [HISS87a].

When a child has more than one parent at the same level, the first parent in the value facet list of AKO slot (is_a or in_of) becomes the primary parent. In case of inheritance for a facet value, the child first inherits from this parent. If the parent frame does not have that facet, it searches recursively through parent's ancestors. If that facet value cannot be inherited through this path, then only it tries inheriting from the other parents one by one The search stops as soon as the first value for the facet is found.

Consider the following example; represented in a graphical manner on the next page:



Now frame c will inherit when needed, the following values for slot kid:

max = 2	(from frame a)
min = 4	(from frame y)
value = [l,m,n,q]	(from frame b)

which are contradictory to each other.

The analogy of this to real life shows that there is a lack of communication in the family resulting in a confused child situation. The child inherits different things from different ancestors irrespective of whether they create contradictions or not. The inheritance here completely depends upon the availability of the facet.

One solution to consider would be to allow the user selection of the frame from which to inherit a value. It could be a combination of methods like global method always inherit first/last value, or local method - using a facet inherit_from :

Example

```
?- def_frame(a:[is_a:[value:[x,y,z]],
                inherit_from:[y]]).
```

Another observation relates to the inheritance search that stops as soon as the first available value is found for the facet. One could consider exploration of all possible values that can be inherited as an alternative choice for the user. This would help the user know all the alternatives that are available before selecting the specific frame to inherit from.

8.2 TRACE FEATURE

Trace_on (and trace_off) feature was included in the package to limit the trace to the steps as they relate to the user's knowledge base. (The trace provided by the Prolog interpreter otherwise is too difficult and detailed to understand because it forces one to look 'under the hood' into the source code of the frame package.) The trace provided in the enhanced frame package, that relates to the user's knowledge base performs well, however, an additional feature can be considered for further enhancements.

With trace_on, as and when any frame package operator gets executed, it traces and displays the decision path of the operators. It also lets the user know if a demon was executed and what parameters were passed. However, it does not trace the details of the source code of demons. (the frame package operators encountered with in the demons are traced exactly as the operators encountered outside the demons.)

A facility that traces the entire demon code would be helpful to the user for debugging the demons. This facility however, should limit to the demon code only; it should not trace the frame package source code.

8.3 LOCAL DEPENDENCY BETWEEN SLOTS

Methods to provide local dependency between slots in a frame would be a nice feature to explore.

For the following knowledge base example file:

```
?- def_frame(material_type:[ ]).
?- def_frame(polyester:[is_a:[value:[material_type]]]).
?- def_frame(wool:[is_a:[value:[material_type]]]).
?- def_frame(price_type:[ ]).
?- def_frame(inexpensive:[is_a:[value:[price_type]]]).
?- def_frame(expensive:[is_a:[value:[price_type]]]).
?- def_frame(very_expensive:[is_a:[value:[price_type]]]).
?- def_frame(two_pc_suit:[
                    .
                    material : [value:[polyester],
                                max:1,
                                type:material_type],
                    price:    [value:[inexpensive],
                               type:price_type],
                    .
                    ]).
```

If the material is changed from 'polyester' to 'wool' in the frame 'two_pc_suit', the price also should change, since wool is expensive. It is quite likely that the user would change the material and forget to change the price. This would result in an incorrect price for two_pc_suit. This problem could be solved if there were some way to define a dependency between the 'material' slot and the 'price' slot for this particular frame such that a change in the value of any one of the slots would automatically prompt the user to change the value of the other slot. In the present package one can do this in an indirect way by writing appropriate demons in the knowledge base. The thought, however, is to have a generic method included in the package that allows a user to simply define a local dependency.

There is a way around it even without defining a local dependency. For example, one could make price a slot in the cotton and silk frames. This would also be a perfectly acceptable solution as the price becomes an integral part of the polyester and wool frames. In fact, one may always choose to use such a technique. But what happens if a large knowledge base has already been created and later the user realizes that certain dependencies should be incorporated. At that point, such a slot dependency operator would be a big help instead of going back to the original knowledge base and redefining all those frames that need such dependencies.

Not much has been found in the literature on local dependency between slots. Jones mentions that frame structures are often not deemed to imply temporal or causal relations between their slots [JONE84,38-39] . The issue of causal relationships between slots must be examined carefully. Adding local dependency between slots seems to be equivalent of adding some "thinking power" to the knowledge representation system.

8.4 HELP FACILITY

For the convenience of the user, an on-line help facility for the frame package operators can be provided. Depending on the level of details in the help facility, this could become quite involved.

8.5 ERROR MESSAGES

The nature of the frame concept and the inheritance mechanism can result in a very complicated and/or a large number of steps in order to perform an operation. This, at times can generate several messages from these different steps. Some of these messages may therefore provide the impression of duplicate messages. In terms of the frame package design, they basically indicate the number of times the underlying reason got asserted in the total process. For future enhancements, elimination of redundant messages can be one of the challenging tasks, particularly because of no definite pattern of these duplications. Perhaps one could write an expert system that will identify these duplications and eliminate them.

8.6 NATURAL LANGUAGE INTERFACE

The audience for the current package is limited to the users who are conversant with Prolog. A Natural Language Interface can potentially make the power of this package available to non Prolog users as well as make it more user friendly. Considering the size, variety and the complexity of the operations of the frame package, the natural language interface will be very involved.

BIBLIOGRAPHY

[BOBR77]

Bobrow, Daniel G., "A Panel on Knowledge Representation", *Joint Conference on Artificial Intelligence 1977*, William Kaufman, Los Altos, Calif., pp. 983-992, 1977.

[BOBR85]

Bobrow, Daniel G., and Winograd, Terry, "An overview of KRL, a Knowledge Representation Language", from *Readings in Knowledge Representation*, eds. Brachman, Ronald J., and Levesque, Hector J., Morgan Kaufman Publishers, Los Altos, Calif., pp. 263-285, 1985.

[BRAC85]

Brachman, Ronald J., Fikes, Richard E., and Levesque, Hector J., "KRYPTON: A Functional Approach to Knowledge Representation", from *Readings in Knowledge Representation*, eds. Brachman, Ronald J., and Levesque, Hector J., Morgan Kaufman Publishers, Los Altos, Calif., pp. 411-429, 1985.

[ENGE80]

Engleman, Carl, Scarl, Ethan A., and Berg, Charles H., "Interactive Frame Instantiation", *First Annual National Conference on Artificial Intelligence 1980*, pp. 184-186, 1980.

[ENGE81]

Engleman, Carl, and Stanton, William M., "An Integrated Frame/Rule Architecture", from *Artificial and Human Intelligence*, eds. Elithron, Alick, and Banerji, Raman, Elsevier Science Publishers, B. V., pp. 141-145, 1981.

[FIKE85]

Fikes, Richard, and Kehler, Tom, "The Role of Frame-Based Representation in Reasoning", *Communications of the ACM*, Vol. 28, No. 9, pp. 904-920, 1985.

[HISS87a]

Hiss, LaMora S., *A Frame Virtual Machine in C-Prolog*, unpublished Masters thesis, School of Computer Science and Technology, Rochester Institute of Technology, Rochester, NY, 1987.

[HISS87b]

Hiss, LaMora S., *User's Manual for a Frame Virtual Machine in C-Prolog*, unpublished document, School of Computer Science and Technology, Rochester Institute of Technology, Rochester, NY, 1987.

[JONE84]

Jones, Karen S., "Frame", from *Catalogue of Artificial Intelligence Tools*, ed. Bundy, Alan, Berlin; NY: Springer-Verlag, pp. 38-39, 1984.

[KUIP75]

Kuipers, Benjamin J., "A Frame for Frames", from *Representation and Understanding*, eds. Bobrow, Daniel G., and Collins, Alan, Academic Press, Inc., New York, San Francisco, London, pp. 151-184, 1975.

[LEVE85]

Levesque, Hector J., and Brachman, Ronald J., "A Fundamental Tradeoff in Knowledge Representation and Reasoning", from *Readings in Knowledge Representation*, eds. Brachman, Ronald J., and Levesque, Hector J., Morgan Kaufman Publishers, Los Altos, Calif., pp. 41-70, 1985.

[MACK81]

Mackworth, Alan K., "Consistency in Networks of Relations", from *Readings in Artificial Intelligence*, eds. Webber, Bonnie L., and Nilson, Nils J., Tioga Publishing Company, Palo Alto, Calif., pp. 69-78, 1981.

[MINS85]

Minsky, Marvin, "A Framework for Representing Knowledge", from *Readings in Knowledge Representation*, eds. Brachman, Ronald J., and Levesque, Hector J., Morgan Kaufman Publishers, Los Altos, Calif., pp. 245-262, 1985.

[PATE84]

Patel-Schneider, Peter F., "Small Can Be Beautiful in Knowledge Representation", *IEEE workshop on Principals of Knowledge Based Systems 1984,* IEEE Computer Society Press, Silver Springs, Md, pp. 11-16, 1984.

[PIGM84]

Pigman, Victoria, "The Interaction Between Assertional and Terminological Knowledge in KRYPTON", *IEEE workshop on Principals of Knowledge Based Systems 1984,* IEEE Computer Society Press, Silver Springs, Md, pp. 3-9, 1984.

[ROSE79]

Rosenberg, Steven and Roberts, Bruce, "CoReference in a Frame Database", *Joint Conference on Artificial Intelligence 1979*, William Kaufman, Los Altos, Calif., pp. 729-734, 1979.

[STEF79]

Stefik, Mark, "An Examination of a Frame-Structured Representation System", *Joint Conference on Artificial Intelligence 1979*, William Kaufman, Los Altos, Calif., pp. 845-852, 1979.

APPENDIX A

EXCERPTS FROM [HISS87a]

2.2. Present Definitions of Frames

Many variations on the frame concept have been formulated since Minsky's first paper on this knowledge representation structure. This study will attempt to give a sampling of some of these concepts rather than an exhaustive listing.

2.2.1. Frames in FRL

In the late 1970's, Bruce Roberts and Ira Goldstein of MIT developed a frame-based language called FRL (Frame Representation Language). FRL was developed as a general purpose tool that was to be integrated into knowledge based applications, and since then, a wide variety of applications have used it. Representative applications domains are office schedules, wheat commodities market, travel, and English text [ROSE79][WINS79].

FRL provides the frame as the basic representational concept and operators that are used to manipulate those frames. A frame represents a concept and consists of a named collection of slots. A slot is composed of an arbitrary number of user and system-defined facets. The facets provide the slot's value as well as meta-knowledge about that value. System defined facets are default, require, if-needed, if-added, and if-removed. The value facet contains the slot's value, while the default facet contains a default value that can be used if the value facet is not present. The require facet establishes procedural constraints on the slot's value. The remaining three system-defined facets allow for procedural attachment in three cases: when a value is added to a slot, when a value is removed from a slot, and when a value is needed but is not present in the slot [ROSE79].

Frames in FRL exist in hierarchies established by AKO (a-kind-of) links. These links are established by the system-defined AKO slot. The value of an AKO slot must be a generic frame of which the current frame is a specialized instance. This hierarchical representation allows for inheritance from generic frames to their specialized instances. The inheritance is of two types: additive and restrictive. Additive inheritance allows a specialization to add new non-contradictory facts to the information inherited from the generic frame. Restrictive inheritance allows the information in a specialization to override the information in the generic ancestor. Besides these two formal methods of inheritance, FRL also allows for "idiosyncratic" forms of inheritance through the use of attached procedures [STEF79].

The set of frame operators in FRL allows for limited manipulation of frames and the information in them and the matching of frames to a given pattern. Frames are put into the knowledge base with an FASSERT operator. This allows establishment of the frame name, list of slots, and the facet names and values. Once created, three operators can act on the information in the frames: FPUT, FGET, and FREMOVE. FPUT allows values to be put in any facet of any slot of any frame. FGET and FREMOVE, on the other hand, only allow the return or removal of the value in the value facet. This means, therefore, that meta-knowledge concerning a slot's value can be augmented by the user, but never removed or retrieved by him.

Besides the manipulation of frames and their slot values, FRL provides an operator to return frames that match a given prototype. This frame-match operator requires two arguments: a reference frame and the prototype or pattern to be matched. The prototype defines what is to be searched for and the reference frame defines the search space by specifying the parent node below which to search.

FRL holds very closely to Minsky's original concept of the frame. This is perhaps to be expected, as FRL was developed rather shortly after Minsky's first paper on the subject. FRL lacks many of the enhancements of later frame-based systems, but provides an obviously useful set of tools which serve as at least part of the conceptual basis of many later frame-based packages.

2.2.2. Frames in KRL

The frames used in KRL, Knowledge Representation Language, are one powerful and complex variation. KRL's ultimate goal was to provide a programming language for building systems for natural language understanding. The needs of KRL were seemingly custom made for the frame concept: knowledge should be organized around conceptual entities with associated descriptions and procedures; a description must be able to accommodate multiple descriptors and incomplete knowledge; and comparison and/or contrast with a known prototype must be a valid method of description.

A description is defined as "fundamentally intensional - the structure of the description can be used in recognizing a conceptual entity and comparing it with others" [BOBR85a,265]. Three underlying operations were defined for descriptions: augmenting a description to incorporate new knowledge; matching descriptions to see if they are compatible; and seeking referents for entities that match a specific description. The basic data structures of KRL are built of these descriptions which are clustered into units. These units are analogous to Minsky's frames and serve as the unique representations of categories and entities. The descriptions are actually referred to as slots. Each slot has a name and can contain procedures, values, or constraints. Unlike Minsky's frames, however, the units of KRL are categorized into seven distinct types: basic, abstract, specialization, individual, manifestation, relation, and proposition. Abstract, basic and specialization units are used principally as prototypes. An abstract unit serves as a prototype whose descriptions will be inherited by entities based on this prototype. Basic units produce a non-overlapping partitioning of the world into different types of entities. Specialization units provide for further distinctions within a basic category. Both individual and manifestation units represent unique entities in the world; individual units represent a named entity, while manifestations represent an unidentified entity. A relation unit represents a predicate, while a proposition unit represents an instantiation of a relation.

Just as KRL has augmented the basic frame concept, KRL also elaborates upon Minsky's concept of procedural attachment. It classifies procedural attachments along two dimensions: when the procedure will be used; and whether the procedure is associated with a prototype or an individual. Knowledge as to when a procedure will be used determines whether it is a servant (which provides the method for carrying out an operation) or a demon (which causes a secondary effect for an event). Servants are invoked when the system needs a procedure in order to apply an operation to a data element. A typical use of a servant might be to describe how to match a descriptor involving a relation. A demon is invoked as a side effect of actions such as accessing or adding data. Demons can be awakened when an action is about to be done or after it is done. This concept of when the procedure would be invoked is the first dimension of procedural attachment. The second dimension involves the type of unit with which a procedure is associated. Procedures associated with individual data objects are called traps, while those associated with prototypes are called triggers. A list of triggers and traps can be associated with each slot within a unit. Triggers and traps can be either servants or demons [BOBR85a].

The data structures of KRL are rather complex, but the originators felt that "any representation language that is to be used in modeling thought or achieving 'intelligent' performance will have to have an extensive and varied repertoire of mechanisms" [BOBR85a,284]. It may well have been the weight and complexity of these features that brought about KRL's collapse, as KRL was ultimately a failure [BRAC85,263].

2.2.3. Frames in the Unit Package

The Unit Package was developed at Stanford University in the late 1970's as a frame-based, interactive knowledge representation tool that could be used in hierarchical planning applications. Knowledge is organized as a partitioned semantic network with frames as its nodes. The Unit Package's intended domain, computer planning of molecular genetics experiments, was narrow enough to make uniform representation of information possible. The intent was to create representations that could be processed by uniform network processing routines. As far as inferencing capability, the package only included property inheritance, pattern matching, and procedural attachment. All other inferencing was the responsibility of an application package. The Unit Package merely provided the representational structure.

As in KRL, the term unit, rather than frame, is used to represent a node. There are actually four kinds of nodes provided in the package. They are instance, schema, indefinite, and description units. Instance and schema units are constant nodes. An instance represents a unique individual and is, therefore, a terminal node. A schema node represents a class and describes attributes necessary for potential progeny. Indefinite and description units are nodes whose identities are unknown. An indefinite node is a variable node that stands for an individual, while a description node is a variable node that stands for a class. These variable nodes may be linked to constant nodes by coreferential or anchoring links.

The links in the semantic network are represented by slots. These slots are structured and have a defined set of attributes. The attributes of the slots are called aspects and correspond to KEE's facets. There are six defined aspects for a slot: name, value, definitional role, inheritance role, default, and datatype. The name aspect is simply the name of the slot, while the value aspect is the value stored in the slot. The definitional role aspect is the role that the slot takes in the definition of the unit. There are six defined roles: part-of, property, relation, super-unit, equivalence, and documentation. The inheritance role aspect dictates the method by which progeny will inherit the slot. There are four defined inheritance methods: same, requirements, optional, and unique. The "same" method indicates that all progeny inherit the same value. The "requirements" method is limited to schema and causes the slot's value to be inherited as a requirement or value-restriction for the corresponding slots in any progeny. The progeny must, therefore, have a value that meets the value-restriction. The "optional" inheritance method is similar to the "requirements" method, but it does not require that progeny have a value for the slot; it requires only that if a value is present, it fulfill the requirements. The "unique" inheritance method is used for slot values that are not to be inherited by progeny; the slot will be inherited, but the slot's value will be unique at each level of the hierarchy.

Like most of the other frame-based representation schemes, the Unit Package provides for procedural attachment. These attached procedures are the main inferencing mechanism in the Unit Package. Attachment is allowed at three distinct places: unit attachment, slot attachment, and datatype attachment. The attached procedures are activated by messages and have specific purposes. The purpose indicates when the procedure should be activated. Sending a message with a purpose-matching token serves to activate a procedure [STEF79].

The Unit Package represents a hybrid conceptual model of sorts. The scheme is very representative of Minsky's frame model and yet still clings tenaciously to the concept and vocabulary of the semantic network. Even with this strong semantic network flavor, however, the Unit Package serves as a fair representative of a frame-based knowledge representation scheme.

2.2.4. Frames in KRYPTON

About 1983, the frame-based language KRYPTON was developed by researchers working at Xerox PARC and Fairchild Laboratory for Artificial Intelligence Research. The creators were concerned about the ways that frames could be accessed and manipulated. They found ambiguity in the factual versus definitional interpretations of frames and in the actual meanings of the representations that are created by frames. Frames can either be interpreted as assertions or descriptions. Interpreting frames as assertions makes expressions of incomplete knowledge or composite descriptions difficult if not impossible. Interpreting frames as descriptions avoids the above problems, but can allow for misinterpretation by a careless user. KRYPTON was designed to overcome these difficulties because it distinguishes between functional and factual information.

This language is not defined in terms of the structures that can be manipulated, but rather in terms of what can be asked or told about the domain. KRYPTON actually uses two representational languages: one for descriptive terms and one for making statements about the world. The terminological language is called TBox and is a simple frame language. It supports expressions of Concepts (frames) and Roles (slots), allows establishment of taxonomies concerning these structures, and answers questions about analytical relationships among the structures. There are are three Concept-defining operators: PrimGeneric, ConGeneric, and VRGeneric. PrimGeneric produces a primitive, undefined Concept which is considered to be a root node in the frame hierarchy. ConGeneric produces a Concept that is a conjunction of an arbitrary number of Concepts. VRGeneric produces a value-restricted Concept. The functions allowed on Concepts, therefore, are conjunction, value restriction, number restriction, decomposition, and specification of necessary-but-not-sufficient conditions for a Concept definition. There are two Role-defining operators: PrimRole and RoleChain. PrimRole produces a primitive, undefined Role, while RoleChain produces a chain of primitive or defined Roles. The functions allowed on Roles are role differentiation, role chaining, decomposition, and specification of necessary-but-not-sufficient conditions for a Role definition.

The assertional language is called ABox. It is used to build descriptive theories about a domain and to answer questions about these domains. ABox provides a first-order predicate calculus language whose predicates come from TBox. Expressions in TBox or ABox are used either to TELL the knowledge base or to ASK the knowledge base. TELLing an ABox expression asserts that it is true; TELLing a TBox expression associates a symbol with the expression. ASKing an ABox expression asks if the expression is true. There are two ASKs for TBox expressions; one asks whether one term subsumes another while the other asks whether two terms are disjoint.

The descriptive methods of KRYPTON are rather cumbersome and the set of operators is small, but this language was created mainly out of academic interest so a full set of operators and ease of use was probably not a goal of the project. The main goal was to control access to the knowledge and to thus control the way that it could be interpreted [BRAC85][PIGM84].

2.2.5. Frames in KANDOR

KANDOR was developed at Fairchild Laboratory for Artificial Intelligence Research in 1984 as an attempt to build a limited knowledge representation system. The limits were of two types. The interface to the knowledge base was intended to be well-defined in hopes of limiting a user's manipulation of the underlying data structures in a way unintended by the designers. The second limit was imposed on the expressional power. This limit hopefully assures that all operations done on the knowledge within the system will terminate in a reasonable period of time. The designers felt these limits made it realistic to believe that KANDOR could be incorporated into a larger knowledge based system.

The basic units of KANDOR's frame-based knowledge representation system are individuals and frames. Individuals represent objects in the real world, while frames model collections of those objects. Slots associate information with an individual or frame; slot fillers are the values associated with the slots.

A KANDOR frame specifies conditions for an individual being an instance-of it, and merely serves as a description. Two types of conditions may be specified by a frame: superframes and restrictions. A superframe condition is satisfied if an individual is an instance-of each of the specified superframes. A restriction condition is simply a constraint on slot fillers. In hopes of controlling computational time when determining subsumptive relationships, KANDOR limits these restrictions to three types. The first type states that at least some number of slot fillers must be an instance-of some frame. A second type of restriction requires that all slot fillers must be instances-of some frame. The third restriction is a cardinality restriction on the number of slot fillers for a specific slot. There are two types of frames in KANDOR: defined and primitive. In defined frames, the conditions are both necessary and sufficient conditions, while in primitive frames the conditions are only necessary.

Procedural attachment, a characteristic of most frame-based knowledge representation schemes, is purposely absent in KANDOR. The designers believed it would jeopardize the character of the system and reduce it to a package for data structure manipulation.

Operators are provided that allow for the creation of individuals, slots, and frames. Purposefully absent are operators that would allow a slot to be removed or a slot filler to be changed. Again, this omission was made in hopes of assuring the integrity of the system. Operators exist that will determine if an individual is an instance-of a frame and if one frame subsumes another.

KANDOR represents a frame-based knowledge scheme that uses a subset of the concepts usually attributed to frames. The absence of default values and procedural attachment produces an inconsistency with Minsky's basic concept. KANDOR is, however, an example of a scheme that has chosen to represent a little less than Minsky suggested in hopes of producing a real-world useable system [PATE84].

2.2.6. Frames in KEE

One branch of research in knowledge representation has lead to the creation of hybrid systems: systems that use more than one knowledge representation paradigm.

KEE (Knowledge Engineering Environment) by Intellicorp is just such a system. This system combines frames and production rules for a hybrid form of representation. The frames provide the structure for describing objects referred to by the rules and the inheritance and taxonomy that allow for some level of deductive capability. Frames may also be used to "house" rules and provide partitioning, indexing, and organization of the rules.

KEE units (frames) are used to represent a class or individual and incorporate sets of attributes called slots. Two kinds of links are provided between frames: member links and subclass links. Member links represent membership in a class, while subclass links represent subsumption of one class by another. Slots come in two basic varieties: own slots and member slots. Own slots can occur in individual or class frames and belong to the class or individual that contains them. Member slots can only occur in class frames and are used to describe attributes of each member rather than the class itself. For example, a class frame, SOPHOMORES, might own an average-age-slot. It does not make sense, however, for an individual student to inherit that slot even though he is a member of the class. It would be more likely for there to be a member age-slot for members of the class SOPHOMORE. Conversely, it would make little sense for the class frame SOPHOMORE to have an age-slot. Slots are composed of facets. Facets are basically descriptions of the attribute represented by the slot. Facets would include such information as cardinality, valid values, and actual values.

KEE provides a form of object-oriented programming by providing procedural attachment to the frame structures. Two forms of procedural attachment are provided: methods and active values. Methods are LISP procedures which are attached to frames and are stored as values of special message responder slots. A method is activated by sending a message to the appropriate slot. Active values are collections of rules or procedures and are stored in slots as demons. An active value is activated by accessing a slot's value.

KEE actually also uses frames to represent its rules. This allows rules to be grouped into classes and associated with a non-rule frame. Several advantages result from this feature. Rules and the classes to which they pertain can be linked. This can increase the efficiency of the operation of a hybrid system since only a subset of the rules need be "active" at any given time [FIKE85].

APPENDIX B

KEYWORDS AND THEIR DEFINITIONS

AKO/ a_kind_of:

In the frame package, is_a and in_of (instance_of) slots are used as 'a_kind_of' links to show subsumptive relationship between two classes (is_a) or between a class and an individual member of that class (in_of). A frame that has an in_of slot is a terminal node, while a frame with an is_a slot is a member of the parent class. A frame with neither is_a or instance_of slot is a root node in the frame hierarchy. Frame package uses this information for inheritance purposes only.

Computational Tractability:

It is related to the amount of time required to complete operations. A quicker performance will implies better computational tractability.

Consistency:

It relates to the consistency (suitability/fitness/congruity) of all the knowledge base elements with each other.

Declarative Knowledge:

In the frame package, the knowledge that is defined through max, min, type, default and value facet is the declarative knowledge.

Default Facet:

The default facet provides a list of values that is to be used for the slot's value if no other values are present.

Demon:

A demon is a prolog predicate that represents procedural knowledge. A demon may be invoked when a slot in a frame is accessed via the value facet operators. The four facets - if_added, if_removed, if_changed and if_needed reflect the four types of demon operations on the value facet of a slot.

Expressive Power:

It is related to the breadth of operations that are available in the knowledge representation language that allows access, modifications, analysis and update of a knowledge base.

If_added:

If_added facet stores the procedural knowledge about the slot's value in terms of prolog predicate(s) or demon(s). A demon in the if_added facet will be invoked when a value(s) is added to a slot.

If_changed:

If_changed facet stores the procedural knowledge about the slot's value in terms of prolog predicate(s) or demon(s). A demon in the if_changed facet will be invoked when a value(s) of a slot is changed.

If_needed:

If_needed facet stores the procedural knowledge about the slot's value in terms of prolog predicate(s) or demon(s). A demon in the if_needed facet will be invoked when a value is requested for a given slot but none is present in the slot.

If_removed:

If_removed facet stores the procedural knowledge about the slot's value in terms of prolog predicate(s) or demon(s). A demon in the if_removed facet will be invoked when a value(s) is removed from a slot.

Internal Information:

Throughout the session, frame package maintains certain information regarding the knowledge base which is completely transparent to the user. This information is very critical to the performance of the package. For example, information regarding the order of various knowledge base files that were loaded during the session, the order of frames in each knowledge base file etc. All this information is being referred to as the internal information to the package.

Facet:

Facets represent descriptive knowledge, procedural knowledge and constraints for a slot's value. These are system defined facets. The value, type, max, min and default facets store declarative knowledge about a slot. if_added, if_removed, if_changed and if_needed facets store procedural knowledge about a slot.

Frame:

Basic conceptual unit of the frame package. A frame is composed of a unique name and a list of any number of slots including zero number of slots.

Knowledge Analyzer:

Knowledge analyzer is an operator available in the frame package, that analyzes the complete knowledge base as it exists at that point of time during a session for consistency.

Max Facet:

The max facet indicates the maximum number of values that the value facet may contain.

Min Facet:

The min facet indicates the minimum number of values that the value facet may contain.

Original knowledge base file /
Knowledge base file in the working directory, and
Knowledge base file in working memory:

The knowledge base file as it stands saved in the working directory is referred to as the original knowledge base file or the knowledge base file in the working directory. When this file is loaded through the frame package, it then becomes available in working memory also. This working memory file is what is referred to as the knowledge base file in working memory.

The same knowledge base file can be present in the working directory as well as in working memory. However, over time their contents can be different from each other as operations are performed on working memory file. (Most frame package operators operate on working memory file only. They do not update the knowledge base file in the working directory (see Chapter 5).)

Procedural Knowledge:

In the frame package, the procedural knowledge is stored in the form of prolog predicates or demons. It is used with if_added, if_needed, if_removed and if_changed facets. These prolog predicates get executed when any one of these facets is invoked.

Slot:

A slot represents knowledge in a frame. In other words, slots are attributes of a frame. A slot is composed of a name and a list of one or more system defined facets.

Subsumptive Relationship:

Frame package uses this concept to determine relationship among frames in a hierarchy. One frame is considered to subsume another if there exists a chain of is_a links from the second frame to the first frame; or there exists an instance_of link from the second frame to the first frame.

Type Facet:

The type facet serves as a constraint on the possible types of values that can appear in the value facet. The value in a type facet must be a legal type predicate. A legal type predicate is defined as a frame name or a logical combination (and, or, not) of frame names.

APPENDIX C

ANIMALKINGDOM KNOWLEDGE BASE FILE

This knowledge base file represents a very small amount of knowledge about five different but interconnected domains: the animal kingdom, body temperature types, body covering types, reproduction types and food sources of animals. While knowledge about any one of these topics could be represented and used as an illustration, this example shows how knowledge hierarchies can gain knowledge from each other. Five knowledge hierarchies are established. The food_source_type hierarchy stores information about food_sources. The body_temp_type hierarchy stores information about body temperature types. Putting knowledge in these hierarchies rather than in the actual frames in the living_thing hierarchy avoids the duplication of information and localizes the information so that additions or deletions of knowledge about food source types or body temperature types are very easy. The living_thing hierarchy stores information concerning the characteristics of different groups within the animal kingdom and information concerning the relationship among those groups [HISS87a,47].

```
/*------------------------------------------------------------------*/
/*.....Filename: animalkingdom......................................*/
/*.....This is a example knowledge base file........................*/
/*------------------------------------------------------------------*/

/*------------------------------------------------------------------*/
/* TEST SLOT DEFINITIONS............................................*/
/*------------------------------------------------------------------*/

?- def_slot(reproduction:
           [max:999]).

/*------------------------------------------------------------------*/
/* TEST FRAME DEFINITIONS...........................................*/
/*------------------------------------------------------------------*/

/*------------------------------------------------------------------*/
/* frames in the REPRODUCTION_TYPE hierarchy........................*/
/*------------------------------------------------------------------*/

?- def_frame(reproduction_type:
           [description:
                 [value:[possible_types_of_reproduction]]]).
?- def_frame(viviparous:
           [is_a:
                 [value:[reproduction_type]],
           description:
                 [value:[birth_to_live_young]]
           ]).
?- def_frame(ovoviviparous:
           [is_a:
                 [value:[reproduction_type]],
           description:
                 [value:[eggs_hatch_inside_mother]]
           ]).
?- def_frame(oviparous:
           [is_a:
                 [value:[reproduction_type]],
           description:
                 [value:[lay_eggs]]
           ]).

/*------------------------------------------------------------------*/
/* frames in the BODY_COVERING_TYPE hierarchy.......................*/
/*------------------------------------------------------------------*/

?- def_frame(body_covering_type:
           [description:
                 [value:[possible_types_of_body_coverings]]]).
?- def_frame(hair:
```

```
            [is_a:
                  [value:[body_covering_type]]]).
?- def_frame(feathers:
            [is_a:
                  [value:[body_covering_type]]]).
?- def_frame(scales:
            [is_a:
                  [value:[body_covering_type]]]).

/*-----------------------------------------------------------------------*/
/* frames in the BODY_TEMP_TYPE hierarchy............................*/
/*-----------------------------------------------------------------------*/

?- def_frame(body_temp_type:
            [description:
                  [value:[possible_body_temp_types]]]).
?- def_frame(cold_blooded:
            [is_a:
                  [value:[body_temp_type]],
            description:
                  [value:[does_not_maintain_constant_temp]]
            ]).
?- def_frame(warm_blooded:
            [is_a:
                  [value:[body_temp_type]],
            description:
                  [value:[maintains_constant_temp]]
            ]).

/*-----------------------------------------------------------------------*/
/* frames in the FOOD_SOURCE_TYPE hierarchy..........................*/
/*-----------------------------------------------------------------------*/

?- def_frame(food_source_type:
            [description:
                  [value:[possible_food_sources]]]).
?- def_frame(omnivorous:
            [is_a:
                  [value:[food_source_type]],
            characteristic_of:
                  [value:[humans],
                   max:999],
            description:
                  [value:[eats_plants_and_animals]]
            ]).
?- def_frame(herbivorous:
            [is_a:
                  [value:[food_source_type]],
            characteristic_of:
                  [value:[rodents,cattle],
                   max:999],
            description:
                 [value:[eats_plants]]
            ]).
?- def_frame(carnivorous:
            [is_a:
                 [value:[food_source_type]],
            characteristic_of:
                 [value:[cats,dogs],
                  max:999],
            description:
                 [value:[eats_animals]]
            ]).

/*----------------------------------------------------------------*/
/* frames in the LIVING_THING hierarchy..........................*/
/*----------------------------------------------------------------*/

?- def_frame(living_thing:[]).
?- def_frame(animal:
            [is_a:
                 [value:[living_thing]],
            food_source:
                 [type:(~photosynthesis),
                 max:999],
            description:
                 [value:[partition_of_living_things]]
            ]).
?- def_frame(mammal:
            [is_a:
                 [value:[animal]],
            food_source:
                 [default:[omnivorous],
                 type: food_source_type],
            body_covering:
                 [value:[hair],
                 type:hair],
            reproduction:
                 [value:[viviparous],
                 type:viviparous],
            body_temperature:
                 [value:[warm_blooded],
                  type: warm_blooded]
            ]).
?- def_frame(bird:
            [is_a:
                 [value:[animal]],
            body_covering:
                 [value:[feathers],
                 type:feathers],
            reproduction:
                 [value:[oviparous],
                type:oviparous],
           body_temperature:
                [value:[warm_blooded],
                type:warm_blooded]
           ]).
?- def_frame(reptile:
           [is_a:
                [value:[animal]],
           body_covering:
                [value:[scales],
                type:scales],
           reproduction:
                [value:[ovoviviparous,oviparous],
                type: ovoviviparous $$ oviparous],
           body_temperature:
                [value:[cold_blooded],
                type:cold_blooded]
           ]).
```

APPENDIX D

A SAMPLE SESSION

The sample session uses 'bookprototype' (short form bk) as the knowledge base file and 'bookprototype_demon' (short form - bk_demon) as a demon file.

These knowledge base files represent knowledge about typesetting of documents. Three knowledge hierarchies are established - typeface_type, font_type and book_prototype. Two of the hierarchies are interrelated and store knowledge about typeface and fonts. These hierarchies represent knowledge about the classification of fonts and the relationships among those classifications. The basic relationship represented is that each regular typeface has an italic and a bold counterpart font. These counterparts are used in a document for quotations, titles, section headings, subtitles, etc. The third hierarchy serves as a book prototype. Information about the structure of a book is found in the hierarchy, as well as information about how to typeset a book. This information about typesetting, however, is stored in a very general manner. Typesetting characteristics such as font, font size and justification are stored in such a way that they are inherited and adjusted at the time that a typesetting decision is made. An example scenario might serve to clarify this description. Typesetting decisions are based on the type of text being set: children's books are set in a different typeface than adults book; textbooks are set in a different typeface than poetry books. If a prototype contained specific information, one would need a different prototype for every text possibility. If, on the other hand, only relationships and adjustments to a hypothetical typeface were stored in the prototype, one prototype might serve for all books. Initial decisions as to which basic typeface to use and what basic size to use will, after inheritance and constraint propagation, leave the prototype completely specified. As can be seen in the example prototype, title gets its font by inheritance and its size by unleashing a demon that enlarges the basic size of a book_component. Once the prototype is designed, a simple change to the book_component frame will change the entire typesetting information in the prototype. The elegance of this method can be seen in the fact that alternative sets of knowledge do not have to be stored; only a general pattern has to be specified [HISS87a,41].

knowledge base file **bookprototype** (short form **bk**)

```
/*------------------------------------------------------------------------*/
/* .. Filename : bookprototype   (short_name : bk)........................*/
/* .. This is an example knowledge base file .............................*/
/*------------------------------------------------------------------------*/
/*========================================================================*/
/* PREDEFINED SLOTS ......................................................*/
/*========================================================================*/
?- def_slot(size:
                 [max : 1,
                  min : 0
                 ]).

/*===========BASIC_TYPEFACE_TYPE HIERARCHY ==============================*/

?- def_frame(typeface_type: [available_types:[if_added:get_others]]).
?- def_frame(roman: [is_a:[value:[typeface_type]]]).

?- def_slot(typeface:
                 [type: typeface_type,
                  max : 1,
                  min : 0 ]).

/*================= FONT, BOLD, and ITALIC TYPE HIERARCHY ==============*/

?- def_frame(font_type: []).
?- def_frame(bold_type: [is_a:[value:[font_type]]]).
?- def_frame(italic_type: [is_a:[value:[font_type]]]).
?- def_frame(regular_type: [is_a:[value:[font_type]]]).
?- def_frame(roman_bold: [is_a:[value:[roman, bold_type]]]).
?- def_frame(roman_italic: [is_a:[value:[roman, italic_type]]]).

/*===================== BOOK COMPONENT HIERARCHY ======================*/

?- def_frame(book_component:
                 [typeface:
                              [value:[roman],
                               max: 1,
                               min : 1,
                               type : typeface_type],
                  size:
                              [value:[11],
                               max: 1,
                               min : 0],
                  font:
                              [value:[regular_type],
                               max: 1,
                               min: 0,
                               type: font_type]
                 ]).
?- def_frame(front_matter:
```

```
                    [is_a:[value:[book_component]]]).
?- def_frame(title_page_component:
                    [is_a:[value:[front_matter]],
                     justification: [value:[centered]],
                     size: [if_needed: enlarge(3)]
                    ]).
?- def_frame(title:
                    [is_a:[value:[title_page_component]],
                     font:[if_needed: make_bold]
                    ]).
?- def_frame(author:
                    [is_a:[value:[title_page_component]],
                     font: [if_needed: make_italic]
                    ]).
?- def_frame(publisher:
                    [is_a:[value:[title_page_component]]]).
?- def_frame(body_matter:
                    [is_a: [value: [book_component]]]).
```

demon file **bookprototype_demon** (short form - **bk_demon**)

```
/*----------------------------------------------------------------------*/
/* .. Filename : bookprototype_demon (short_name: bk_demon)............*/
/* .. This is an example demon file ...................................*/
/*----------------------------------------------------------------------*/


/*==================== DEMONS FOR BOOKPROTOTYPE =========================*/

/*============= if_needed demons and associated procedures  ===== =======*/

enlarge(Enlarge_By,Frame,size,Value) :-
            find_ancestor_characteristic(Frame,size,Answer),
            Value is Answer + Enlarge_By.

find_characteristic(Component,Characteristic,Answer) :-
                  return_value_from_slot(Component,Characteristic,Answer).

find_ancestor_characteristic(Component,Characteristic,Answer) :-
                  (return_value_from_slot(Component,in_of,Ancestors);
                   return_value_from_slot(Component,is_a,Ancestors)),
                  member(AnAncestor,Ancestors),
                  find_characteristic(AnAncestor,Characteristic,Answer).

make_italic(Frame,font,Value) :-
                find_characteristic(Frame,typeface,Answer),
                member(SingleAnswer,Answer),
                frame_match_subset(
                        [is_a:[value:[italic_type,SingleAnswer]]], Value).

make_bold(Frame,font,Value) :-
                find_characteristic(Frame,typeface,Answer),
                member(SingleAnswer,Answer),
                frame_match_subset(
                        [is_a:[value:[bold_type,SingleAnswer]]], Value).

/*===========if_added demons and their associated procedures ============*/

get_others(_,_,[]).

/* when more than one value is added */
get_others(Frame,Slot,Value) :-
        Value = [H|T],
        get_more_inf(H),
        get_others(Frame,Slot,T).

get_others(Frame,Slot,Value) :-
        atom(Value),
        get_more_inf(Value).
get_more_inf(Value) :-
            write('You want to add the typeface '),
            write( Value),
            nl,
            write('Is this a regular font (y,or,n)? '),
            read(Y),
            (member(Y,[y]), enter_new_regular_font(Value)).

enter_new_regular_font(Font) :-
            add_frame(bk,Font:[is_a:[value:[regular_type,typeface_type]]]),
            write('What is the italic form of '),
            write(Font),
            write('?'),
            nl,
            read(Answer),
            add_frame(bk,Answer:[is_a:[value:[italic_type,Font]]]),
            write('What is the bold form of '),
            write(Font),
            write('?'),
            nl,
            read(BoldAnswer),
            add_frame(bk,BoldAnswer:[is_a:[value:[bold_type,Font]]]).
```

Sample Session

(using Enhanced Frame Package)

```
balsa-axb9535[10] prolog /usr/local/lib/prolog/frameboot
C-Prolog version 1.4
[ Restoring file /usr/local/lib/prolog/frameboot ]

yes
| ?- start.
Do you want to enforce a consistent system? [y or n]  y.

yes
| ?- load(bk).
slot size loaded successfully.
yes
frame typeface_type loaded successfully.
yes
frame roman loaded successfully.
yes
slot typeface loaded successfully.
yes
frame font_type loaded successfully.
yes
frame bold_type loaded successfully.
yes
frame italic_type loaded successfully.
yes
frame regular_type loaded successfully.
yes
frame roman_bold loaded successfully.
yes
frame roman_italic loaded successfully.
yes
frame book_component loaded successfully.
yes
frame front_matter loaded successfully.
yes
frame title_page_component loaded successfully.
yes
frame title loaded successfully.
yes
frame author loaded successfully.
yes
frame publisher loaded successfully.
yes
frame body_matter loaded successfully.
yes
bk consulted 2508 bytes 8.76666 sec.

yes
| ?- load_demon(bk_demon).
bk_demon reconsulted 1896 bytes 0.666673 sec.

yes
```

```
| ?- trace_on.

yes
| ?- ret_val(title_page_component,size,Value).
--> no value facet value available in KB for slot in_of in frame title_pag
onent
--> value = [front_matter] from slot is_a in frame title_page_component
--> value = [11] inherited from slot size in frame book_component
--> Demon enlarge(3,title_page_component,size,14) is called from slot size
ame title_page_component
--> value = 14 by activating if_needed demon from slot size in frame title
component

Value = 14

yes
| ?- trace_off.

yes
| ?- pf(title).
title:
     [
     is_a:
          value:[title_page_component]
     font:
          if_needed:make_bold
     ]
yes
| ?- ret_val(title,font,Value).

Value = [roman_bold] ;

no
| ?- print_tree(title_page_component).

PARENT FRAME(S):
                    front_matter

QUERY FRAME:
                    title_page_component

CHILD FRAME(S):
                    title
                    author
                    publisher



yes
| ?- change_val(book_component,typeface,baskerville,roman).
Value  baskerville  does not exist in value facet.
```

```
Hence operation fails

no
| ?- pf(book_component).
book_component:
     [
     typeface:
          value:[roman]
          max:1
          min:1
          type:typeface_type
     size:
          value:[11]
          max:1
          min:0
     font:
          value:[regular_type]
          max:1
          min:0
          type:font_type
     ]
yes
| ?- change_val(book_component,typeface,roman,baskerville).
Type violation for baskerville wrt given type typeface_type
  The operation fails.

no
| ?- add_val(typeface_type,available_types,[baskerville,times]).
Entries in value facet outside min max range.
There is a cardinality violation in slot available_types:[value:[baskervil
es],if_added:get_others] .
In frame typeface_type  max = 1 and Min = 0 .
The operation fails.

no
| ?- pf(typeface_type).
typeface_type:
     [
     available_types:
          if_added:get_others
     ]
yes
| ?- add_max(typeface_type,available_types,5).

yes
| ?- add_val(typeface_type,available_types,[baskerville,times]).
You want to add the typeface baskerville
Is this a regular font (y,or,n)? y.
What is the italic form of baskerville?
|: baskerville_italic.
What is the bold form of baskerville?
```

```
|: baskerville_bold.
You want to add the typeface times
Is this a regular font (y,or,n)? y.
What is the italic form of times?
|: times_italic.
What is the bold form of times?
|: times_bold.

yes
| ?- pf(typeface_type).
typeface_type:
     [
     available_types:
          value:[baskerville,times]
          max:5
          if_added:get_others
     ]
yes
| ?- change_val(book_component,typeface,roman,baskerville).

yes
| ?- pf(book_component).
book_component:
     [
     typeface:
          value:[baskerville]
          max:1
          min:1
          type:typeface_type
     size:
          value:[11]
          max:1
          min:0
     font:
          value:[regular_type]
          max:1
          min:0
          type:font_type
     ]
yes
| ?- ret_val(title,font,Value).

Value = [baskerville_bold]

yes
| ?- ka.
slot size alright.
yes
frame typeface_type is consistent.
yes
frame roman is consistent.
```

```
yes
slot typeface alright.
yes
frame font_type is consistent.
yes
frame bold_type is consistent.
yes
frame italic_type is consistent.
yes
frame regular_type is consistent.
yes
frame roman_bold is consistent.
yes
frame roman_italic is consistent.
yes
There is a type violation in slot typeface:[value:[baskerville],max:1,min:
:typeface_type] in frame book_component
Value not defined as typeface_type YET.
<<< Definition will not get considered here if defined later in KB. >>>

no
frame front_matter is consistent.
yes
frame title_page_component is consistent.
yes
frame title is consistent.
yes
frame author is consistent.
yes
frame publisher is consistent.
yes
frame body_matter is consistent.
yes
frame baskerville is consistent.
yes
frame baskerville_italic is consistent.
yes
frame baskerville_bold is consistent.
yes
frame times is consistent.
yes
frame times_italic is consistent.
yes
frame times_bold is consistent.
yes
/tmp/frm1_1989 consulted 2596 bytes 12.25 sec.
/tmp/frm2_1986 consulted 3148 bytes 0.900016 sec.

yes
| ?- check_frame(baskerville).
frame baskerville is consistent.
```

```
yes
/tmp/frm1_1989 consulted 116 bytes 0.600014 sec.
/tmp/frm2_1986 consulted 116 bytes 0.0666951 sec.

yes
| ?- check_frame(book_component).
frame book_component is consistent.
yes
/tmp/frm1_1989 consulted 416 bytes 1.00002 sec.
/tmp/frm2_1986 consulted 416 bytes 0.116675 sec.

yes
| ?- ret_max(title,font,Max).

Max = 1

yes
| ?- trace_on.

yes
| ?- ret_min(front_matter,typeface,Min).
--> min = 1 inherited from slot typeface in frame book_component

Min = 1

yes
| ?- trace_off.

yes
| ?- show_file(bk).
File -->  bk   Frames/Slots -->  [[size],typeface_type,roman,[typeface],f
pe,bold_type,italic_type,regular_type,roman_bold,roman_italic,book_compone
nt_matter,title_page_component,title,author,publisher,body_matter,baskervi
skerville_italic,baskerville_bold,times,times_italic,times_bold]

yes
| ?- frame_subsumes(title_page_component,Ans).

Ans = title ;

Ans = author ;

Ans = publisher ;

no
| ?- frame_match_exact([is_a:[value:[regular_type,typeface_type]]],Ans).

Ans = [times,baskerville] ;

no
| ?- frame_match_subset([is_a:[value:[typeface_type]]],Ans).
```

```
Ans = [roman,times,baskerville] ;

no
| ?- save_file(bk,bk_new).

yes
| ?- exit.
Did you save? Confirm exit! y.

[ Prolog execution halted ]
balsa-axb9535[11] exit
```

bk_new

(saved knowledge base file using the Enhanced Frame Package)

```
?- def_slot(size:
          [max:1,
               min:0
          ]).

?- def_frame(typeface_type:
          [available_types:
               [value:[baskerville,times],
               max:5,
               if_added:get_others]
          ]).

?- def_frame(roman:
          [is_a:
               [value:[typeface_type]]
          ]).

?- def_slot(typeface:
          [type:typeface_type,
               max:1,
               min:0
          ]).

?- def_frame(font_type:
          []).

?- def_frame(bold_type:
          [is_a:
               [value:[font_type]]
          ]).

?- def_frame(italic_type:
          [is_a:
               [value:[font_type]]
          ]).

?- def_frame(regular_type:
          [is_a:
               [value:[font_type]]
          ]).

?- def_frame(roman_bold:
          [is_a:
               [value:[roman,bold_type]]
          ]).

?- def_frame(roman_italic:
          [is_a:
               [value:[roman,italic_type]]
          ]).
```

```
?- def_frame(book_component:
          [typeface:
               [value:[baskerville],
               max:1,
               min:1,
               type:typeface_type],
          size:
               [value:[11],
               max:1,
               min:0],
          font:
               [value:[regular_type],
               max:1,
               min:0,
               type:font_type]
          ]).

?- def_frame(front_matter:
          [is_a:
               [value:[book_component]]
          ]).

?- def_frame(title_page_component:
          [is_a:
               [value:[front_matter]],
          justification:
               [value:[centered]],
          size:
               [if_needed:enlarge(3)]
          ]).

?- def_frame(title:
          [is_a:
               [value:[title_page_component]],
          font:
               [if_needed:make_bold]
          ]).

?- def_frame(author:
          [is_a:
               [value:[title_page_component]],
          font:
               [if_needed:make_italic]
          ]).

?- def_frame(publisher:
          [is_a:
               [value:[title_page_component]]
          ]).

?- def_frame(body_matter:
```

```
          [is_a:
               [value:[book_component]]
          ]).

?- def_frame(baskerville:
          [is_a:
               [value:[regular_type,typeface_type]]
          ]).

?- def_frame(baskerville_italic:
          [is_a:
               [value:[italic_type,baskerville]]
          ]).

?- def_frame(baskerville_bold:
          [is_a:
               [value:[bold_type,baskerville]]
          ]).

?- def_frame(times:
          [is_a:
               [value:[regular_type,typeface_type]]
          ]).

?- def_frame(times_italic:
          [is_a:
               [value:[italic_type,times]]
          ]).

?- def_frame(times_bold:
          [is_a:
               [value:[bold_type,times]]
          ]).
```

INDEX OF FRAME PACKAGE OPERATORS